1b. SAGE Quadratic Forms Source Code ($\sim$ 22,000 lines Python Code in 187 pages)

[Hanh] – J. Hanke

(Included in SAGE)

http://hg.sagemath.org/sage-main/file/f24ce048fa66/sage/quadratic_forms

```
from binary_qf import BinaryQF, BinaryQF_reduced_representatives

from quadratic_form import QuadraticForm, DiagonalQuadraticForm, is_QuadraticFor
m

from random_quadraticform import random_quadraticform, random_quadraticform_with
_conditions

from extras import least_quadratic_nonresidue, extend_to_primitive, is_triangula
r_number

from special_values import gamma__exact, zeta__exact, QuadraticBernoulliNumber,
\
      quadratic_L_function__exact, quadratic_L_function__numerical

from genera.genus import is_GlobalGenus, is_2_adic_genus
#is_trivial_symbol

from constructions import BezoutianQuadraticForm, HyperbolicPlane_quadratic_form

from sage.quadratic_forms.projective_iterators import normalized_finite_projecti
ve_space_generator


from square_classes import local_squareclass_representatives_list, is_SquareClas
s, SquareClass

from weak_approx import weak_approx_for_numbers_over_QQ, weak_approx_for_squarec
lasses_over_QQ,\
                                        strong_approx_for_squareclasses_by_Q
Q_except_at_one_prime

from quadratic_space import QuadraticSpace, local_quadratic_space_anisotropic_di
mension_by_invariants, \
                                        local_quadratic_space_core_invariant
s_from_invariants, \
                                        local_quadratic_space_by_invariants,
\
                                        local_quadratic_space_GHY_to_Standar
d_invariants, \
                                        local_quadratic_space_Standard_to_GH
Y_invariants, \
                                        find_locally_represented_number, \
                                        rational_quadratic_space_from_local_
space_list

from quadratic_lattice import QuadraticLattice

from symmetric_bilinear import DiagonalMatrix, SymmetricBilinearSpace, Symmetric
BilinearLattice

from lattice import Lattice

from localization import Qv
```

```
r"""
Binary Quadratic Forms with Integer Coefficients.

This module provides a specialized class for working with a binary quadratic
form 'a x^2 + b x y + c y^2', stored as a triple of integers '(a, b, c)'.

EXAMPLES::

    sage: Q = BinaryQF([1,2,3])
    sage: Q
    x^2 + 2*x*y  + 3*y^2
    sage: Q.discriminant()
    -8
    sage: Q.reduced_form()
    x^2 + 2*y^2
    sage: Q(1, 1)
    6

TESTS::

    sage: Q == loads(dumps(Q))
    True

AUTHORS:

- Jon Hanke (2006-08-08):

  - Appended to add the methods :func:`BinaryQF_reduced_representatives`,
    :meth:`~BinaryQF.is_reduced`, and ``__add__`` on 8-3-2006 for Coding Sprint
    #2.
  - Added Documentation and :meth:`~BinaryQF.complex_point` method on 8-8-2006.

- Nick Alexander: add doctests and clean code for Doc Days 2
- William Stein (2009-08-05): composition; some ReSTification.
- William Stein (2009-09-18): make immutable.
- Jon Hanke (2011-02-05):  Add spectral decomp. and some nice pictures.
"""

#*****************************************************************************
#       Copyright (C) 2006--2011 William Stein and Jon Hanke
#
#  Distributed under the terms of the GNU General Public License (GPL)
#
#    This code is distributed in the hope that it will be useful,
#    but WITHOUT ANY WARRANTY; without even the implied warranty of
#    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
#    General Public License for more details.
#
#  The full text of the GPL is available at:
#
#                  http://www.gnu.org/licenses/
#*****************************************************************************

from sage.libs.pari.all import pari
from sage.rings.all import (is_fundamental_discriminant, ZZ, divisors, QQ, RR)
from sage.structure.sage_object import SageObject
from sage.misc.cachefunc import cached_method

from sage.matrix.constructor import Matrix
from sage.modules.free_module_element import vector
from sage.functions.other import sqrt, ceil
from sage.functions.trig import arccos
from sage.plot.all import Graphics, point, ellipse, text, graphics_array


class BinaryQF(SageObject):
    """
    A binary quadratic form over 'ZZ'.
```



```
    INPUT:

    - 'v' -- a list or tuple of 3 entries:  [a,b,c], or a quadratic homogeneous
      polynomial in two variables with integer coefficients


    OUTPUT:

    the binary quadratic form a*x^2 + b*x*y + c*y^2.

    EXAMPLES::

        sage: b = BinaryQF([1,2,3])
        sage: b.discriminant()
        -8
        sage: R.<x, y> = ZZ[]
        sage: BinaryQF(x^2 + 2*x*y + 3*y^2) == b
        True
    """
        # Initializes the form with a 3-element list
        def __init__(self, abc):
            r"""
        Creates the binary quadratic form 'ax^2 + bxy + cy^2' from the
        triple [a,b,c] over 'ZZ' or from a polynomial.

        INPUT:

        - ''abc'' -- 3-tuple of integers, or a quadratic homogeneous polynomial
          in two variables with integer coefficients

        EXAMPLES::

            sage: Q = BinaryQF([1,2,3]); Q
            x^2 + 2*x*y + 3*y^2
            sage: Q = BinaryQF([1,2])
            Traceback (most recent call last):
            ...
            TypeError: Binary quadratic form must be given by a list of three coefficients

            sage: R.<x, y> = ZZ[]
            sage: f = x^2 + 2*x*y + 3*y^2
            sage: BinaryQF(f)
            x^2 + 2*x*y + 3*y^2
            sage: BinaryQF(f + x)
            Traceback (most recent call last):
            ...
            TypeError: Binary quadratic form must be given by a quadratic homogeneous bivariate integer polynomial

        TESTS::

            sage: BinaryQF(0)
            0
        """
            if isinstance(abc, (list, tuple)):
                if len(abc) != 3:
                    # Check we have three coefficients
                    raise TypeError, "Binary quadratic form must be given by a list of three coefficients"
                self._a, self._b, self._c = [ZZ(x) for x in abc]
            else:
                f = abc
                from sage.rings.polynomial.multi_polynomial_element import is_MPolyn
omial
                if f.is_zero():
                    self._a, self._b, self._c = [ZZ(0), ZZ(0), ZZ(0)]
                elif (is_MPolynomial(f) and f.is_homogeneous() and f.base_ring() ==
ZZ
                        and f.degree() == 2 and f.parent().ngens() == 2):
                    x, y = f.parent().gens()
                    self._a, self._b, self._c = [f.monomial_coefficient(mon) for mon
```

```
in [x**2, x*y, y**2]]
            else:
                raise TypeError, "Binary quadratic form must be given by a quadratic homogeneous bi
variate integer polynomial"

    def _pari_init_(self):
        """
    Used to convert this quadratic form to Pari.

    EXAMPLES::

        sage: f = BinaryQF([2,3,4]); f
        2*x^2 + 3*x*y + 4*y^2
        sage: f._pari_init_()
        'Qfb(2,3,4)'
        sage: pari(f)
        Qfb(2, 3, 4)
        sage: type(pari(f))
        <type 'sage.libs.pari.gen.gen'>
        sage: gp(f)
        Qfb(2, 3, 4)
        sage: type(gp(f))
        <class 'sage.interfaces.gp.GpElement'>
    """
        return 'Qfb(%s,%s,%s)'%(self._a,self._b,self._c)

    def __mul__(self, right):
        """
    Gauss composition of binary quadratic forms.  The result is
    not reduced.

    EXAMPLES:

    We explicitly compute in the group of classes of positive
    definite binary quadratic forms of discriminant -23.

    ::

        sage: R = BinaryQF_reduced_representatives(-23); R
        [x^2 + x*y + 6*y^2, 2*x^2 - x*y + 3*y^2, 2*x^2 + x*y + 3*y^2]
        sage: R[0] * R[0]
        x^2 + x*y + 6*y^2
        sage: R[1] * R[1]
        4*x^2 + 3*x*y + 2*y^2
        sage: (R[1] * R[1]).reduced_form()
        2*x^2 + x*y + 3*y^2
        sage: (R[1] * R[1] * R[1]).reduced_form()
        x^2 + x*y + 6*y^2

    """
        if not isinstance(right, BinaryQF):
            raise TypeError, "both self and right must be binary quadratic forms"
        # There could be more elegant ways, but qfbcompraw isn't
        # wrapped yet in the PARI C library.  We may as well settle
        # for the below, until somebody simply implements composition
        # from scratch in Cython.
        v = list(pari('qfbcompraw(%s,%s)'%(self._pari_init_(),
                                           right._pari_init_())))
        return BinaryQF(v)

    def __getitem__(self, n):
        """
    Return the n-th component of this quadratic form.

    If this form is 'a x^2 + b x y + c y^2', the 0-th component is 'a',
    the 1-st component is 'b', and '2'-nd component is 'c'.

    Indexing is like lists -- negative indices and slices are allowed.
```



```
    EXAMPLES::


        sage: Q = BinaryQF([2,3,4])
        sage: Q[0]
        2
        sage: Q[2]
        4
        sage: Q[:2]
        (2, 3)
        sage: tuple(Q)
        (2, 3, 4)
        sage: list(Q)
        [2, 3, 4]
    """
        return (self._a, self._b, self._c)[n]

    def __call__(self, *args):
        r"""
    Evaluate this quadratic form at a point.

    INPUT:

    - args -- x and y values, as a pair x, y or a list, tuple, or
      vector

    EXAMPLES::


        sage: Q = BinaryQF([2, 3, 4])
        sage: Q(1, 2)
        24

    TESTS::

        sage: Q = BinaryQF([2, 3, 4])
        sage: Q([1, 2])
        24
        sage: Q((1, 2))
        24
        sage: Q(vector([1, 2]))
        24
    """
        if len(args) == 1:
            args = args[0]
        x, y = args
        return (self._a * x + self._b * y) * x + self._c * y**2

    def __cmp__(self, right):
        """
    Returns True if self and right are identical: the same coefficients.

    EXAMPLES::


        sage: P = BinaryQF([2,2,3])
        sage: Q = BinaryQF([2,2,3])
        sage: R = BinaryQF([1,2,3])
        sage: P == Q # indirect doctest
        True
        sage: P == R # indirect doctest
        False

    TESTS::

        sage: P == P
        True
        sage: Q == P
        True
```

```
    sage: R == P
    False
    sage: P == 2
    False
"""
        if not isinstance(right, BinaryQF):
            return cmp(type(self), type(right))
        return cmp((self._a,self._b,self._c), (right._a,right._b,right._c))

    def __add__(self, Q):
        """
Returns the component−wise sum of two forms.

That is, given 'a_1 x^2 + b_1 x y + c_1 y^2' and 'a_2 x^2 + b_2 x y +
c_2 y^2', returns the form
'(a_1 + a_2) x^2 + (b_1 + b_2) x y + (c_1 + c_2) y^2.'

EXAMPLES::


    sage: P = BinaryQF([2,2,3]); P
    2*x^2 + 2*x*y + 3*y^2
    sage: Q = BinaryQF([−1,2,2]); Q
    −x^2 + 2*x*y + 2*y^2
    sage: P + Q
    x^2 + 4*x*y + 5*y^2
    sage: P + Q == BinaryQF([1,4,5]) # indirect doctest
    True

TESTS::

    sage: Q + P == BinaryQF([1,4,5]) # indirect doctest
    True
"""
        return BinaryQF([self._a + Q._a, self._b + Q._b, self._c + Q._c])

    def __sub__(self, Q):
        """
Returns the component−wise difference of two forms.

That is, given 'a_1 x^2 + b_1 x y + c_1 y^2' and 'a_2 x^2 + b_2 x y +
c_2 y^2', returns the form
'(a_1 − a_2) x^2 + (b_1 − b_2) x y + (c_1 − c_2) y^2.'

EXAMPLES::


    sage: P = BinaryQF([2,2,3]); P
    2*x^2 + 2*x*y + 3*y^2
    sage: Q = BinaryQF([−1,2,2]); Q
    −x^2 + 2*x*y + 2*y^2
    sage: P − Q
    3*x^2 + y^2
    sage: P − Q == BinaryQF([3,0,1]) # indirect doctest
    True

TESTS::

    sage: Q − P == BinaryQF([3,0,1]) # indirect doctest
    False
    sage: Q − P != BinaryQF([3,0,1]) # indirect doctest
    True
"""
        return BinaryQF([self._a - Q._a, self._b - Q._b, self._c - Q._c])

    def _repr_(self):
        """
Display the quadratic form.
```



```
EXAMPLES::


    sage: Q = BinaryQF([1,2,3]); Q # indirect doctest
    x^2 + 2*x*y + 3*y^2

    sage: Q = BinaryQF([−1,2,3]); Q
    −x^2 + 2*x*y + 3*y^2

    sage: Q = BinaryQF([0,0,0]); Q
    0
"""
        return repr(self.polynomial())

    def _latex_(self):
        """
Return latex representation of this binary quadratic form.

EXAMPLES::

    sage: f = BinaryQF((778,1115,400)); f
    778*x^2 + 1115*x*y + 400*y^2
    sage: latex(f) # indirect doctest
    778 x^{2} + 1115 x y + 400 y^{2}
"""
        return self.polynomial()._latex_()

    @cached_method
    def polynomial(self):
        """
Returns the binary quadratic form as a homogeneous 2−variable
polynomial.

EXAMPLES::

    sage: Q = BinaryQF([1,2,3])
    sage: Q.polynomial()
    x^2 + 2*x*y + 3*y^2

    sage: Q = BinaryQF([−1,−2,3])
    sage: Q.polynomial()
    −x^2 − 2*x*y + 3*y^2

    sage: Q = BinaryQF([0,0,0])
    sage: Q.polynomial()
    0
"""
        return self(ZZ['x,y'].gens())

    @cached_method
    def discriminant(self):
        """
Returns the discriminant 'b^2 − 4ac' of the binary
form 'ax^2 + bxy + cy^2'.

EXAMPLES::


    sage: Q = BinaryQF([1,2,3])
    sage: Q.discriminant()
    −8
"""
        return self._b**2 - 4 * self._a * self._c

    @cached_method
    def has_fundamental_discriminant(self):
        """
Checks if the discriminant D of this form is a fundamental
discriminant (i.e. D is the smallest element of its
```

squareclass with D = 0 or 1 mod 4).

EXAMPLES::

```
    sage: Q = BinaryQF([1,0,1])
    sage: Q.discriminant()
    -4
    sage: Q.has_fundamental_discriminant()
    True

    sage: Q = BinaryQF([2,0,2])
    sage: Q.discriminant()
    -16
    sage: Q.has_fundamental_discriminant()
    False
"""
        return is_fundamental_discriminant(self.discriminant())

    @cached_method
    def is_primitive(self):
        """
```

Checks if the form 'ax^2 + bxy + cy^2' satisfies '\gcd(a,b,c)=1', i.e., is primitive.

EXAMPLES::

```
    sage: Q = BinaryQF([6,3,9])
    sage: Q.is_primitive()
    False

    sage: Q = BinaryQF([1,1,1])
    sage: Q.is_primitive()
    True

    sage: Q = BinaryQF([2,2,2])
    sage: Q.is_primitive()
    False

    sage: rqf = BinaryQF_reduced_representatives(-23*9)
    sage: [qf.is_primitive() for qf in rqf]
    [True, True, True, False, True, True, False, False, True]
    sage: rqf
    [x^2 + x*y + 52*y^2,
    2*x^2 - x*y + 26*y^2,
    2*x^2 + x*y + 26*y^2,
    3*x^2 + 3*x*y + 18*y^2,
    4*x^2 - x*y + 13*y^2,
    4*x^2 + x*y + 13*y^2,
    6*x^2 - 3*x*y + 9*y^2,
    6*x^2 + 3*x*y + 9*y^2,
    8*x^2 + 7*x*y + 8*y^2]
    sage: [qf for qf in rqf if qf.is_primitive()]
    [x^2 + x*y + 52*y^2,
    2*x^2 - x*y + 26*y^2,
    2*x^2 + x*y + 26*y^2,
    4*x^2 - x*y + 13*y^2,
    4*x^2 + x*y + 13*y^2,
    8*x^2 + 7*x*y + 8*y^2]
    """
        from sage.rings.arith import gcd
        return gcd([self._a, self._b, self._c])==1

    @cached_method
    def is_weakly_reduced(self):
        """
```

Checks if the form 'ax^2 + bxy + cy^2' satisfies '|b| \leq a \leq c', i.e., is weakly reduced.

EXAMPLES::

```
    sage: Q = BinaryQF([1,2,3])
    sage: Q.is_weakly_reduced()
    False

    sage: Q = BinaryQF([2,1,3])
    sage: Q.is_weakly_reduced()
    True

    sage: Q = BinaryQF([1,-1,1])
    sage: Q.is_weakly_reduced()
    True
    """
        if self.discriminant() >= 0:
            raise NotImplementedError, "only implemented for negative discriminants"
        return (abs(self._b) <= self._a) and (self._a <= self._c)

    @cached_method
    def reduced_form(self):
        """
```

Return the unique reduced form equivalent to ''self''. See also :meth:'~is_reduced'.

EXAMPLES::

```
    sage: a = BinaryQF([33,11,5])
    sage: a.is_reduced()
    False
    sage: b = a.reduced_form(); b
    5*x^2 - x*y + 27*y^2
    sage: b.is_reduced()
    True

    sage: a = BinaryQF([15,0,15])
    sage: a.is_reduced()
    True
    sage: b = a.reduced_form(); b
    15*x^2 + 15*y^2
    sage: b.is_reduced()
    True
    """
        if self.discriminant() >= 0 or self._a < 0:
            raise NotImplementedError, "only implemented for positive definite forms"
        if not self.is_reduced():
            v = list(pari('Vec(qfbred(Qfb(%s,%s,%s)))'%(self._a,self._b,self._c)))
            return BinaryQF(v)
        else:
            return self

    def is_equivalent(self, right):
        """
```

Return true if self and right are equivalent, i.e., have the same reduced form.

INPUT:

- ''right'' -- a binary quadratic form

EXAMPLES::

```
    sage: a = BinaryQF([33,11,5])
    sage: b = a.reduced_form(); b
    5*x^2 - x*y + 27*y^2
    sage: a.is_equivalent(b)
    True
    sage: a.is_equivalent(BinaryQF((3,4,5)))
    False
```

```
    """
        if not isinstance(right, BinaryQF):
            raise TypeError, "right must be a binary quadratic form"
        return self.reduced_form() == right.reduced_form()

    @cached_method
    def is_reduced(self):
        """
    Checks if the quadratic form is reduced, i.e., if the form
    'ax^2 + bxy + cy^2' satisfies 'lb|\leq a \leq c', and
    that 'b\geq 0' if either 'a = b' or 'a = c'.

    EXAMPLES::

       sage: Q = BinaryQF([1,2,3])
       sage: Q.is_reduced()
       False

       sage: Q = BinaryQF([2,1,3])
       sage: Q.is_reduced()
       True

       sage: Q = BinaryQF([1,-1,1])
       sage: Q.is_reduced()
       False

       sage: Q = BinaryQF([1,1,1])
       sage: Q.is_reduced()
       True
    """
        return (-self._a < self._b <= self._a < self._c) or \
               (ZZ(0) <= self._b <= self._a == self._c)

    def complex_point(self):
        r"""
    Returns the point in the complex upper half-plane associated
    to this (positive definite) quadratic form.

    For positive definite forms with negative discriminants, this is a
    root '\tau' of 'a x^2 + b x + c' with the imaginary part of '\tau'
    greater than 0.

    EXAMPLES::

       sage: Q = BinaryQF([1,0,1])
       sage: Q.complex_point()
       1.00000000000000*I
    """
        if self.discriminant() >= 0:
            raise NotImplementedError, "only implemented for negative discriminant"
        R = ZZ['x']
        x = R.gen()
        Q1 = R(self.polynomial()(x,1))
        return [z  for z in Q1.complex_roots()  if z.imag() > 0][0]

    def matrix_action_left(self, M):
        r"""
    Return the binary quadratic form resulting from the left action
    of the 2-by-2 matrix ''M'' on the quadratic form ''self''.

    Here the action of the matrix 'M = \begin{pmatrix} a & b \\ c & d
    \end{pmatrix}' on the form 'Q(x, y)' produces the form 'Q(ax+by,
    cx+dy)'.

    EXAMPLES::

       sage: Q = BinaryQF([2, 1, 3]); Q
       2*x^2 + x*y + 3*y^2
       sage: M = matrix(ZZ, [[1, 2], [3, 5]])
```



```
       sage: Q.matrix_action_left(M)
       32*x^2 + 109*x*y + 93*y^2
    """
        v, w = M.columns()
        a1 = self(v)
        c1 = self(w)
        b1 = self(v + w) - a1 - c1
        return BinaryQF([a1, b1, c1])

    def matrix_action_right(self, M):
        r"""
    Return the binary quadratic form resulting from the right action
    of the 2-by-2 matrix ''M'' on the quadratic form ''self''.

    Here the action of the matrix 'M = \begin{pmatrix} a & b \\ c & d
    \end{pmatrix}' on the form 'Q(x, y)' produces the form 'Q(ax+cy,
    bx+dy)'.

    EXAMPLES::

       sage: Q = BinaryQF([2, 1, 3]); Q
       2*x^2 + x*y + 3*y^2
       sage: M = matrix(ZZ, [[1, 2], [3, 5]])
       sage: Q.matrix_action_right(M)
       16*x^2 + 83*x*y + 108*y^2
    """
        v, w = M.rows()
        a1 = self(v)
        c1 = self(w)
        b1 = self(v + w) - a1 - c1
        return BinaryQF([a1, b1, c1])



    def matrix_Gram(self):
        """
    Returns the (symmetric) Gram matrix G of the
    quadratic form, which expresses Q(x,y) as the
    matrix product Q(x,y) = [x y] * G * [x]
                            [y].

    This matrix has base_ring QQ.

    """
        return Matrix(QQ, 2, 2, [QQ(self._a), QQ(self._b)/2, QQ(self._b)/2, QQ(s
elf._c)])


    def matrix_Hessian(self):
        """
    Returns the Hessian matrix (of mixed partial derivatives)
    of the quadratic form, which is twice the Gram matrix.
    This matrix has base_ring ZZ.
    """
        return Matrix(ZZ, 2, 2, [2 * self._a, self._b, self._b, 2*self._c])



    def eigenspaces(self):
        """
    Return the eigenspaces of the Hessian matrix of this quadratic form.

    TO DO: Cache this, and make sure the BQF is immutable!
    """
        return self.matrix_Hessian().eigenspaces()



    def spectral_decomposition(self):
```

```
        """
    Return the spectral decomposition matrices (D, X) so that
    the Hessian matrix H (which is twice the Gram matrix of Q)
    can be written as H = X^{t} * D * X where D is a diagonal
    matrix and X is an orthogonal matrix.

    INPUT:
        None

    OUTPUT:
        D -- a diagonal matrix
        X -- an orthogonal matrix

    EXAMPLES:
        sage: B = BinaryQF([1,1,1])
        sage: B.eigenspaces()
        [
        (3, Vector space of degree 2 and dimension 1 over Rational Field
        User basis matrix:
        [1 1]),
        (1, Vector space of degree 2 and dimension 1 over Rational Field
        User basis matrix:
        [ 1 -1])
        ]

        sage: B = BinaryQF([1,0,2])
        sage: B.eigenspaces()
        [
        (4, Vector space of degree 2 and dimension 1 over Rational Field
        User basis matrix:
        [0 1]),
        (2, Vector space of degree 2 and dimension 1 over Rational Field
        User basis matrix:
        [1 0])
        ]
    """
        E = self.eigenspaces()

        ## Make the diagonal matrix D
        D_list = []
        for W in E:
            D_list += [W[0][0]] * W[0][1].dimension()
        D = Matrix(QQ, 2, 2, 0)
        for i in range(2):
            D[i,i] = D_list[i]


        ## Make the orthogonal matrix X
        M_rows = []
        for W in E:
            for v in W.basis():
                v_len = sqrt(v.dot_product(v))
                M_rows.append(v / v_len)
        X = Matrix(M_rows).transpose()

        ## Return the spectral decomposition
        return D, X



    def major_axis(self):
        """
    This assumes that the quadratic form is definite,
    and returns a vector in the eigenspace with the largest
    size eigenvalue.  If there is a single eigenspace, then
    we return the standard basis vector e1 = (1,0)!

    INPUT:
        None
```



```
    OUTPUT:
        a vector

    EXAMPLES:
        sage: B = BinaryQF([1,1,1])
        sage: B.major_axis()
        (1, -1)
        sage: B.minor_axis()
        (1, 1)

        sage: B = BinaryQF([1,0,2])
        sage: B.major_axis()
        (1, 0)
        sage: B.minor_axis()
        (0, 1)

        sage: B = BinaryQF([1,0,1])
        sage: B.major_axis()
        (1, 0)
        sage: B.minor_axis()
        (0, 1)

    """
        E = self.eigenspaces()
        if len(E) == 1:
            return vector([1,0])
        else:
            ## Choose the *smallest* eigenvalue for the major axis
            if abs(E[0][0]) < abs(E[1][0]):
                return E[0][1].basis()[0]
            else:
                return E[1][1].basis()[0]


    def minor_axis(self):
        """
    This assumes that the quadratic form is definite,
    and returns a vector in the eigenspace with the largest
    size eigenvalue.  If there is a single eigenspace, then
    we return the standard basis vector e2 = (0,1).

    INPUT:
        None

    OUTPUT:
        a vector

    EXAMPLES:
        sage: B = BinaryQF([1,1,1])
        sage: B.major_axis()
        (1, -1)
        sage: B.minor_axis()
        (1, 1)

        sage: B = BinaryQF([1,0,2])
        sage: B.major_axis()
        (1, 0)
        sage: B.minor_axis()
        (0, 1)

        sage: B = BinaryQF([1,0,1])
        sage: B.major_axis()
        (1, 0)
        sage: B.minor_axis()
        (0, 1)

    """
        E = self.eigenspaces()
```

```
        if len(E) == 1:
            return vector([0,1])
        else:
            ## Choose the *largest* eigenvalue for the major axis
            if abs(E[0][0]) > abs(E[1][0]):
                return E[0][1].basis()[0]
            else:
                return E[1][1].basis()[0]



    def plot_level_set(self, m, x_max = None, y_max = None, legend_option=None,
return_graphic_object=True, opacity=None, \
                       ellipse_alpha=0.3, ellipse_fill=True, ellipse_rgbcolor=(1
,0,0), \
                       intersection_point_rgbcolor=(0,0,1), intersection_point_s
ize=15, \
                       background_point_rgbcolor=(0,0,1), background_point_size=
4):
        """
    Returns a graphic object that shows the level set of
    this binary quadratic form with respect to the standard
    basis, showing both the background lattice, the level set
    ellipse, and the lattice points on this level set.

    TO DO: Allow arguments for:
      x_max
      y_max
      legend_center -- ok
      ----------------
      x_range
      y_range
      point_color
      point_size
      ellipse_color
      ellipse_thickness
      intersection_point_size
      intersection_point_color
      return_plot
      aspect_ratio
      legend_center
      hide_legend

    INPUT:
      m -- an integer

    OUTPUT:
      a graphics object

    EXAMPLES:
      B = BinaryQF([1,1,1])
      sage: B.plot_level_set(13)
      sage: B.plot_level_set(409)
      sage: B.plot_level_set(11, legend_center=(3,3.5))
    """

        ## legend_label='$x^2 + xy + y^2$'
        ##  legend_label = str(Q([i,j]))

                    #G += point([(i,j)], size=15, rgbcolor=(1,0,0))
        #G += circle((0,0), sqrt(13), rgbcolor=(1,0,0))


        sqrt_m = sqrt(m)

        ## Find the eigenvalues and eigenvectors
        v1 = self.major_axis()
        v2 = self.minor_axis()
        euclid_len1 = sqrt(v1.dot_product(v1))
```



```
        euclid_len2 = sqrt(v2.dot_product(v2))
        Q_len1 = sqrt(self(v1))
        Q_len2 = sqrt(self(v2))

        ## Find the ellipse information
        if v1[1] < 0:
            upper_v1 = -v1   ## ensure the major axis vector is in upper half-pl
ane, so Arccos gives the correct angle!
        else:
            upper_v1 = v1
        angle = arccos(RR(upper_v1[0])/euclid_len1)
        major_radius = sqrt_m * euclid_len1 / Q_len1
        minor_radius = sqrt_m * euclid_len2 / Q_len2


        ## Determine bounds for our lattice point region
        x_max = ceil(major_radius)
        y_max = x_max

        ## Draw the lattice and intersection points
        G = Graphics()
        for i in range(-x_max, x_max + 1):
            for j in range(-y_max, y_max + 1):
                if self([i,j]) != m:
                    G += point([(i,j)], rgbcolor=background_point_rgbcolor, size
=background_point_size)
                else:
                    G += point([(i,j)], rgbcolor=intersection_point_rgbcolor, si
ze=intersection_point_size)


        ## Draw the shaded ellipse
        G += ellipse((0,0), major_radius, minor_radius, angle, fill=ellipse_fill
, alpha=ellipse_alpha, rgbcolor=ellipse_rgbcolor)


        ## Add a label
        if legend_option == None:
            legend_center = (x_max - 1, y_max - 1 + .5)
        elif legend_option == "top_right":
            legend_center = (.5 * x_max, 1.2 * y_max)
        elif isinstance(legend_option, (tuple, list)) and len(legend_option) ==
2:
            legend_center = legend_option
        else:
            legend_center = (0,0)
        Label = text('$' + str(self.polynomial()).replace('*', '') + ' = ' + str(
m) + '$', legend_center, fontsize=15, rgbcolor=(0,0,1))
        Label.axes(False)

        #print "legend_option = " + str(legend_option)
        #print "legend_center = " + str(legend_center)

        if not legend_option in ["hidden", "bottom", "top", "separate"]:
            print "Adding the label!"
            G += Label


        ## Return the plotted graphic
        G.axes_labels(['$x$','$y$'])
        G.set_aspect_ratio(1)
        if return_graphic_object:
            if legend_option == "separate":
                return G, Label
            elif legend_option == "top":
                GA = graphics_array([[Label],[G]])
                return GA
            elif legend_option == "bottom":
                GA = graphics_array([[G],[Label]])
```

```
                return GA
            else:
                return G
        else:
            return G.show()




def BinaryQF_reduced_representatives(D, primitive_only=False):
    r"""
    Returns a list of inequivalent reduced representatives for the
    equivalence classes of positive definite binary forms of
    discriminant D.

    INPUT:

    - 'D' -- (integer) A negative discriminant.

    - ''primitive_only'' -- (bool, default False) flag controlling whether only
      primitive forms are included.

    OUTPUT:

    (list) A lexicographically-ordered list of inequivalent reduced
    representatives for the equivalence classes of positive definite binary
    forms of discriminant 'D'.  If ''primitive_only'' is ''True'' then
    imprimitive forms (which only exist when 'D' is not fundamental) are
    omitted; otherwise they are included.

    EXAMPLES::

        sage: BinaryQF_reduced_representatives(-4)
        [x^2 + y^2]

        sage: BinaryQF_reduced_representatives(-163)
        [x^2 + x*y + 41*y^2]

        sage: BinaryQF_reduced_representatives(-12)
        [x^2 + 3*y^2, 2*x^2 + 2*x*y + 2*y^2]

        sage: BinaryQF_reduced_representatives(-16)
        [x^2 + 4*y^2, 2*x^2 + 2*y^2]

        sage: BinaryQF_reduced_representatives(-63)
        [x^2 + x*y + 16*y^2, 2*x^2 - x*y + 8*y^2, 2*x^2 + x*y + 8*y^2, 3*x^2 + 3*x*y + 6*y^2, 4*x^2 + x*y + 4*y^2]

    The number of inequivalent reduced binary forms with a fixed negative
    fundamental discriminant D is the class number of the quadratic field
    'Q(\sqrt{D})'::

        sage: len(BinaryQF_reduced_representatives(-13*4))
        2
        sage: QuadraticField(-13*4, 'a').class_number()
        2
        sage: p=next_prime(2^20); p
        1048583
        sage: len(BinaryQF_reduced_representatives(-p))
        689
        sage: QuadraticField(-p, 'a').class_number()
        689

        sage: BinaryQF_reduced_representatives(-23*9)
        [x^2 + x*y + 52*y^2,
        2*x^2 - x*y + 26*y^2,
        2*x^2 + x*y + 26*y^2,
        3*x^2 + 3*x*y + 18*y^2,
```



```
        4*x^2 - x*y + 13*y^2,
        4*x^2 + x*y + 13*y^2,
        6*x^2 - 3*x*y + 9*y^2,
        6*x^2 + 3*x*y + 9*y^2,
        8*x^2 + 7*x*y + 8*y^2]
        sage: BinaryQF_reduced_representatives(-23*9, primitive_only=True)
        [x^2 + x*y + 52*y^2,
        2*x^2 - x*y + 26*y^2,
        2*x^2 + x*y + 26*y^2,
        4*x^2 - x*y + 13*y^2,
        4*x^2 + x*y + 13*y^2,
        8*x^2 + 7*x*y + 8*y^2]

    TESTS::

        sage: BinaryQF_reduced_representatives(5)
        Traceback (most recent call last):
        ...
        ValueError: discriminant must be negative and congruent to 0 or 1 modulo 4
    """
        D = ZZ(D)
        if not ( D < 0 and (D % 4 in [0,1])):
            raise ValueError, "discriminant must be negative and congruent to 0 or 1 modulo 4"

        # For a fundamental discriminant all forms are primitive so we need not chec
k:
        if primitive_only:
            primitive_only = not is_fundamental_discriminant(D)

        form_list = []

        from sage.misc.all import xsrange
        from sage.rings.arith import gcd

        # Only iterate over positive a and over b of the same
        # parity as D such that 4a^2 + D <= b^2 <= a^2
        for a in xsrange(1,1+((-D)//3).isqrt()):
            a4 = 4*a
            s = D + a*a4
            w = 1+(s-1).isqrt() if s > 0 else 0
            if w%2 != D%2: w += 1
            for b in xsrange(w,a+1,2):
                t = b*b-D
                if t % a4 == 0:
                    c = t // a4
                    if (not primitive_only) or gcd([a,b,c])==1:
                        if b>0 and a>b and c>a:
                            form_list.append(BinaryQF([a,-b,c]))
                        form_list.append(BinaryQF([a,b,c]))

        form_list.sort()
        return form_list
```

```
"""
Some Extras
"""
##
## Some extra routines to make the QuadraticForm class more useful.
##

from sage.rings.integer import is_Integer
from sage.rings.all import ZZ
from sage.rings.polynomial.polynomial_element import is_Polynomial
from sage.rings.polynomial.polynomial_ring_constructor import PolynomialRing
from sage.quadratic_forms.quadratic_form import QuadraticForm



def BezoutianQuadraticForm(f, g):
    r"""
  Compute the Bezoutian of two polynomials defined over a common base ring.  This is defined by

  .. math::

     {\rm Bez}(f, g) := \frac{f(x) g(y) - f(y) g(x)}{y - x}

  and has size defined by the maximum of the degrees of 'f' and 'g'.

  INPUT:

  - 'f', 'g' -- polynomials in 'R[x]', for some ring 'R'

  OUTPUT:

  a quadratic form over 'R'

  EXAMPLES::

     sage: R = PolynomialRing(ZZ, 'x')
     sage: f = R([1,2,3])
     sage: g = R([2,5])
     sage: Q = BezoutianQuadraticForm(f, g) ; Q
     Quadratic form in 2 variables over Integer Ring with coefficients:
     [ 1 -12 ]
     [ * -15 ]

  AUTHORS:

  - Fernando Rodriguez-Villegas, Jonathan Hanke -- added on 11/9/2008

  """
    ## Check that f and g are polynomials with a common base ring
    if not is_Polynomial(f) or not is_Polynomial(g):
        raise TypeError, "Oops!  One of your inputs is not a polynomial. =("
    if f.base_ring() != g.base_ring():                    ## TO DO:  Change this
to allow coercion!
        raise TypeError, "Oops!  These polynomials are not defined over the same coefficient ring."

    ## Initialize the quadratic form
    R = f.base_ring()
    P = PolynomialRing(R, ['x','y'])
    a, b = P.gens()
    n = max(f.degree(), g.degree())
    Q = QuadraticForm(R, n)

    ## Set the coefficients of Bezoutian
    bez_poly = (f(a) * g(b) - f(b) * g(a)) // (b - a)    ## Truncated (exact) di
vision here
    for i in range(n):
        for j in range(i, n):
            if i == j:
                Q[i,j] = bez_poly.coefficient({a:i,b:j})
            else:
                Q[i,j] = bez_poly.coefficient({a:i,b:j}) * 2

    return Q


def HyperbolicPlane_quadratic_form(R, r=1):
    """
  Constructs the direct sum of 'r' copies of the quadratic form 'xy'
  representing a hyperbolic plane defined over the base ring 'R'.

  INPUT:

  - 'R': a ring
  - 'n' (integer, default 1) number of copies

  EXAMPLES::

     sage: HyperbolicPlane_quadratic_form(ZZ)
     Quadratic form in 2 variables over Integer Ring with coefficients:
     [ 0 1 ]
     [ * 0 ]

  """
    r = ZZ(r)
    ## Check that the multiplicity is a natural number
    if r < 1:
        raise TypeError, "The multiplicity r must be a natural number."

    H = QuadraticForm(R, 2, [0, 1, 0])
    return sum([H  for i in range(r-1)], H)
```

```
r"""
Optimised Cython code for counting congruence solutions
"""

include "../ext/cdefs.pxi"
include "../ext/gmp.pxi"

from sage.rings.arith import valuation, kronecker_symbol, is_prime
from sage.rings.finite_rings.integer_mod import IntegerMod, Mod
from sage.rings.finite_rings.integer_mod_ring import IntegerModRing

from sage.rings.integer_ring import ZZ

from sage.rings.finite_rings.integer_mod cimport IntegerMod_gmp
from sage.sets.set import Set




def extract_sublist_indices(Biglist, Smalllist):
    """
    Returns the indices of Biglist which index the entries of
    Smalllist appearing in Biglist.  (Note that Smalllist may not be a
    sublist of Biglist.)

    NOTE 1: This is an internal routine which deals with re-indexing
    lists, and is not exported to the QuadraticForm namespace!

    NOTE 2: This should really by applied only when BigList has no
    repeated entries.

    TO DO: *** Please revisit this routine, and eliminate it! ***

    INPUT:
        Biglist, Smalllist -- two lists of a common type, where Biglist has no
        repeated entries.

    OUTPUT:
        a list of integers >= 0

    EXAMPLES::

        sage: from sage.quadratic_forms.quadratic_form__local_density_congruence
 import extract_sublist_indices

        sage: biglist = [1,3,5,7,8,2,4]
        sage: sublist = [5,3,2]
        sage: sublist == [biglist[i]  for i in extract_sublist_indices(biglist,
sublist)]  ## Ok whenever Smalllist is a sublist of Biglist
        True

        sage: extract_sublist_indices([1,2,3,6,9,11], [1,3,2,9])
        [0, 2, 1, 4]

        sage: extract_sublist_indices([1,2,3,6,9,11], [1,3,10,2,9,0])
        [0, 2, 1, 4]

        sage: extract_sublist_indices([1,3,5,3,8], [1,5])
        Traceback (most recent call last):
        ...
        TypeError: Biglist must not have repeated entries!
    """
    ## Check that Biglist has no repeated entries
    Big_set = Set(Biglist)
    if len(Set(Biglist)) != len(Biglist):
        raise TypeError, "Biglist must not have repeated entries!"

    ## Extract the indices of Biglist needed to make Sublist
    index_list = []
```



```
    for x in Smalllist:
        try:
            index_list.append(Biglist.index(x))
        except ValueError:                                      ## This happens whe
n an entry of Smalllist is not contained in Biglist
            None

    ## Return the list if indices
    return index_list



def count_modp__by_gauss_sum(n, p, m, Qdet):
    """
    Returns the number of solutions of Q(x) = m over the finite field
    Z/pZ, where p is a prime number > 2 and Q is a non-degenerate
    quadratic form of dimension n >= 1 and has Gram determinant Qdet.

    REFERENCE:
        These are defined in Table 1 on p363 of Hanke's "Local
        Densities..." paper.

    INPUT:

    - n -- an integer >= 1
    - p -- a prime number > 2
    - m -- an integer
    - Qdet -- a integer which is non-zero mod p

    OUTPUT:
        an integer >= 0

    EXAMPLES::

        sage: from sage.quadratic_forms.count_local_2 import count_modp__by_gaus
s_sum

        sage: count_modp__by_gauss_sum(3, 3, 0, 1)    ## for Q = x^2 + y^2 + z^2
  => Gram Det = 1 (mod 3)
        9
        sage: count_modp__by_gauss_sum(3, 3, 1, 1)    ## for Q = x^2 + y^2 + z^2
  => Gram Det = 1 (mod 3)
        6
        sage: count_modp__by_gauss_sum(3, 3, 2, 1)    ## for Q = x^2 + y^2 + z^2
  => Gram Det = 1 (mod 3)
        12

        sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
        sage: [Q.count_congruence_solutions(3, 1, m, None, None) == count_modp__
by_gauss_sum(3, 3, m, 1)  for m in range(3)]
        [True, True, True]


        sage: count_modp__by_gauss_sum(3, 3, 0, 2)    ## for Q = x^2 + y^2 + 2*z
^2  => Gram Det = 2 (mod 3)
        9
        sage: count_modp__by_gauss_sum(3, 3, 1, 2)    ## for Q = x^2 + y^2 + 2*z
^2  => Gram Det = 2 (mod 3)
        12
        sage: count_modp__by_gauss_sum(3, 3, 2, 2)    ## for Q = x^2 + y^2 + 2*z
^2  => Gram Det = 2 (mod 3)
        6

        sage: Q = DiagonalQuadraticForm(ZZ, [1,1,2])
        sage: [Q.count_congruence_solutions(3, 1, m, None, None) == count_modp__
by_gauss_sum(3, 3, m, 2)  for m in range(3)]
        [True, True, True]
```

```
    """
    ## Check that Qdet is non-degenerate
    if Qdet % p == 0:
        raise RuntimeError, "Qdet must be non-zero."

    ## Check that p is prime > 0
    if not is_prime(p) or p == 2:
        raise RuntimeError, "p must be a prime number > 2."

    ## Check that n >= 1
    if n < 1:
        raise RuntimeError, "the dimension n must be >= 1."

    ## Compute the Gauss sum
    neg1 = -1
    if (m % p == 0):
        if (n % 2 != 0):
            count = (p**(n-1))
        else:
            count = (p**(n-1)) + (p-1) * (p**((n-2)/2)) * kronecker_symbol(((neg
1**(n/2)) * Qdet) % p, p)
    else:
        if (n % 2 != 0):
            count = (p**(n-1)) + (p**((n-1)/2)) * kronecker_symbol(((neg1**((n-1
)/2)) * Qdet * m) % p, p)
        else:
            count = (p**(n-1)) - (p**((n-2)/2)) * kronecker_symbol(((neg1**(n/2)
) * Qdet) % p, p)

    ## Return the result
    return count




cdef CountAllLocalTypesNaive_cdef(Q, p, k, m, zvec, nzvec):
    """
    This Cython routine is documented in its Python wrapper method
    QuadraticForm.count_congruence_solutions_by_type().
    """
    cdef long n, i
    cdef long a, b    ## Used to quickly evaluate Q(v)
    cdef long ptr     ## Used to increment the vector
    cdef long solntype    ## Used to store the kind of solution we find


    ## Some shortcuts and definitions
    n = Q.dim()
    R = p ** k
    Q1 = Q.base_change_to(IntegerModRing(R))


    ## Cython Variables
    cdef IntegerMod_gmp zero, one
    zero = IntegerMod_gmp(IntegerModRing(R), 0)
    one = IntegerMod_gmp(IntegerModRing(R), 1)



    ## Initialize the counting vector
    count_vector = [0  for i in range(6)]

    ## Initialize v = (0, ... , 0)
    v = [Mod(0, R)  for i in range(n)]
```



```
    ## Some declarations to speed up the loop
    R_n = R ** n
    m1 = Mod(m, R)

    ## Count the local solutions
    for i from 0 <= i < R_n:

        ## Perform a carry (when value = R-1) until we can increment freely
        ptr = len(v)
        while ((ptr > 0) and (v[ptr-1] == R-1)):
            v[ptr-1] += 1
            ptr += -1

        ## Only increment if we're not already at the zero vector =)
        if (ptr > 0):
            v[ptr-1] += 1


        ## Evaluate Q(v) quickly
        tmp_val = Mod(0, R)
        for a from 0 <= a < n:
            for b from a <= b < n:
                tmp_val += Q1[a,b] * v[a] * v[b]

        ## Sort the solution by it's type
        #if (Q1(v) == m1):
        if (tmp_val == m1):
            solntype = local_solution_type_cdef(Q1, p, v, zvec, nzvec)
            if (solntype != 0):
                count_vector[solntype] += 1


    ## Generate the Bad-type and Total counts
    count_vector[3] = count_vector[4] + count_vector[5]
    count_vector[0] = count_vector[1] + count_vector[2] + count_vector[3]

    ## Return the solution counts
    return count_vector



def CountAllLocalTypesNaive(Q, p, k, m, zvec, nzvec):
    """
    This is an internal routine, which is called by
    :meth:`sage.quadratic_forms.quadratic_form.QuadraticForm.count_congruence_so
lutions_by_type
    QuadraticForm.count_congruence_solutions_by_type`. See the documentation of
    that method for more details.

    INPUT:

    - `Q` -- quadratic form over `\ZZ`
    - `p` -- prime number > 0
    - `k` -- an integer > 0
    - `m` -- an integer (depending only on mod `p^k`)
    - ``zvec``, ``nzvec`` -- a list of integers in ``range(Q.dim())``, or ``None
``

    OUTPUT:
        a list of six integers `\ge 0` representing the solution types: ``[All,
        Good, Zero, Bad, BadI, BadII]``


    EXAMPLES::

        sage: from sage.quadratic_forms.count_local_2 import CountAllLocalTypesN
aive
        sage: Q = DiagonalQuadraticForm(ZZ, [1,2,3])
```

```
        sage: CountAllLocalTypesNaive(Q, 3, 1, 1, None, None)
        [6, 6, 0, 0, 0, 0]
        sage: CountAllLocalTypesNaive(Q, 3, 1, 2, None, None)
        [6, 6, 0, 0, 0, 0]
        sage: CountAllLocalTypesNaive(Q, 3, 1, 0, None, None)
        [15, 12, 1, 2, 0, 2]

    """
    return CountAllLocalTypesNaive_cdef(Q, p, k, m, zvec, nzvec)




cdef local_solution_type_cdef(Q, p, w, zvec, nzvec):
    """
    Internal routine to check if a given solution vector w (of Q(w) =
    m mod p^k) is of a certain local type and satisfies certain
    congruence conditions mod p.

    NOTE: No internal checking is done to test if p is a prime >=2, or
    that Q has the same size as w.

    """
    cdef long i
    cdef long n

    n = Q.dim()


    ## Check if the solution satisfies the zvec "zero" congruence conditions
    ## (either zvec is empty or its components index the zero vector mod p)
    if (zvec == None) or (len(zvec) == 0):
        zero_flag = True
    else:
        zero_flag = False
        i = 0
        while ( (i < len(zvec)) and ((w[zvec[i]] % p) == 0) ):  ## Increment so
long as our entry is zero (mod p)
            i += 1
        if (i == len(zvec)):      ## If we make it through all entries then the
solution is zero (mod p)
            zero_flag = True


    ## DIAGNOSTIC
    #print "IsLocalSolutionType: Finished the Zero congruence condition test \n"

    if (zero_flag == False):
        return <long> 0

    ## DIAGNOSTIC
    #print "IsLocalSolutionType: Passed the Zero congruence condition test \n"


    ## Check if the solution satisfies the nzvec "nonzero" congruence conditions
    ## (nzvec is non-empty and its components index a non-zero vector mod p)
    if (nzvec == None):
        nonzero_flag = True
    elif (len(nzvec) == 0):
        nonzero_flag = False           ## Trivially no solutions in this case!
    else:
        nonzero_flag = False
        i = 0
        while ((nonzero_flag == False) and (i < len(nzvec))):
            if ((w[nzvec[i]] % p) != 0):
                nonzero_flag = True           ## The non-zero condition is satis
```



```
fied when we find one non-zero entry
            i += 1

    if (nonzero_flag == False):
        return <long> 0


    ## Check if the solution has the appropriate (local) type:
    ## -------------------------------------------------------

    ## 1: Check Good-type
    for i from 0 <= i < n:
        if (((w[i] % p) != 0)  and ((Q[i,i] % p) != 0)):
            return <long> 1
    if (p == 2):
        for i from 0 <= i < (n - 1):
            if (((Q[i,i+1] % p) != 0) and (((w[i] % p) != 0) or ((w[i+1] % p) !=
 0))):
                return <long> 1


    ## 2: Check Zero-type
    Zero_flag = True
    for i from 0 <= i < n:
        if ((w[i] % p) != 0):
            Zero_flag = False
    if (Zero_flag == True):
        return <long> 2


    ## Check if wS1 is zero or not
    wS1_nonzero_flag = False
    for i from 0 <= i < n:

        ## Compute the valuation of each index, allowing for off-diagonal terms
        if (Q[i,i] == 0):
            if (i == 0):
                val = valuation(Q[i,i+1], p)    ## Look at the term to the right
            elif (i == n - 1):
                val = valuation(Q[i-1,i], p)    ## Look at the term above
            else:
                val = valuation(Q[i,i+1] + Q[i-1,i], p)    ## Finds the valuatio
n of the off-diagonal term since only one isn't zero
        else:
            val = valuation(Q[i,i], p)

        ## Test each index
        if ((val == 1) and ((w[i] % p) != 0)):
            wS1_nonzero_flag = True


    ## 4: Check Bad-type I
    if (wS1_nonzero_flag == True):
        #print " Bad I Soln :  " + str(w)
        return <long> 4


    ## 5: Check Bad-type II
    if (wS1_nonzero_flag == False):
        #print " Bad II Soln :  " + str(w)
        return <long> 5


    ## Error if we get here! =o
    print "   Solution vector is " + str(w)
    print "   and Q is \n" + str(Q) + "\n"
    raise RuntimeError, "Error in IsLocalSolutionType: Should not execute this l
ine... =( \n"
```

```
from random import random
from sage.matrix.constructor import matrix
from sage.matrix.matrix import is_Matrix
from sage.rings.arith import legendre_symbol
from sage.rings.rational_field import QQ
from sage.rings.integer_ring import ZZ
from sage.rings.infinity import infinity
from sage.misc.functional import squarefree_part

def is_triangular_number(n):
    """
    Determines if the integer n is a triangular number.
    (I.e. determine if n = a*(a+1)/2 for some natural number a.)
    If so, return the number a, otherwise return False.

    Note: As a convention, n=0 is considered triangular for the
    number a=0 only (and not for a=-1).

    WARNING: Any non-zero value will return True, so this will test as
    True iff n is triangular and not zero.  If n is zero, then this
    will return the integer zero, which tests as False, so one must test

        if is_triangular_number(n) != False:

    instead of

        if is_triangular_number(n):

    to get zero to appear triangular.


    INPUT:
        an integer

    OUTPUT:
        either False or a non-negative integer

    EXAMPLES:
        sage: is_triangular_number(3)
        2
        sage: is_triangular_number(1)
        1
        sage: is_triangular_number(2)
        False
        sage: is_triangular_number(0)
        0
        sage: is_triangular_number(-1)
        False
        sage: is_triangular_number(-11)
        False
        sage: is_triangular_number(-1000)
        False
        sage: is_triangular_number(-0)
        0
        sage: is_triangular_number(10^6 * (10^6 +1)/2)
        1000000
    """
    if n < 0:
        return False
    elif n == 0:
        return ZZ(0)
    else:
        from sage.functions.all import sqrt
        ## Try to solve for the integer a
        try:
            disc_sqrt = ZZ(sqrt(1+8*n))
            a = ZZ( (ZZ(-1) + disc_sqrt) / ZZ(2) )
            return a
        except:
            return False
```



```
def extend_to_primitive(A_input):
    """
    Given a matrix (resp. list of vectors), extend it to a square
    matrix (resp. list of vectors), such that its determinant is the
    gcd of its minors (i.e. extend the basis of a lattice to a
    "maximal" one in Z^n).

    Author(s): Gonzalo Tornaria and Jonathan Hanke.

    INPUT:
        a matrix, or a list of length n vectors (in the same space)

    OUTPUT:
        a square matrix, or a list of n vectors (resp.)

    EXAMPLES:
        sage: A = Matrix(ZZ, 3, 2, range(6))
        sage: extend_to_primitive(A)
        [ 0  1  0]
        [ 2  3  0]
        [ 4  5 -1]

        sage: extend_to_primitive([vector([1,2,3])])
        [(1, 2, 3), (0, 1, 0), (0, 0, 1)]

    """
    ## Deal with a list of vectors
    if not is_Matrix(A_input):
        A = matrix(A_input)      ## Make a matrix A with the given rows.
        vec_output_flag = True
    else:
        A = A_input
        vec_output_flag = False


    ## Arrange for A  to have more columns than rows.
    if A.is_square():
        return A
    if A.nrows() > A.ncols():
        return extend_to_primitive(A.transpose()).transpose()

    ## Setup
    k = A.nrows()
    n = A.ncols()
    R = A.base_ring()

    # Smith normal form transformation, assuming more columns than rows
    V = A.smith_form()[2]

    ## Extend the matrix in new coordinates, then switch back.
    B = A * V
    B_new = matrix(R, n-k, n)
    for i in range(n-k):
        B_new[i, n-i-1] = 1
    C = B.stack(B_new)
    D = C * V**(-1)

    ## DIAGNOSTIC
    #print "A = ", A, "\n"
    #print "B = ", B, "\n"
    #print "C = ", C, "\n"
    #print "D = ", D, "\n"

    # Normalize for a positive determinant
```

```
    if D.det() < 0:
        D.rescale_row(n-1, -1)

    ## Return the current information
    if  vec_output_flag:
        return D.rows()
    else:
        return D

def least_quadratic_nonresidue(p):
    """
```

Returns the smallest positive integer quadratic non−residue in Z/pZ for primes p>2.

EXAMPLES::

```
sage: least_quadratic_nonresidue(5)
2
sage: [least_quadratic_nonresidue(p) for p in prime_range(3,100)]
[2, 2, 3, 2, 2, 3, 2, 5, 2, 3, 2, 3, 2, 5, 2, 2, 2, 2, 7, 5, 3, 2, 3, 5]
```

TESTS:

Raises an error if input is a positive composite integer.

::

```
sage: least_quadratic_nonresidue(20)
Traceback (most recent call last):
...
ValueError: Oops!  p must be a prime number > 2.
```

Raises an error if input is 2. This is because every integer is a quadratic residue modulo 2.

::

```
sage: least_quadratic_nonresidue(2)
Traceback (most recent call last):
...
ValueError: Oops!  There are no quadratic non−residues in Z/2Z.
```

```
    """
    from sage.functions.all import floor
    p1 = abs(p)

    ## Deal with the prime p = 2 and |p| <= 1.
    if p1 == 2:
        raise ValueError, "Oops!  There are no quadratic non−residues in Z/2Z."
    if p1 < 2:
        raise ValueError, "Oops!  p must be a prime number > 2."

    ## Find the smallest non-residue mod p
    ## For 7/8 of primes the answer is 2, 3 or 5:
    if p%8 in (3,5):
        return ZZ(2)
    if p%12 in (5,7):
        return ZZ(3)
    if p%5 in (2,3):
        return ZZ(5)
    ## default case (first needed for p=71):
    if not p.is_prime():
        raise ValueError, "Oops!  p must be a prime number > 2."
    from sage.misc.misc import xsrange
    for r in xsrange(7,p):
        if legendre_symbol(r, p) == -1:
            return ZZ(r)
```

```
from sage.modules.free_module import FreeModule_submodule_with_basis_pid

from copy import deepcopy



##############################
## Code for the Lattice class ##
##############################

class Lattice():
    """
    This is a class that gives a finitely generated submodule of a
    K−vectorspace over its ring of integers O_K, where K is a number
    field.

    TO DO: Eventually add support for S−integers as well, where S is a
    set of places of K.
    """


    def __init__(self, V, basis):
        """
        Initializes with the syntax:

            Lattice(V, list_of_generators)
            Lattice(V, row_matrix_of_generators)



        Note:  When subclassing this class overload the __init__() and ambient_space() methods, and change the internal variable.
        """
        ## Check that V is a vectorspace over a number field or its completion a
t a place


        ## Check that basis is a matrix or list of vectors


        ## Check that the ring of integers is a PID
        #raise NotImplementedError, "Presently we only have support for modules
over principal ideal domains."


        ## Initilize the lattice
        self.__lattice_module = FreeModule_submodule_with_basis_pid(V, basis)
        self.__base_ring = V.base_field().ring_of_integers()


    def __repr__(self):
        """
        Returns a string representing the lattice.

        EXAMPLES:
            sage: L = Lattice(QQ^3, [[1,0,0], [1,2,3]])
            sage: L.__repr__()
            Lattice in Vector space of dimension 3 over Rational Field generated by over the ring Integer Ring by
            [
            (1, 0, 0),
            (1, 2, 3)
            ]
        """
        return "Lattice in " + str(self.ambient_space()) + \
               " generated by over the ring " + str(self.base_ring()) + \
               " by \n" + str(self.generators())
```



```


    def ambient_space(self):
        """
        Returns the ambient vector space that this lattice sits in.
        """
        return self.__lattice_module.ambient_vector_space()


    def ambient_dimension(self):
        """
        Returns the ambient vector space that this lattice sits in.
        """
        return self.ambient_space().dim()


    def base_ring(self):
        """
        Returns the ring for which this is a module.
        """
        return deepcopy(self.__base_ring)



    def rank(self):
        """
        Determines if the lattice spans the ambient vector space.
        """
        return self.__lattice_module.rank()


    def is_full_rank(self):
        """
        Determines if the lattice spans the ambient vector space.
        """
        return self.rank() == self.ambient_dimension()


    def is_free(self):
        """
        Determines if the lattice has a (free) basis.
        """
        return True


    def sum_with(self, other):
        """
        Find the sum of this lattice with the lattice L (in the same vector space).
        """
        ## Check that both lattices live on the same ambient space
        if not self.ambient_space() == other.ambient_space():
            raise TypeError, "The two lattices live on different ambient spaces!"

        ## Return the lattice generated by generators of both lattices
        return self.__init__(self.ambient_space(), self.generators() + other.gen
erators())


    def intersect_with(self, other):
        """
        Find the intersection of this lattice with the lattice L (in the same vector space).
        """
        ## Check that both lattices live on the same ambient space
        if not self.ambient_space() == other.ambient_space():
            raise TypeError, "The two lattices live on different ambient spaces!"

        ## Return the intersection of the two quadratic latties
        intersection_basis = self.__lattice_free_module.intersection(other.__lat
tice_free_module).basis()
```

```
        return self.__init__(self.quadratic_space(), intersection_basis)


    def basis(self):
        """
    Returns the list of the basis vectors for the lattice, if the generators form a basis.
    """
        return self.__lattice_module.basis()


    def basis_matrix_of_columns(self):
        """
    Returns a matrix whose columns are the ordered basis for the lattice (if a basis exists).
    """
        return self.__lattice_module.basis_matrix().transpose()


    def basis_matrix_of_rows(self):
        """
    Returns a matrix whose rows are the ordered basis for the lattice (if a basis exists).
    """
        return self.__lattice_module.basis_matrix()


    def generators(self):
        """
    Returns the list of generating vectors for the lattice.
    """
        return self.__lattice_module.basis()


    def generator_matrix_of_columns(self):
        """
    Returns a matrix whose columns are the (ordered) generators for the lattice.
    """
        return self.__lattice_module.basis_matrix().transpose()


    def generator_matrix_of_rows(self):
        """
    Returns a matrix whose rows are the (ordered) generators for the lattice.
    """
        return self.__lattice_module.basis_matrix()


    def apply_linear_transformation_on_left(self, U):
        """
    Returns the lattice generated by the generators of this
    lattice (as a matrix of row vectors), after left-multiplying
    by the matrix U.
    """
        return self.__init__(self.ambient_space(), U * self.generator_matrix_of_
rows())


    def apply_linear_transformation_on_right(self, U):
        """
    Returns the lattice generated by the generators of this
    lattice (as a matrix of column vectors), after right-multiplying
    by the matrix U.
    """
        return self.__init__(self.ambient_space(), U * self.generator_matrix_of_
columns())
```

```
## Routines related to localization of QQ at various places



from sage.rings.padics.factory import Qp
from sage.rings.real_mpfr import RR
from sage.rings.infinity import Infinity



def Qv(v):
    """
  Returns the localization of the rational numbers QQ at the place v.

  INPUT:
    v -- a positive prime number or Infinity

  OUTPUT:
    a local field (RR or Qp(v))

  EXAMPLES:
    sage: Qv(5) == Qp(5)
    True
    sage: Qv(Infinity) == RR
    True

  """
    if v == Infinity:
        return RR
    else:
        return Qp(v)
```

```
from sage.misc.misc import verbose

from random import random
from sage.functions.other import floor, sqrt
from sage.matrix.constructor import matrix
from sage.matrix.matrix import is_Matrix
from sage.rings.arith import valuation, kronecker_symbol, legendre_symbol, hilbe
rt_symbol, is_prime
from sage.rings.rational_field import QQ
from sage.rings.integer_ring import ZZ
from sage.rings.infinity import infinity
from sage.misc.functional import squarefree_part

from sage.rings.polynomial.polynomial_ring_constructor import PolynomialRing
from sage.rings.all import GF
from sage.modules.free_module_element import vector




def find_isotropic_vector_at_prime(G):
    """
  Returns a vector for the bilinear form G over GF(p) which is
  isotropic, and False if there is no such vector.

  INPUT:
    G -- the Gram matrix of a form over GF(p)

  OUTPUT:
    a vector over GF(p)

  EXAMPLES:
    sage: from sage.quadratic_forms.maximal_extras import find_isotropic_vector_at_prime
    sage: A = matrix(GF(3), 2, 2, [3,1,1,3])
    sage: v = find_isotropic_vector_at_prime(A)
    sage: v*A*v.transpose() == 0
    True

  """
    d = G.nrows()


    ## Deal with dimension 0 forms
    if d == 0:
        return False


    p = G.parent().base_ring().characteristic()
    ## Check that G % p is non-degenerate... or allow it an use the kernel.

    ## DIAGNSOTIC
    verbose("G = " + str(G))

    G_det = G.det()                                 ################  WARNING -- TH
IS IS A SAGE ERROR OVER GF(2)!!!
    if G_det == 0:
        raise NotImplementedError, "Must input a non-degenerate matrix over GF(p)."


    ## Deal with dimension 1 forms
    if d == 1:
        return False

    ## Deal with dimension 2 forms
    if d == 2:
        if (p != 2) and (legendre_symbol(-G_det.lift(), p) != 1):       ## Check
 if we don't have a hyperbolic plane!
```



```
            return False


    ## Deal with dimension >=3 forms or hyperbolic plane (so we have isotropic v
ectors)!
    PR = PolynomialRing(GF(p), 'y')
    y = PR.gen()

    while True:                                                         ## This must te
rminate since n >= 3
        ## Choose a random (non-zero) linear polynomial vector in L#/L
        v1 = vector([PR(ZZ.random_element(p))  for i in range(d)])
        while v1 == v1.parent().zero_vector():
            v1 = vector([PR(ZZ.random_element(p))  for i in range(d)])
        v2 = vector([PR(ZZ.random_element(p))  for i in range(d)])
        while v2 == v2.parent().zero_vector():
            v2 = vector([PR(ZZ.random_element(p))  for i in range(d)])
        v = v1 + y * v2

        G1 = matrix(PR, G)
        F = (v * G1 * v.transpose())[0]

        ## Deal with every vector being isotropic
        if F == 0:
            return vector(GF(p), v1)

        ## Otherwise find roots
        F_roots = F.roots()
        if len(F_roots) != 0:
            a = F_roots[0][0]  ## Take the first root over F_p

            new_v = v1 + a*v2
            if new_v != new_v.parent().zero_vector():
                return vector(GF(p), new_v)



def find_basis_of_maximal_isotropic_subspace(G):
    """
  Find a basis of a maximal isotropic subspace of G over GF(p).

  INPUT:
    G -- the Gram matrix of a form over GF(p)

  OUTPUT:
    a matrix of row vectors over GF(p)

  EXAMPLES:
    sage: from sage.quadratic_forms.maximal_extras import find_basis_of_maximal_isotropic_subspace

    sage: MM = matrix(GF(5), 6, 6, [0, 0, 1, 2, 2, 2, 0, 0, 0, 1, 0, 1, 1, 0, 2, 2, 3, 0, 2, 1, 2, 3, 1, 2, 2, 0, 3, 1, 1, 0, 2, 1, 0
, 2, 0, 1])
    sage: find_basis_of_maximal_isotropic_subspace(MM)  ## random
    [2 0 0 3 0 2]
    [0 3 0 1 3 1]
    [0 0 2 2 2 3]

  """
    n = G.nrows()
    p = G.parent().base_ring().characteristic()

    ## Make the transformation matrix (of rows!!!)
    T = matrix(GF(p), 0, n, [])

    ## Find one isotropic vector
    v = find_isotropic_vector_at_prime(G)

    ## Check if we're done.
    if v == False:
```

```
        return T

    ## Find a basis for V^\perp
    K = (G * v.transpose()).kernel().basis_matrix()

    ## DIAGNOSTIC
    verbose("v = " + str(v))
    verbose("K = " + str(K))

    ## Find the first non-zero entry of v, to use to decide which kernel basis v
ector to replace with v.
    for i in range(n):
        if v[i] != 0:
            v_nz_index = i
            break
    ## Find the associated basis vector (using heavily the row echelon form of t
he output)
    for i in range(K.nrows()):
        if K[i, v_nz_index] != 0:
            K_nz_index = i
            break

    ## DIAGNOSIC
    verbose("v_nz_index = " + str(v_nz_index))
    verbose("K_nz_index = " + str(K_nz_index))

    ## Extract the kernel basis excluding v
    K1 = K.matrix_from_rows([j for j in range(K.nrows())  if j != K_nz_index])
    G1 = K1 * G * K1.transpose()

    ## Perform the recursion
    T1 = find_basis_of_maximal_isotropic_subspace(G1)
    T_last = T1 * K1
    T_new = (T_last.transpose().augment(v.transpose())).transpose()   ## Augment
 T_last by adding the row v

    ## DIAGNOSTIC
    verbose("Found T_new of dimension " + str(T_new.nrows()))

    return T_new



def even_neighbor_of_bilinear_gram_matrix(Gram):
    """
  Returns an even neighbor of an odd lattice, which is the lattice
  itself if Gram is even, and the even sublattice of Gram if there
  is no even neighbor.

  INPUT:
    G -- the symmetric (Gram) matrix of a form over ZZ

  OUTPUT:
    a symmetric matrix over ZZ with even diagonal

  EXAMPLES:
    sage: from sage.quadratic_forms.maximal_extras import even_neighbor_of_bilinear_gram_matrix

    sage: B =  matrix(ZZ, 8, 8, [1  for i in range(64)]) + 2
    sage: E, T = even_neighbor_of_bilinear_gram_matrix(B)
    sage: E
    [12  8  8  8  8  8  8  8]
    [ 8  8  6  6  6  6  6  6]
    [ 8  6  8  6  6  6  6  6]
    [ 8  6  6  8  6  6  6  6]
    [ 8  6  6  6  8  6  6  6]
    [ 8  6  6  6  6  8  6  6]
    [ 8  6  6  6  6  6  8  6]
    [ 8  6  6  6  6  6  6  8]
```



```
    sage: T
    [2 1 1 1 1 1 1 1]
    [0 1 0 0 0 0 0 0]
    [0 0 1 0 0 0 0 0]
    [0 0 0 1 0 0 0 0]
    [0 0 0 0 1 0 0 0]
    [0 0 0 0 0 1 0 0]
    [0 0 0 0 0 0 1 0]
    [0 0 0 0 0 0 0 1]

    sage: B = matrix(ZZ, 8, 8, 1)
    sage: E, T = even_neighbor_of_bilinear_gram_matrix(B)  ## This should be the E8 lattice!
    sage: E
    [ 4  2  2  2  2  9 -7 -7]
    [ 2  2  1  1  1  5 -4 -4]
    [ 2  1  2  1  1  5 -4 -4]
    [ 2  1  1  2  1  5 -4 -4]
    [ 2  1  1  1  2  5 -4 -4]
    [ 9  5  5  5  5 22 -17 -17]
    [-7 -4 -4 -4 -4 -17  14  13]
    [-7 -4 -4 -4 -4 -17  13  14]
    sage: T
    [  2   1   1   1   1   1 9/2 -7/2 -7/2]
    [  0   1   0   0   0   0 1/2 -1/2 -1/2]
    [  0   0   1   0   0   0 1/2 -1/2 -1/2]
    [  0   0   0   1   0   0 1/2 -1/2 -1/2]
    [  0   0   0   0   1   0 1/2 -1/2 -1/2]
    [  0   0   0   0   0   0 1/2  1/2 -1/2]
    [  0   0   0   0   0   0 1/2 -1/2  1/2]
    [  0   0   0   0   0   0 1/2 -1/2 -1/2]
    sage: E.det()
    1
  """
    n = Gram.nrows()
    T = matrix(ZZ, n, n, 1)

    Gram_even,trans_even,is_even = even_sublattice_of_bilinear_gram_matrix(Gram)

    ## we already have found an even lattice
    if is_even:
        return Gram, T


################################################################################
###############
    ## better way
    # Diagonalise the  Gram_of_max_lat modulo 2 (if this is odd, otherwise we ar
e fine anyhow)
    # Then either the even sublattice is already the maximal even overlattice
    # or this is generated by the even sublattice and 1/2 of the sum of all (n o
r (n-1))
    # basisvectors of odd norm

    Gram_diag, trans_diagmod2, numberofodd = diagonalise_mod_2(Gram)

    #print "Gram = \n" + str(Gram)
    #print "Gramdiag = \n" + str(Gram_diag)
    #print "trans_diagmod2 = \n" + str(trans_diagmod2)
    #print "numberofodd = \n" + str(numberofodd)


        ## should return a unimodular integral matrix s.t. Gram_diag = trans_dia
gmod2.transpose()*Gram_of_max_lat*trans_diagmod2
        ## is a diagonal matrix modulo 2
        ## and an integer numberofodd, such that the first numberofodd vectors i
n trans_diagmod2 have odd norm
        ## the others have even norm (so if det = 2 then numberoofodd = n-1 else
 n )
    oddity = 0
```

```
    for i in range(numberofodd):
         oddity += Gram_diag[i,i]
    if oddity % 8 == 0:
        halfcolumns = vector([1  for j in range(numberofodd)] + [0  for j in ran
ge(numberofodd, n)])
        newtrans =  matrix(ZZ,(2*trans_diagmod2).augment(halfcolumns.transpose()
)).transpose()
        newtrans_lll = newtrans.LLL().matrix_from_rows(range(1,n+1)).transpose()


        #print "halfcolumns = \n" + str(halfcolumns)
        #print "newtrans = \n" + str(newtrans)
        #print "newtrans_lll = \n" + str(newtrans_lll)
        #print "newtrans_lll.transpose() * Gram_even * newtrans_lll/4 = \n" + st
r(newtrans_lll.transpose() * Gram_even * newtrans_lll/4)


        ## Make the new matrix and transformation for non-zero oddity (mod 8)
        new_A = newtrans_lll.transpose() * Gram_even * newtrans_lll / 4
        new_T = trans_even * newtrans_lll / 2
        print "new_A = \n", new_A
        print "new_T = \n", new_T
        return matrix(ZZ, new_A), new_T

    return Gram_even,  trans_even   ## if oddity is not zero mod 8 then the even
 sublattice is already maximal even
```

```
def split_one_odd_vector_mod2(A):
    """
  This function returns G,T, where T is a unimodular integral matrix
  G = T^tr A T which has first diagonal entry odd and all other
  entries in the first row and column of G are even such that the
  lower right (n−1) submatrix is again odd or has all entries
  divisible by 2.  The flag mult2 says whether the lower right is
  zero mod 2.

  (Note: Here internally we are working again with vectors as columns)

  INPUT:
    A −− an odd symmetric matrix

  OUTPUT:
    G −− odd symmetric matrix with first row/column zero except for the upper left entry (which is odd).
    T −− integral unimodular matrix
    mult2 −− boolean

  EXAMPLES:

  """
    n = A.nrows()
    T = matrix(ZZ,n,n,0)
    ## Find the first odd diagonal entry
    odd_ind = -1
    for i in range(n):
        if A[i,i] % 2 != 0:
            odd_ind = i
            break

    ## Find the associated transformation (shifting the odd entry to the upper-l
eft corner, and we clear the first row/column of the form)
    T[odd_ind, 0] = 1
```



```
    for i in range(odd_ind):
        T[i, i+1] = 1
        if A[odd_ind, i] % 2 == 1:
            T[odd_ind, i+1] = 1
    for i in range(odd_ind+1, n):
        T[i,i] = 1
        if A[odd_ind, i] % 2 == 1:
            T[odd_ind, i] = 1

    ## Make the new matrix and check if it is odd anywhere on the diagonal (in w
hich case we return)
    G = T.transpose() * A * T
    for i in range(1,n):
        if G[i,i] % 2 == 1:
            return G, T, False

    ## Check the off-diagonal entries looking for odd entries
    mult2 = True
    for i in range(1, n):
        if mult2 == False:
            break
        for j in range(i+1, n):
            if G[i,j] % 2 == 1:
                mult2 = False
                odd_i = i
                break

    if mult2 == True:
        return G, T, mult2

    ## Deal with the presence of odd entries:    [ WARNING: THIS DOESN'T SEEM TO
 USE A UNIMODULAR MATRIX HERE! ]
    ##------------------------------------
    T1 = matrix(ZZ,n,n,0)

    ## Find the associated transformation (to arrange that the associated submat
rix has one diagonal entry)
    T1[odd_i, 0] = 1
    T1[0, 0] = 1
    for i in range(1, n):
        T1[i,i] = 1
        if (G[0, i] + G[odd_i, i]) % 2 == 1:
            T1[0, i] = 1

    ## Return the new matrix
    T2 = T * T1
    G = T2.transpose() * A * T2
    return G, T2, False
```

```
def diagonalise_mod_2(A):
    """
  A is an odd symmetric matrix.
  this function returns G,T,a, where T is a unimodular integral matrix
  G = T^tr A T (work again with columns) is diagonal modulo 2
  where the first a diagonal entries of G are odd and all other entries of G are even

  INPUT:
    A −− an odd symmetrix matrix of a form over ZZ

  OUTPUT:
    G −− a symmetric matrix,
    T −− a det 1 integral matrix
    a −− an integer

  EXAMPLES:
```

```
"""
   n = A.nrows()
   T = matrix(ZZ,n,n,1)

   ## Find the first odd diagonal entry
   odd_ind = -1
   for i in range(n):
       if A[i,i] % 2 != 0:
           odd_ind = i
           break
   if odd_ind == -1:
       raise TypeError,"matrix is even"

   mult2 = False
   G = matrix(ZZ,A)
   dimodd = 0
   while not mult2:
       G, T1, mult2 = split_one_odd_vector_mod2(G)
       T = T * (matrix(ZZ, dimodd, dimodd, 1)).block_sum(T1)
       G.subdivide(1,1)
       G = G.subdivision(1,1)
       dimodd += 1
   return T.transpose()*A*T, T, dimodd
```

```
from sage.rings.all import ZZ, Field, Ring
from sage.modules.free_module_element import vector
from sage.rings.arith import is_prime, divisors
from itertools import product
from sage.misc.misc import prod


def normalized_finite_projective_space_generator(n, R):
        """
    Iterator for looping over all normalized vectors in a finite
    projective space P^n(Z/R*Z) or P^n(R) for a finite ring R.  We
    say a vector v is normalized if it has the form

       v = (0, ..., 0, 1, *, ..., *)

    where * represents freely chosen elements of R which are non-zero

    This assumes that R has an iterator and that either R is a
    field, or has methods R.is_unit() and R.is_zero()


    TO DO: This could be moved to a generator in the file
    sage.schemes.generic.projective_space once it supports
    constructions like:

       #sage: R = QuotientRing(ZZ, 7*ZZ)
       #sage: ProjectiveSpace(3, R)

    which currently raises an error!


    INPUT:
       n -- an integer >= 1
       R -- either an integer > 1 or a finite ring.

    OUTPUT:
       a vector of length n, with coefficients in ZZ or in R

    EXAMPLES:
       sage: for v in normalized_finite_projective_space_generator(3, 2): print "v = ", v
       v =  (1, 0, 0)
       v =  (1, 0, 1)
       v =  (1, 1, 0)
       v =  (1, 1, 1)
       v =  (0, 1, 0)
       v =  (0, 1, 1)
       v =  (0, 0, 1)

       sage: for v in normalized_finite_projective_space_generator(3, 3): print "v = ", v
       v =  (1, 0, 0)
       v =  (1, 0, 1)
       v =  (1, 0, 2)
       v =  (1, 1, 0)
       v =  (1, 1, 1)
       v =  (1, 1, 2)
       v =  (1, 2, 0)
       v =  (1, 2, 1)
       v =  (1, 2, 2)
       v =  (0, 1, 0)
       v =  (0, 1, 1)
       v =  (0, 1, 2)
       v =  (0, 0, 1)

            sage: R = QuotientRing(ZZ, 3*ZZ)
            sage: for v in normalized_finite_projective_space_generator(3, R): print "v = ", v
            Traceback (most recent call last):
```



```
            ...
            NotImplementedError: object does not support iteration

    """
        ## Sanity Check -- n >= 1
        if not ((n in ZZ) and (n > 0)):
            raise TypeError, "The number n (giving the dimension of projective space) must be an integ
er > 0."


        ## Setup the first vector, depending on the field_info:
        ## ----------------------------------------------------

        ## Enumerate over [0, ..., R-1]
        if (R in ZZ) and (R > 1):
            if is_prime(R):
                Field_flag = True
            else:
                Field_flag = False

            for i in range(n):
                for d in divisors(R)[:-1]:          ## Run through all proper po
sitive divisors of R.
                    for v in product( * (n-1-i)*[range(R)]):

                        ## Ensure that each non-unit component of v is zero  (wh
ich is automatically true over a field).
                        if Field_flag or (prod([gcd(v[k], R) == 1  for k in rang
e(len(v))]) == 1):
                            yield vector([ZZ(0) for k in range(i)] + [ZZ(d)] + l
ist(v))



        ## Enumerate over a (presumably finite) ring
        elif isinstance(R, Ring):

            if isinstance(R, Field):
                Field_flag = True
                nonzero_ideal_generators = [1]
            else:
                Field_flag = False
                units = [u   for u in R  if u.is_unit()]

                ## Find all non-zero ideal generators
                nonzero_ideal_generators = [1]
                for x in R:
                    if not x.is_zero() and not x.is_unit():

                        ## Add x to the list of no associate if it is already li
sted
                        if [u  for u in units  if x*u in nonzero_ideal_generator
s] == []:
                            nonzero_ideal_generators.append(x)


            for i in range(n):
                for d in nonzero_ideal_generators:
                    for v in product( * (n-1-i)*[R]):

                        ## Ensure that each non-unit component of v is zero  (wh
ich is automatically true over a field).
                        if Field_flag or (prod([v[k].is_unit() or v[k].is_zero()
  for k in range(len(v))]) == 1):
                            yield vector([R(0) for k in range(i)] + [d] + list(v
))


        ## Raise an exception if we don't have a ring or an integer > 1.
```

```
        else:
            raise TypeError, "Invalid R was passed in.  We need a finite ring or an integer > 1."
```

```
"""
Automorphisms of Quadratic Forms
"""
from sage.interfaces.gp import gp
from sage.matrix.constructor import Matrix
from sage.rings.rational_field import QQ
from sage.rings.integer_ring import ZZ
from sage.misc.mrange import mrange

from sage.modules.free_module_element import vector
from sage.rings.arith import GCD
from sage.misc.sage_eval import sage_eval
from sage.misc.misc import SAGE_ROOT

import tempfile, os
from random import random


def basis_of_short_vectors(self, show_lengths=False, safe_flag=True):
    """
    Return a basis for 'ZZ^n' made of vectors with minimal lengths Q('v').

    The safe_flag allows us to select whether we want a copy of the
    output, or the original output.  By default safe_flag = True, so
    we return a copy of the cached information.  If this is set to
    False, then the routine is much faster but the return values are
    vulnerable to being corrupted by the user.

    OUTPUT:
      a list of vectors, and optionally a list of values for each vector.

    EXAMPLES::

      sage: Q = DiagonalQuadraticForm(ZZ, [1,3,5,7])
      sage: Q.basis_of_short_vectors()
      [(1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1)]
      sage: Q.basis_of_short_vectors(True)
      ([(1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1)], [1, 3, 5, 7])

    """
    ## Try to use the cached results
    try:
        ## Return the appropriate result
        if show_lengths:
            if safe_flag:
                return deep_copy(self.__basis_of_short_vectors), deepcopy(self._
_basis_of_short_vectors_lengths)
            else:
                return self.__basis_of_short_vectors, self.__basis_of_short_vect
ors_lengths
        else:
            if safe_flag:
                return deepcopy(self.__basis_of_short_vectors)
            else:
                return deepcopy(self.__basis_of_short_vectors)
    except:
        pass


    ## Set an upper bound for the number of vectors to consider
    Max_number_of_vectors = 10000

    ## Generate a PARI matrix string for the associated Hessian matrix
    M_str = str(gp(self.matrix()))


    ## Run through all possible minimal lengths to find a spanning set of vector
s
    n = self.dim()
```



```
    #MS = MatrixSpace(QQ, n)
    M1 = Matrix([[0]])
    vec_len = 0
    while M1.rank() < n:

        ## DIAGONSTIC
        #print
        #print "Starting with vec_len = ", vec_len
        #print "M_str = ", M_str

        vec_len += 1
        gp_mat = gp.qfminim(M_str, vec_len, Max_number_of_vectors)[3].mattranspo
se()
        number_of_vecs = ZZ(gp_mat.matsize()[1])
        vector_list = []
        for i in range(number_of_vecs):
            #print "List at", i, ":", list(gp_mat[i+1,])
            new_vec = vector([ZZ(x)  for x in list(gp_mat[i+1,])])
            vector_list.append(new_vec)


        ## DIAGNOSTIC
        #print "number_of_vecs = ", number_of_vecs
        #print "vector_list = ", vector_list


        ## Make a matrix from the short vectors
        if len(vector_list) > 0:
            M1 = Matrix(vector_list)


        ## DIAGNOSTIC
        #print "matrix of vectors = \n", M1
        #print "rank of the matrix = ", M1.rank()



    #print " vec_len = ", vec_len
    #print M1


    ## Organize these vectors by length (and also introduce their negatives)
    max_len = vec_len/2
    vector_list_by_length = [[]  for _ in range(max_len + 1)]
    for v in vector_list:
        l = self(v)
        vector_list_by_length[l].append(v)
        vector_list_by_length[l].append(vector([-x  for x in v]))


    ## Make a matrix from the column vectors (in order of ascending length).
    sorted_list = []
    for i in range(len(vector_list_by_length)):
        for v in vector_list_by_length[i]:
            sorted_list.append(v)
    sorted_matrix = Matrix(sorted_list).transpose()


    ## Determine a basis of vectors of minimal length
    pivots = sorted_matrix.pivots()
    basis = [sorted_matrix.column(i) for i in pivots]
    pivot_lengths = [self(v)  for v in basis]


    ## DIAGNOSTIC
    #print "basis = ", basis
    #print "pivot_lengths = ", pivot_lengths
```

```
    ## Cache the result
    self.__basis_of_short_vectors = basis
    self.__basis_of_short_vectors_lenghts = pivot_lengths

    ## Clear the GP prompts (since we only have about 65,000 of them!)
    gp.clear_prompts()


    ## Return the appropriate result
    if show_lengths:
        return basis, pivot_lengths
    else:
        return basis




def short_vector_list_up_to_length(self, len_bound, up_to_sign_flag=False):
    """
  Return a list of lists of short vectors `v`, sorted by length, with
  Q(`v`) < len_bound.  The list in output `[i]` indexes all vectors of
  length `i`.  If the up_to_sign_flag is set to True, then only one of
  the vectors of the pair `[v, –v]` is listed.

  Note:  This processes the PARI/GP output to always give elements of type `ZZ`.

  OUTPUT:
    a list of lists of vectors.

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,3,5,7])
    sage: Q.short_vector_list_up_to_length(3)
    [[(0, 0, 0, 0)], [(1, 0, 0, 0), [–1, 0, 0, 0]], []]
    sage: Q.short_vector_list_up_to_length(4)
    [[(0, 0, 0, 0)],
     [(1, 0, 0, 0), [–1, 0, 0, 0]],
     [],
     [(0, 1, 0, 0), [0, –1, 0, 0]]]
    sage: Q.short_vector_list_up_to_length(5)
    [[(0, 0, 0, 0)],
     [(1, 0, 0, 0), [–1, 0, 0, 0]],
     [],
     [(0, 1, 0, 0), [0, –1, 0, 0]],
     [(1, 1, 0, 0),
     [–1, –1, 0, 0],
     (–1, 1, 0, 0),
     [1, –1, 0, 0],
     (2, 0, 0, 0),
     [–2, 0, 0, 0]]]
    sage: Q.short_vector_list_up_to_length(5, True)
    [[(0, 0, 0, 0)],
     [(1, 0, 0, 0)],
     [],
     [(0, 1, 0, 0)],
     [(1, 1, 0, 0), (–1, 1, 0, 0), (2, 0, 0, 0)]]

  """
    ## Set an upper bound for the number of vectors to consider
    Max_number_of_vectors = 10000

    ## Generate a PARI matrix string for the associated Hessian matrix
    M_str = str(gp(self.matrix()))

    ## Generate the short vectors
    gp_mat = gp.qfminim(M_str, 2*len_bound-2, Max_number_of_vectors)[3].mattrans
pose()
    number_of_rows = gp_mat.matsize()[1]
    gp_mat_vector_list = [vector([ZZ(x)  for x in gp_mat[i+1,]])  for i in range
```



```
(number_of_rows)]

    ## Sort the vectors into lists by their length
    vec_list = [[]  for i in range(len_bound)]
    for v in gp_mat_vector_list:
        vec_list[self(v)].append(v)
        if up_to_sign_flag == False:
            vec_list[self(v)].append([-x  for x in v])

    ## Add the zero vector by hand
    zero_vec = vector([ZZ(0)  for _ in range(self.dim())])
    vec_list[0].append(zero_vec)

    ## Clear the GP prompts (since we only have about 65,000 of them!)
    gp.clear_prompts()

    ## Return the sorted list
    return vec_list



def short_primitive_vector_list_up_to_length(self, len_bound, up_to_sign_flag=Fa
lse):
    """
  Return a list of lists of short primitive vectors `v`, sorted by length, with
  Q(`v`) < len_bound.  The list in output `[i]` indexes all vectors of
  length `i`.  If the up_to_sign_flag is set to True, then only one of
  the vectors of the pair `[v, –v]` is listed.

  Note:  This processes the PARI/GP output to always give elements of type `ZZ`.

  OUTPUT:
    a list of lists of vectors.

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,3,5,7])
    sage: Q.short_vector_list_up_to_length(5, True)
    [[(0, 0, 0, 0)],
     [(1, 0, 0, 0)],
     [],
     [(0, 1, 0, 0)],
     [(1, 1, 0, 0), (–1, 1, 0, 0), (2, 0, 0, 0)]]
    sage: Q.short_primitive_vector_list_up_to_length(5, True)
    [[], [(1, 0, 0, 0)], [], [(0, 1, 0, 0)], [(1, 1, 0, 0), (–1, 1, 0, 0)]]


  """
    ## Get a list of short vectors
    full_vec_list = self.short_vector_list_up_to_length(len_bound, up_to_sign_fl
ag)

    ## Make a new list of the primitive vectors
    prim_vec_list = [[v  for v in L  if GCD(list(v)) == 1]   for L in full_vec_l
ist]                 ## TO DO:  Arrange for GCD to take a vector argument!

    ## Return the list of primitive vectors
    return prim_vec_list



## ---------------------------------------------------------------------------
-----------------------


def automorphisms(self):
    """
  Return a list of the automorphisms of the quadratic form.
```

```
    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
        sage: Q.number_of_automorphisms()                # optional -- souvigner
        48
        sage: 2^3 * factorial(3)
        48
        sage: len(Q.automorphisms())
        48

    ::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,3,5,7])
        sage: Q.number_of_automorphisms()                # optional -- souvigner
        16
        sage: aut = Q.automorphisms()
        sage: len(aut)
        16
        sage: print([Q(M) == Q for M in aut])
        [True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, True]

        sage: Q = QuadraticForm(ZZ, 3, [2, 1, 2, 2, 1, 3])
        sage: Q.automorphisms()
        [
        [1 0 0]  [-1  0  0]
        [0 1 0]  [ 0 -1  0]
        [0 0 1], [ 0  0 -1]
        ]

    ::

        sage: Q = DiagonalQuadraticForm(ZZ, [1, -1])
        sage: Q.automorphisms()
        Traceback (most recent call last):
        ...
        ValueError: not a definite form in QuadraticForm.automorphisms()
    """
        ## only for definite forms
        if not self.is_definite():
            raise ValueError, "not a definite form in QuadraticForm.automorphisms()"

        ## Check for a cached value
        try:
            return self.__automorphisms
        except:
            pass


        ## Find a basis of short vectors, and their lengths
        basis, pivot_lengths = self.basis_of_short_vectors(show_lengths=True)

        ## List the relevant vectors by length
        max_len = max(pivot_lengths)
        vector_list_by_length = self.short_primitive_vector_list_up_to_length(max_le
n + 1)


        ## Make the matrix A:e_i |--> v_i to our new basis.
        A = Matrix(basis).transpose()
        Ainv = A.inverse()
        #A1 = A.inverse() * A.det()
        #Q1 = A1.transpose() * self.matrix() * A1       ## This is the matrix of Q
        #Q = self.matrix() * A.det()**2
        Q2 = A.transpose() * self.matrix() * A       ## This is the matrix of Q in t
he new basis
        Q3 = self.matrix()
```



```
        ## Determine all automorphisms
        n = self.dim()
        Auto_list = []
        #ct = 0

        ## DIAGNOSTIC
        #print "n = " + str(n)
        #print "pivot_lengths = " + str(pivot_lengths)
        #print "vector_list_by_length = " + str(vector_list_by_length)
        #print "length of vector_list_by_length = " + str(len(vector_list_by_length)
)

        for index_vec in mrange([len(vector_list_by_length[pivot_lengths[i]])  for i
 in range(n)]):
            M = Matrix([vector_list_by_length[pivot_lengths[i]][index_vec[i]]   for
i in range(n)]).transpose()
            #Q1 = self.matrix()
            #if self(M) == self:
            #ct += 1
            #print "ct = ", ct, "   M = "
            #print M
            #print
            if M.transpose() * Q3 * M == Q2:       ## THIS DOES THE SAME THING! =(
                Auto_list.append(M * Ainv)


        ## Cache the answer and return the list
        self.__automorphisms = Auto_list
        self.__number_of_automorphisms = len(Auto_list)
        return Auto_list



###################
   ## TO DO:  Be sure to fix the SOUVIGNER CODE binaries, and change the 'use_co
de="Souvigner"' below!
###################




def number_of_automorphisms(self, recompute=False, use_code="Souvigner"):
    """
    Return a list of the number of automorphisms (of det 1 and -1) of
    the quadratic form.

    If recompute is True, then we will recompute the cached result.

    OUTPUT:
        an integer >= 2.

    EXAMPLES::

        sage: Q = QuadraticForm(ZZ, 3, [1, 0, 0, 1, 0, 1], unsafe_initialization=True, number_of_automorphisms=-1)
        sage: Q.list_external_initializations()
        ['number_of_automorphisms']
        sage: Q.number_of_automorphisms()
        -1
        sage: Q.number_of_automorphisms(recompute=True)          # optional -- souvigner
        48
        sage: Q.list_external_initializations()                  # optional -- souvigner
        []

    ::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1])
        sage: Q.number_of_automorphisms()                        # optional -- souvigner
        384
        sage: 2^4 * factorial(4)
```

```
    384

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1, -1])
    sage: Q.number_of_automorphisms()
    Traceback (most recent call last):
    ...
    ValueError: not a definite form in QuadraticForm.number_of_automorphisms()

  """
    ## only for definite forms
    if not self.is_definite():
        raise ValueError, "not a definite form in QuadraticForm.number_of_automorphisms()"

    ## Try to use the cached version if we can
    if not recompute:
        try:
            #print "Using the cached number of automorphisms."
            #print "We stored", self.__number_of_automorphisms
            return self.__number_of_automorphisms
        except:
            pass

    ## Otherwise cache and return the result
    #print "Recomputing the number of automorphisms based on the list of automor
phisms."
    if use_code == "Souvigner":
        self.__number_of_automorphisms = self.number_of_automorphisms__souvigner
()    ## This is the default
    elif use_code == "Python":
        self.__number_of_automorphisms = len(self.automorphisms())
   ## This is now deprecated.
    else:
        raise TypeError, "The option for use_code you passed is not recognized.  Try 'Souvigner' (default
) or 'Python' (very slow)."


    try:
        self._external_initialization_list.remove('number_of_automorphisms')
    except:
        pass  ## Do nothing if the removal fails, since it might not be in the l
ist (causing an error)!
    return self.__number_of_automorphisms



def number_of_automorphisms__souvigner(self):
    """
  Uses the Souvigner code to compute the number of automorphisms.

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1,1])
    sage: Q.number_of_automorphisms__souvigner()                    # optional -- souvigner
    3840
    sage: 2^5 * factorial(5)
    3840

  """
    ## Find an LLL-reduced quadratic form globally equivalent to this one
    Q1 = self.LLL_reduced()

    ## Write an input text file
    F_filename = '/tmp/tmp_isom_input' + str(random()) + ".txt"
    F = open(F_filename, 'w')
    #F = tempfile.NamedTemporaryFile(prefix='tmp_auto_input', suffix=".txt")   #
# This fails because the Souvigner code doesn't like random filenames (hyphens a
re bad...)!
```



```
    F.write("#1\n")
    n = self.dim()
    F.write(str(n) + "x0\n")      ## Use the lower-triangular form
    for i in range(n):
        for j in range(i+1):
            if i == j:
                F.write(str(2 * Q1[i,j]) + " ")
            else:
                F.write(str(Q1[i,j]) + " ")
        F.write("\n")
    F.flush()
    #print "Input filename = ", F.name
    #os.system("less " + F.name)

    ## Call the Souvigner automorphism code
    souvigner_auto_path = SAGE_ROOT + "/local/bin/Souvigner_AUTO"                    ##
FIX THIS LATER!!!
    G1 = tempfile.NamedTemporaryFile(prefix='tmp_auto_ouput', suffix=".txt")
    #print "Output filename = ", G1.name
    os.system(souvigner_auto_path + " '" +  F.name + "'>'" + G1.name +"'")


    ## Read the output
    G2 = open(G1.name, 'r')
    for line in G2:
        if line.startswith("|Aut| = "):
            num_of_autos = sage_eval(line.lstrip("|Aut| = "))
            F.close()
            G1.close()
            G2.close()
            os.system("rm -f " + F_filename)
            #os.system("rm -f " + G1.name)
            return num_of_autos

    ## Raise and error if we're here:
    raise RuntimeError, "Oops! There is a problem..."



def set_number_of_automorphisms(self, num_autos):
    """
  Set the number of automorphisms to be the value given.  No error
  checking is performed, to this may lead to erroneous results.

  The fact that this result was set externally is recorded in the
  internal list of external initializations, accessible by the
  method list_external_initializations().

  Return a list of the number of
  automorphisms (of det 1 and -1) of the quadratic form.

  OUTPUT:
    None

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1, 1, 1])
    sage: Q.list_external_initializations()
    []
    sage: Q.set_number_of_automorphisms(-3)
    sage: Q.number_of_automorphisms()
    -3
    sage: Q.list_external_initializations()
    ['number_of_automorphisms']

  """
    self.__number_of_automorphisms = num_autos
    text = 'number_of_automorphisms'
    if not text in self._external_initialization_list:
```

```
        self._external_initialization_list.append(text)



### TODO
# def Nipp_automorphism_testing(self):
#     """
#     Testing the automorphism routine against Nipp's Tables
#
#         --- MOVE THIS ELSEWHERE!!! ---
#
#     """
#     for i in range(20):
#         Q = QuadraticForm(ZZ, 4, Nipp[i][2])
#         my_num = Q.number_of_automorphisms()
#         nipp_num = Nipp.number_of_automorphisms(i)
#         print "    i = " + str(i) + "  my_num = " + str(my_num) + "  nipp_num
= " + str(nipp_num)
#
```

```
"""
Counting Congruence Solutions

This file provides more user-friendly Python front-ends to the Cython code in
:module:`sage.quadratic_forms.count_local`.
"""
#################################################################
## Methods for counting/computing the number of representations ##
## of a number by a quadratic form in Z/(p^k)Z of various types ##
#################################################################


from sage.quadratic_forms.count_local_2 import CountAllLocalTypesNaive



def count_congruence_solutions_as_vector(self, p, k, m, zvec, nzvec):
    """
    Gives the number of integer solution vectors `x` satisfying the
    congruence Q(`x`) `= m (mod p^k)` of each solution type (i.e. All,
    Good, Zero, Bad, BadI, BadII) which satisfy the additional
    congruence conditions of having certain coefficients = 0 (mod `p`)
    and certain collections of coefficients not congruent to the zero
    vector (mod `p`).

    A solution vector `x` satisfies the additional congruence conditions
    specified by zvec and nzvec (and therefore is counted) iff both of
    the following conditions hold:

        1) `x[i] == 0 (mod p)` for all `i` in zvec

        2) `x[i] != 0 (mod p) for all i` in nzvec


    REFERENCES: See Hanke's (????) paper "Local Densities and explicit
            bounds...", p??? for the definitions of the solution types and
        congruence conditions.

    INPUT:
        `p` -- prime number > 0

        `k` -- an integer > 0

        `m` -- an integer (depending only on mod `p^k`)

        zvec, nzvec -- a list of integers in range(self.dim()), or None

    OUTPUT:
        a list of six integers >= 0 representing the solution types:
            [All, Good, Zero, Bad, BadI, BadII]


    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,2,3])
        sage: Q.count_congruence_solutions_as_vector(3, 1, 1, [], [])
        [0, 0, 0, 0, 0, 0]
        sage: Q.count_congruence_solutions_as_vector(3, 1, 1, None, [])
        [0, 0, 0, 0, 0, 0]
        sage: Q.count_congruence_solutions_as_vector(3, 1, 1, [], None)
        [6, 6, 0, 0, 0, 0]
        sage: Q.count_congruence_solutions_as_vector(3, 1, 1, None, None)
        [6, 6, 0, 0, 0, 0]
        sage: Q.count_congruence_solutions_as_vector(3, 1, 2, None, None)
        [6, 6, 0, 0, 0, 0]
        sage: Q.count_congruence_solutions_as_vector(3, 1, 0, None, None)
        [15, 12, 1, 2, 0, 2]

    """
```



```
    return CountAllLocalTypesNaive(self, p, k, m, zvec, nzvec)




##########################################
##/// Front-ends for our counting routines //
##########################################

def count_congruence_solutions(self, p, k, m, zvec, nzvec):
    """
    Counts all solutions of Q(`x`) `= m (mod p^k)` satisfying the
    additional congruence conditions described in
    QuadraticForm.count_congruence_solutions_as_vector().

    INPUT:

        `p` -- prime number > 0

        `k` -- an integer > 0

        `m` -- an integer (depending only on mod `p^k`)

        zvec, nzvec -- a list of integers in range(self.dim()), or None

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,2,3])
        sage: Q.count_congruence_solutions(3, 1, 0, None, None)
        15

    """
    return CountAllLocalTypesNaive(self, p, k, m, zvec, nzvec)[0]



def count_congruence_solutions__good_type(self, p, k, m, zvec, nzvec):
    """
    Counts the good-type solutions of Q(x) = m (mod p^k) satisfying the
    additional congruence conditions described in
    QuadraticForm.count_congruence_solutions_as_vector().

    INPUT:

        `p` -- prime number > 0

        `k` -- an integer > 0

        `m` -- an integer (depending only on mod `p^k`)

        zvec, nzvec -- a list of integers up to dim(Q)

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,2,3])
        sage: Q.count_congruence_solutions__good_type(3, 1, 0, None, None)
        12

    """
    return CountAllLocalTypesNaive(self, p, k, m, zvec, nzvec)[1]



def count_congruence_solutions__zero_type(self, p, k, m, zvec, nzvec):
    """
    Counts the zero-type solutions of Q(`x`) = `m (mod p^k)` satisfying the
    additional congruence conditions described in
```

QuadraticForm.count_congruence_solutions_as_vector().

INPUT:

'p' -- prime number > 0

'k' -- an integer > 0

'm' -- an integer (depending only on mod 'p^k')

zvec, nzvec -- a list of integers up to dim(Q)

EXAMPLES::

```
sage: Q = DiagonalQuadraticForm(ZZ, [1,2,3])
sage: Q.count_congruence_solutions__zero_type(3, 1, 0, None, None)
1
```

```
    """
        return CountAllLocalTypesNaive(self, p, k, m, zvec, nzvec)[2]


def count_congruence_solutions__bad_type(self, p, k, m, zvec, nzvec):
    """
```

Counts the bad-type solutions of Q('x') '= m (mod p^k)' satisfying the additional congruence conditions described in QuadraticForm.count_congruence_solutions_as_vector().

INPUT:

'p' -- prime number > 0

'k' -- an integer > 0

'm' -- an integer (depending only on mod 'p^k')

zvec, nzvec -- a list of integers up to dim(Q)

EXAMPLES::

```
sage: Q = DiagonalQuadraticForm(ZZ, [1,2,3])
sage: Q.count_congruence_solutions__bad_type(3, 1, 0, None, None)
2
```

```
    """
        return CountAllLocalTypesNaive(self, p, k, m, zvec, nzvec)[3]


def count_congruence_solutions__bad_type_I(self, p, k, m, zvec, nzvec):
    """
```

Counts the bad-typeI solutions of Q('x') = 'm (mod p^k)' satisfying the additional congruence conditions described in QuadraticForm.count_congruence_solutions_as_vector().

INPUT:

'p' -- prime number > 0

'k' -- an integer > 0

'm' -- an integer (depending only on mod 'p^k')

zvec, nzvec -- a list of integers up to dim(Q)

EXAMPLES::

```
sage: Q = DiagonalQuadraticForm(ZZ, [1,2,3])
sage: Q.count_congruence_solutions__bad_type_I(3, 1, 0, None, None)
0
```



```
    """
        return CountAllLocalTypesNaive(self, p, k, m, zvec, nzvec)[4]


def count_congruence_solutions__bad_type_II(self, p, k, m, zvec, nzvec):
    """
```

Counts the bad-typeII solutions of Q('x') '= m (mod p^k)' satisfying the additional congruence conditions described in QuadraticForm.count_congruence_solutions_as_vector().

INPUT:

'p' -- prime number > 0

'k' -- an integer > 0

'm' -- an integer (depending only on mod 'p^k')

zvec, nzvec -- a list of integers up to dim(Q)

EXAMPLES::

```
sage: Q = DiagonalQuadraticForm(ZZ, [1,2,3])
sage: Q.count_congruence_solutions__bad_type_II(3, 1, 0, None, None)
2
```

```
    """
        return CountAllLocalTypesNaive(self, p, k, m, zvec, nzvec)[5]
```

```
"""
Equivalence Testing
"""

from sage.matrix.constructor import Matrix
from sage.misc.mrange import mrange
from sage.rings.arith import hilbert_symbol, prime_divisors, is_prime, valuation
, GCD, legendre_symbol
from sage.rings.integer_ring import ZZ

from quadratic_form import is_QuadraticForm
from quadratic_form__genus import CS_genus_symbol_list


from sage.misc.misc import SAGE_ROOT

import tempfile, os

from random import random


#################################################################################
## Routines to test if two quadratic forms over ZZ are globally equivalent.   ##
## (For now, we require both forms to be positive definite.)                  ##
#################################################################################

def is_globally_equivalent__souvigner(self, other, return_transformation=False):
    """
    Uses the Souvigner code to test for global equivalence of positive definite quadratic forms!

    INPUT:
        a QuadraticForm

    OUTPUT:
        boolean, and optionally a matrix

    EXAMPLES::

        sage: Q = QuadraticForm(ZZ, 3, [1, 0, -1, 2, -1, 5])
        sage: Q1 = QuadraticForm(ZZ, 3, [8, 6, 5, 3, 4, 2])
        sage: M = Q.is_globally_equivalent__souvigner(Q1, True) ; M   # optional -- souvigner
        [ 0  0 -1]
        [ 1  0  0]
        [-1  1  1]
        sage: Q1(M) == Q                                # optional -- souvigner
        True

    """
    ## SANITY CHECK:  The Souvigner code requires matrices to be positive defini
te!
    if not (self.is_positive_definite() and other.is_positive_definite()):
        raise TypeError, "Both quadratic forms need to be positive definite to use the Souvigner code!"


    ## Find LLL-reduced quadratic forms globally equivalent to our given ones
    Q1 = self.LLL_reduced()
    Q2 = other.LLL_reduced()

    ## Write an input text file
    F_filename = '/tmp/tmp_isom_input__' + str(ZZ.random_element(100000)) + ".txt"
    F = open(F_filename, 'w')
    #F = tempfile.NamedTemporaryFile(prefix='tmp_isom_input', suffix=".txt")  ##
 This failed because it may have hyphens, which are interpreted badly by the Sou
vigner code.
    F.write("\n #1 \n")

    ## Write the first form
    n = self.dim()
    F.write(str(n) + "x0 \n")       ## Use the lower-triangular form
```



```
    for i in range(n):
        for j in range(i+1):
            if i == j:
                F.write(str(2 * Q1[i,j]) + " ")
            else:
                F.write(str(Q1[i,j]) + " ")
        F.write("\n")

    ## Write the second form
    F.write("\n")
    n = self.dim()
    F.write(str(n) + "x0 \n")       ## Use the lower-triangular form
    for i in range(n):
        for j in range(i+1):
            if i == j:
                F.write(str(2 * Q2[i,j]) + " ")
            else:
                F.write(str(Q2[i,j]) + " ")
        F.write("\n")
    F.flush()
    #print "Input filename = ", F.name
    #os.system("less " + F.name)

    ## Call the Souvigner automorphism code
    souvigner_isom_path = SAGE_ROOT + "/local/bin/Souvigner_ISOM"
    G1 = tempfile.NamedTemporaryFile(prefix='tmp_isom_output', suffix=".txt")
    #print "Output filename = ", G1.name
    #print  "Executing the shell command:   " + souvigner_isom_path + " '" +  F.
name + "' > '" + G1.name + "'"
    os.system(souvigner_isom_path + " '" +  F.name + "'>'" + G1.name +"'")


    ## Read the output
    G2 = open(G1.name, 'r')
    line = G2.readline()
    if line.startswith("Error:"):
        raise RuntimeError, "There is a problem using the souvigner code... " + line
    elif line.find("not iso") != -1:     ## Checking if this text appears, if so t
hen they're not isomorphic/isometric!
        return False
    else:
        ## Decide whether to read the transformation matrix, and return true
        if not return_transformation:
            F.close()
            G1.close()
            G2.close()
            os.system("rm -f " + F_filename)
            return True
        else:
            ## Try to read the isomorphism matrix
            M = Matrix(ZZ, n, n)
            for i in range(n):
                new_row_text = G2.readline().split()
                #print new_row_text
                for j in range(n):
                    M[i,j] = new_row_text[j]

            ## Remove temporary files and return the value
            F.close()
            G1.close()
            G2.close()
            os.system("rm -f " + F_filename)
            if return_transformation:
                return M.transpose()
            else:
                return True
            #return True, M

    ## Raise and error if we're here:
```

```
    raise RuntimeError, "Oops! There is a problem..."



def is_globally_equivalent_to(self, other, return_matrix=False, check_theta_to_precision=30, check_local_equivalence=True, use_code="Souvigner"):
    """
    Determines if the current quadratic form is equivalent to the
    given form over ZZ.  If return_matrix is True, then we also return
    the transformation matrix M so that self(M) == other.

    INPUT:
        a QuadraticForm

    OUTPUT:
        boolean, and optionally a matrix

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1])
        sage: M = Matrix(ZZ, 4, 4, [1,2,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1])
        sage: Q1 = Q(M)
        sage: Q.(Q1)                                    # optional -- souvigner
        True
        sage: MM = Q.is_globally_equivalent_to(Q1, return_matrix=True)  # optional -- souvigner
        sage: Q(MM) == Q1                               # optional -- souvigner
        True

    ::

        sage: Q1 = QuadraticForm(ZZ, 3, [1, 0, -1, 2, -1, 5])
        sage: Q2 = QuadraticForm(ZZ, 3, [2, 1, 2, 2, 1, 3])
        sage: Q3 = QuadraticForm(ZZ, 3, [8, 6, 5, 3, 4, 2])
        sage: Q1.is_globally_equivalent_to(Q2)          # optional -- souvigner
        False
        sage: Q1.is_globally_equivalent_to(Q3)          # optional -- souvigner
        True
        sage: M = Q1.is_globally_equivalent_to(Q3, True) ; M        # optional -- souvigner
        [-1 -1  0]
        [ 1  1  1]
        [-1  0  0]
        sage: Q1(M) == Q3                               # optional -- souvigner
        True

    ::

        sage: Q = DiagonalQuadraticForm(ZZ, [1, -1])
        sage: Q.is_globally_equivalent_to(Q)
        Traceback (most recent call last):
        ...
        ValueError: not a definite form in QuadraticForm.is_globally_equivalent_to()

    """
    ## only for definite forms
    if not self.is_definite():
        raise ValueError, "not a definite form in QuadraticForm.is_globally_equivalent_to()"

    ## Check that other is a QuadraticForm
    #if not isinstance(other, QuadraticForm):
    if not is_QuadraticForm(other):
        raise TypeError, "Oops!  You must compare two quadratic forms, but the argument is not a quadratic form. =("



    if use_code == "Souvigner":
        ## Now use the Souvigner code by default! =)
        return other.is_globally_equivalent__souvigner(self, return_matrix)    # Note: We switch this because the Souvigner code has the opposite mapping conve
```



```
ntion to us.  (It takes the second argument to the first!)


    elif use_code == "Python":
        ## Check if the forms are locally equivalent
        if (check_local_equivalence == True):
            if not self.is_locally_equivalent_to(other):
                return False

        ## Check that the forms have the same theta function up to the desired precision (this can be set so that it determines the cusp form)
        if check_theta_to_precision != None:
            if self.theta_series(check_theta_to_precision, var_str='', safe_flag=False) != other.theta_series(check_theta_to_precision, var_str='', safe_flag=False):
                return False


        ## Make all possible matrices which give an isomorphism -- can we do this more intelligently?
        ## ------------------------------------------------------------------------------------------

        ## Find a basis of short vectors for one form, and try to match them with vectors of that length in the other one.
        basis_for_self, self_lengths = self.basis_of_short_vectors(show_lengths=True)
        max_len = max(self_lengths)
        short_vectors_of_other = other.short_vector_list_up_to_length(max_len + 1)

        ## Make the matrix A:e_i |--> v_i to our new basis.
        A = Matrix(basis_for_self).transpose()
        Q2 = A.transpose() * self.matrix() * A       ## This is the matrix of 'self' in the new basis
        Q3 = other.matrix()

        ## Determine all automorphisms
        n = self.dim()
        Auto_list = []

        ## DIAGNOSTIC
        #print "n = " + str(n)
        #print "pivot_lengths = " + str(pivot_lengths)
        #print "vector_list_by_length = " + str(vector_list_by_length)
        #print "length of vector_list_by_length = " + str(len(vector_list_by_length))

        for index_vec in mrange([len(short_vectors_of_other[self_lengths[i]])  for i in range(n)]):
            M = Matrix([short_vectors_of_other[self_lengths[i]][index_vec[i]] for i in range(n)]).transpose()
            if M.transpose() * Q3 * M == Q2:
                if return_matrix:
                    return A * M.inverse()
                else:
                    return True

        ## If we got here, then there is no isomorphism
        return False

    else:
        raise TypeError, "The use_code argument must be either 'Python' or 'Souvigner'."


def is_locally_equivalent_to(self, other, check_primes_only=False, force_jordan_equivalence_test=False):
    """
    Determines if the current quadratic form (defined over ZZ) is
```

```
    locally equivalent to the given form over the real numbers and the
    ‘p‘–adic integers for every prime p.

    This works by comparing the local Jordan decompositions at every
    prime, and the dimension and signature at the real place.

    INPUT:
        a QuadraticForm

    OUTPUT:
        boolean

    EXAMPLES::

        sage: Q1 = QuadraticForm(ZZ, 3, [1, 0, –1, 2, –1, 5])
        sage: Q2 = QuadraticForm(ZZ, 3, [2, 1, 2, 2, 1, 3])
        sage: Q1.is_globally_equivalent_to(Q2)                  # optional –– souvigner
        False
        sage: Q1.is_locally_equivalent_to(Q2)
        True

    """
        ## TO IMPLEMENT:
        if self.det() == 0:
            raise NotImplementedError, "OOps!  We need to think about whether this still works for degen
erate forms...  especially check the signature."

        ## Check that both forms have the same dimension and base ring
        if (self.dim() != other.dim()) or (self.base_ring() != other.base_ring()):
            return False

        ## Check that the determinant and level agree
        if (self.det() != other.det()) or (self.level() != other.level()):
            return False

        ## ---------------------------------------------------

        ## Test equivalence over the real numbers
        if self.signature() != other.signature():
            return False

        ## Test equivalence over Z_p for all primes
        if (self.base_ring() == ZZ) and (force_jordan_equivalence_test == False):

            ## Test equivalence with Conway-Sloane genus symbols (default over ZZ)
            if self.CS_genus_symbol_list() != other.CS_genus_symbol_list():
                return False
        else:
            ## Test equivalence via the O'Meara criterion.
            for p in prime_divisors(ZZ(2) * self.det()):
                #print "checking the prime p = ", p
                if not self.has_equivalent_Jordan_decomposition_at_prime(other, p):
                    return False

        ## All tests have passed!
        return True



def has_equivalent_Jordan_decomposition_at_prime(self, other, p):
    """
    Determines if the given quadratic form has a Jordan decomposition
    equivalent to that of self.

    INPUT:
        a QuadraticForm

    OUTPUT:
```



```
        boolean

    EXAMPLES::

        sage: Q1 = QuadraticForm(ZZ, 3, [1, 0, –1, 1, 0, 3])
        sage: Q2 = QuadraticForm(ZZ, 3, [1, 0, 0, 2, –2, 6])
        sage: Q3 = QuadraticForm(ZZ, 3, [1, 0, 0, 1, 0, 11])
        sage: [Q1.level(), Q2.level(), Q3.level()]
        [44, 44, 44]
        sage: Q1.has_equivalent_Jordan_decomposition_at_prime(Q2,2)
        False
        sage: Q1.has_equivalent_Jordan_decomposition_at_prime(Q2,11)
        False
        sage: Q1.has_equivalent_Jordan_decomposition_at_prime(Q3,2)
        False
        sage: Q1.has_equivalent_Jordan_decomposition_at_prime(Q3,11)
        True
        sage: Q2.has_equivalent_Jordan_decomposition_at_prime(Q3,2)
        True
        sage: Q2.has_equivalent_Jordan_decomposition_at_prime(Q3,11)
        False

    """
        ## Sanity Checks
        #if not isinstance(other, QuadraticForm):
        if type(other) != type(self):
            raise TypeError, "Oops!  The first argument must be of type QuadraticForm."
        if not is_prime(p):
            raise TypeError, "Oops!  The second argument must be a prime number."

        ## Get the relevant local normal forms quickly
        self_jordan = self.jordan_blocks_by_scale_and_unimodular(p, safe_flag= False
)
        other_jordan = other.jordan_blocks_by_scale_and_unimodular(p, safe_flag=Fals
e)

        ## DIAGNOSTIC
        #print "self_jordan = ", self_jordan
        #print "other_jordan = ", other_jordan


        ## Check for the same number of Jordan components
        if len(self_jordan) != len(other_jordan):
            return False


        ## Deal with odd primes:  Check that the Jordan component scales, dimensions
, and discriminants are the same
        if p != 2:
            for i in range(len(self_jordan)):
                if (self_jordan[i][0] != other_jordan[i][0]) \
                   or (self_jordan[i][1].dim() != other_jordan[i][1].dim()) \
                   or (legendre_symbol(self_jordan[i][1].det() * other_jordan[i][1].
det(), p) != 1):
                    return False

            ## All tests passed for an odd prime.
            return True


        ## For p = 2:  Check that all Jordan Invariants are the same.
        elif p == 2:

            ## Useful definition
            t = len(self_jordan)          ## Define t = Number of Jordan components


            ## Check that all Jordan Invariants are the same (scale, dim, and norm)
            for i in range(t):
```

```
            if (self_jordan[i][0] != other_jordan[i][0]) \
               or (self_jordan[i][1].dim() != other_jordan[i][1].dim()) \
               or (valuation(GCD(self_jordan[i][1].coefficients()), p) != valuat
ion(GCD(other_jordan[i][1].coefficients()), p)):
                return False

        ## DIAGNOSTIC
        #print "Passed the Jordan invariant test."


        ## Use O'Meara's isometry test 93:29 on p277.
        ## ------------------------------------------

        ## List of norms, scales, and dimensions for each i
        scale_list = [ZZ(2)**self_jordan[i][0]  for i in range(t)]
        norm_list = [ZZ(2)**(self_jordan[i][0] + valuation(GCD(self_jordan[i][1]
.coefficients()), 2))  for i in range(t)]
        dim_list = [(self_jordan[i][1].dim())  for i in range(t)]

        ## List of Hessian determinants and Hasse invariants for each Jordan (su
b)chain
        ## (Note: This is not the same as O'Meara's Gram determinants, but ratio
s are the same!)  -- NOT SO GOOD...
        ## But it matters in condition (ii), so we multiply all by 2 (instead of
 dividing by 2 since only square-factors matter, and it's easier.)
        j = 0
        self_chain_det_list = [ self_jordan[j][1].Gram_det() * (scale_list[j]**d
im_list[j])]
        other_chain_det_list = [ other_jordan[j][1].Gram_det() * (scale_list[j]*
*dim_list[j])]
        self_hasse_chain_list = [ self_jordan[j][1].scale_by_factor(ZZ(2)**self_
jordan[j][0]).hasse_invariant__OMeara(2) ]
        other_hasse_chain_list = [ other_jordan[j][1].scale_by_factor(ZZ(2)**oth
er_jordan[j][0]).hasse_invariant__OMeara(2) ]

        for j in range(1, t):
            self_chain_det_list.append(self_chain_det_list[j-1] * self_jordan[j]
[1].Gram_det() * (scale_list[j]**dim_list[j]))
            other_chain_det_list.append(other_chain_det_list[j-1] * other_jordan
[j][1].Gram_det() * (scale_list[j]**dim_list[j]))
            self_hasse_chain_list.append(self_hasse_chain_list[j-1] \
                                         * hilbert_symbol(self_chain_det_list[j-
1], self_jordan[j][1].Gram_det(), 2) \
                                         * self_jordan[j][1].hasse_invariant__OM
eara(2))
            other_hasse_chain_list.append(other_hasse_chain_list[j-1] \
                                          * hilbert_symbol(other_chain_det_list[
j-1], other_jordan[j][1].Gram_det(), 2) \
                                          * other_jordan[j][1].hasse_invariant__
OMeara(2))


        ## SANITY CHECK -- check that the scale powers are strictly increasing

        for i in range(1, len(scale_list)):
            if scale_list[i-1] >= scale_list[i]:
                raise RuntimeError, "Oops!  There is something wrong with the Jordan Decompo
sition -- the given scales are not strictly increasing!"


        ## DIAGNOSTIC
        #print "scale_list = ", scale_list
        #print "norm_list = ", norm_list
        #print "dim_list = ", dim_list
        #print
        #print "self_chain_det_list = ", self_chain_det_list
        #print "other_chain_det_list = ", other_chain_det_list
        #print "self_hasse_chain_list = ", self_hasse_chain_list
        #print "other_hasse_chain_det_list = ", other_hasse_chain_list
```



```
        ## Test O'Meara's two conditions
        for i in range(t-1):

            ## Condition (i): Check that their (unit) ratio is a square (but it
suffices to check at most mod 8).
            modulus = norm_list[i] * norm_list[i+1] / (scale_list[i] ** 2)
            if modulus > 8:
                modulus = 8
            if (modulus > 1) and (((self_chain_det_list[i] / other_chain_det_lis
t[i]) % modulus) != 1):
                #print "Failed when i =", i, " in condition 1."
                return False

            ## Check O'Meara's condition (ii) when appropriate
            if norm_list[i+1] % (4 * norm_list[i]) == 0:
                if self_hasse_chain_list[i] * hilbert_symbol(norm_list[i] * othe
r_chain_det_list[i], -self_chain_det_list[i], 2) \
                    != other_hasse_chain_list[i] * hilbert_symbol(norm_list[i
], -other_chain_det_list[i], 2):      ## Nipp conditions
                    #print "Failed when i =", i, " in condition 2."
                    return False


        ## All tests passed for the prime 2.
        return True

    else:
        raise TypeError, "Oops!  This should not have happened."
```

```
def QFEvaluateVector(Q, v):
    """
    Evaluate this quadratic form Q on a vector or matrix of elements
    coercible to the base ring of the quadratic form.  If a vector
    is given then the output will be the ring element Q(v), but if a
    matrix is given then the output will be the quadratic form Q'
    which in matrix notation is given by:

            Q' = v^t * Q * v.

    Note: This is a Python wrapper for the fast evaluation routine
    QFEvaluateVector_cdef().  This routine is for internal use and is
    called more conveniently as Q(M).

    INPUT:
        Q -- QuadraticForm over a base ring R
        v -- a tuple or list (or column matrix) of Q.dim() elements of R

    OUTPUT:
        an element of R

    EXAMPLES:
        sage: from sage.quadratic_forms.quadratic_form__evaluate import QFEvalua
teVector
        sage: Q = QuadraticForm(ZZ, 4, range(10)); Q
        Quadratic form in 4 variables over Integer Ring with coefficients:
        [ 0 1 2 3 ]
        [ * 4 5 6 ]
        [ * * 7 8 ]
        [ * * * 9 ]
        sage: QFEvaluateVector(Q, (1,0,0,0))
        0
        sage: QFEvaluateVector(Q, (1,0,1,0))
        9

    """
    return QFEvaluateVector_cdef(Q, v)



cdef QFEvaluateVector_cdef(Q, v):
    """
    Routine to quickly evaluate a quadratic form Q on a vector v.  See
    the Python wrapper function QFEvaluate() above for details.

    """
    ## If we are passed a matrix A, return the quadratic form Q(A(x))
    ## (In matrix notation: A^t * Q * A)
    n = Q.dim()

    tmp_val = Q.base_ring()(0)
    for i from 0 <= i < n:
        for j from i <= j < n:
            tmp_val += Q[i,j] * v[i] * v[j]

    ## Return the value (over R)
    return Q.base_ring()._coerce_(tmp_val)



def QFEvaluateMatrix(Q, M, Q2):
    """
    Evaluate this quadratic form Q on a matrix M of elements coercible
    to the base ring of the quadratic form, which in matrix notation
    is given by:

            Q2 = M^t * Q * M.
```



```
    Note: This is a Python wrapper for the fast evaluation routine
    QFEvaluateMatrix_cdef().  This routine is for internal use and is
    called more conveniently as Q(M).  The inclusion of Q2 as an
    argument is to avoid having to create a QuadraticForm here, which
    for now creates circular imports.

    INPUT:
        Q -- QuadraticForm over a base ring R
        M -- a Q.dim() x Q2.dim() matrix of elements of R

    OUTPUT:
        Q2 -- a QuadraticForm over R

    EXAMPLES:
        sage: from sage.quadratic_forms.quadratic_form__evaluate import QFEvalua
teMatrix
        sage: Q = QuadraticForm(ZZ, 4, range(10)); Q
        Quadratic form in 4 variables over Integer Ring with coefficients:
        [ 0 1 2 3 ]
        [ * 4 5 6 ]
        [ * * 7 8 ]
        [ * * * 9 ]
        sage: Q2 = QuadraticForm(ZZ, 2)
        sage: M = Matrix(ZZ, 4, 2, [1,0,0,0, 0,1,0,0]); M
        [1 0]
        [0 0]
        [0 1]
        [0 0]
        sage: QFEvaluateMatrix(Q, M, Q2)
        Quadratic form in 2 variables over Integer Ring with coefficients:
        [ 0 2 ]
        [ * 7 ]

    """
    return QFEvaluateMatrix_cdef(Q, M, Q2)



cdef QFEvaluateMatrix_cdef(Q, M, Q2):
    """
    Routine to quickly evaluate a quadratic form Q on a matrix M.  See
    the Python wrapper function QFEvaluateMatrix() above for details.

    """
    ## Create the new quadratic form
    n = Q.dim()
    m = Q2.dim()

    ## TO DO: Check the dimensions of M are compatible with those of Q and Q2

    ## Evaluate Q(M) into Q2
    for k from 0 <= k < m:
        for l from k <= l < m:
            tmp_sum = Q2.base_ring()(0)
            for i from 0 <= i < n:
                for j from i <= j < n:
                    if (k == l):
                        tmp_sum += Q[i,j] * (M[i,k] * M[j,l])
                    else:
                        tmp_sum += Q[i,j] * (M[i,k] * M[j,l] + M[i,l] * M[j,k])
            Q2[k,l] = tmp_sum
    return Q2
```

```
## Routines to enumerate classes in a genus or spinor genus, and spinor genera i
n a genus.




from sage.rings.all import ZZ
from sage.rings.arith import next_prime, prime_divisors
from sage.rings.fast_arith import prime_range









def class_number(self):
    """
  Returns the number of (improper) classes in the genus of the given quadratic form.

  TO DO: Add Caching ability!



  INPUT:
    None

  OUTPUT:
    a positive integer

  EXAMPLES:
    sage: Q = DiagonalQuadraticForm(ZZ, [1,48,144])
    sage: Q.class_number() == 4
    True

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1])
    sage: Q.class_number() == 1
    True

  """
    ## Use a cached value of the class number if it exists!


    ## Test if the class number is 1 quickly
    if self.conway_mass() == 1/self.number_of_automorphisms():
        return 1

    ## Compute the class number by counting the genus representatives
    return len(self.genus_representatives())









def genus_representatives(self):
    """
  Find a set of inequivalent representatives for the classes in the
  genus of the given quadratic form.

  Warning: If there is a problem now -- the algorithm will not
  terminate.  It's better to compute the spinor genera in the gennus
  adelically first, so we know which primes to use!


  INPUT:
    None
```



```
  OUTPUT:
    A list of non-isometric quadratic forms representing all classes in the genus.

  EXAMPLES:
    sage: Q = DiagonalQuadraticForm(ZZ, [1, 48, 144])
    sage: GR = Q.genus_representatives()
    sage: len(GR)
    4
    sage: GR
    [Quadratic form in 3 variables over Integer Ring with coefficients:
    [ 1 0 0 ]
    [ * 48 0 ]
    [ * * 144 ]
    ,
    Quadratic form in 3 variables over Integer Ring with coefficients:
    [ 697 0 3168 ]
    [ * 48 0 ]
    [ * * 3600 ]
    ,
    Quadratic form in 3 variables over Integer Ring with coefficients:
    [ 25 96 192 ]
    [ * 1200 1920 ]
    [ * * 912 ]
    ,
    Quadratic form in 3 variables over Integer Ring with coefficients:
    [ 97 576 864 ]
    [ * 1200 960 ]
    [ * * 3792 ]
    ]



  """
    ## Sanity Check:  Base ring = ZZ only!
    if self.base_ring() != ZZ:
        raise NotImplementedError, "The genus representatives routine is only implemented over the b
ase ring ZZ."


    ## TO DO: USE A CACHED RESULT IF POSSIBLE...



    ## Compute p-neighbors for some prime p>2 not dividing det(Q)
    p_initial = next_prime(max([2] + prime_divisors(self.det())) + 1)
    genus_reps_list = self.p_neighbors_up_to_isometry(p_initial)


    ## Compute the mass and see if we're done
    mass = self.conway_mass()
    reps_auto_list = [ZZ(1)/Q.number_of_automorphisms()  for Q in genus_reps_lis
t]
    mass_of_reps = sum(reps_auto_list)

    print "reps_auto_list = " + str(reps_auto_list)
    print "mass_of_reps = " + str(mass_of_reps)
    print "mass = " + str(mass)


    ## Look for primes where some p-neighbor gives a new (inequivalent) form, un
til the mass formula is satisfied
    new_prime_list = []
    for p in prime_range(next_prime(p_initial), 200):
  ## THIS IS AWFUL -- CHANGE THIS TO HAVE AN EXIT CONDITION!

        ## Check if the automorphisms fill out the mass (so we're done)
        if mass == mass_of_reps:
            return genus_reps_list
```

```
        elif mass < mass_of_reps:
            raise RuntimeError, "We have a problem since the mass = " + str(mass) + \
                 " < the mass of our genus representatives " + str(mass_of_reps) + \
                 ", which have reciprocal automorphism sizes " + str(reps_auto_list) + "."

        ## Generate a p-neighbor
        tmp_neighbor = self.compute_p_neighbor_once(p)

        print "Found some p-neighbor for p = " + str(p)

        ## Test that it is not isometric to any known forms
        new_genus_rep = True
        for rep in genus_reps_list:
            if tmp_neighbor.is_globally_equivalent_to(rep):
                new_genus_rep = False
                break              ## Don't test past one successful isometry


        ## Generate its p_initial neighbors if we get a new form.
        if new_genus_rep:
            new_prime_list.append(p)
            new_reps_list = tmp_neighbor.p_neighbors_up_to_isometry(p)
            reps_auto_list += [Q.number_of_automorphisms()  for Q in new_reps_li
st]
            mass_of_reps = sum(reps_auto_list)


    ## Sanity Check:
    raise RuntimeError,    "After checking the primes < 200, we didn't make the entire genus... =( "



def spinor_genus_representatives(self):
    """
  Returns the spinor genus of the given quadratic form.

  TO DO: WE NEED TO FIND A PRIME WHICH ONLY GIVES ONE SPINOR GENUS FROM ITS NEIGHBORS
!


  INPUT:
    None

  OUTPUT:
    A list of integrally inequivalent quadratic forms

  EXAMPLES:
    sage: Q = DiagonalQuadraticForm(ZZ, [1,48,144])
    sage: len(Q.spinor_genus_representatives()) == 2
    True

  """
    raise NotImplementedError, "We need to find the correct prime to use first!"
```

```
"""
Local and Global Genus Symbols

"""


#############################################################
##                                                         ##
##  Wrappers for the Genus/Genus Symbol Code in ../genera/ ##
##                                                         ##
#############################################################

from sage.quadratic_forms.genera.genus import Genus, LocalGenusSymbol, \
        is_GlobalGenus, is_2_adic_genus, canonical_2_adic_compartments, \
        canonical_2_adic_trains, canonical_2_adic_reduction, \
        basis_complement, p_adic_symbol, is_even_matrix, \
        split_odd,  trace_diag_mod_8, two_adic_symbol
        #is_trivial_symbol
        #GenusSymbol_p_adic_ring, GenusSymbol_global_ring

## Removed signature_pair_of_matrix due to a circular import issue.

## NOTE: Removed the signature routine here... and rewrote it for now.


from sage.rings.integer_ring import IntegerRing
from sage.rings.arith import is_prime, prime_divisors



def global_genus_symbol(self):
    """
    Returns the genus of a two times a quadratic form over ZZ.  These
    are defined by a collection of local genus symbols (a la Chapter
    15 of Conway–Sloane), and a signature.

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,2,3,4])
        sage: Q.global_genus_symbol()
        Genus of [2 0 0 0]
        [0 4 0 0]
        [0 0 6 0]
        [0 0 0 8]

    ::

        sage: Q = QuadraticForm(ZZ, 4, range(10))
        sage: Q.global_genus_symbol()
        Genus of [ 0  1  2  3]
        [ 1  8  5  6]
        [ 2  5 14  8]
        [ 3  6  8 18]

    """
    ## Check that the form is defined over ZZ
    if not self.base_ring() == IntegerRing():
        raise TypeError, "Oops!  The quadratic form is not defined over the integers."

    ## Return the result
    try:
        return Genus(self.Hessian_matrix())
    except:
        raise TypeError, "Oops!  There is a problem computing the genus symbols for this form."



def local_genus_symbol(self, p):
    """
```



```
    Returns the Conway–Sloane genus symbol of 2 times a quadratic form
    defined over ZZ at a prime number p.  This is defined (in the
    Genus_Symbol_p_adic_ring() class in the quadratic_forms/genera
    subfolder) to be a list of tuples (one for each Jordan component
    p^m*A at p, where A is a unimodular symmetric matrix with
    coefficients the p–adic integers) of the following form:

      1. If p>2 then return triples of the form [‘m‘, ‘n‘, ‘d‘] where
         ‘m‘ = valuation of the component

         ‘n‘ = rank of A

         ‘d‘ = det(A) in {1,u} for normalized quadratic non–residue u.

      2. If p=2 then return quintuples of the form [‘m‘,‘n‘,‘s‘, ‘d‘, ‘o‘] where
         ‘m‘ = valuation of the component

         ‘n‘ = rank of A

         ‘d‘ = det(A) in {1,3,5,7}

         ‘s‘ = 0 (or 1) if A is even (or odd)

         ‘o‘ = oddity of A (= 0 if s = 0) in Z/8Z
           = the trace of the diagonalization of A

    NOTE: The Conway–Sloane convention for describing the prime ’p =
    –1’ is not supported here, and neither is the convention for
    including the ’prime’ Infinity.  See note on p370 of Conway–Sloane
    (3rd ed) for a discussion of this convention.

    INPUT:

      –‘p‘ –– a prime number > 0

    OUTPUT:
      Returns a Conway–Sloane genus symbol at p, which is an
      instance of the Genus_Symbol_p_adic_ring class.

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,2,3,4])
        sage: Q.local_genus_symbol(2)
        Genus symbol at 2 : [[1, 2, 3, 1, 4], [2, 1, 1, 1, 1], [3, 1, 1, 1, 1]]
        sage: Q.local_genus_symbol(3)
        Genus symbol at 3 : [[0, 3, 1], [1, 1, –1]]
        sage: Q.local_genus_symbol(5)
        Genus symbol at 5 : [[0, 4, 1]]

    """
    ## Check that p is prime and that the form is defined over ZZ.
    if not is_prime(p):
        raise TypeError, "Oops!  The number " + str(p) + " isn't prime."
    if not self.base_ring() == IntegerRing():
        raise TypeError, "Oops!  The quadratic form is not defined over the integers."

    ## Return the result
    try:
        M = self.Hessian_matrix()
        return LocalGenusSymbol(M, p)
    except:
        raise TypeError, "Oops!  There is a problem computing the local genus symbol at the prime " +
str(p) + " for this form."
```

```
def CS_genus_symbol_list(self, force_recomputation=False):
    """
```

Returns the list of Conway–Sloane genus symbols in increasing order of primes dividing 2*det.

EXAMPLES::

```
    sage: Q = DiagonalQuadraticForm(ZZ, [1,2,3,4])
    sage: Q.CS_genus_symbol_list()
    [Genus symbol at 2 : [[1, 2, 3, 1, 4], [2, 1, 1, 1, 1], [3, 1, 1, 1, 1]],
    Genus symbol at 3 : [[0, 3, 1], [1, 1, -1]]]

    """
    ## Try to use the cached list
    if force_recomputation == False:
        try:
            return self.__CS_genus_symbol_list
        except:
            pass

    ## Otherwise recompute and cache the list
    list_of_CS_genus_symbols = [ ]

    for p in prime_divisors(2 * self.det()):
        list_of_CS_genus_symbols.append(self.local_genus_symbol(p))

    self.__CS_genus_symbol_list = list_of_CS_genus_symbols
    return list_of_CS_genus_symbols
```

```
"""
Local Density Congruence

"""


#################################################################################
## Methods which compute the local densities for representing a number
## by a quadratic form at a prime (possibly subject to additional
## congruence conditions).
#################################################################################

from copy import deepcopy

from sage.sets.set import Set
from sage.rings.rational_field import QQ
from sage.rings.arith import valuation, kronecker_symbol
from sage.rings.integer_ring import ZZ
from sage.misc.misc import prod, verbose

from sage.quadratic_forms.count_local_2 import count_modp__by_gauss_sum, extract
_sublist_indices




def count_modp_solutions__by_Gauss_sum(self, p, m):
    """
    Returns the number of solutions of `Q(x) = m (mod p)` of a
    non-degenerate quadratic form over the finite field `Z/pZ`,
    where `p` is a prime number > 2.

    Note: We adopt the useful convention that a zero-dimensional
    quadratic form has exactly one solution always (i.e. the empty
    vector).

    These are defined in Table 1 on p363 of Hanke's "Local
       Densities..." paper.

    INPUT:
      `p` -- a prime number > 2

      `m` -- an integer

    OUTPUT:
      an integer >= 0

    EXAMPLES::

      sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
      sage: [Q.count_modp_solutions__by_Gauss_sum(3, m)  for m in range(3)]
      [9, 6, 12]

    ::

      sage: Q = DiagonalQuadraticForm(ZZ, [1,1,2])
      sage: [Q.count_modp_solutions__by_Gauss_sum(3, m)  for m in range(3)]
      [9, 12, 6]

    """
    if self.dim() == 0:
        return 1
    else:
        return count_modp__by_gauss_sum(self.dim(), p, m, self.Gram_det())
```



```
def local_good_density_congruence_odd(self, p, m, Zvec, NZvec):
    """
    Finds the Good-type local density of Q representing `m` at `p`.
    (Assuming that `p` > 2 and Q is given in local diagonal form.)

    The additional congruence condition arguments Zvec and NZvec can
    be either a list of indices or None.  Zvec = [] is equivalent to
    Zvec = None which both impose no additional conditions, but NZvec
    = [] returns no solutions always while NZvec = None imposes no
    additional condition.

    TO DO: Add type checking for Zvec, NZvec, and that Q is in local
    normal form.

    INPUT:
      Q -- quadratic form assumed to be diagonal and p-integral

      `p` -- a prime number

      `m` -- an integer

      Zvec, NZvec -- non-repeating lists of integers in range(self.dim()) or None

    OUTPUT:
      a rational number

    EXAMPLES::

      sage: Q = DiagonalQuadraticForm(ZZ, [1,2,3])
      sage: Q.local_good_density_congruence_odd(3, 1, None, None)
      2/3

    ::

      sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1])
      sage: Q.local_good_density_congruence_odd(3, 1, None, None)
      8/9

    """
      n = self.dim()

      ## Put the Zvec congruence condition in a standard form
      if Zvec == None:
          Zvec = []


      ## Sanity Check on Zvec and NZvec:
      ## -------------------------------
      Sn = Set(range(n))
      if (Zvec != None) and (len(Set(Zvec) + Sn) > n):
          raise RuntimeError, "Zvec must be a subset of {0, ..., n-1}."
      if (NZvec != None) and (len(Set(NZvec) + Sn) > n):
          raise RuntimeError, "NZvec must be a subset of {0, ..., n-1}."



      ## Assuming Q is diagonal, find the indices of the p-unit (diagonal) entries
      UnitVec = [i  for i in range(n)  if (self[i,i] % p) != 0]
      NonUnitVec = list(Set(range(n)) - Set(UnitVec))


      ## Take cases on the existence of additional non-zero congruence conditions
(mod p)
      UnitVec_minus_Zvec = list(Set(UnitVec) - Set(Zvec))
      NonUnitVec_minus_Zvec = list(Set(NonUnitVec) - Set(Zvec))
      Q_Unit_minus_Zvec = self.extract_variables(UnitVec_minus_Zvec)

      if (NZvec == None):
          if m % p != 0:
```

```
            total = Q_Unit_minus_Zvec.count_modp_solutions__by_Gauss_sum(p, m) *
 p**len(NonUnitVec_minus_Zvec)          ## m != 0 (mod p)
        else:
            total = (Q_Unit_minus_Zvec.count_modp_solutions__by_Gauss_sum(p, m)
- 1) * p**len(NonUnitVec_minus_Zvec)    ## m == 0 (mod p)

    else:
        UnitVec_minus_ZNZvec = list(Set(UnitVec) - (Set(Zvec) + Set(NZvec)))
        NonUnitVec_minus_ZNZvec = list(Set(NonUnitVec) - (Set(Zvec) + Set(NZvec)
))
        Q_Unit_minus_ZNZvec = self.extract_variables(UnitVec_minus_ZNZvec)

        if m % p != 0:         ## m != 0 (mod p)
            total = Q_Unit_minus_Zvec.count_modp_solutions__by_Gauss_sum(p, m) *
 p**len(NonUnitVec_minus_Zvec) \
                  - Q_Unit_minus_ZNZvec.count_modp_solutions__by_Gauss_sum(p,
m) * p**len(NonUnitVec_minus_ZNZvec)
        else:                  ## m == 0 (mod p)
            total = (Q_Unit_minus_Zvec.count_modp_solutions__by_Gauss_sum(p, m)
- 1) * p**len(NonUnitVec_minus_Zvec) \
                  - (Q_Unit_minus_ZNZvec.count_modp_solutions__by_Gauss_sum(p,
 m) - 1) * p**len(NonUnitVec_minus_ZNZvec)

    ## Return the Good-type representation density
    good_density = QQ(total) / p**(n-1)
    return good_density



def local_good_density_congruence_even(self, m, Zvec, NZvec):
    """
  Finds the Good--type local density of Q representing 'm' at 'p=2'.
  (Assuming Q is given in local diagonal form.)


  The additional congruence condition arguments Zvec and NZvec can
  be either a list of indices or None.  Zvec = [] is equivalent to
  Zvec = None which both impose no additional conditions, but NZvec
  = [] returns no solutions always while NZvec = None imposes no
  additional condition.

  WARNING: Here the indices passed in Zvec and NZvec represent
  indices of the solution vector 'x' of Q('x') = 'm (mod p^k)', and *not*
  the Jordan components of Q.  They therefore are required (and
  assumed) to include either all or none of the indices of a given
  Jordan component of Q.  This is only important when 'p=2' since
  otherwise all Jordan blocks are 1x1, and so there the indices and
  Jordan blocks coincide.

  TO DO: Add type checking for Zvec, NZvec, and that Q is in local
  normal form.


  INPUT:
    Q -- quadratic form assumed to be block diagonal and 2-integral

    'p' -- a prime number

    'm' -- an integer

    Zvec, NZvec -- non-repeating lists of integers in range(self.dim()) or None

  OUTPUT:
    a rational number

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,2,3])
```



```
    sage: Q.local_good_density_congruence_even(1, None, None)
    1

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1])
    sage: Q.local_good_density_congruence_even(1, None, None)
    1
    sage: Q.local_good_density_congruence_even(2, None, None)
    3/2
    sage: Q.local_good_density_congruence_even(3, None, None)
    1
    sage: Q.local_good_density_congruence_even(4, None, None)
    1/2

  ::

    sage: Q = QuadraticForm(ZZ, 4, range(10))
    sage: Q[0,0] = 5
    sage: Q[1,1] = 10
    sage: Q[2,2] = 15
    sage: Q[3,3] = 20
    sage: Q
    Quadratic form in 4 variables over Integer Ring with coefficients:
    [ 5 1 2 3 ]
    [ * 10 5 6 ]
    [ * * 15 8 ]
    [ * * * 20 ]
    sage: Q.theta_series(20)
    1 + 2*q^5 + 2*q^10 + 2*q^14 + 2*q^15 + 2*q^16 + 2*q^18 + O(q^20)
    sage: Q.local_normal_form(2)
    Quadratic form in 4 variables over Integer Ring with coefficients:
    [ 0 1 0 0 ]
    [ * 0 0 0 ]
    [ * * 0 1 ]
    [ * * * 0 ]
    sage: Q.local_good_density_congruence_even(1, None, None)
    3/4
    sage: Q.local_good_density_congruence_even(2, None, None)
    1
    sage: Q.local_good_density_congruence_even(5, None, None)
    3/4

  """
    n = self.dim()

    ## Put the Zvec congruence condition in a standard form
    if Zvec == None:
        Zvec = []


    ## Sanity Check on Zvec and NZvec:
    ## -------------------------------
    Sn = Set(range(n))
    if (Zvec != None) and (len(Set(Zvec) + Sn) > n):
        raise RuntimeError, "Zvec must be a subset of {0, ..., n-1}."
    if (NZvec != None) and (len(Set(NZvec) + Sn) > n):
        raise RuntimeError, "NZvec must be a subset of {0, ..., n-1}."



    ## Find the indices of x for which the associated Jordan blocks are non-zero
 mod 8    TO DO: Move this to special Jordan block code separately!
    ## ---------------------------------------------------------------------
------
    Not8vec = []
    for i in range(n):

        ## DIAGNOSTIC
```

```
        verbose(" i = " + str(i))
        verbose(" n = " + str(n))
        verbose(" Not8vec = " + str(Not8vec))

        nz_flag = False

        ## Check if the diagonal entry isn't divisible 8
        if  ((self[i,i] % 8) != 0):
            nz_flag = True

        ## Check appropriate off-diagonal entries aren't divisible by 8
        else:

            ## Special check for first off-diagonal entry
            if ((i == 0) and ((self[i,i+1] % 8) != 0)):
                nz_flag = True

            ## Special check for last off-diagonal entry
            elif ((i == n-1) and ((self[i-1,i] % 8) != 0)):
                nz_flag = True

            ## Check for the middle off-diagonal entries
            else:
                if ( (i > 0)  and  (i < n-1)  and  (((self[i,i+1] % 8) != 0) or
((self[i-1,i] % 8) != 0)) ):
                    nz_flag = True

        ## Remember the (vector) index if it's not part of a Jordan block of nor
m divisible by 8
        if (nz_flag == True):
            Not8vec += [i]



    ## Compute the number of Good-type solutions mod 8:
    ## -----------------------------------------------

    ## Setup the indexing sets for additional zero congruence solutions
    Q_Not8 = self.extract_variables(Not8vec)
    Not8 = Set(Not8vec)
    Is8 = Set(range(n)) - Not8

    Z = Set(Zvec)
    Z_Not8 = Not8.intersection(Z)
    Z_Is8 = Is8.intersection(Z)
    Is8_minus_Z = Is8 - Z_Is8


    ## DIAGNOSTIC
    verbose("Z = " + str(Z))
    verbose("Z_Not8 = " + str(Z_Not8))
    verbose("Z_Is8 = " + str(Z_Is8))
    verbose("Is8_minus_Z = " + str(Is8_minus_Z))


    ## Take cases on the existence of additional non-zero congruence conditions
(mod 2)
    if NZvec == None:
        total = (4 ** len(Z_Is8)) * (8 ** len(Is8_minus_Z)) \
            * Q_Not8.count_congruence_solutions__good_type(2, 3, m, list(Z_Not8)
, None)
    else:
        ZNZ = Z + Set(NZvec)
        ZNZ_Not8 = Not8.intersection(ZNZ)
        ZNZ_Is8 = Is8.intersection(ZNZ)
        Is8_minus_ZNZ = Is8 - ZNZ_Is8

        ## DIAGNOSTIC
        verbose("ZNZ = " + str(ZNZ))
```



```
        verbose("ZNZ_Not8 = " + str(ZNZ_Not8))
        verbose("ZNZ_Is8 = " + str(ZNZ_Is8))
        verbose("Is8_minus_ZNZ = " + str(Is8_minus_ZNZ))

        total = (4 ** len(Z_Is8)) * (8 ** len(Is8_minus_Z)) \
            * Q_Not8.count_congruence_solutions__good_type(2, 3, m, list(Z_Not8)
, None) \
            - (4 ** len(ZNZ_Is8)) * (8 ** len(Is8_minus_ZNZ)) \
            * Q_Not8.count_congruence_solutions__good_type(2, 3, m, list(ZNZ_Not
8), None)


    ## DIAGNOSTIC
    verbose("total = " + str(total))


    ## Return the associated Good-type representation density
    good_density = QQ(total) / 8**(n-1)
    return good_density




def local_good_density_congruence(self, p, m, Zvec=None, NZvec=None):
    """
  Finds the Good-type local density of Q representing 'm' at 'p'.
  (Front end routine for parity specific routines for p.)

  TO DO: Add Documentation about the additional congruence
  conditions Zvec and NZvec.



  INPUT:
    Q -- quadratic form assumed to be block diagonal and p-integral

    'p' -- a prime number

    'm' -- an integer

    Zvec, NZvec -- non-repeating lists of integers in range(self.dim()) or None

  OUTPUT:
    a rational number

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,2,3])
    sage: Q.local_good_density_congruence(2, 1, None, None)
    1
    sage: Q.local_good_density_congruence(3, 1, None, None)
    2/3

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1])
    sage: Q.local_good_density_congruence(2, 1, None, None)
    1
    sage: Q.local_good_density_congruence(3, 1, None, None)
    8/9

  """
    ## DIAGNOSTIC
    verbose(" In local_good_density_congruence with ")
    verbose(" Q is: \n" + str(self))
```

```
    verbose(" p = " + str(p))
    verbose(" m = " + str(m))
    verbose(" Zvec = " + str(Zvec))
    verbose(" NZvec = " + str(NZvec))

    ## Put the Zvec congruence condition in a standard form
    if Zvec == None:
        Zvec = []


    n = self.dim()

    ## Sanity Check on Zvec and NZvec:
    ## -------------------------------
    Sn = Set(range(n))
    if (Zvec != None) and (len(Set(Zvec) + Sn) > n):
        raise RuntimeError, "Zvec must be a subset of {0, ..., n-1}."
    if (NZvec != None) and (len(Set(NZvec) + Sn) > n):
        raise RuntimeError, "NZvec must be a subset of {0, ..., n-1}."



    ## Check that Q is in local normal form -- should replace this with a diagon
alization check?
    ##   (it often may not be since the reduction procedure
    ##   often mixes up the order of the valuations...)
    #
    #if (self != self.local_normal_form(p))
    #    print "Warning in local_good_density_congruence: Q is not in local norm
al form! \n";




    ## Decide which routine to use to compute the Good-type density
    if (p > 2):
        return self.local_good_density_congruence_odd(p, m, Zvec, NZvec)

    if (p == 2):
        #print "\n Using the (p=2) Local_Good_Density_Even routine! \n"
        return self.local_good_density_congruence_even(m, Zvec, NZvec)

    raise RuntimeError, "\n Error in Local_Good_Density: The 'prime' p = " + str(p) + " is < 2. \n"







def local_zero_density_congruence(self, p, m, Zvec=None, NZvec=None):
    """
```

Finds the Zero–type local density of Q representing ‘m‘ at ‘p‘,
allowing certain congruence conditions mod p.

INPUT:

Q –– quadratic form assumed to be block diagonal and ‘p‘–integral

‘p‘ –– a prime number

‘m‘ –– an integer

Zvec, NZvec –– non–repeating lists of integers in range(self.dim()) or None

OUTPUT:

a rational number

EXAMPLES::


```
    sage: Q = DiagonalQuadraticForm(ZZ, [1,2,3])
    sage: Q.local_zero_density_congruence(2, 2, None, None)
    0
    sage: Q.local_zero_density_congruence(2, 4, None, None)
    1/2
    sage: Q.local_zero_density_congruence(3, 6, None, None)
    0
    sage: Q.local_zero_density_congruence(3, 9, None, None)
    2/9

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1])
    sage: Q.local_zero_density_congruence(2, 2, None, None)
    0
    sage: Q.local_zero_density_congruence(2, 4, None, None)
    1/4
    sage: Q.local_zero_density_congruence(3, 6, None, None)
    0
    sage: Q.local_zero_density_congruence(3, 9, None, None)
    8/81

  """
    ## DIAGNOSTIC
    verbose(" In local_zero_density_congruence with ")
    verbose(" Q is: \n" + str(self))
    verbose(" p = " + str(p))
    verbose(" m = " + str(m))
    verbose(" Zvec = " + str(Zvec))
    verbose(" NZvec = " + str(NZvec))

    ## Put the Zvec congruence condition in a standard form
    if Zvec == None:
        Zvec = []


    n = self.dim()

    ## Sanity Check on Zvec and NZvec:
    ## -------------------------------
    Sn = Set(range(n))
    if (Zvec != None) and (len(Set(Zvec) + Sn) > n):
        raise RuntimeError, "Zvec must be a subset of {0, ..., n-1}."
    if (NZvec != None) and (len(Set(NZvec) + Sn) > n):
        raise RuntimeError, "NZvec must be a subset of {0, ..., n-1}."


    p2 = p * p

    ## Check some conditions for no zero-type solutions to exist
    if ((m % (p2) != 0) or (NZvec != None)):
        return 0

    ## Use the reduction procedure to return the result
    return self.local_density_congruence(p, m / p2, None, None) / p**(self.dim()
 - 2)







def local_badI_density_congruence(self, p, m, Zvec=None, NZvec=None):
    """
```

Finds the Bad–type I local density of Q representing ‘m‘ at ‘p‘.
(Assuming that p > 2 and Q is given in local diagonal form.)

```
INPUT:
    Q -- quadratic form assumed to be block diagonal and 'p'-integral

    'p' -- a prime number

    'm' -- an integer

    Zvec, NZvec -- non-repeating lists of integers in range(self.dim()) or None

OUTPUT:
    a rational number

EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,2,3])
    sage: Q.local_badI_density_congruence(2, 1, None, None)
    0
    sage: Q.local_badI_density_congruence(2, 2, None, None)
    1
    sage: Q.local_badI_density_congruence(2, 4, None, None)
    0
    sage: Q.local_badI_density_congruence(3, 1, None, None)
    0
    sage: Q.local_badI_density_congruence(3, 6, None, None)
    0
    sage: Q.local_badI_density_congruence(3, 9, None, None)
    0

::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1])
    sage: Q.local_badI_density_congruence(2, 1, None, None)
    0
    sage: Q.local_badI_density_congruence(2, 2, None, None)
    0
    sage: Q.local_badI_density_congruence(2, 4, None, None)
    0
    sage: Q.local_badI_density_congruence(3, 2, None, None)
    0
    sage: Q.local_badI_density_congruence(3, 6, None, None)
    0
    sage: Q.local_badI_density_congruence(3, 9, None, None)
    0

::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,3,3,9])
    sage: Q.local_badI_density_congruence(3, 1, None, None)
    0
    sage: Q.local_badI_density_congruence(3, 3, None, None)
    4/3
    sage: Q.local_badI_density_congruence(3, 6, None, None)
    4/3
    sage: Q.local_badI_density_congruence(3, 9, None, None)
    0
    sage: Q.local_badI_density_congruence(3, 18, None, None)
    0


"""
    ## DIAGNOSTIC
    verbose(" In local_badI_density_congruence with ")
    verbose(" Q is: \n" + str(self))
    verbose(" p = " + str(p))
    verbose(" m = " + str(m))
    verbose(" Zvec = " + str(Zvec))
    verbose(" NZvec = " + str(NZvec))
```



```
    ## Put the Zvec congruence condition in a standard form
    if Zvec == None:
        Zvec = []


    n = self.dim()



    ## Sanity Check on Zvec and NZvec:
    ## -------------------------------
    Sn = Set(range(n))
    if (Zvec != None) and (len(Set(Zvec) + Sn) > n):
        raise RuntimeError, "Zvec must be a subset of {0, ..., n-1}."
    if (NZvec != None) and (len(Set(NZvec) + Sn) > n):
        raise RuntimeError, "NZvec must be a subset of {0, ..., n-1}."



    ## Define the indexing set S_0, and determine if S_1 is empty:
    ## -----------------------------------------------------------
    S0 = []
    S1_empty_flag = True    ## This is used to check if we should be computing B
I solutions at all!
                            ## (We should really to this earlier, but S1 must be
 non-zero to proceed.)

    ## Find the valuation of each variable (which will be the same over 2x2 bloc
ks),
    ## remembering those of valuation 0 and if an entry of valuation 1 exists.
    for i in range(n):

        ## Compute the valuation of each index, allowing for off-diagonal terms
        if (self[i,i] == 0):
            if (i == 0):
                val = valuation(self[i,i+1], p)    ## Look at the term to the ri
ght
            else:
                if (i == n-1):
                    val = valuation(self[i-1,i], p)    ## Look at the term above
                else:
                    val = valuation(self[i,i+1] + self[i-1,i], p)    ## Finds th
e valuation of the off-diagonal term since only one isn't zero
        else:
            val = valuation(self[i,i], p)

        if (val == 0):
            S0 += [i]
        elif (val == 1):
            S1_empty_flag = False    ## Need to have a non-empty S1 set to proce
ed with Bad-type I reduction...




    ## Check that S1 is non-empty and p|m to proceed, otherwise return no soluti
ons.
    if (S1_empty_flag == True) or (m % p != 0):
        return 0

    ## Check some conditions for no bad-type I solutions to exist
    if (NZvec != None) and (len(Set(S0).intersection(Set(NZvec))) != 0):
        return 0



    ## Check that the form is primitive...                    WHY DO WE NEED TO
 DO THIS?!?
```

```
    if (S0 == []):
        print " Using Q = " + str(self)
        print " and p = " + str(p)
        raise RuntimeError, "Oops! The form is not primitive!"



    ## DIAGNOSTIC
    verbose(" m = " + str(m) + "  p = " + str(p))
    verbose(" S0 = " + str(S0))
    verbose(" len(S0) = " + str(len(S0)))



    ## Make the form Qnew for the reduction procedure:
    ## ----------------------------------------------
    Qnew = deepcopy(self)                                   ## TO DO:  DO
 THIS WITHOUT A copy(). =)
    for i in range(n):
        if i in S0:
            Qnew[i,i] = p * Qnew[i,i]
            if ((p == 2) and (i < n-1)):
                Qnew[i,i+1] = p * Qnew[i,i+1]
        else:
            Qnew[i,i] = Qnew[i,i] / p
            if ((p == 2) and (i < n-1)):
                Qnew[i,i+1] = Qnew[i,i+1] / p



    ## DIAGNOSTIC
    verbose("\n\n Check of Bad-type I reduction: \n")
    verbose(" Q is " + str(self))
    verbose(" Qnew is " + str(Qnew))
    verbose(" p = " + str(p))
    verbose(" m / p = " + str(m/p))
    verbose(" NZvec " + str(NZvec))



    ## Do the reduction
    Zvec_geq_1 = list(Set([i  for i in Zvec  if i not in S0]))
    if NZvec == None:
        NZvec_geq_1 = NZvec
    else:
        NZvec_geq_1 = list(Set([i  for i in NZvec  if i not in S0]))

    return QQ(p**(1 - len(S0))) * Qnew.local_good_density_congruence(p, m / p, Z
vec_geq_1, NZvec_geq_1)







def local_badII_density_congruence(self, p, m, Zvec=None, NZvec=None):
    """
  Finds the Bad-type II local density of Q representing 'm' at 'p'.
  (Assuming that 'p' > 2 and Q is given in local diagonal form.)


  INPUT:
    Q -- quadratic form assumed to be block diagonal and p-integral

    'p' -- a prime number

    'm' -- an integer
```



```
    Zvec, NZvec -- non-repeating lists of integers in range(self.dim()) or None

  OUTPUT:
    a rational number

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,2,3])
    sage: Q.local_badII_density_congruence(2, 1, None, None)
    0
    sage: Q.local_badII_density_congruence(2, 2, None, None)
    0
    sage: Q.local_badII_density_congruence(2, 4, None, None)
    0
    sage: Q.local_badII_density_congruence(3, 1, None, None)
    0
    sage: Q.local_badII_density_congruence(3, 6, None, None)
    0
    sage: Q.local_badII_density_congruence(3, 9, None, None)
    0
    sage: Q.local_badII_density_congruence(3, 27, None, None)
    0

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,3,3,9,9])
    sage: Q.local_badII_density_congruence(3, 1, None, None)
    0
    sage: Q.local_badII_density_congruence(3, 3, None, None)
    0
    sage: Q.local_badII_density_congruence(3, 6, None, None)
    0
    sage: Q.local_badII_density_congruence(3, 9, None, None)
    4/27
    sage: Q.local_badII_density_congruence(3, 18, None, None)
    4/9

  """
    ## DIAGNOSTIC
    verbose(" In local_badII_density_congruence with ")
    verbose(" Q is: \n" + str(self))
    verbose(" p = " + str(p))
    verbose(" m = " + str(m))
    verbose(" Zvec = " + str(Zvec))
    verbose(" NZvec = " + str(NZvec))

    ## Put the Zvec congruence condition in a standard form
    if Zvec == None:
        Zvec = []


    n = self.dim()


    ## Sanity Check on Zvec and NZvec:
    ## -------------------------------
    Sn = Set(range(n))
    if (Zvec != None) and (len(Set(Zvec) + Sn) > n):
        raise RuntimeError, "Zvec must be a subset of {0, ..., n-1}."
    if (NZvec != None) and (len(Set(NZvec) + Sn) > n):
        raise RuntimeError, "NZvec must be a subset of {0, ..., n-1}."


    ## Define the indexing sets S_i:
    ## -----------------------------
    S0 = []
    S1 = []
    S2plus = []
```

```
    for i in range(n):

        ## Compute the valuation of each index, allowing for off-diagonal terms
        if (self[i,i] == 0):
            if (i == 0):
                val = valuation(self[i,i+1], p)    ## Look at the term to the ri
ght
            elif (i == n-1):
                val = valuation(self[i-1,i], p)    ## Look at the term above
            else:
                val = valuation(self[i,i+1] + self[i-1,i], p)    ## Finds the va
luation of the off-diagonal term since only one isn't zero
        else:
            val = valuation(self[i,i], p)

        ## Sort the indices into disjoint sets by their valuation
        if (val == 0):
            S0 += [i]
        elif (val == 1):
            S1 += [i]
        elif (val >= 2):
            S2plus += [i]



    ## Check that S2 is non-empty and p^2 divides m to proceed, otherwise return
 no solutions.
    p2 = p * p
    if (S2plus == []) or (m % p2 != 0):
        return 0




    ## Check some conditions for no bad-type II solutions to exist
    if (NZvec != None) and (len(Set(S2plus).intersection(Set(NZvec))) == 0):
        return 0



    ## Check that the form is primitive...                      WHY IS THIS NECES
SARY?
    if (S0 == []):
        print " Using Q = " + str(self)
        print " and p = " + str(p)
        raise RuntimeError, "Oops! The form is not primitive!"




    ## DIAGNOSTIC
    verbose("\n Entering BII routine ")
    verbose(" S0 is " + str(S0))
    verbose(" S1 is " + str(S1))
    verbose(" S2plus is " + str(S2plus))
    verbose(" m = " + str(m) + "  p = " + str(p))




    ## Make the form Qnew for the reduction procedure:
    ## ----------------------------------------------
    Qnew = deepcopy(self)                                          ## TO DO:  DO
 THIS WITHOUT A copy(). =)
    for i in range(n):
        if i in S2plus:
            Qnew[i,i] = Qnew[i,i] / p2
            if (p == 2) and (i < n-1):
```



```
                Qnew[i,i+1] = Qnew[i,i+1] / p2

    ## DIAGNOSTIC
    verbose("\n\n Check of Bad-type II reduction: \n")
    verbose(" Q is " + str(self))
    verbose(" Qnew is " + str(Qnew))



    ## Perform the reduction formula
    Zvec_geq_2 = list(Set([i  for i in Zvec  if i in S2plus]))
    if NZvec == None:
        NZvec_geq_2 = NZvec
    else:
        NZvec_geq_2 = list(Set([i  for i in NZvec  if i in S2plus]))

    return QQ(p**(len(S2plus) + 2 - n)) \
        * (Qnew.local_density_congruence(p, m / p2, Zvec_geq_2, NZvec_geq_2) \
        - Qnew.local_density_congruence(p, m / p2, S2plus , NZvec_geq_2))




def local_bad_density_congruence(self, p, m, Zvec=None, NZvec=None):
    """
    Finds the Bad-type local density of Q representing
    'm' at 'p', allowing certain congruence conditions mod 'p'.

    INPUT:
        Q -- quadratic form assumed to be block diagonal and p-integral

        'p' -- a prime number

        'm' -- an integer

        Zvec, NZvec -- non-repeating lists of integers in range(self.dim()) or None

    OUTPUT:
        a rational number

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,2,3])
        sage: Q.local_bad_density_congruence(2, 1, None, None)
        0
        sage: Q.local_bad_density_congruence(2, 2, None, None)
        1
        sage: Q.local_bad_density_congruence(2, 4, None, None)
        0
        sage: Q.local_bad_density_congruence(3, 1, None, None)
        0
        sage: Q.local_bad_density_congruence(3, 6, None, None)
        0
        sage: Q.local_bad_density_congruence(3, 9, None, None)
        0
        sage: Q.local_bad_density_congruence(3, 27, None, None)
        0

    ::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,3,3,9,9])
        sage: Q.local_bad_density_congruence(3, 1, None, None)
        0
        sage: Q.local_bad_density_congruence(3, 3, None, None)
        4/3
        sage: Q.local_bad_density_congruence(3, 6, None, None)
        4/3
```

```
    sage: Q.local_bad_density_congruence(3, 9, None, None)
    4/27
    sage: Q.local_bad_density_congruence(3, 18, None, None)
    4/9
    sage: Q.local_bad_density_congruence(3, 27, None, None)
    8/27

  """
    return self.local_badI_density_congruence(p, m, Zvec, NZvec) + self.local_ba
dII_density_congruence(p, m, Zvec, NZvec)



#######################################################
## local_density and local_density_congruence routines ##
#######################################################

def local_density_congruence(self, p, m, Zvec=None, NZvec=None):
    """
  Finds the local density of Q representing 'm' at 'p',
  allowing certain congruence conditions mod 'p'.

  INPUT:
    Q -- quadratic form assumed to be block diagonal and p-integral

    'p' -- a prime number

    'm' -- an integer

    Zvec, NZvec -- non-repeating lists of integers in range(self.dim()) or None

  OUTPUT:
    a rational number

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1])
    sage: Q.local_density_congruence(p=2, m=1, Zvec=None, NZvec=None)
    1
    sage: Q.local_density_congruence(p=3, m=1, Zvec=None, NZvec=None)
    8/9
    sage: Q.local_density_congruence(p=5, m=1, Zvec=None, NZvec=None)
    24/25
    sage: Q.local_density_congruence(p=7, m=1, Zvec=None, NZvec=None)
    48/49
    sage: Q.local_density_congruence(p=11, m=1, Zvec=None, NZvec=None)
    120/121

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,2,3])
    sage: Q.local_density_congruence(2, 1, None, None)
    1
    sage: Q.local_density_congruence(2, 2, None, None)
    1
    sage: Q.local_density_congruence(2, 4, None, None)
    3/2
    sage: Q.local_density_congruence(3, 1, None, None)
    2/3
    sage: Q.local_density_congruence(3, 6, None, None)
    4/3
    sage: Q.local_density_congruence(3, 9, None, None)
    14/9
    sage: Q.local_density_congruence(3, 27, None, None)
    2

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,3,3,9,9])
```



```
    sage: Q.local_density_congruence(3, 1, None, None)
    2
    sage: Q.local_density_congruence(3, 3, None, None)
    4/3
    sage: Q.local_density_congruence(3, 6, None, None)
    4/3
    sage: Q.local_density_congruence(3, 9, None, None)
    2/9
    sage: Q.local_density_congruence(3, 18, None, None)
    4/9

  """
    return self.local_good_density_congruence(p, m, Zvec, NZvec) \
                + self.local_zero_density_congruence(p, m, Zvec, NZvec) \
                + self.local_bad_density_congruence(p, m, Zvec, NZvec)



def local_primitive_density_congruence(self, p, m, Zvec=None, NZvec=None):
    """
  Finds the primitive local density of Q representing
  'm' at 'p', allowing certain congruence conditions mod 'p'.

  Note: The following routine is not used internally, but is included for consistency.

  INPUT:
    Q -- quadratic form assumed to be block diagonal and p-integral

    'p' -- a prime number

    'm' -- an integer

    Zvec, NZvec -- non-repeating lists of integers in range(self.dim()) or None

  OUTPUT:
    a rational number

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1])
    sage: Q.local_primitive_density_congruence(p=2, m=1, Zvec=None, NZvec=None)
    1
    sage: Q.local_primitive_density_congruence(p=3, m=1, Zvec=None, NZvec=None)
    8/9
    sage: Q.local_primitive_density_congruence(p=5, m=1, Zvec=None, NZvec=None)
    24/25
    sage: Q.local_primitive_density_congruence(p=7, m=1, Zvec=None, NZvec=None)
    48/49
    sage: Q.local_primitive_density_congruence(p=11, m=1, Zvec=None, NZvec=None)
    120/121

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,2,3])
    sage: Q.local_primitive_density_congruence(2, 1, None, None)
    1
    sage: Q.local_primitive_density_congruence(2, 2, None, None)
    1
    sage: Q.local_primitive_density_congruence(2, 4, None, None)
    1
    sage: Q.local_primitive_density_congruence(3, 1, None, None)
    2/3
    sage: Q.local_primitive_density_congruence(3, 6, None, None)
    4/3
    sage: Q.local_primitive_density_congruence(3, 9, None, None)
    4/3
    sage: Q.local_primitive_density_congruence(3, 27, None, None)
    4/3
```

```
::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,3,3,9,9])
    sage: Q.local_primitive_density_congruence(3, 1, None, None)
    2
    sage: Q.local_primitive_density_congruence(3, 3, None, None)
    4/3
    sage: Q.local_primitive_density_congruence(3, 6, None, None)
    4/3
    sage: Q.local_primitive_density_congruence(3, 9, None, None)
    4/27
    sage: Q.local_primitive_density_congruence(3, 18, None, None)
    4/9
    sage: Q.local_primitive_density_congruence(3, 27, None, None)
    8/27
    sage: Q.local_primitive_density_congruence(3, 81, None, None)
    8/27
    sage: Q.local_primitive_density_congruence(3, 243, None, None)
    8/27

"""
    return self.local_good_density_congruence(p, m, Zvec, NZvec) \
                + self.local_bad_density_congruence(p, m, Zvec, NZvec)
```

```
"""
Local Density Interfaces
"""
## // This is needed in the filter for primitivity...
## #include "../max-min.h"

from copy import deepcopy

from sage.rings.arith import valuation
from sage.rings.rational_field import QQ, RationalField




def local_density(self, p, m):
    """
  Gives the local density -- should be called by the user. =)

  NOTE: This screens for imprimitive forms, and puts the quadratic
  form in local normal form, which is a *requirement* of the
  routines performing the computations!

  INPUT:
    'p' -- a prime number > 0
    'm' -- an integer

  OUTPUT:
    a rational number

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1])  ## NOTE: This is already in local normal form for *all* primes
p!
    sage: Q.local_density(p=2, m=1)
    1
    sage: Q.local_density(p=3, m=1)
    8/9
    sage: Q.local_density(p=5, m=1)
    24/25
    sage: Q.local_density(p=7, m=1)
    48/49
    sage: Q.local_density(p=11, m=1)
    120/121

  """
    n = self.dim()
    if (n == 0):
        raise TypeError, "Oops!  We currently don't handle 0-dim'l forms. =("

    ## Find the local normal form and p-scale of Q      --  Note: This uses the v
aluation ordering of local_normal_form.
    ##                                                     TO DO:  Write a separ
ate p-scale and p-norm routines!
    Q_local = self.local_normal_form(p)
    if n == 1:
        p_valuation = valuation(Q_local[0,0], p)
    else:
        p_valuation = min(valuation(Q_local[0,0], p), valuation(Q_local[0,1], p)
)

    ## If m is less p-divisible than the matrix, return zero
    if ((m != 0) and (valuation(m,p) < p_valuation)):   ## Note: The (m != 0) co
ndition protects taking the valuation of zero.
        return QQ(0)


    ## If the form is imprimitive, rescale it and call the local density routine
    p_adjustment = QQ(1) / p**p_valuation
```



```
    m_prim = QQ(m) / p**p_valuation
    Q_prim = Q_local.scale_by_factor(p_adjustment)

    ## Return the densities for the reduced problem
    return Q_prim.local_density_congruence(p, m_prim)




def local_primitive_density(self, p, m):
    """
  Gives the local primitive density -- should be called by the user. =)

  NOTE: This screens for imprimitive forms, and puts the
  quadratic form in local normal form, which is a *requirement* of
  the routines performing the computations!

  INPUT:
    'p' -- a prime number > 0
    'm' -- an integer

  OUTPUT:
    a rational number

  EXAMPLES::

    sage: Q = QuadraticForm(ZZ, 4, range(10))
    sage: Q[0,0] = 5
    sage: Q[1,1] = 10
    sage: Q[2,2] = 15
    sage: Q[3,3] = 20
    sage: Q
    Quadratic form in 4 variables over Integer Ring with coefficients:
    [ 5 1 2 3 ]
    [ * 10 5 6 ]
    [ * * 15 8 ]
    [ * * * 20 ]
    sage: Q.theta_series(20)
    1 + 2*q^5 + 2*q^10 + 2*q^14 + 2*q^15 + 2*q^16 + 2*q^18 + O(q^20)
    sage: Q.local_normal_form(2)
    Quadratic form in 4 variables over Integer Ring with coefficients:
    [ 0 1 0 0 ]
    [ * 0 0 0 ]
    [ * * 0 1 ]
    [ * * * 0 ]

    sage: Q.local_primitive_density(2, 1)
    3/4
    sage: Q.local_primitive_density(5, 1)
    24/25

    sage: Q.local_primitive_density(2, 5)
    3/4
    sage: Q.local_density(2, 5)
    3/4

  """
    n = self.dim()
    if (n == 0):
        raise TypeError, "Oops!  We currently don't handle 0-dim'l forms. =("

    ## Find the local normal form and p-scale of Q      --  Note: This uses the v
aluation ordering of local_normal_form.
    ##                                                     TO DO:  Write a separ
ate p-scale and p-norm routines!
    Q_local = self.local_normal_form(p)
    if n == 1:
        p_valuation = valuation(Q_local[0,0], p)
    else:
```

```
        p_valuation = min(valuation(Q_local[0,0], p), valuation(Q_local[0,1], p)
)


    ## If m is less p-divisible than the matrix, return zero
    if ((m != 0) and (valuation(m,p) < p_valuation)):   ## Note: The (m != 0) co
ndition protects taking the valuation of zero.
        return QQ(0)


    ## If the form is imprimitive, rescale it and call the local density routine
    p_adjustment = QQ(1) / p**p_valuation
    m_prim = QQ(m) / p**p_valuation
    Q_prim = Q_local.scale_by_factor(p_adjustment)

    ## Return the densities for the reduced problem
    return Q_prim.local_primitive_density_congruence(p, m_prim)
```

```
"""
Local Field Invariants
"""
#*****************************************************************************
#       Copyright (C) 2007 William Stein and Jonathan Hanke
#
#  Distributed under the terms of the GNU General Public License (GPL)
#
#    This code is distributed in the hope that it will be useful,
#    but WITHOUT ANY WARRANTY; without even the implied warranty of
#    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
#    General Public License for more details.
#
#  The full text of the GPL is available at:
#
#                  http://www.gnu.org/licenses/
#*****************************************************************************


############################################################################
## TO DO: Add routines for hasse invariants at all places, anisotropic
## places, is_semi_definite, and support for number fields.
############################################################################


import copy

from sage.rings.integer_ring import ZZ
from sage.rings.rational_field import QQ
from sage.rings.real_mpfr import RR
from sage.rings.arith import prime_divisors, valuation, hilbert_symbol
from sage.functions.all import sgn
from sage.rings.fraction_field import FractionField
from sage.matrix.matrix_space import MatrixSpace
from sage.rings.arith import GCD


## Routines to compute local (p-adic) invariants of a quadratic form Q:
## (Note: Here Q is the matrix so that Q(x) = x^t * Q * x.)
## ---------------------------------------------------------------------

def rational_diagonal_form(self, return_matrix=False):
    """
    Returns a diagonal form equivalent to Q over the fraction field of
    its defining ring.  If the return_matrix is True, then we return
    the transformation matrix performing the diagonalization as the
    second argument.

    INPUT:
        none

    OUTPUT:
        Q -- the diagonalized form of this quadratic form
        (optional) T -- matrix which diagonalizes Q (over it's fraction field)

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,3,5,7])
        sage: Q.rational_diagonal_form(return_matrix=True)
        (Quadratic form in 4 variables over Rational Field with coefficients:
        [ 1 0 0 0 ]
        [ * 3 0 0 ]
        [ * * 5 0 ]
        [ * * * 7 ]
        ,
        [1 0 0 0]
        [0 1 0 0]
        [0 0 1 0]
        [0 0 0 1])
```



```
    ::

        sage: Q1 = QuadraticForm(ZZ, 4, [1, 1, 0, 0, 1, 0, 0, 1, 0, 18])
        sage: Q1
        Quadratic form in 4 variables over Integer Ring with coefficients:
        [ 1 1 0 0 ]
        [ * 1 0 0 ]
        [ * * 1 0 ]
        [ * * * 18 ]
        sage: Q1.rational_diagonal_form(return_matrix=True)
        (Quadratic form in 4 variables over Rational Field with coefficients:
        [ 1 0 0 0 ]
        [ * 3/4 0 0 ]
        [ * * 1 0 ]
        [ * * * 18 ]
        ,
        [   1 -1/2   0   0]
        [   0   1   0   0]
        [   0   0   1   0]
        [   0   0   0   1])

    """
        n = self.dim()
        Q = copy.deepcopy(self)
        Q.__init__(FractionField(self.base_ring()), self.dim(), self.coefficients())
        MS = MatrixSpace(Q.base_ring(), n, n)
        T = MS(1)

        ## Clear the entries one row at a time.
        for i in range(n):

            ## Deal with rows where the diagonal entry is zero.
            if Q[i,i] == 0:

                ## SANITY CHECK:  Raise an error in characteristic 2!
                R = self.base_ring()
                if R(2) == R(0):
                    raise NotImplementedError, "In characteristic 2 we can't make non-zero pivots in
without more work!"

                ## Look for a non-zero entry and use it to make the diagonal non-zer
o (if it exists)
                for j in range(i+1, n):
                    if Q[i,j] != 0:

                        ## Find a shearing that makes the diagonal entry non-zero
                        bad_shearing_factor = -Q[j,j] / Q[i,j]
                        if bad_shearing_factor == 1:
                            shear_factor = 2
                        else:
                            shear_factor = 1

                        ## Make the shearing transformation
                        temp = MS(1)
                        temp[j, i] = shear_factor

                        ## Apply the transformation
                        Q = Q(temp)
                        T = T * temp
                        break

            ## Create a matrix which deals with off-diagonal entries (all at once fo
r each row)
            temp = MS(1)
            for j in range(i+1, n):
                if Q[i,j] != 0:
                    temp[i,j] = -Q[i,j] / (Q[i,i] * 2)    ## This should only occur
when Q[i,i] != 0, which the above step guarantees.
```

```
        Q = Q(temp)
        T = T * temp


    ## Return the appropriate output
    if return_matrix:
        return Q, T
    else:
        return Q




def signature_vector(self):
    """
  Returns the triple '(p, n, z)' of integers where

  – 'p' = number of positive eigenvalues
  – 'n' = number of negative eigenvalues
  – 'z' = number of zero eigenvalues

  for the symmetric matrix associated to Q.

  INPUT:

  (none)

  OUTPUT:

  a triple of integers >= 0

  EXAMPLES::

     sage: Q = DiagonalQuadraticForm(ZZ, [1,0,0,–4])
     sage: Q.signature_vector()
     (1, 1, 2)

  ::

     sage: Q = DiagonalQuadraticForm(ZZ, [1,2,–3,–4])
     sage: Q.signature_vector()
     (2, 2, 0)

  ::

     sage: Q = QuadraticForm(ZZ, 4, range(10)); Q
     Quadratic form in 4 variables over Integer Ring with coefficients:
     [ 0 1 2 3 ]
     [ * 4 5 6 ]
     [ * * 7 8 ]
     [ * * * 9 ]
     sage: Q.signature_vector()
     (3, 1, 0)

  """
    diag = self.rational_diagonal_form()
    p = 0
    n = 0
    z = 0
    for i in range(diag.dim()):
        if diag[i,i] > 0:
            p += 1
        elif diag[i,i] < 0:
            n += 1
        else:
            z += 1
```



```
    ## TO DO: Cache this result?

    return (p, n, z)



def signature(self):
    """
  Returns the signature of the quadratic form, defined as:

    number of positive eigenvalues – number of negative eigenvalues

  of the matrix of the quadratic form.

  INPUT:
    None

  OUTPUT:
    an integer

  EXAMPLES:
    sage: Q = DiagonalQuadraticForm(ZZ, [1,0,0,–4,3,11,3])
    sage: Q.signature()
    3

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,2,–3,–4])
    sage: Q.signature()
    0

  ::

    sage: Q = QuadraticForm(ZZ, 4, range(10)); Q
    Quadratic form in 4 variables over Integer Ring with coefficients:
    [ 0 1 2 3 ]
    [ * 4 5 6 ]
    [ * * 7 8 ]
    [ * * * 9 ]
    sage: Q.signature()
    2

  """
    (p, n, z) = self.signature_vector()
    return p - n




def hasse_invariant(self, p):
    """
  Computes the Hasse invariant at a prime 'p', as given on p55 of
  Cassels's book.  If Q is diagonal with coefficients 'a_i', then the
  (Cassels) Hasse invariant is given by

  .. math::

    c_p = \prod_{i < j} (a_i, a_j)_p

  where '(a,b)_p' is the Hilbert symbol at 'p'.  The underlying
  quadratic form must be non–degenerate over 'Q_p' for this to make
  sense.

  WARNING: This is different from the O'Meara Hasse invariant, which
  allows 'i <= j' in the product.  That is given by the method
  hasse_invariant__OMeara(p).

  NOTE: We should really rename this hasse_invariant__Cassels(), and
```

set hasse_invariant() as a front−end to it.

INPUT:
‘p‘ −− a prime number > 0

OUTPUT:
1 or −1

EXAMPLES::

```
sage: Q = QuadraticForm(ZZ, 2, [1,2,3])
sage: Q.rational_diagonal_form()
Quadratic form in 2 variables over Rational Field with coefficients:
[ 1 0 ]
[ * 2 ]
sage: [Q.hasse_invariant(p) for p in prime_range(20)]
[1, 1, 1, 1, 1, 1, 1, 1]
sage: [Q.hasse_invariant__OMeara(p) for p in prime_range(20)]
[1, 1, 1, 1, 1, 1, 1, 1]
```

::

```
sage: Q = DiagonalQuadraticForm(ZZ, [1,−1])
sage: [Q.hasse_invariant(p) for p in prime_range(20)]
[1, 1, 1, 1, 1, 1, 1, 1]
sage: [Q.hasse_invariant__OMeara(p) for p in prime_range(20)]
[1, 1, 1, 1, 1, 1, 1, 1]
```

::

```
sage: Q = DiagonalQuadraticForm(ZZ, [1,−1, −1])
sage: [Q.hasse_invariant(p) for p in prime_range(20)]
[−1, 1, 1, 1, 1, 1, 1, 1]
sage: [Q.hasse_invariant__OMeara(p) for p in prime_range(20)]
[1, 1, 1, 1, 1, 1, 1, 1]
```

```
    """
        ## TO DO: Need to deal with the case n=1 separately somewhere!

        Diag = self.rational_diagonal_form()

        ## DIAGNOSTIC
        #print "\n Q = " + str(self)
        #print "\n Q diagonalized at p = " + str(p) + " gives " + str(Diag)

        hasse_temp = 1
        n = Diag.dim()

        for j in range(n-1):
            for k in range(j+1, n):
                hasse_temp = hasse_temp * hilbert_symbol(Diag[j,j], Diag[k,k], p)

        return hasse_temp


def hasse_invariant__OMeara(self, p):
    """
```

Computes the O’Meara Hasse invariant at a prime ‘p‘, as given on p167 of O’Meara’s book. If Q is diagonal with coefficients ‘a_i‘, then the (Cassels) Hasse invariant is given by

.. math::

c_p = \prod_{i <= j} (a_i, a_j)_p

where ‘(a,b)_p‘ is the Hilbert symbol at ‘p‘.

WARNING: This is different from the (Cassels) Hasse invariant, which

only allows ‘i < j‘ in the product. That is given by the method hasse_invariant(p).

INPUT:
‘p‘ −− a prime number > 0

OUTPUT:
1 or −1

EXAMPLES::

```
sage: Q = QuadraticForm(ZZ, 2, [1,2,3])
sage: Q.rational_diagonal_form()
Quadratic form in 2 variables over Rational Field with coefficients:
[ 1 0 ]
[ * 2 ]
sage: [Q.hasse_invariant(p) for p in prime_range(20)]
[1, 1, 1, 1, 1, 1, 1, 1]
sage: [Q.hasse_invariant__OMeara(p) for p in prime_range(20)]
[1, 1, 1, 1, 1, 1, 1, 1]
```

::

```
sage: Q = DiagonalQuadraticForm(ZZ, [1,−1])
sage: [Q.hasse_invariant(p) for p in prime_range(20)]
[1, 1, 1, 1, 1, 1, 1, 1]
sage: [Q.hasse_invariant__OMeara(p) for p in prime_range(20)]
[1, 1, 1, 1, 1, 1, 1, 1]
```

::

```
sage: Q = DiagonalQuadraticForm(ZZ, [1,−1, −1])
sage: [Q.hasse_invariant(p) for p in prime_range(20)]
[−1, 1, 1, 1, 1, 1, 1, 1]
sage: [Q.hasse_invariant__OMeara(p) for p in prime_range(20)]
[1, 1, 1, 1, 1, 1, 1, 1]
```

```
    """
        ## TO DO: Need to deal with the case n=1 separately somewhere!

        Diag = self.rational_diagonal_form()

        ## DIAGNOSTIC
        #print "\n Q = " + str(self)
        #print "\n Q diagonalized at p = " + str(p) + " gives " + str(Diag)

        hasse_temp = 1
        n = Diag.dim()

        for j in range(n-1):
            for k in range(j, n):
                hasse_temp = hasse_temp * hilbert_symbol(Diag[j,j], Diag[k,k], p)

        return hasse_temp



def is_hyperbolic(self, p):
    """
```

Checks if the quadratic form is a sum of hyperbolic planes over the p−adic numbers Q_p.

REFERENCES:
This criteria follows from Cassels’s "Rational Quadratic Forms":
– local invariants for hyperbolic plane (Lemma 2.4, p58)
– direct sum formulas (Lemma 2.3 on p58)

```
    INPUT:
      'p' -- a prime number > 0

    OUTPUT:
      boolean

    EXAMPLES::

      sage: Q = DiagonalQuadraticForm(ZZ, [1,1])
      sage: Q.is_hyperbolic("infinity")
      False
      sage: Q.is_hyperbolic(2)
      False
      sage: Q.is_hyperbolic(3)
      False
      sage: Q.is_hyperbolic(5)     ## Here -1 is a square, so it's true.
      True
      sage: Q.is_hyperbolic(7)
      False
      sage: Q.is_hyperbolic(13)    ## Here -1 is a square, so it's true.
      True

    """
      ## False for odd-dim'l forms
      if self.dim() % 2 != 0:
          return False

      ## True for the zero form
      if self.dim == 0:
          return True

      ## Compare local invariants
      ## (Note: since the dimension is even, the extra powers of 2 in
      ##        self.det() := Det(2*Q) don't affect the answer!)
      m = ZZ(self.dim() / 2)
      if p == "infinity":
          return (self.signature() == 0)

      elif p == 2:
          return QQ(self.det() * (-1)**m).is_padic_square(p) and (self.hasse_invar
iant(p) == (-1)**m)     ## Actually, this -1 is the Hilbert symbol (-1,-1)_p

      else:
          return QQ(self.det() * (-1)**m).is_padic_square(p) and (self.hasse_invar
iant(p) == 1)



def is_anisotropic(self, p):
      """
    Checks if the quadratic form is anisotropic over the p-adic numbers 'Q_p'.

    INPUT:
      'p' -- a prime number > 0

    OUTPUT:
      boolean

    EXAMPLES::

      sage: Q = DiagonalQuadraticForm(ZZ, [1,1])
      sage: Q.is_anisotropic(2)
      True
      sage: Q.is_anisotropic(3)
      True
      sage: Q.is_anisotropic(5)
      False

    ::
```



```
      sage: Q = DiagonalQuadraticForm(ZZ, [1,-1])
      sage: Q.is_anisotropic(2)
      False
      sage: Q.is_anisotropic(3)
      False
      sage: Q.is_anisotropic(5)
      False

    ::

      sage: [DiagonalQuadraticForm(ZZ, [1, -least_quadratic_nonresidue(p)]).is_anisotropic(p)  for p in prime_range(3,
30)]
      [True, True, True, True, True, True, True, True, True]

    ::

      sage: [DiagonalQuadraticForm(ZZ, [1, -least_quadratic_nonresidue(p), p, -p*least_quadratic_nonresidue(p)]).is_
anisotropic(p)  for p in prime_range(3, 30)]
      [True, True, True, True, True, True, True, True, True]

    """
      n = self.dim()
      D = self.det()

      ## TO DO: Should check that p is prime

      if (n >= 5):
          return False;

      if (n == 4):
          return ( QQ(D).is_padic_square(p) and (self.hasse_invariant(p) == - hilb
ert_symbol(-1,-1,p)) )

      if (n == 3):
          return (self.hasse_invariant(p) != hilbert_symbol(-1, -D, p))

      if (n == 2):
          return (not QQ(-D).is_padic_square(p))

      if (n == 1):
          return (self[0,0] != 0)

      raise NotImplementedError, "Oops!  We haven't established a convention for 0-dim'l quadratic form
s... =("


def is_isotropic(self, p):
      """
    Checks if Q is isotropic over the p-adic numbers 'Q_p'.

    INPUT:
      'p' -- a prime number > 0

    OUTPUT:
      boolean

    EXAMPLES::

      sage: Q = DiagonalQuadraticForm(ZZ, [1,1])
      sage: Q.is_isotropic(2)
      False
      sage: Q.is_isotropic(3)
      False
      sage: Q.is_isotropic(5)
      True

    ::
```

```
    sage: Q = DiagonalQuadraticForm(ZZ, [1,-1])
    sage: Q.is_isotropic(2)
    True
    sage: Q.is_isotropic(3)
    True
    sage: Q.is_isotropic(5)
    True

  ::

    sage: [DiagonalQuadraticForm(ZZ, [1, -least_quadratic_nonresidue(p)]).is_isotropic(p)  for p in prime_range(3, 3
0)]
    [False, False, False, False, False, False, False, False, False]

  ::

    sage: [DiagonalQuadraticForm(ZZ, [1, -least_quadratic_nonresidue(p), p, -p*least_quadratic_nonresidue(p)]).is_
isotropic(p)  for p in prime_range(3, 30)]
    [False, False, False, False, False, False, False, False, False]

  """
    return not self.is_anisotropic(p)


def anisotropic_primes(self):
    """
  Returns a list with all of the anisotropic primes of the quadratic form.


  INPUT:
    None

  OUTPUT:
    Returns a list of prime numbers >0.

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
    sage: Q.anisotropic_primes()
    [2]

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1])
    sage: Q.anisotropic_primes()
    [2]

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1,1])
    sage: Q.anisotropic_primes()
    []

  """

    ## Look at all prime divisors of 2 * Det(Q) to find the anisotropic primes..
.
    possible_primes = prime_divisors(2 * self.det())
    AnisoPrimes = []

    ## DIAGNSOTIC
    #print " Possible anisotropic primes are: " + str(possible_primes)

    for p in possible_primes:
        if (self.is_anisotropic(p)):
            AnisoPrimes += [p]

    ## DIAGNSOTIC
    #print " leaving anisotropic_primes..."
```



```
    return AnisoPrimes



def compute_definiteness(self):
    """
  Computes whether the given quadratic form is positive-definite,
  negative-definite, indefinite, degenerate, or the zero form.

  This caches one of the following strings in self.__definiteness_string:
  "pos_def", "neg_def", "indef", "zero", "degenerate".  It is called
  from all routines like:

    is_positive_definite(), is_negative_definite(), is_indefinite(), etc.

  Note:  A degenerate form is considered neither definite nor indefinite.
  Note:  The zero-dim'l form is considered both positive definite and negative definite.

  INPUT:
    QuadraticForm

  OUTPUT:
    boolean

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1,1])
    sage: Q.compute_definiteness()
    sage: Q.is_positive_definite()
    True
    sage: Q.is_negative_definite()
    False
    sage: Q.is_indefinite()
    False
    sage: Q.is_definite()
    True

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [])
    sage: Q.compute_definiteness()
    sage: Q.is_positive_definite()
    True
    sage: Q.is_negative_definite()
    True
    sage: Q.is_indefinite()
    False
    sage: Q.is_definite()
    True

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,0,-1])
    sage: Q.compute_definiteness()
    sage: Q.is_positive_definite()
    False
    sage: Q.is_negative_definite()
    False
    sage: Q.is_indefinite()
    False
    sage: Q.is_definite()
    False

  """
    ## Sanity Check
    if not ((self.base_ring() == ZZ) or (self.base_ring() == QQ) or (self.base_r
ing() == RR)):
```

```
        raise NotImplementedError, "Oops!  We can only check definiteness over ZZ, QQ, and RR for
now."

    ## Some useful variables
    n = self.dim()
    M = self.matrix()


    ## Deal with the zero-diml form
    if n == 0:
        self.__definiteness_string = "zero"
        return


    sig_pos, sig_neg, sig_zer = self.signature_vector()

    ## Determine and cache the definiteness string
    if sig_zer > 0:
        self.__definiteness_string = "degenerate"
        return
    elif sig_neg == n:
        self.__definiteness_string = "neg_def"
        return
    elif sig_pos == n:
        self.__definiteness_string = "pos_def"
        return
    else:
        self.__definiteness_string = "indefinite"
        return



def compute_definiteness_string_by_determinants(self):
    """
  Compute the (positive) definiteness of a quadratic form by looking
  at the signs of all of its upper–left subdeterminants.  See also
  self.compute_definiteness() for more documentation.

  INPUT:
    None

  OUTPUT:
    string describing the definiteness

  EXAMPLES:
    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1,1])
    sage: Q.compute_definiteness_string_by_determinants()
    'pos_def'

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [])
    sage: Q.compute_definiteness_string_by_determinants()
    'zero'

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,0,–1])
    sage: Q.compute_definiteness_string_by_determinants()
    'degenerate'

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,–1])
    sage: Q.compute_definiteness_string_by_determinants()
    'indefinite'

  ::
```



```
    sage: Q = DiagonalQuadraticForm(ZZ, [–1,–1])
    sage: Q.compute_definiteness_string_by_determinants()
    'neg_def'

  """
    ## Sanity Check
    if not ((self.base_ring() == ZZ) or (self.base_ring() == QQ) or (self.base_r
ing() == RR)):
        raise NotImplementedError, "Oops!  We can only check definiteness over ZZ, QQ, and RR for
now."

    ## Some useful variables
    n = self.dim()
    M = self.matrix()


    ## Deal with the zero-diml form
    if n == 0:
        return "zero"



    ## Deal with degenerate forms
    if self.det() == 0:
        return "degenerate"


    ## Check the sign of the ratios of consecutive determinants of the upper tri
angular r x r submatrices
    first_coeff = self[0,0]
    for r in range(1,n+1):
        I = range(r)
        new_det = M.matrix_from_rows_and_columns(I, I).det()

        ## Check for a (non-degenerate) zero -- so it's indefinite
        if new_det == 0:
            return "indefinite"


        ## Check for a change of signs in the upper r x r submatrix -- so it's i
ndefinite
        if sgn(first_coeff)**r != sgn(new_det):
            return "indefinite"


    ## Here all ratios of determinants have the correct sign, so the matrix is (
pos or neg) definite.
    if first_coeff > 0:
        return "pos_def"
    else:
        return "neg_def"




def is_positive_definite(self):
    """
  Determines if the given quadratic form is positive–definite.

  Note:  A degenerate form is considered neither definite nor indefinite.
  Note:  The zero–dim'l form is considered both positive definite and negative definite.

  INPUT:
    None

  OUTPUT:
    boolean –– True or False
```

```
    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,3,5])
        sage: Q.is_positive_definite()
        True

    ::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,-3,5])
        sage: Q.is_positive_definite()
        False

    """
        ## Try to use the cached value
        try:
            def_str = self.__definiteness_string
        except:
            self.compute_definiteness()
            def_str = self.__definiteness_string

        ## Return the answer
        return (def_str == "pos_def") or (def_str == "zero")



def is_negative_definite(self):
        """
    Determines if the given quadratic form is negative-definite.

    Note:  A degenerate form is considered neither definite nor indefinite.
    Note:  The zero-dim'l form is considered both positive definite and negative definite.

    INPUT:
        None

    OUTPUT:
        boolean -- True or False

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [-1,-3,-5])
        sage: Q.is_negative_definite()
        True

    ::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,-3,5])
        sage: Q.is_negative_definite()
        False

    """
        ## Try to use the cached value
        try:
            def_str = self.__definiteness_string
        except:
            self.compute_definiteness()
            def_str = self.__definiteness_string

        ## Return the answer
        return (def_str == "neg_def") or (def_str == "zero")



def is_indefinite(self):
        """
    Determines if the given quadratic form is indefinite.

    Note:  A degenerate form is considered neither definite nor indefinite.
```



```
    Note:  The zero-dim'l form is not considered indefinite.

    INPUT:
        None

    OUTPUT:
        boolean -- True or False

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [-1,-3,-5])
        sage: Q.is_indefinite()
        False

    ::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,-3,5])
        sage: Q.is_indefinite()
        True

    """
        ## Try to use the cached value
        try:
            def_str = self.__definiteness_string
        except:
            self.compute_definiteness()
            def_str = self.__definiteness_string

        ## Return the answer
        return def_str == "indefinite"


def is_definite(self):
        """
    Determines if the given quadratic form is (positive or negative) definite.

    Note:  A degenerate form is considered neither definite nor indefinite.
    Note:  The zero-dim'l form is considered indefinite.

    INPUT:
        None

    OUTPUT:
        boolean -- True or False

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [-1,-3,-5])
        sage: Q.is_definite()
        True

    ::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,-3,5])
        sage: Q.is_definite()
        False

    """
        ## Try to use the cached value
        try:
            def_str = self.__definiteness_string
        except:
            self.compute_definiteness()
            def_str = self.__definiteness_string

        ## Return the answer
        return (def_str == "pos_def") or (def_str == "neg_def") or (def_str == "zero")
```

```
"""
Local Normal Form

"""

#*****************************************************************************
#       Copyright (C) 2007 William Stein and Jonathan Hanke
#
#  Distributed under the terms of the GNU General Public License (GPL)
#
#    This code is distributed in the hope that it will be useful,
#    but WITHOUT ANY WARRANTY; without even the implied warranty of
#    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
#    General Public License for more details.
#
#  The full text of the GPL is available at:
#
#                  http://www.gnu.org/licenses/
#*****************************************************************************

import copy
from sage.rings.infinity import Infinity
from sage.rings.integer_ring import IntegerRing, ZZ
from sage.rings.rational_field import QQ
from sage.rings.arith import GCD, valuation, is_prime


#from sage.misc.functional import ideal          ## TODO: This can probably be r
emoved!



def find_entry_with_minimal_scale_at_prime(self, p):
    """
    Finds the entry of the quadratic form with minimal scale at the
    prime p, preferring diagonal entries in case of a tie.  (I.e. If
    we write the quadratic form as a symmetric matrix M, then this
    entry M[i,j] has the minimal valuation at the prime p.)

    Note: This answer is independent of the kind of matrix (Gram or
    Hessian) associated to the form.

    INPUT:
        'p' -- a prime number > 0

    OUTPUT:
        a pair of integers >= 0

    EXAMPLES::

        sage: Q = QuadraticForm(ZZ, 2, [6, 2, 20]); Q
        Quadratic form in 2 variables over Integer Ring with coefficients:
        [ 6 2 ]
        [ * 20 ]
        sage: Q.find_entry_with_minimal_scale_at_prime(2)
        (0, 1)
        sage: Q.find_entry_with_minimal_scale_at_prime(3)
        (1, 1)
        sage: Q.find_entry_with_minimal_scale_at_prime(5)
        (0, 0)

    """
    n = self.dim()
    min_val = Infinity
    ij_index = None
    val_2 = valuation(2, p)
    for d in range(n):            ## d = difference j-i
        for e in range(n - d):    ## e is the length of the diagonal with value
d.
```



```
            ## Compute the valuation of the entry
            if d == 0:
                tmp_val = valuation(self[e, e+d], p)
            else:
                tmp_val = valuation(self[e, e+d], p) - val_2

            ## Check if it's any smaller than what we have
            if tmp_val < min_val:
                ij_index = (e,e+d)
                min_val = tmp_val

    ## Return the result
    return ij_index



def local_normal_form(self, p):
    """
    Returns the a locally integrally equivalent quadratic form over
    the p-adic integers Z_p which gives the Jordan decomposition.  The
    Jordan components are written as sums of blocks of size <= 2 and
    are arranged by increasing scale, and then by increasing norm.
    (This is equivalent to saying that we put the 1x1 blocks before
    the 2x2 blocks in each Jordan component.)

    INPUT:
        'p' -- a positive prime number.

    OUTPUT:
        a quadratic form over ZZ

    WARNING:  Currently this only works for quadratic forms defined over ZZ.

    EXAMPLES::

        sage: Q = QuadraticForm(ZZ, 2, [10,4,1])
        sage: Q.local_normal_form(5)
        Quadratic form in 2 variables over Integer Ring with coefficients:
        [ 1 0 ]
        [ * 6 ]

    ::

        sage: Q.local_normal_form(3)
        Quadratic form in 2 variables over Integer Ring with coefficients:
        [ 10 0 ]
        [ * 15 ]

        sage: Q.local_normal_form(2)
        Quadratic form in 2 variables over Integer Ring with coefficients:
        [ 1 0 ]
        [ * 6 ]

    """
    ## Sanity Checks
    if (self.base_ring() != IntegerRing()):
        raise NotImplementedError, "Oops!  This currently only works for quadratic forms defined ove
r IntegerRing(). =("
    if not ((p>=2) and is_prime(p)):
        raise TypeError, "Oops!  p is not a positive prime number. =("

    ## Some useful local variables
    Q = copy.deepcopy(self)
    Q.__init__(self.base_ring(), self.dim(), self.coefficients())

    ## Prepare the final form to return
    Q_Jordan = copy.deepcopy(self)
```

```
    Q_Jordan.__init__(self.base_ring(), 0)


    while Q.dim() > 0:
        n = Q.dim()

        ## Step 1: Find the minimally p-divisible matrix entry, preferring diago
nals
        ## ------------------------------------------------------------------------
----
        (min_i, min_j) = Q.find_entry_with_minimal_scale_at_prime(p)
        if min_i == min_j:
            min_val = valuation(2 * Q[min_i, min_j], p)
        else:
            min_val = valuation(Q[min_i, min_j], p)

        ## Error if we still haven't seen non-zero coefficients!
        if (min_val == Infinity):
            raise RuntimeError, "Oops!  The original matrix is degenerate. =("


        ## Step 2: Arrange for the upper leftmost entry to have minimal valuatio
n
        ## ------------------------------------------------------------------------
-
        if (min_i == min_j):
            block_size = 1
            Q.swap_variables(0, min_i, in_place = True)
        else:
            ## Work in the upper-left 2x2 block, and replace it by its 2-adic eq
uivalent form
            Q.swap_variables(0, min_i, in_place = True)
            Q.swap_variables(1, min_j, in_place = True)

            ## 1x1 => make upper left the smallest
            if (p != 2):
                block_size = 1;
                Q.add_symmetric(1, 0, 1, in_place = True)
            ## 2x2 => replace it with the appropriate 2x2 matrix
            else:
                block_size = 2

        ## DIAGNOSTIC
        #print "\n Finished Step 2 \n";
        #print "\n Q is: \n" + str(Q)  + "\n";
        #print "  p is: " + str(p)
        #print "  min_val is: " + str( min_val)
        #print "  block_size is: " + str(block_size)
        #print "\n Starting Step 3 \n"

        ## Step 3: Clear out the remaining entries
        ##  ---------------------------------------
        min_scale = p ** min_val                                ## This is the mini
mal valuation of the Hessian matrix entries.

        ##DIAGNOSTIC
        #print "Starting Step 3:"
        #print "----------------"
        #print "  min_scale is: " + str(min_scale)


        ## Perform cancellation over Z by ensuring divisibility
        if (block_size == 1):
            a = 2 * Q[0,0]
            for j in range(block_size, n):
                b = Q[0, j]
                g = GCD(a, b)

                ## DIAGNSOTIC
```



```
                #print "Cancelling from a 1x1 block:"
                #print "----------------------------"
                #print "  Cancelling entry with index (" + str(upper_left) + ",
" + str(j) + ")"
                #print "  entry = " + str(b)
                #print "  gcd = " + str(g)
                #print "  a = " + str(a)
                #print "  b = " + str(b)
                #print "  a/g = " + str(a/g) + "   (used for stretching)"
                #print "  -b/g = " + str(-b/g) + "   (used for cancelling)"

                ## Sanity Check:  a/g is a p-unit
                if valuation (g, p) != valuation(a, p):
                    raise RuntimeError, "Oops!  We have a problem with our rescaling not preservi
ng p-integrality!"

                Q.multiply_variable(ZZ(a/g), j, in_place = True)   ## Ensures th
at the new b entry is divisible by a
                Q.add_symmetric(ZZ(-b/g), j, 0, in_place = True)  ## Performs th
e cancellation


        elif (block_size == 2):
            a1 = 2 * Q[0,0]
            a2 = Q[0, 1]
            b1 = Q[1, 0]      ## This is the same as a2
            b2 = 2 * Q[1, 1]

            big_det = (a1*b2 - a2*b1)
            small_det = big_det / (min_scale * min_scale)

            ## Cancels out the rows/columns of the 2x2 block
            for j in range(block_size, n):
                a = Q[0, j]
                b = Q[1, j]

                ## Ensures an integral result (scale jth row/column by big_det)
                Q.multiply_variable(big_det, j, in_place = True)

                ## Performs the cancellation (by producing -big_det * jth row/co
lumn)
                Q.add_symmetric(ZZ(-(a*b2 - b*a2)), j, 0, in_place = True)
                Q.add_symmetric(ZZ(-(-a*b1 + b*a1)), j, 1, in_place = True)

                ## Now remove the extra factor (non p-unit factor) in big_det we
 introduced above
                Q.divide_variable(ZZ(min_scale * min_scale), j, in_place = True)


            ## DIAGNOSTIC
            #print "Cancelling out a 2x2 block:"
            #print "---------------------------"
            #print "  a1 = " + str(a1)
            #print "  a2 = " + str(a2)
            #print "  b1 = " + str(b1)
            #print "  b2 = " + str(b2)
            #print "  big_det = " + str(big_det)
            #print "  min_scale = " + str(min_scale)
            #print "  small_det = " + str(small_det)
            #print "  Q = \n", Q

            ## Uses Cassels's proof to replace the remaining 2 x 2 block
            if (((1 + small_det) % 8) == 0):
                Q[0, 0] = 0
                Q[1, 1] = 0
                Q[0, 1] = min_scale
            elif (((5 + small_det) % 8) == 0):
                Q[0, 0] = min_scale
                Q[1, 1] = min_scale
```

```
                Q[0, 1] = min_scale
            else:
                raise RuntimeError, "Error in LocalNormal: Impossible behavior for a 2x2 block! \n"


        ## Check that the cancellation worked, extract the upper-left block, and
 trim Q to handle the next block.
        for i in range(block_size):
            for j in range(block_size, n):
                if Q[i,j] != 0:
                    raise RuntimeError, "Oops!  The cancellation didn't work properly at entry ("
+ str(i) + "," + str(j) + ")."
        Q_Jordan = Q_Jordan + Q.extract_variables(range(block_size))
        Q = Q.extract_variables(range(block_size, n))

    return Q_Jordan



def jordan_blocks_by_scale_and_unimodular(self, p, safe_flag=True):
    """
    Returns a list of pairs '(s_i, L_i)' where 'L_i' is a maximal
    'p^{s_i}'-unimodular Jordan component which is further decomposed into
    block diagonals of block size '\le 2'. For each 'L_i' the 2x2 blocks are
    listed after the 1x1 blocks (which follows from the convention of the
    :meth:'local_normal_form' method).

    ..note ::

        The decomposition of each 'L_i' into smaller block is not unique!

    The ''safe_flag'' argument allows us to select whether we want a copy of
    the output, or the original output.  By default ''safe_flag = True'', so we
    return a copy of the cached information.  If this is set to ''False'', then
    the routine is much faster but the return values are vulnerable to being
    corrupted by the user.

    INPUT:

    - 'p' -- a prime number > 0.

    OUTPUT:

    A list of pairs '(s_i, L_i)' where:

    - 's_i' is an integer,
    - 'L_i' is a block-diagonal unimodular quadratic form over '\ZZ_p'.

    .. note::

        These forms 'L_i' are defined over the 'p'-adic integers, but by a
        matrix over '\ZZ' (or '\QQ'?).

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,9,5,7])
        sage: Q.jordan_blocks_by_scale_and_unimodular(3)
        [(0,
        Quadratic form in 3 variables over Integer Ring with coefficients:
        [ 1 0 0 ]
        [ * 5 0 ]
        [ * * 7 ]
        ),
        (2,
        Quadratic form in 1 variables over Integer Ring with coefficients:
        [ 1 ]
        )]
```



```
    ::

        sage: Q2 = QuadraticForm(ZZ, 2, [1,1,1])
        sage: Q2.jordan_blocks_by_scale_and_unimodular(2)
        [(-1,
         Quadratic form in 2 variables over Integer Ring with coefficients:
        [ 2 2 ]
        [ * 2 ]
        )]
        sage: Q = Q2 + Q2.scale_by_factor(2)
        sage: Q.jordan_blocks_by_scale_and_unimodular(2)
        [(-1,
         Quadratic form in 2 variables over Integer Ring with coefficients:
        [ 2 2 ]
        [ * 2 ]
        ),
         (0,
         Quadratic form in 2 variables over Integer Ring with coefficients:
        [ 2 2 ]
        [ * 2 ]
        )]

    """
        ## Try to use the cached result
        try:
            if safe_flag:
                return copy.deepcopy(self.__jordan_blocks_by_scale_and_unimodular_di
ct[p])
            else:
                return self.__jordan_blocks_by_scale_and_unimodular_dict[p]
        except:
            ## Initialize the global dictionary if it doesn't exist
            if not hasattr(self, '__jordan_blocks_by_scale_and_unimodular_dict'):
                self.__jordan_blocks_by_scale_and_unimodular_dict = {}


        ## Deal with zero dim'l forms
        if self.dim() == 0:
            return []


        ## Find the Local Normal form of Q at p
        Q1 = self.local_normal_form(p)


        ## Parse this into Jordan Blocks
        n = Q1.dim()
        tmp_Jordan_list = []
        i = 0
        start_ind = 0
        if (n >= 2) and (Q1[0,1] != 0):
            start_scale = valuation(Q1[0,1], p) - 1
        else:
            start_scale = valuation(Q1[0,0], p)

        while (i < n):

            ## Determine the size of the current block
            if (i == n-1) or (Q1[i,i+1] == 0):
                block_size = 1
            else:
                block_size = 2

            ## Determine the valuation of the current block
            if block_size == 1:
                block_scale = valuation(Q1[i,i], p)
            else:
                block_scale = valuation(Q1[i,i+1], p) - 1
```

```
        ## Process the previous block if the valuation increased
        if block_scale > start_scale:
            tmp_Jordan_list += [(start_scale, Q1.extract_variables(range(start_i
nd, i)).scale_by_factor(ZZ(1) / (QQ(p)**(start_scale))))]
            start_ind = i
            start_scale = block_scale

        ## Increment the index
        i += block_size

    ## Add the last block
    tmp_Jordan_list += [(start_scale, Q1.extract_variables(range(start_ind, n)).
scale_by_factor(ZZ(1) / QQ(p)**(start_scale)))]


    ## Cache the result
    self.__jordan_blocks_by_scale_and_unimodular_dict[p] = tmp_Jordan_list

    ## Return the result
    return tmp_Jordan_list




def jordan_blocks_in_unimodular_list_by_scale_power(self, p):
    """
    Returns a list of Jordan components, whose component at index i
    should be scaled by the factor p^i.

    This is only defined for integer-valued quadratic forms
    (i.e. forms with base_ring ZZ), and the indexing only works
    correctly for p=2 when the form has an integer Gram matrix.

    INPUT:
        self -- a quadratic form over ZZ, which has integer Gram matrix if p == 2
        `p` -- a prime number > 0

    OUTPUT:
        a list of p-unimodular quadratic forms

    EXAMPLES::

        sage: Q = QuadraticForm(ZZ, 3, [2, -2, 0, 3, -5, 4])
        sage: Q.jordan_blocks_in_unimodular_list_by_scale_power(2)
        Traceback (most recent call last):
        ...
        TypeError: Oops!  The given quadratic form has a Jordan component with a negative scale exponent!
        This routine requires an integer-matrix quadratic form for the output indexing to work properly!

        sage: Q.scale_by_factor(2).jordan_blocks_in_unimodular_list_by_scale_power(2)
        [Quadratic form in 2 variables over Integer Ring with coefficients:
        [ 0 2 ]
        [ * 0 ]
        ,
        Quadratic form in 0 variables over Integer Ring with coefficients:
        ,
        Quadratic form in 1 variables over Integer Ring with coefficients:
        [ 345 ]
        ]

        sage: Q.jordan_blocks_in_unimodular_list_by_scale_power(3)
        [Quadratic form in 2 variables over Integer Ring with coefficients:
        [ 2 0 ]
        [ * 10 ]
        ,
        Quadratic form in 1 variables over Integer Ring with coefficients:
        [ 2 ]
        ]
```



```
    """
    ## Sanity Check
    if self.base_ring() != ZZ:
        raise TypeError, "Oops!  This method only makes sense for integer-valued quadratic forms (i.e. d
efined over ZZ)."

    ## Deal with zero dim'l forms
    if self.dim() == 0:
        return []

    ## Find the Jordan Decomposition
    list_of_jordan_pairs = self.jordan_blocks_by_scale_and_unimodular(p)
    scale_list =  [P[0] for P in list_of_jordan_pairs]
    s_max = max(scale_list)
    if min(scale_list) < 0:
        raise TypeError, "Oops!  The given quadratic form has a Jordan component with a negative scale e
xponent!\n" \
        + "This routine requires an integer-matrix quadratic form for the output indexing to work properly!"

    ## Make the new list of unimodular Jordan components
    zero_form = copy.deepcopy(self)
    zero_form.__init__(ZZ, 0)
    list_by_scale = [zero_form  for _ in range(s_max+1)]
    for P in  list_of_jordan_pairs:
        list_by_scale[P[0]] = P[1]

    ## Return the new list
    return list_by_scale
```

```
"""
Local Representation Conditions
"""
#############################################################################
## Class for keeping track of the local conditions for representability ##
## of numbers by a quadratic form over ZZ (and eventually QQ also).     ##
#############################################################################

from copy import deepcopy

from sage.rings.integer_ring import IntegerRing, ZZ
from sage.rings.arith import prime_divisors, valuation, is_square
from sage.quadratic_forms.extras import least_quadratic_nonresidue
from sage.rings.infinity import infinity
from sage.misc.functional import numerator, denominator
from sage.rings.rational_field import QQ



class QuadraticFormLocalRepresentationConditions():
    """
    Creates a class for dealing with the local conditions of a
    quadratic form, and checking local representability of numbers.

    EXAMPLES::

        sage: Q4 = DiagonalQuadraticForm(ZZ, [1,1,1,1])
        sage: Q4.local_representation_conditions()
        This form represents the p-adic integers Z_p for all primes p except
        [].  For these and the reals, we have:
            Reals:   [0, +Infinity]
        sage: Q4.is_locally_represented_number(1)
        True
        sage: Q4.is_locally_universal_at_all_primes()
        True
        sage: Q4.is_locally_universal_at_all_places()
        False
        sage: L = [m  for m in range(-5, 100)  if Q4.is_locally_represented_number(m)]
        sage: L == range(100)
        True

    ::

        sage: Q3 = DiagonalQuadraticForm(ZZ, [1,1,1])
        sage: Q3.local_representation_conditions()
        This form represents the p-adic integers Z_p for all primes p except
        [2].  For these and the reals, we have:
            Reals:   [0, +Infinity]
            p = 2:   [0, 0, 0, +Infinity, 0, 0, 0, 0]
        sage: E = [m  for m in range(100)  if not Q3.is_locally_represented_number(m)]
        sage: E1 = [m  for m in range(1,100)  if m / 2**(2*floor(valuation(m,2)/2)) % 8 == 7]
        sage: E == E1
        True
        sage: E
        [7, 15, 23, 28, 31, 39, 47, 55, 60, 63, 71, 79, 87, 92, 95]

    ::

        sage: Q2 = DiagonalQuadraticForm(ZZ, [1,1])
        sage: Q2.local_representation_conditions()
        This 2-dimensional form represents the p-adic integers of even
        valuation for all primes p except [2].
        For these and the reals, we have:
            Reals:   [0, +Infinity]
            p = 2:   [0, +Infinity, 0, +Infinity, 0, +Infinity, 0, +Infinity]
        sage: Q2.is_locally_universal_at_all_places()
        False
        sage: Q2.is_locally_universal_at_all_primes()
        False
```



```
        sage: L = [m  for m in range(-5, 25)  if Q2.is_locally_represented_number(m)]
        sage: L1 = [0] + [m  for m in range(1,25)  \
             if len([p  for p in prime_factors(squarefree_part(ZZ(m)))  if (p % 4) == 3]) % 2 == 0]
        sage: L == L1
        True
        sage: L
        [0, 1, 2, 4, 5, 8, 9, 10, 13, 16, 17, 18, 20, 21]

    ::

        sage: Q1 = DiagonalQuadraticForm(ZZ, [1])
        sage: Q1.local_representation_conditions()
        This 1-dimensional form only represents square multiples of 1.
        sage: L = [m  for m in range(100)  if Q1.is_locally_represented_number(m)]
        sage: L
        [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

    ::

        sage: Q0 = DiagonalQuadraticForm(ZZ, [])
        sage: Q0.local_representation_conditions()
        This 0-dimensional form only represents zero.
        sage: L = [m  for m in range(100)  if Q0.is_locally_represented_number(m)]
        sage: L
        [0]

    """

    def __init__(self, Q):
        """
        Takes a QuadraticForm and computes its local conditions (if
        they don't already exist).  The recompute_flag overrides the
        previously computed conditions if they exist, and stores the
        new conditions.

        INPUT:
            Q -- Quadratic form over ZZ

        OUTPUT:
            a  QuadraticFormLocalRepresentationConditions object

        EXAMPLES::

            sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1])
            sage: from sage.quadratic_forms.quadratic_form__local_representation_conditions import QuadraticFormLocal
RepresentationConditions
            sage: QuadraticFormLocalRepresentationConditions(Q)
            This form represents the p-adic integers Z_p for all primes p except
            [].  For these and the reals, we have:
                Reals:   [0, +Infinity]

        """

            ## Check that the form Q is integer-valued (we can relax this later)
            if Q.base_ring() != ZZ:
                raise TypeError, "We require that the quadratic form be defined over ZZ (integer-values) fo
r now."


            ## Basic structure initialization
            self.local_repn_array = []     ## List of all local conditions
            self.dim = Q.dim()       ## We allow this to be any non-negative integer
.
            self.exceptional_primes = [infinity]

            ## Deal with the special cases of 0 and 1-dimensional forms
            if self.dim == 0:
                self.coeff = None
                return
```

```
        elif self.dim == 1:
            self.coeff = Q[0,0]
            return
        else:
            self.coeff = None


        ## Compute the local conditions at the real numbers (i.e. "p = infinity"
)
        ## ---------------------------------------------------------------------
-
        M = Q.matrix()
        E = M.eigenspaces()
        M_eigenvalues = [E[i][0]  for i in range(len(E))]

        pos_flag = infinity
        neg_flag = infinity

        for e in M_eigenvalues:
            if e > 0:
                pos_flag = 0
            elif e < 0:
                neg_flag = 0

        real_vec = [infinity, pos_flag, neg_flag, None, None, None, None, None,
None]
        self.local_repn_array.append(real_vec)


        ## Compute the local conditions for representability:
        ## --------------------------------------------------
        N = Q.level()
        level_primes = prime_divisors(N)
        prime_repn_modulus_list = [p**(valuation(4*N, p) + 2)  for p in level_pr
imes]

        ## Make a table of local normal forms for each p | N
        local_normal_forms = [Q.local_normal_form(p)  for p in level_primes]

        ## Check local representability conditions for each prime
        for i in range(len(level_primes)):
            p = level_primes[i]
            tmp_local_repn_vec = [p, None, None, None, None, None, None, None, N
one]
            sqclass = self.squareclass_vector(p)

            ## Check the representability in each Z_p squareclass
            for j in range(len(sqclass)):
                m = sqclass[j]
                k = 0
                repn_flag = False

                while ((repn_flag == False) and (m < 4 * N * p * p)):
                    if (local_normal_forms[i].local_density(p, m) > 0):
                        tmp_local_repn_vec[j+1] = k
                        repn_flag = True
                    k = k + 1
                    m = m * p * p

                ## If we're not represented, write "infinity" to signify
                ## that this squareclass is fully obstructed
                if (repn_flag == False):
                    tmp_local_repn_vec[j+1] = infinity

            ## Test if the conditions at p give exactly Z_p when dim >=3, or
            ## if we represent the elements of even valuation >= 2 when dim = 2.
            omit_flag = True
            if self.dim >= 2:
                ## Check that all entries are zero or 'None'
```



```
                for x in tmp_local_repn_vec[1:]:
                    if not ((x == 0) or (x == None)):
                        omit_flag = False

            ## Add the results for this prime if there is a congruence obstructi
on
            if omit_flag == False:
                self.local_repn_array.append(tmp_local_repn_vec)
                self.exceptional_primes.append(p)


    def __repr__(self):
        """
    Print the local conditions.

    INPUT:
      none

    OUTPUT:
      string

    TO DO:  Improve the output for the real numbers, and special output for locally universality.
    Also give names to the squareclasses, so it's clear what the output means! =)

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,1])
        sage: from sage.quadratic_forms.quadratic_form__local_representation_conditions import QuadraticFormLocal
RepresentationConditions
        sage: C = QuadraticFormLocalRepresentationConditions(Q)
        sage: C.__repr__()
        'This 2-dimensional form represents the p-adic integers of even\nvaluation for all primes p except [2].\nFor the
se and the reals, we have:\n    Reals:  [0, +Infinity]\n    p = 2:  [0, +Infinity, 0, +Infinity, 0, +Infinity, 0, +Infinity]\n'

    """
        if self.dim == 0:
            out_str = "This 0-dimensional form only represents zero."
        elif self.dim == 1:
            out_str = "This 1-dimensional form only represents square multiples of " + str(self.co
eff) + "."
        elif self.dim == 2:
            out_str = "This 2-dimensional form represents the p-adic integers of even\n"
            out_str += "valuation for all primes p except " + str(self.exceptional_primes[1
:]) + ".\n"
            out_str += "For these and the reals, we have:\n"
        else:
            out_str = "This form represents the p-adic integers Z_p for all primes p except \n"
            out_str += str(self.exceptional_primes[1:]) + ".  For these and the reals, we h
ave:\n"

        for v in self.local_repn_array:
            if v[0] == infinity:
                out_str += "   " + "Reals:  " + str(v[1:3]) + "\n"
            elif v[0] == 2:
                out_str += "   " + "p = 2:  " + str(v[1:]) + "\n"
            else:
                out_str += "   " + "p = " + str(v[0]) + ":  " + str(v[1:5]) + "\n"

        return out_str



    def __eq__(self, right):
        """
    Determines if two sets of local conditions are equal.

    INPUT:
      right -- a QuadraticFormLocalRepresentationConditions object
```

```
    OUTPUT:
      boolean

    EXAMPLES::

        sage: Q1 = DiagonalQuadraticForm(ZZ, [1,1])
        sage: Q2 = DiagonalQuadraticForm(ZZ, [1,1,1])
        sage: Q3 = DiagonalQuadraticForm(ZZ, [1,3,5,7])
        sage: Q4 = DiagonalQuadraticForm(ZZ, [1,1,1,1])

        sage: Q1.local_representation_conditions() == Q2.local_representation_conditions()
        False
        sage: Q1.local_representation_conditions() == Q3.local_representation_conditions()
        False
        sage: Q1.local_representation_conditions() == Q4.local_representation_conditions()
        False
        sage: Q2.local_representation_conditions() == Q3.local_representation_conditions()
        False
        sage: Q3.local_representation_conditions() == Q4.local_representation_conditions()
        True
    """
        if not isinstance(right, QuadraticFormLocalRepresentationConditions):
            return False

        ## Check the dimensions agree when they affect the kind of representatio
n conditions.
        if ((self.dim <= 2) or (right.dim <= 2)) and self.dim != right.dim:
            return False

        ## Check equality by dimension
        if self.dim == 0:
            return True
        elif self.dim == 1:
            return self.coeff == right.coeff     ## Compare coefficients in dime
nsion 1 (since ZZ has only one unit square)
        else:
            return (self.exceptional_primes == right.exceptional_primes) \
                and (self.local_repn_array == right.local_repn_array)


    def squareclass_vector(self, p):
        """
    Gives a vector of integers which are normalized
    representatives for the 'p'-adic rational squareclasses
    (or the real squareclasses) at the prime 'p'.

    INPUT:
      'p' -- a positive prime number or "infinity".

    OUTPUT:
      a list of integers

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
        sage: from sage.quadratic_forms.quadratic_form__local_representation_conditions import QuadraticFormLocal
RepresentationConditions
        sage: C = QuadraticFormLocalRepresentationConditions(Q)
        sage: C.squareclass_vector(5)
        [1, 2, 5, 10]

    """
        if p == infinity:
            return [1, -1]
        elif p == 2:
            return [1, 3, 5, 7, 2, 6, 10, 14]
        else:
            r = least_quadratic_nonresidue(p)
            return [1, r, p, p*r]
```



```
    def local_conditions_vector_for_prime(self, p):
        """
    Returns a local representation vector for the (possibly infinite) prime 'p'.

    INPUT:
      'p' -- a positive prime number.  (Is 'infinity' allowed here?)

    OUTPUT:
      a list of integers

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
        sage: from sage.quadratic_forms.quadratic_form__local_representation_conditions import QuadraticFormLocal
RepresentationConditions
        sage: C = QuadraticFormLocalRepresentationConditions(Q)
        sage: C.local_conditions_vector_for_prime(2)
        [2, 0, 0, 0, +Infinity, 0, 0, 0, 0]
        sage: C.local_conditions_vector_for_prime(3)
        [3, 0, 0, 0, 0, None, None, None, None]

    """
        ## Check if p is non-generic
        if p in self.exceptional_primes:
            return deepcopy(self.local_repn_array[self.exceptional_primes.index(
p)])

        ## Otherwise, generate a vector at this (finite) prime
        if self.dim >= 3:
            if p == 2:
                return [2, 0, 0, 0, 0, 0, 0, 0, 0]
            else:
                return [p, 0, 0, 0, 0, None, None, None, None]

        elif self.dim == 2:
            if p == 2:
                return [2, 0, 0, 0, 0, infinity, infinity, infinity, infinity]
            else:
                return [p, 0, 0, infinity, infinity, None, None, None, None]

        elif self.dim == 1:
            v = [p, None, None, None, None, None, None, None, None]
            sqclass = self.squareclass_vector(p)

            for i in range(len(sq_class)):
                if QQ(self.coeff / sqclass[i]).is_padic_square(p):    ## Note:Th
is should happen only once!
                    nu = valuation(self.coeff / sqclass[i], p) / 2
                else:
                    v[i+1] = infinity

        elif self.dim == 0:
            if p == 2:
                return [2, infinity, infinity, infinity, infinity, infinity, inf
inity, infinity, infinity]
            else:
                return [p, infinity, infinity, infinity, infinity, None, None, N
one, None]

        raise RuntimeError, "Error... The dimension stored should be a non-negative integer!"


    def is_universal_at_prime(self, p):
        """
    Determines if the (integer-valued/rational) quadratic form represents all of 'Z_p'.
```

INPUT:
‘p‘ –– a positive prime number or `"infinity"`.

OUTPUT:
boolean

EXAMPLES::

```
sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
sage: from sage.quadratic_forms.quadratic_form__local_representation_conditions import QuadraticFormLocal
RepresentationConditions
sage: C = QuadraticFormLocalRepresentationConditions(Q)
sage: C.is_universal_at_prime(2)
False
sage: C.is_universal_at_prime(3)
True
sage: C.is_universal_at_prime(infinity)
False
```

```
"""
        ## Check if the prime behaves generically for n >= 3.
        if (self.dim >= 3) and not (p in self.exceptional_primes):
            return True

        ## Check if the prime behaves generically for n <= 2.
        if (self.dim <= 2) and not (p in self.exceptional_primes):
            return False

        ## Check if the prime is "infinity" (for the reals)
        if p == infinity:
            v = self.local_repn_array[0]
            if p != v[0]:
                raise RuntimeError, "Error... The first vector should be for the real numbers!"
            return (v[1:3] == [0,0])      ## True iff the form is indefinite

        ## Check non-generic "finite" primes
        v = self.local_conditions_vector_for_prime(p)
        Zp_univ_flag = True
        for nu in v[1:]:
            if (nu != None) and ((nu != 0) or (nu == infinity)):
                Zp_univ_flag = False
        return Zp_univ_flag


    def is_universal_at_all_finite_primes(self):
        """
```

Determines if the quadratic form represents ‘Z_p‘ for all finite/non–archimedian primes.

INPUT:
none

OUTPUT:
boolean

EXAMPLES::

```
sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
sage: from sage.quadratic_forms.quadratic_form__local_representation_conditions import QuadraticFormLocal
RepresentationConditions
sage: C = QuadraticFormLocalRepresentationConditions(Q)
sage: C.is_universal_at_all_finite_primes()
False
```

::

```
sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1])
sage: from sage.quadratic_forms.quadratic_form__local_representation_conditions import QuadraticFormLocal
RepresentationConditions
```



```
sage: C = QuadraticFormLocalRepresentationConditions(Q)
sage: C.is_universal_at_all_finite_primes()
True
```

```
    """
        ## Check if dim <= 2.
        if self.dim <= 2:
            return False

        ## Check that all non-generic finite primes are universal
        univ_flag = True
        for p in self.exceptional_primes[1:]:      ## Omit p = "infinity" here
            univ_flag = univ_flag and self.is_universal_at_prime(p)
        return univ_flag


    def is_universal_at_all_places(self):
        """
```

Determines if the quadratic form represents ‘Z_p‘ for all finite/non–archimedian primes, and represents all real numbers.

INPUT:
none

OUTPUT:
boolean

EXAMPLES::

```
sage: from sage.quadratic_forms.quadratic_form__local_representation_conditions import QuadraticFormLocal
RepresentationConditions
```

::

```
sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
sage: C = QuadraticFormLocalRepresentationConditions(Q)
sage: C.is_universal_at_all_places()
False
```

::

```
sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1])
sage: C = QuadraticFormLocalRepresentationConditions(Q)
sage: C.is_universal_at_all_places()
False
```

::

```
sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1,–1])
sage: C = QuadraticFormLocalRepresentationConditions(Q)     # long time (8.5 s)
sage: C.is_universal_at_all_places()                        # long time
True
```

```
    """
        ## Check if dim <= 2.
        if self.dim <= 2:
            return False

        ## Check that all non-generic finite primes are universal
        for p in self.exceptional_primes:
            if not self.is_universal_at_prime(p):
                return False
        return True



    def is_locally_represented_at_place(self, m, p):
        """
```

Determines if the rational number m is locally represented by the

quadratic form at the (possibly infinite) prime ‘p‘.

INPUT:

‘m‘ –– an integer

‘p‘ –– a positive prime number or `"infinity"`.

OUTPUT:
boolean

EXAMPLES::

sage: from sage.quadratic_forms.quadratic_form__local_representation_conditions import QuadraticFormLocal
RepresentationConditions

sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
sage: C = QuadraticFormLocalRepresentationConditions(Q)
sage: C.is_locally_represented_at_place(7, 2)
False
sage: C.is_locally_represented_at_place(1, 3)
True
sage: C.is_locally_represented_at_place(–1, infinity)
False
sage: C.is_locally_represented_at_place(1, infinity)
True
sage: C.is_locally_represented_at_place(0, infinity)
True

```
    """
        ## Sanity Check
        if not m in QQ:
            raise TypeError, "Oops!  m = " + str(m) +   " is not a rational number!"

       ## Representing zero
        if m == 0:
            return True

        ## 0-dim'l forms
        if self.dim == 0:    ## Here m != 0
            return False

        ## 1-dim'l forms
        if self.dim == 1:
            m1 = QQ(m) / self.coeff
            if p == infinity:
                return (m1 > 0)
            else:
                return (valuation(m1, p) >= 0) and m1.is_padic_square(p)

        ## >= 2-dim'l forms
        local_vec = self.local_conditions_vector_for_prime(p)

        ## Check the real place
        if p == infinity:
            if m > 0:
                return local_vec[1] == 0
            elif m < 0:
                return local_vec[2] == 0
            else:   ## m == 0
                return True

        ## Check at a finite place
        sqclass = self.squareclass_vector(p)
        for s in sqclass:
            #print "m =", m, "   s =", s, "   m/s =", (QQ(m)/s)
            if (QQ(m)/s).is_padic_square(p):
                nu = valuation(m/s, p)
                return local_vec[sqclass.index(s) + 1] <= (nu / 2)
```



```
    def is_locally_represented(self, m):
        """
```

Determines if the rational number ‘m‘ is locally represented by
the quadratic form (allowing vectors with coefficients in ‘Z_p‘ at all
places).

INPUT:
‘m‘ –– an integer

OUTPUT:
boolean

EXAMPLES::

sage: from sage.quadratic_forms.quadratic_form__local_representation_conditions import QuadraticFormLocal
RepresentationConditions

sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
sage: C = QuadraticFormLocalRepresentationConditions(Q)
sage: C.is_locally_represented(7)
False
sage: C.is_locally_represented(28)
False
sage: C.is_locally_represented(11)
True
sage: C.is_locally_represented(QQ(1)/QQ(2))
False

```
    """
        ## Representing zero
        if m == 0:
            return True

        ## 0-dim'l forms
        if self.dim == 0:    ## Here m != 0
            return False

        ## 1-dim'l forms
        if self.dim == 1:
            m1 = m / self.coeff
            return (m1 in ZZ) and is_square(m1)

        ## Check the generic primes (when n = 2 or n >= 3)
        m_primes = prime_divisors(numerator(m) * denominator(m))
        for p in m_primes:
            if not p in self.exceptional_primes:
               val = valuation(m, p)
               if (val < 0):
                   return False

        ## Check the non-generic primes (when n = 2 or n >= 3)
        for p in self.exceptional_primes:
            if not self.is_locally_represented_at_place(m, p):
                return False

        ## If we got here, we're locally represented!
        return True



## --------------------  End of QuadraticFormLocalRepresentationConditions Class
   ----------------------


def local_representation_conditions(self, recompute_flag=False, silent_flag=Fals
```

```
e):
        """
    WARNING: THIS ONLY WORKS CORRECTLY FOR FORMS IN >=3 VARIABLES,
       WHICH ARE LOCALLY UNIVERSAL AT ALMOST ALL PRIMES!

    This class finds the local conditions for a number to be integrally
    represented by an integer-valued quadratic form.  These conditions
    are stored in "self.__local_representability_conditions" and
    consist of a list of 9 element vectors, with one for each prime
    with a local obstruction (though only the first 5 are meaningful
    unless 'p=2' ).  The first element is always the prime 'p' where the
    local obstruction occurs, and the next 8 (or 4) entries represent
    square-classes in the 'p'-adic integers 'Z_p', and are labeled by the
    'Q_p' square-classes 't*(Q_p)^2' with 't' given as follows:

       'p > 2'  ==>  [ *  1  u  p  u p  *  *  *  * ]

       'p = 2'  ==>  [ *  1  3  5  7  2  6  10  14 ]

    The integer appearing in each place tells us how 'p'-divisible a
    number needs to be in that square-class in order to be locally
    represented by Q.  A negative number indicates that the entire 'Q_p'
    square-class is not represented, while a positive number 'x' indicates
    that 't*p^{(2*x)} (Z_p)^2' is locally represented but 't*p^{(2*(x-1))}'
    '(Z_p)^2' is not.

    As an example, the vector

       [2  3  0  0  0  0  2  0  infinity]

    tells us that all positive integers are locally represented at p=2
    except those of the forms:

       '2^6 * u * r^2' with  'u = 1 (mod 8)'

       '2^5 * u * r^2' with 'u = 3 (mod 8)'

       '2 * u * r^2' with 'u = 7 (mod 8)'

    At the real numbers, the vector which looks like

       [infinity, 0, infinity, None, None, None, None, None, None]

    means that Q is negative definite (i.e. the 0 tells us all
    positive reals are represented).  The real vector always appears,
    and is listed before the other ones.

    INPUT:
       none

    OUTPUT:
       A list of 9-element vectors describing the representation
       obstructions at primes dividing the level.

    EXAMPLES::

       sage: Q = DiagonalQuadraticForm(ZZ, [])
       sage: Q.local_representation_conditions()
       This 0-dimensional form only represents zero.

       sage: Q = DiagonalQuadraticForm(ZZ, [5])
       sage: Q.local_representation_conditions()
       This 1-dimensional form only represents square multiples of 5.

       sage: Q1 = DiagonalQuadraticForm(ZZ, [1,1])
       sage: Q1.local_representation_conditions()
       This 2-dimensional form represents the p-adic integers of even
       valuation for all primes p except [2].
       For these and the reals, we have:
```



```
       Reals:   [0, +Infinity]
       p = 2:   [0, +Infinity, 0, +Infinity, 0, +Infinity, 0, +Infinity]


       sage: Q1 = DiagonalQuadraticForm(ZZ, [1,1,1])
       sage: Q1.local_representation_conditions()
       This form represents the p-adic integers Z_p for all primes p except
       [2].  For these and the reals, we have:
        Reals:   [0, +Infinity]
        p = 2:   [0, 0, 0, +Infinity, 0, 0, 0, 0]

       sage: Q1 = DiagonalQuadraticForm(ZZ, [1,1,1,1])
       sage: Q1.local_representation_conditions()
       This form represents the p-adic integers Z_p for all primes p except
       [].  For these and the reals, we have:
        Reals:   [0, +Infinity]

       sage: Q1 = DiagonalQuadraticForm(ZZ, [1,3,3,3])
       sage: Q1.local_representation_conditions()
       This form represents the p-adic integers Z_p for all primes p except
       [3].  For these and the reals, we have:
        Reals:   [0, +Infinity]
        p = 3:   [0, 1, 0, 0]

       sage: Q2 = DiagonalQuadraticForm(ZZ, [2,3,3,3])
       sage: Q2.local_representation_conditions()
       This form represents the p-adic integers Z_p for all primes p except
       [3].  For these and the reals, we have:
        Reals:   [0, +Infinity]
        p = 3:   [1, 0, 0, 0]

       sage: Q3 = DiagonalQuadraticForm(ZZ, [1,3,5,7])
       sage: Q3.local_representation_conditions()
       This form represents the p-adic integers Z_p for all primes p except
       [].  For these and the reals, we have:
        Reals:   [0, +Infinity]

     """
        ## Recompute the local conditions if they don't exist or the recompute_flag
is set.
        if (not hasattr(self, "__local_representability_conditions")) or (recompute_flag == Tru
e):
            self.__local_representability_conditions = QuadraticFormLocalRepresentat
ionConditions(self)

        ## Return the local conditions if the silent_flag is not set.
        if not silent_flag:
            return self.__local_representability_conditions



def is_locally_universal_at_prime(self, p):
        """
    Determines if the (integer-valued/rational) quadratic form represents all of 'Z_p'.

    INPUT:
       'p' -- a positive prime number or "infinity".

    OUTPUT:
       boolean

    EXAMPLES::

       sage: Q = DiagonalQuadraticForm(ZZ, [1,3,5,7])
       sage: Q.is_locally_universal_at_prime(2)
       True
       sage: Q.is_locally_universal_at_prime(3)
       True
       sage: Q.is_locally_universal_at_prime(5)
```

```
    True
    sage: Q.is_locally_universal_at_prime(infinity)
    False

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
    sage: Q.is_locally_universal_at_prime(2)
    False
    sage: Q.is_locally_universal_at_prime(3)
    True
    sage: Q.is_locally_universal_at_prime(5)
    True
    sage: Q.is_locally_universal_at_prime(infinity)
    False

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,-1])
    sage: Q.is_locally_universal_at_prime(infinity)
    True

  """
    self.local_representation_conditions(silent_flag=True)
    return self.__local_representability_conditions.is_universal_at_prime(p)


def is_locally_universal_at_all_primes(self):
    """
  Determines if the quadratic form represents 'Z_p' for all finite/non-archimedian primes.

  INPUT:
    none

  OUTPUT:
    boolean

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,3,5,7])
    sage: Q.is_locally_universal_at_all_primes()
    True

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1])
    sage: Q.is_locally_universal_at_all_primes()
    True

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
    sage: Q.is_locally_universal_at_all_primes()
    False

  """
    self.local_representation_conditions(silent_flag=True)
    return self.__local_representability_conditions.is_universal_at_all_finite_p
rimes()


def is_locally_universal_at_all_places(self):
    """
  Determines if the quadratic form represents 'Z_p' for all
  finite/non-archimedian primes, and represents all real numbers.

  INPUT:
```



```
    none

  OUTPUT:
    boolean

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,3,5,7])
    sage: Q.is_locally_universal_at_all_places()
    False

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1])
    sage: Q.is_locally_universal_at_all_places()
    False

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1,-1])
    sage: Q.is_locally_universal_at_all_places()       # long time (8.5 s)
    True

  """
    self.local_representation_conditions(silent_flag=True)
    return self.__local_representability_conditions.is_universal_at_all_places()


def is_locally_represented_number_at_place(self, m, p):
    """
  Determines if the rational number m is locally represented by the
  quadratic form at the (possibly infinite) prime 'p'.

  INPUT:

    'm' -- an integer

    'p' -- a prime number > 0 or 'infinity'

  OUTPUT:
    boolean

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
    sage: Q.is_locally_represented_number_at_place(7, infinity)
    True
    sage: Q.is_locally_represented_number_at_place(7, 2)
    False
    sage: Q.is_locally_represented_number_at_place(7, 3)
    True
    sage: Q.is_locally_represented_number_at_place(7, 5)
    True
    sage: Q.is_locally_represented_number_at_place(-1, infinity)
    False
    sage: Q.is_locally_represented_number_at_place(-1, 2)
    False

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1,-1])
    sage: Q.is_locally_represented_number_at_place(7, infinity)    # long time (8.5 s)
    True
    sage: Q.is_locally_represented_number_at_place(7, 2)           # long time
    True
    sage: Q.is_locally_represented_number_at_place(7, 3)           # long time
    True
    sage: Q.is_locally_represented_number_at_place(7, 5)           # long time
```

```
    True

  """
    self.local_representation_conditions(silent_flag=True)
    return self.__local_representability_conditions.is_locally_represented_at_pl
ace(m, p)



def is_locally_represented_number(self, m):
    """
  Determines if the rational number m is locally represented by the quadratic form.

  INPUT:
    'm' -- an integer

  OUTPUT:
    boolean

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
    sage: Q.is_locally_represented_number(2)
    True
    sage: Q.is_locally_represented_number(7)
    False
    sage: Q.is_locally_represented_number(-1)
    False
    sage: Q.is_locally_represented_number(28)
    False
    sage: Q.is_locally_represented_number(0)
    True

  """
    self.local_representation_conditions(silent_flag=True)
    return self.__local_representability_conditions.is_locally_represented(m)
```

```
"""
Conway–Sloane masses
"""
from sage.rings.integer_ring import ZZ
from sage.rings.rational_field import QQ
from sage.rings.arith import kronecker_symbol, legendre_symbol, prime_divisors,
is_prime, fundamental_discriminant
from sage.symbolic.constants import pi
from sage.misc.misc import prod
from sage.quadratic_forms.special_values import gamma__exact, zeta__exact, quadr
atic_L_function__exact
from sage.functions.all import floor
from sage.symbolic.expression import Expression, is_Expression



def parity(self, allow_rescaling_flag=True):
    """
    Returns the parity ("even" or "odd") of an integer–valued quadratic
    form over 'ZZ', defined up to similitude/rescaling of the form so that
    its Jordan component of smallest scale is unimodular.  After this
    rescaling, we say a form is even if it only represents even numbers,
    and odd if it represents some odd number.

    If the 'allow_rescaling_flag' is set to False, then we require that
    the quadratic form have a Gram matrix with coefficients in 'ZZ', and
    look at the unimodular Jordan block to determine its parity.  This
    returns an error if the form is not integer–matrix, meaning that it
    has Jordan components at 'p=2' which do not have an integer scale.

    We determine the parity by looking for a 1x1 block in the 0–th
    Jordan component, after a possible rescaling.

    INPUT:
        self –– a quadratic form with base_ring 'ZZ', which we may
                require to have integer Gram matrix.

    OUTPUT:
        One of the strings: "even" or "odd"

    EXAMPLES::

        sage: Q = QuadraticForm(ZZ, 3, [4, –2, 0, 2, 3, 2]); Q
        Quadratic form in 3 variables over Integer Ring with coefficients:
        [ 4 –2 0 ]
        [ * 2 3 ]
        [ * * 2 ]
        sage: Q.parity()
        'even'

    ::

        sage: Q = QuadraticForm(ZZ, 3, [4, –2, 0, 2, 3, 1]); Q
        Quadratic form in 3 variables over Integer Ring with coefficients:
        [ 4 –2 0 ]
        [ * 2 3 ]
        [ * * 1 ]
        sage: Q.parity()
        'even'

    ::

        sage: Q = QuadraticForm(ZZ, 3, [4, –2, 0, 2, 2, 2]); Q
        Quadratic form in 3 variables over Integer Ring with coefficients:
        [ 4 –2 0 ]
        [ * 2 2 ]
        [ * * 2 ]
        sage: Q.parity()
        'even'
```



```
    ::

        sage: Q = QuadraticForm(ZZ, 3, [4, –2, 0, 2, 2, 1]); Q
        Quadratic form in 3 variables over Integer Ring with coefficients:
        [ 4 –2 0 ]
        [ * 2 2 ]
        [ * * 1 ]
        sage: Q.parity()
        'odd'

    """
        ## Deal with 0-dim'l forms
        if self.dim() == 0:
            return "even"

        ## Identify the correct Jordan component to use.
        Jordan_list = self.jordan_blocks_by_scale_and_unimodular(2)
        scale_pow_list = [J[0]  for J in Jordan_list]
        min_scale_pow = min(scale_pow_list)
        if allow_rescaling_flag:
            ind = scale_pow_list.index(min_scale_pow)
        else:
            if min_scale_pow < 0:
                raise TypeError, "Oops! If rescaling is not allowed, then we require our form to have an inte
gral Gram matrix."
            ind = scale_pow_list.index(0)


        ## Find the component of scale (power) zero, and then look for an odd dim'l
component.
        J0 = Jordan_list[ind]
        Q0 = J0[1]

        ## The lattice is even if there is no component of scale (power) 0
        if J0 == None:
            return "even"

        ## Look for a 1x1 block in the 0-th Jordan component (which by
        ## convention of the local_normal_form routine will appear first).
        if Q0.dim() == 1:
            return "odd"
        elif Q0[0,1] == 0:
            return "odd"
        else:
            return "even"



def is_even(self, allow_rescaling_flag=True):
    """
    Returns true iff after rescaling by some appropriate factor, the
    form represents no odd integers.  For more details, see parity().

    Requires that Q is defined over 'ZZ'.

    EXAMPLES::

        sage: Q = QuadraticForm(ZZ, 2, [1, 0, 1])
        sage: Q.is_even()
        False
        sage: Q = QuadraticForm(ZZ, 2, [1, 1, 1])
        sage: Q.is_even()
        True

    """
        return self.parity(allow_rescaling_flag) == "even"
```

```
def is_odd(self, allow_rescaling_flag=True):
    """
  Returns true iff after rescaling by some appropriate factor, the
  form represents some odd integers.  For more details, see parity().

  Requires that Q is defined over 'ZZ'.

  EXAMPLES::

    sage: Q = QuadraticForm(ZZ, 2, [1, 0, 1])
    sage: Q.is_odd()
    True
    sage: Q = QuadraticForm(ZZ, 2, [1, 1, 1])
    sage: Q.is_odd()
    False

  """
    return self.parity(allow_rescaling_flag) == "odd"



def conway_species_list_at_odd_prime(self, p):
    """
  Returns an integer called the 'species' which determines the type
  of the orthogonal group over the finite field 'F_p'.

  This assumes that the given quadratic form is a unimodular Jordan
  block at an odd prime 'p'.  When the dimension is odd then this
  number is always positive, otherwise it may be positive or
  negative (or zero, but that is considered positive by convention).

  Note: The species of a zero dim'l form is always 0+, so we
  interpret the return value of zero as positive here! =)

  INPUT:
    a positive prime number

  OUTPUT:
    a list of integers

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, range(1,10))
    sage: Q.conway_species_list_at_odd_prime(3)
    [6, 2, 1]

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, range(1,8))
    sage: Q.conway_species_list_at_odd_prime(3)
    [5, 2]
    sage: Q.conway_species_list_at_odd_prime(5)
    [-6, 1]

  """
    ## Sanity Check:
    if not ((p>2) and is_prime(p)):
        raise TypeError, "Oops!  We are assuming that p is an odd positive prime number."

    ## Deal with the zero-dim'l form
    if self.dim() == 0:
        return [0]

    ## List the (unscaled/unimodular) Jordan blocks by their scale power
    jordan_list = self.jordan_blocks_in_unimodular_list_by_scale_power(p)

    ## Make a list of species (including the two zero-dim'l forms missing at eit
her end of the list of Jordan blocks)
    species_list = []
```



```
    for tmp_Q in jordan_list:

        ## Some useful variables
        n = tmp_Q.dim()
        d = tmp_Q.det()

        ## Determine the species
        if (n % 2 != 0):                                ## Deal with odd dim'l forms
            species = n
        elif (n % 4 == 2) and (p % 4 == 3):             ## Deal with even dim'l form
s
            species = (-1) * legendre_symbol(d, p) * n
        else:
            species = legendre_symbol(d, p) * n

        ## Append the species to the list
        species_list.append(species)

    ## Return the species list
    return species_list



def conway_species_list_at_2(self):
    """
  Returns an integer called the 'species' which determines the type
  of the orthogonal group over the finite field 'F_p'.

  This assumes that the given quadratic form is a unimodular Jordan
  block at an odd prime 'p'.  When the dimension is odd then this
  number is always positive, otherwise it may be positive or
  negative.

  Note: The species of a zero dim'l form is always 0+, so we
  interpret the return value of zero as positive here! =)

  OUTPUT:
    a list of integers

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, range(1,10))
    sage: Q.conway_species_list_at_2()
    [1, 5, 1, 1, 1, 1]

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, range(1,8))
    sage: Q.conway_species_list_at_2()
    [1, 3, 1, 1, 1]

  """
    ## Some useful variables
    n = self.dim()
    d = self.det()

    ## Deal with the zero-dim'l form
    if n == 0:
        return 0

    ## List the (unscaled/unimodular) Jordan blocks by their scale power
    jordan_list = self.jordan_blocks_in_unimodular_list_by_scale_power(2)

    ## Make a list of species (including the two zero-dim'l forms missing at eit
her end of the list of Jordan blocks)
    species_list = []

    if jordan_list[0].parity() == "odd":        ## Add an entry for the unlisted
 "-1" Jordan component as well.
```

```
        species_list.append(1)

    for i in range(len(jordan_list)):           ## Add an entry for each (listed
) Jordan component

        ## Make the number 2*t in the C-S Table 1.
        d = jordan_list[i].dim()
        if jordan_list[i].is_even():
            two_t = d
        else:
            two_t = ZZ(2) * floor((d-1) / 2)

        ## Determine if the form is bound
        if len(jordan_list) == 1:
            is_bound = False
        elif i == 0:
            is_bound = jordan_list[i+1].is_odd()
        elif i == len(jordan_list) - 1:
            is_bound = jordan_list[i-1].is_odd()
        else:
            is_bound = jordan_list[i-1].is_odd() or jordan_list[i+1].is_odd()

        ## Determine the species
        octane = jordan_list[i].conway_octane_of_this_unimodular_Jordan_block_at
_2()
        if is_bound or (octane == 2) or (octane == 6):
            species = two_t + 1
        elif (octane == 0) or (octane == 1) or (octane == 7):
            species = two_t
        else:
            species = (-1) * two_t

        ## Append the species to the list
        species_list.append(species)


    if jordan_list[-1].is_odd():        ## Add an entry for the unlisted "s_max
+ 1" Jordan component as well.
        species_list.append(1)

    ## Return the species list
    return species_list




def conway_octane_of_this_unimodular_Jordan_block_at_2(self):
    """
    Determines the 'octane' of this full unimodular Jordan block at
    the prime 'p=2'.  This is an invariant defined '(mod 8)', ad.

    This assumes that the form is given as a block diagonal form with
    unimodular blocks of size <= 2 and the 1x1 blocks are all in the upper
    leftmost position.

    INPUT:
      none

    OUTPUT:
      an integer 0 <= x <= 7

    EXAMPLES::

      sage: Q = DiagonalQuadraticForm(ZZ, [1,3,5,7])
      sage: Q.conway_octane_of_this_unimodular_Jordan_block_at_2()
      0
      sage: Q = DiagonalQuadraticForm(ZZ, [1,5,13])
      sage: Q.conway_octane_of_this_unimodular_Jordan_block_at_2()
      3
```



```
      sage: Q = DiagonalQuadraticForm(ZZ, [3,7,13])
      sage: Q.conway_octane_of_this_unimodular_Jordan_block_at_2()
      7

    """
    ## Deal with 'even' forms
    if self.parity() == "even":
        d = self.Gram_matrix().det()
        if (d % 8 == 1) or (d % 8 == 7):
            return 0
        else:
            return 4

    ## Deal with 'odd' forms by diagonalizing, and then computing the octane.
    n = self.dim()
    u = self[0,0]
    tmp_diag_vec = [None  for i in range(n)]
    tmp_diag_vec[0] = u       ## This should be an odd integer!
    ind = 1                   ## The next index to diagonalize


    ## Use u to diagonalize the form -- WHAT ARE THE POSSIBLE LOCAL NORMAL FORMS
?
    while ind < n:

        ## Check for a 1x1 block and diagonalize it
        if (ind == (n-1)) or (self[ind, ind+1] == 0):
            tmp_diag_vec[ind] = self[ind, ind]
            ind += 1

        ## Diagonalize the 2x2 block
        else:
            B = self[ind, ind+1]
            if (B % 2 != 0):
                raise RuntimeError, "Oops, we expected the mixed term to be even! "

            a = self[ind, ind]
            b = ZZ(B / ZZ(2))
            c = self[ind+1, ind+1]
            tmp_disc = b * b - a * c

            ## Perform the diagonalization
            if (tmp_disc % 8 == 1):                 ## 2xy
                tmp_diag_vec[ind] = 1
                tmp_diag_vec[ind+1] = -1
                ind += 2
            elif(tmp_disc % 8 == 5):                ## 2x^2 + 2xy + 2y^2
                tmp_diag_vec[0] = 3*u
                tmp_diag_vec[ind] = -u
                tmp_diag_vec[ind+1] = -u
                ind += 2
                u = tmp_diag_vec[0]
            else:
                raise RuntimeError, "Oops!  This should not happen -- the odd 2x2 blocks have disc
1 or 5 (mod 8)."

    ## Compute the octane
    octane = 0
    for a in tmp_diag_vec:
        if a % 4 == 1:
            octane += 1
        elif a % 4 == 3:
            octane += -1
        else:
            raise RuntimeError, "Oops!  The diagonal elements should all be odd... =("

    ## Return its value
    return octane % 8
```

```
def conway_diagonal_factor(self, p):
    """
  Computes the diagonal factor of Conway's 'p'-mass.

  INPUT:
    'p' -- a prime number > 0

  OUTPUT:
    a rational number > 0

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, range(1,6))
    sage: Q.conway_diagonal_factor(3)
    81/256

  """
    ## Get the species list at p
    if p == 2:
        species_list = self.conway_species_list_at_2()
    else:
        species_list = self.conway_species_list_at_odd_prime(p)

    ## Evaluate the diagonal factor
    diag_factor = QQ(1)
    for s in species_list:
        if s == 0:
            pass
        elif s % 2 == 1:                    ## Note: Here always s > 0.
            diag_factor = diag_factor / (2 * prod([1 - QQ(p)**(-i)  for i in ran
ge(2, s, 2)]))
        else:
            diag_factor = diag_factor / (2 * prod([1 - QQ(p)**(-i)  for i in ran
ge(2, abs(s), 2)]))
            s_sign = ZZ(s / abs(s))
            diag_factor = diag_factor / (ZZ(1) - s_sign * QQ(p) ** ZZ(-abs(s) /
ZZ(2)))

    ## Return the diagonal factor
    return diag_factor



def conway_cross_product_doubled_power(self, p):
    """
  Computes twice the power of p which evaluates the 'cross product'
  term in Conway's mass formula.

  INPUT:
    'p' -- a prime number > 0

  OUTPUT:
    a rational number

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, range(1,8))
    sage: Q.conway_cross_product_doubled_power(2)
    18
    sage: Q.conway_cross_product_doubled_power(3)
    10
    sage: Q.conway_cross_product_doubled_power(5)
    6
    sage: Q.conway_cross_product_doubled_power(7)
    6
    sage: Q.conway_cross_product_doubled_power(11)
    0
```



```
    sage: Q.conway_cross_product_doubled_power(13)
    0

  """
    doubled_power = 0
    dim_list = [J.dim()  for J in self.jordan_blocks_in_unimodular_list_by_scale
_power(p)]
    for i in range(len(dim_list)):
        for j in range(i):
            doubled_power += (i-j) * dim_list[i] * dim_list[j]

    return doubled_power



def conway_type_factor(self):
    """
  This is a special factor only present in the mass formula when 'p=2'.

  INPUT:
    none

  OUTPUT:
    a rational number

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, range(1,8))
    sage: Q.conway_type_factor()
    4

  """
    jordan_list = self.jordan_blocks_in_unimodular_list_by_scale_power(2)
    n2 = sum([J.dim()  for J in jordan_list  if J.is_even()])
    n11 = sum([1  for i in range(len(jordan_list) - 1)  if jordan_list[i].is_odd
() and jordan_list[i+1].is_odd()])

    return ZZ(2)**(n11 - n2)



def conway_p_mass(self, p):
    """
  Computes Conway's 'p'-mass.

  INPUT:
    'p' -- a prime number > 0

  OUTPUT:
    a rational number > 0

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, range(1, 6))
    sage: Q.conway_p_mass(2)
    16/3
    sage: Q.conway_p_mass(3)
    729/256

  """
    ## Compute the first two factors of the p-mass
    p_mass = self.conway_diagonal_factor(p) * (p ** (self.conway_cross_product_d
oubled_power(p) / ZZ(2)))

    ## Multiply by the 'type factor' when p = 2
    if p == 2:
        p_mass *= self.conway_type_factor()

    ## Return the result
```

```
    return p_mass



def conway_standard_p_mass(self, p):
    """
  Computes the standard (generic) Conway–Sloane ‘p‘–mass.

  INPUT:
    ‘p‘ –– a prime number > 0

  OUTPUT:
    a rational number > 0

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
    sage: Q.conway_standard_p_mass(2)
    2/3

  """
    ## Some useful variables
    n = self.dim()
    if n % 2 == 0:
        s = n / ZZ(2)
    else:
        s = (n+1) / ZZ(2)

    ## Compute the inverse of the generic p-mass
    p_mass_inv = ZZ(2) * prod([1-p**(-i)  for i in range(2, 2*s, 2)])
    if n % 2 == 0:
        D = (-1)**s * self.det() * (2**n)   ##   We should have something like
D = (-1)**s * self.det() / (2**n), but that's not an integer and here we only ca
re about the square-class.
        #d = self.det()   ## Note: No normalizing power of 2 is needed since the
 power is even.
        #if not ((p == 2) or (d % p == 0)):
        p_mass_inv *= (1 - kronecker_symbol(fundamental_discriminant(D), p) * p*
*(-s))

    ## Return the standard p-mass
    return ZZ(1) / p_mass_inv



def conway_standard_mass(self):
    """
  Returns the infinite product of the standard mass factors.

  INPUT:
    none

  OUTPUT:
    a rational number > 0

  EXAMPLES::

    sage: Q = QuadraticForm(ZZ, 3, [2, –2, 0, 3, –5, 4])
    sage: Q.conway_standard_mass()
    1/6

  ::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
    sage: Q.conway_standard_mass()
    1/6

  """
    n = self.dim()
```



```
    if n % 2 == 0:
        s = n / 2
    else:
        s = (n+1) / 2

    ## DIAGNOSTIC
    #print "n = ", n
    #print "s = ", s
    #print "Gamma Factor = \n", prod([gamma__exact(j / ZZ(2))  for j in range(1,
 n+1)])
    #print "Zeta Factor = \n", prod([zeta__exact(2*k)  for k in range(1, s)])
    #print "Pi Factor = \n", pi**((-1) * n * (n+1) / ZZ(4))

    generic_mass = 2 * pi**((-1) * n * (n+1) / ZZ(4)) \
            * prod([gamma__exact(j / ZZ(2))  for j in range(1, n+1)]) \
            * prod([zeta__exact(2*k)  for k in range(1, s)])

    if n % 2 == 0:
        D = (-1)**s * self.det() * (2**n)   ##   We should have something like
D = (-1)**s * self.det() / (2**n), but that's not an integer and here we only ca
re about the square-class.
        generic_mass *= quadratic_L_function__exact(s, D)

    return generic_mass



def conway_mass(self):
    """
  Compute the mass by using the Conway–Sloane mass formula.

  INPUT:
    none

  OUTPUT:
    a rational number > 0

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
    sage: Q.conway_mass()
    1/48

    sage: Q = DiagonalQuadraticForm(ZZ, [7,1,1])
    sage: Q.conway_mass()
    3/16

    sage: Q = QuadraticForm(ZZ, 3, [7, 2, 2, 2, 0, 2]) + DiagonalQuadraticForm(ZZ, [1])
    sage: Q.conway_mass()
    3/32

  """
    ## Try to use the cached result
    try:
        return self.__conway_mass
    except:
        ## Double the form so it's integer-matrix
        Q = self.scale_by_factor(2)

        ## Compute the standard mass
        mass = Q.conway_standard_mass()

        ## Adjust the p-masses when p|2d
        d = self.det()
        for p in prime_divisors(2*d):
            mass *= (Q.conway_p_mass(p) / Q.conway_standard_p_mass(p))

        ## Simplify any radicals that may appear in the mass
        if isinstance(mass, Expression):
```

```
            mass = mass.simplify_radical()

        ## Cache and return the (simplified) result
        self.__conway_mass = QQ(abs(mass))
        return self.__conway_mass


## =======================================================



#def conway_generic_mass(self):
#    """
#    Computes the generic mass given as
#         2 \pi^{-n(n+1)/4} \prod_{j=1}^{n} \Gamma\(\tfrac{j}{2}\)
#        \zeta(2) \cdots \zeta(2s-2) \zeta_{D}(s)
#    where $n = 2s$ or $2s-1$ depending on the parity of $n$,
#    and $D = (-1)^{s} d$.  We interpret the symbol $\(\frac{D}{p}\)$
#    as 0 if $p\mid 2d$.
#    (Conway and Sloane, Mass formula paper, p??)
#
#    This is possibly equal to
#        2^{-td} * \tau(G) *[\prod_{i=1}^{t} \zeta(1-2i) ]* L(1-t, \chi)
#    where $\dim(Q) = n = 2t$ or $2t+1$, and the last factor is omitted
#    when $n$ is odd.
#    (GHY, Prop 7.4 and 7.5, p121)
#    """
#    RR = RealField(200)
#    n = self.dim()
#    if n % 2 == 0:
#        s = n / 2
#    else:
#        s = (n-1) / 2
#
#    ## Form the generic zeta product
#    ans = 2 * RR(pi)^(-n * (n+1) / 4)
#    for j in range(1,n+1):
#        ans *= gamma(RR(j/2))
#    for j in range(2, 2*s, 2):  ## j = 2, ..., 2s-2
#        ans *= zeta(RR(j))
#
#    ## Extra L-factor for even dimensional forms  -- DO THIS!!!
#    raise NotImplementedError, "This routine is not finished yet... =("
#
#    ## Return the answer
#    return ans




#def conway_p_mass_adjustment(self, p):
#    """
#    Computes the adjustment to give the p-mass from the generic mass.
#    """
#    pass


##########################################################################
```

```
"""
Shimura Mass
"""
#####################################################
## Routines to compute the mass of a quadratic form ##
#####################################################

## Import all general mass finding routines
from sage.quadratic_forms.quadratic_form__mass__Siegel_densities import \
        mass__by_Siegel_densities, \
        Pall_mass_density_at_odd_prime, \
        Watson_mass_at_2, \
        Kitaoka_mass_at_2, \
        mass_at_two_by_counting_mod_power
from sage.quadratic_forms.quadratic_form__mass__Conway_Sloane_masses import \
        parity, \
        is_even, \
        is_odd, \
        conway_species_list_at_odd_prime, \
        conway_species_list_at_2, \
        conway_octane_of_this_unimodular_Jordan_block_at_2, \
        conway_diagonal_factor, \
        conway_cross_product_doubled_power, \
        conway_type_factor, \
        conway_p_mass, \
        conway_standard_p_mass, \
        conway_standard_mass, \
        conway_mass
#        conway_generic_mass, \
#        conway_p_mass_adjustment

################################################



def has_class_number_one(self):
    """
    Uses the mass formula to check that the class number of the
    quadratic form is one.  (I.e., the quadratic form has one class in
    its genus.)

    TO DO/FIX: This gives the wrong result for one variable forms,
    which should always have class number one!


    INPUT:
        None

    OUTPUT:
        Boolean

    EXAMPLES:
        sage: CN1_truth = [DiagonalQuadraticForm(ZZ, n*[1]).has_class_number_one()  for n in range(2, 10)]; CN1_truth
        [True, True, True, True, True, True, True, False, False, False]

    """
    ## See if the mass is realized by this one class.
    mass = self.conway_mass()
    num_of_autos = self.number_of_automorphisms()
    return mass == 1/num_of_autos
```



```
def shimura_mass__maximal(self):
    """
    Use Shimura's exact mass formula to compute the mass of a maximal
    quadratic lattice. This works for any totally real number field,
    but has a small technical restriction when 'n' is odd.

    INPUT:
        none

    OUTPUT:
        a rational number

    EXAMPLE::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
        sage: Q.shimura_mass__maximal()

    """
    pass



def GHY_mass__maximal(self):
    """
    Use the GHY formula to compute the mass of a (maximal?) quadratic
    lattice. This works for any number field.

    Reference:  See [GHY, Prop 7.4 and 7.5, p121] and [GY, Thrm 10.20, p25].

    INPUT:
        none

    OUTPUT:
        a rational number

    EXAMPLE::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
        sage: Q.GHY_mass__maximal()

    """
    pass
```

```
"""
Local Masses and Siegel Densities
"""
#############################################################################
#####################
##  Computes the local masses (rep'n densities of a form by itself) for a quadra
tic forms over ZZ
##      using the papers of Pall [PSPUM VIII (1965), pp95--105]  for p>2, and Wa
tson [Mathematika
##      23, no. 1, (1976), pp 94--106] for p=2.  These formulas will also work f
or any local field
##          which is unramified at p=2.
##
##  Copyright by Jonathan Hanke 2007 <jonhanke@gmail.com>
#############################################################################
#####################

import copy

from sage.misc.misc import prod
from sage.misc.mrange import mrange
from sage.functions.all import floor
from sage.rings.integer_ring import ZZ
from sage.rings.finite_rings.integer_mod_ring import IntegerModRing
from sage.rings.rational_field import QQ
from sage.rings.arith import legendre_symbol, kronecker, prime_divisors
from sage.functions.all import sgn
from sage.quadratic_forms.special_values import gamma__exact, zeta__exact, quadr
atic_L_function__exact
from sage.misc.functional import squarefree_part
from sage.symbolic.constants import pi
from sage.matrix.matrix_space import MatrixSpace


def mass__by_Siegel_densities(self, odd_algorithm="Pall", even_algorithm="Watson")
:
    """
    Gives the mass of transformations (det 1 and -1).

    WARNING: THIS IS BROKEN RIGHT NOW... =(

    Optional Arguments:

    - When p > 2  --  odd_algorithm = "Pall" (only one choice for now)
    - When p = 2  --  even_algorithm = "Kitaoka" or "Watson"

    REFERENCES:

    - Nipp's Book "Tables of Quaternary Quadratic Forms".
    - Papers of Pall (only for p>2) and Watson (for 'p=2' -- tricky!).
    - Siegel, Milnor-Hussemoller, Conway-Sloane Paper IV, Kitoaka (all of which
      have problems...)

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1])
        sage: Q.mass__by_Siegel_densities()
        1/384
        sage: Q.mass__by_Siegel_densities() - (2^Q.dim() * factorial(Q.dim()))^(-1)
        0

    ::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
        sage: Q.mass__by_Siegel_densities()
        1/48
        sage: Q.mass__by_Siegel_densities() - (2^Q.dim() * factorial(Q.dim()))^(-1)
        0
```



```
    """
    ## Setup
    n = self.dim()
    s = floor((n-1)/2)
    if n % 2 != 0:
        char_d = squarefree_part(2*self.det())   ## Accounts for the det as a QF
    else:
        char_d = squarefree_part(self.det())

    ## Form the generic zeta product
    generic_prod = ZZ(2) * (pi)**(-ZZ(n) * (n+1) / 4)
    ########################################
    generic_prod *= (self.det())**(ZZ(n+1)/2)  ## ***** This uses the Hessian De
terminant ********
    ########################################
    #print "gp1 = ", generic_prod
    generic_prod *= prod([gamma__exact(ZZ(j)/2)  for j in range(1,n+1)])
    #print "\n---", [(ZZ(j)/2, gamma__exact(ZZ(j)/2))  for j in range(1,n+1)]
    #print "\n---", prod([gamma__exact(ZZ(j)/2)  for j in range(1,n+1)])
    #print "gp2 = ", generic_prod
    generic_prod *= prod([zeta__exact(ZZ(j))  for j in range(2, 2*s+1, 2)])
    #print "\n---", [zeta__exact(ZZ(j))  for j in range(2, 2*s+1, 2)]
    #print "\n---", prod([zeta__exact(ZZ(j))  for j in range(2, 2*s+1, 2)])
    #print "gp3 = ", generic_prod
    if (n % 2 == 0):
        generic_prod *= ZZ(1) * quadratic_L_function__exact(n/2, (-1)**(n/2) * c
har_d)
        #print " NEW = ", ZZ(1) * quadratic_L_function__exact(n/2, (-1)**(n/2) *
 char_d)
        #print
    #print "gp4 = ", generic_prod

    #print "generic_prod =", generic_prod

    ## Determine the adjustment factors
    adj_prod = 1
    for p in prime_divisors(2 * self.det()):
        ## Cancel out the generic factors
        p_adjustment = prod([1 - ZZ(p)**(-j)  for j in range(2, 2*s+1, 2)])
        if (n % 2 == 0):
            p_adjustment *= ZZ(1) * (1 - kronecker((-1)**(n/2) * char_d, p) * ZZ
(p)**(-n/2))
            #print " EXTRA = ", ZZ(1) * (1 - kronecker((-1)**(n/2) * char_d, p)
* ZZ(p)**(-n/2))
        #print "Factor to cancel the generic one:", p_adjustment

        ## Insert the new mass factors
        if p == 2:
            if even_algorithm == "Kitaoka":
                p_adjustment = p_adjustment / self.Kitaoka_mass_at_2()
            elif even_algorithm == "Watson":
                p_adjustment = p_adjustment / self.Watson_mass_at_2()
            else:
                raise TypeError, "There is a problem -- your even_algorithm argument is invalid.  Tr
y again. =("
        else:
            if odd_algorithm == "Pall":
                p_adjustment = p_adjustment / self.Pall_mass_density_at_odd_prim
e(p)
            else:
                raise TypeError, "There is a problem -- your optional arguments are invalid.  Try agai
n. =("

        #print "p_adjustment for p =", p, "is", p_adjustment

        ## Put them together (cumulatively)
        adj_prod *= p_adjustment

    #print "Cumulative adj_prod =", adj_prod
```

```
        ## Extra adjustment for the case of a 2-dimensional form.
    #if (n == 2):
    #    generic_prod *= 2


    ## Return the mass
    mass = generic_prod * adj_prod
    return mass



def Pall_mass_density_at_odd_prime(self, p):
    """
  Returns the local representation density of a form (for
  representing itself) defined over 'ZZ', at some prime 'p>2'.

  REFERENCES:
    Pall's article "The Weight of a Genus of Positive n-ary Quadratic Forms"
    appearing in Proc. Symp. Pure Math. VIII (1965), pp95--105.

  INPUT:
    'p' -- a prime number > 2.

  OUTPUT:
    a rational number.

  EXAMPLES::

    sage: Q = QuadraticForm(ZZ, 3, [1,0,0,1,0,1])
    sage: Q.Pall_mass_density_at_odd_prime(3)
    [(0, Quadratic form in 3 variables over Integer Ring with coefficients:
    [ 1 0 0 ]
    [ * 1 0 ]
    [ * * 1 ]
    )] [(0, 3, 8)] [8/9] 8/9
    8/9

  """
    ## Check that p is a positive prime -- unnecessary since it's done implicitl
y in the next step. =)
    if p<=2:
        raise TypeError, "Oops!  We need p to be a prime > 2."

    ## Step 1: Obtain a p-adic (diagonal) local normal form, and
    ## compute the invariants for each Jordan block.
    jordan_list = self.jordan_blocks_by_scale_and_unimodular(p)
    modified_jordan_list = [(a, Q.dim(), Q.det())  for (a,Q) in jordan_list]
 ## List of pairs (scale, det)
    #print jordan_list
    #print modified_jordan_list


    ## Step 2: Compute the list of local masses for each Jordan block
    jordan_mass_list = []
    for (s,n,d) in modified_jordan_list:
        generic_factor = prod([1 - p**(-2*j)  for j in range(1, floor((n-1)/2)+1
)])
        #print "generic factor: ", generic_factor
        if (n % 2 == 0):
            m = n/2
            generic_factor *= (1 + legendre_symbol(((-1)**m) * d, p) * p**(-m))
        #print "jordan_mass: ", generic_factor
        jordan_mass_list = jordan_mass_list + [generic_factor]


    ## Step 3: Compute the local mass $\al_p$ at p.
        MJL = modified_jordan_list
    s = len(modified_jordan_list)
```



```
    M = [sum([MJL[j][1]  for j in range(i, s)])  for i in range(s-1)]    ## Note
: It's s-1 since we don't need the last M.
    #print "M = ", M
    nu = sum([M[i] * MJL[i][0] * MJL[i][1]  for i in range(s-1)]) - ZZ(sum([J[0]
 * J[1] * (J[1]-1)  for J in MJL]))/ZZ(2)
    p_mass = prod(jordan_mass_list)
    p_mass *= 2**(s-1) * p**nu

    print jordan_list, MJL, jordan_mass_list, p_mass

    ## Return the result
    return p_mass


def Watson_mass_at_2(self):
    """
  Returns the local mass of the quadratic form when 'p=2', according
  to Watson's Theorem 1 of "The 2-adic density of a quadratic form"
  in Mathematika 23 (1976), pp 94--106.

  INPUT:
    none

  OUTPUT:
    a rational number

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
    sage: Q.Watson_mass_at_2()            ## WARNING:  WE NEED TO CHECK THIS CAREFULLY!
    384

  """
    ## Make a 0-dim'l quadratic form (for initialization purposes)
    Null_Form = copy.deepcopy(self)
    Null_Form.__init__(ZZ, 0)

    ## Step 0: Compute Jordan blocks and bounds of the scales to keep track of
    Jordan_Blocks = self.jordan_blocks_by_scale_and_unimodular(2)
    scale_list = [B[0]  for B in Jordan_Blocks]
    s_min = min(scale_list)
    s_max = max(scale_list)

    ## Step 1: Compute dictionaries of the diagonal block and 2x2 block for each
 scale
    diag_dict = dict((i, Null_Form)  for i in range(s_min-2, s_max + 4))     ##
Initialize with the zero form
    dim2_dict = dict((i, Null_Form)  for i in range(s_min, s_max + 4))       ##
Initialize with the zero form
    for (s,L) in Jordan_Blocks:
        i = 0
        while (i < L.dim()-1) and (L[i,i+1] == 0):      ## Find where the 2x2 bl
ocks start
            i = i + 1
        if i < (L.dim() - 1):
            diag_dict[s] = L.extract_variables(range(i))                 ## Diago
nal Form
            dim2_dict[s+1] = L.extract_variables(range(i, L.dim()))     ## Non-d
iagonal Form
        else:
            diag_dict[s] = L

    #print "diag_dict = ", diag_dict
    #print "dim2_dict = ", dim2_dict
    #print "Jordan_Blocks = ", Jordan_Blocks

    ## Step 2: Compute three dictionaries of invariants (for n_j, m_j, nu_j)
    n_dict = dict((j,0)  for j in range(s_min+1, s_max+2))
    m_dict = dict((j,0)  for j in range(s_min, s_max+4))
```

```
    for (s,L) in Jordan_Blocks:
        n_dict[s+1] = L.dim()
        if diag_dict[s].dim() == 0:
            m_dict[s+1] = ZZ(1)/ZZ(2) * L.dim()
        else:
            m_dict[s+1] = floor(ZZ(L.dim() - 1) / ZZ(2))
            #print " ==>", ZZ(L.dim() - 1) / ZZ(2), floor(ZZ(L.dim() - 1) / ZZ(2
))

    nu_dict = dict((j,n_dict[j+1] - 2*m_dict[j+1])  for j in range(s_min, s_max+
1))
    nu_dict[s_max+1] = 0

    #print "n_dict = ", n_dict
    #print "m_dict = ", m_dict
    #print "nu_dict = ", nu_dict

    ## Step 3: Compute the e_j dictionary
    eps_dict = {}
    for j in range(s_min, s_max+3):
        two_form = (diag_dict[j-2] + diag_dict[j] + dim2_dict[j]).scale_by_facto
r(2)
        j_form = (two_form + diag_dict[j-1]).base_change_to(IntegerModRing(4))

        if j_form.dim() == 0:
            eps_dict[j] = 1
        else:
            iter_vec = [4] * j_form.dim()
            alpha = sum([True  for x in mrange(iter_vec)  if j_form(x) == 0])
            beta = sum([True  for x in mrange(iter_vec)  if j_form(x) == 2])
            if alpha > beta:
                eps_dict[j] = 1
            elif alpha == beta:
                eps_dict[j] = 0
            else:
                eps_dict[j] = -1

    #print "eps_dict = ", eps_dict


    ## Step 4: Compute the quantities nu, q, P, E for the local mass at 2
    nu = sum([j * n_dict[j] * (ZZ(n_dict[j] + 1) / ZZ(2) + \
              sum([n_dict[r]  for r in range(j+1, s_max+2)]))  for j in range(s_
min+1, s_max+2)])
    q = sum([sgn(nu_dict[j-1] * (n_dict[j] + sgn(nu_dict[j])))  for j in range(s
_min+1, s_max+2)])
    P = prod([ prod([1 - QQ(4)**(-i)  for i in range(1, m_dict[j]+1)])  for j in
 range(s_min+1, s_max+2)])
    E = prod([ZZ(1)/ZZ(2) * (1 + eps_dict[j] * QQ(2)**(-m_dict[j]))  for j in ra
nge(s_min, s_max+3)])

    #print "\nFinal Summary:"
    #print "nu =", nu
    #print "q = ", q
    #print "P = ", P
    #print "E = ", E


    ## Step 5: Compute the local mass for the prime 2.
    mass_at_2 = QQ(2)**(nu - q) * P / E
    return mass_at_2



def Kitaoka_mass_at_2(self):
    """
  Returns the local mass of the quadratic form when 'p=2', according
  to Theorem 5.6.3 on pp108--9 of Kitaoka's Book "The Arithmetic of
    Quadratic Forms".
```



```
  INPUT:
    none

  OUTPUT:
    a rational number > 0

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
    sage: Q.Kitaoka_mass_at_2()  ## WARNING:  WE NEED TO CHECK THIS CAREFULLY!
    1/2

  """
    ## Make a 0-dim'l quadratic form (for initialization purposes)
    Null_Form = copy.deepcopy(self)
    Null_Form.__init__(ZZ, 0)

    ## Step 0: Compute Jordan blocks and bounds of the scales to keep track of
    Jordan_Blocks = self.jordan_blocks_by_scale_and_unimodular(2)
    scale_list = [B[0]  for B in Jordan_Blocks]
    s_min = min(scale_list)
    s_max = max(scale_list)

    ## Step 1: Compute dictionaries of the diagonal block and 2x2 block for each
 scale
    diag_dict = dict((i, Null_Form)  for i in range(s_min-2, s_max + 4))     ##
Initialize with the zero form
    dim2_dict = dict((i, Null_Form)  for i in range(s_min, s_max + 4))       ##
Initialize with the zero form
    for (s,L) in Jordan_Blocks:
        i = 0
        while (i < L.dim()-1) and (L[i,i+1] == 0):      ## Find where the 2x2 bl
ocks start
            i = i + 1
        if i < (L.dim() - 1):
            diag_dict[s] = L.extract_variables(range(i))                ## Diago
nal Form
            dim2_dict[s+1] = L.extract_variables(range(i, L.dim()))     ## Non-d
iagonal Form
        else:
            diag_dict[s] = L

    #print "diag_dict = ", diag_dict
    #print "dim2_dict = ", dim2_dict
    #print "Jordan_Blocks = ", Jordan_Blocks


    ##################  START EDITING HERE  ##################

    ## Compute q := sum of the q_j
    q = 0
    for j in range(s_min, s_max + 1):
        if diag_dict[j].dim() > 0:                ## Check that N_j is odd (i.e.
rep'ns an odd #)
            if diag_dict[j+1].dim() == 0:
                q += Jordan_Blocks[j][1].dim()        ## When N_{j+1} is "even",
 add n_j
            else:
                q += Jordan_Blocks[j][1].dim() + 1    ## When N_{j+1} is "odd",
add n_j + 1

    ## Compute P = product of the P_j
    P = QQ(1)
    for j in range(s_min, s_max + 1):
        tmp_m = dim2_dict[j].dim() / 2
        P *= prod([QQ(1) - QQ(4**(-k))  for j in range(1, tmp_m + 1)])

    ## Compute the product E := prod_j (1 / E_j)
```

```
    E = QQ(1)
    for j in range(s_min - 1, s_max + 2):
        if (diag_dict[j-1].dim() == 0) and (diag_dict[j+1].dim() == 0) and \
           ((diag_dict[j].dim() != 2) or (((diag_dict[j][0,0] - diag_dict[j][1,1
]) % 4) != 0)):

            ## Deal with the complicated case:
            tmp_m = dim2_dict[j].dim() / 2
            if dim2_dict[j].is_hyperbolic(2):
                E *= 2 / (1 + 2**(-tmp_m))
            else:
                E *= 2 / (1 - 2**(-tmp_m))

        else:
            E *= 2

    ## DIAGNOSTIC
    #print "\nFinal Summary:"
    #print "nu =", nu
    #print "q = ", q
    #print "P = ", P
    #print "E = ", E

    ## Compute the exponent w
    w = QQ(0)
    for j in range(s_min, s_max+1):
        n_j = Jordan_Blocks[j][1].dim()
        for k in range(j+1, s_max+1):
            n_k = Jordan_Blocks[k][1].dim()
            w += j * n_j * (n_k + QQ(n_j + 1) / 2)


    ## Step 5: Compute the local mass for the prime 2.
    mass_at_2 = (QQ(2)**(w - q)) * P * E
    return mass_at_2



def mass_at_two_by_counting_mod_power(self, k):
    """
  Computes the local mass at 'p=2' assuming that it's stable '(mod 2^k)'.

  Note: This is **way** too slow to be useful, even when k=1!!!

  TO DO: Remove this routine, or try to compile it!


  INPUT:
    k -- an integer >= 1

  OUTPUT:
    a rational number

  EXAMPLE::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
    sage: Q.mass_at_two_by_counting_mod_power(1)
    4

  """
    R = IntegerModRing(2**k)
    Q1 = self.base_change_to(R)
    n = self.dim()
    MS = MatrixSpace(R, n)

    ct = sum([1  for x in mrange([2**k] * (n**2))  if Q1(MS(x)) == Q1])   ## Cou
nt the solutions mod 2^k
    two_mass = ZZ(1)/2 * (ZZ(ct) / ZZ(2)**(k*n*(n-1)/2))
    return two_mass
```

```
########################################
## Routines relating to maximal lattices ##
########################################

from sage.misc.misc import verbose
from copy import deepcopy

from sage.rings.all import ZZ, QQ
from sage.rings.all import GF
from sage.matrix.constructor import matrix
from sage.functions.other import sqrt
from sage.misc.functional import squarefree_part
from sage.rings.arith import prime_divisors

from sage.quadratic_forms.maximal_extras import \
    find_basis_of_maximal_isotropic_subspace, \
    diagonalise_mod_2, \
    even_neighbor_of_bilinear_gram_matrix


#from sage.quadratic_forms.quadratic_form import DiagonalQuadraticForm

#from sage.quadratic_forms.quadratic_form import QuadraticForm



def is_maximal(self):
    """
  Returns if the current quadratic form is maximal (i.e. the
  quadratic form of a maximal integer–valued lattice in its rational
  quadratic space).

  INPUT:
    none

  OUTPUT:
    boolean

  EXAMPLES:
    sage: Q3 = DiagonalQuadraticForm(ZZ, [1,1,1])
    sage: Q3.is_maximal()
    True

    sage: Q4 = DiagonalQuadraticForm(ZZ, [1,1,1,1])
    sage: Q4.is_maximal()
    False

    sage: Q4_sub = QuadraticForm(ZZ, 4, [1, 0, 0, 1,  1, 0, 1,  1, 1,  1])
    sage: Q4_sub.is_maximal()
    True

  """
    Q_maximal = self.maximal_form()
    if self.det() == Q_maximal.det():
        return True
    else:
        return False
```



```
def maximal_form__Deprecated(self, return_transformation=False):
    """
  Return a quadratic form (for a maximal lattice) containing the
  given form.

  If return_transformation is True, then we also return the integral
  linear transformation that achieves this.

  INPUT:

  OUTPUT:
    a quadratic form Q, or a pair (Q,T) where T is a square matrix of size Q.dim()

  EXAMPLES:
    sage: Q3 = DiagonalQuadraticForm(ZZ, [1,1,1])
    sage: Q3.maximal_form()
    Quadratic form in 3 variables over Integer Ring with coefficients:
    [ 2 2 0 ]
    [ * 1 0 ]
    [ * * 1 ]

    sage: Q4 = DiagonalQuadraticForm(ZZ, [1,1,1,1])
    sage: Q4.maximal_form()
    Quadratic form in 4 variables over Integer Ring with coefficients:
    [ 2 2 2 2 ]
    [ * 1 1 1 ]
    [ * * 1 1 ]
    [ * * * 1 ]

    sage: Q1 = DiagonalQuadraticForm(ZZ, [1])
    sage: Q1.maximal_form()
    Quadratic form in 1 variables over Integer Ring with coefficients:
    [ 1 ]

    sage: Q1 = DiagonalQuadraticForm(ZZ, [4])
    sage: Q1.maximal_form()
    Quadratic form in 1 variables over Integer Ring with coefficients:
    [ 1 ]

    sage: Q1 = DiagonalQuadraticForm(ZZ, [18])
    sage: Q1.maximal_form()
    Quadratic form in 1 variables over Integer Ring with coefficients:
    [ 2 ]

  """
    n = self.dim()

    Big_trans = matrix(QQ, n, n, 1)    ## The cumulative transformation to arriv
e at the Watson form B


    ## Compute the exponent of L#/L
    B = self.matrix()

    ## Precompute the exponent/index data once
    ed = B.elementary_divisors()
    max_ed = ed[-1]
    f = max_ed.squarefree_part()
    fa = sqrt(f * max_ed)
    a = fa / f

    #print
    #print "f = ", f
    #print "a = ", a
    #print "fa = ", fa

    ## Try to find the Watson superlattice
    while a != 1:
        B_inv = B.inverse()
```

```
        generator_mat = matrix(ZZ, max_ed * B_inv)
        X = generator_mat.augment(matrix(ZZ, n, n, a))
        gen_lll = X.transpose().LLL().transpose().matrix_from_columns(range(n,2*
n))
        ##assume that the last rows returned by LLL constitue the basis of the l
attice

        B = matrix(ZZ, gen_lll.transpose() * B * gen_lll / (a*a))

        ## Save the transformation matrix
        Big_trans = Big_trans * gen_lll/a

        ## Recompute the index/exponent data for L#/L
        ed = B.elementary_divisors()
        max_ed = ed[-1]

        f = max_ed.squarefree_part()

        ## DIAGNOSTIC
        #print "X = ", X
        #print "gen_lll = ", gen_lll
        verbose("B = " + str(B))
        verbose("")
        verbose("f = " + str(f))

        fa = sqrt(f * max_ed)
        a = fa / f

        ## DIAGNOSTIC
        verbose("a = " + str(a))
        verbose("fa = " + str(fa))



    ## Deal with the 1-dimensional case:
    if n == 1:
        new_coeff = squarefree_part(self[0,0])
        Q_maximal = deepcopy(self)
        Q_maximal.__init__(ZZ, 1, [new_coeff])
        if return_transformation:
            return Q_maximal, Matrix(QQ, 1, 1, [sqrt(self[0,0]/new_coeff)])
        else:
            return Q_maximal




    ## Return the Watson form (for a superlattice)
    verbose("Watson lattice B = " + str(B))
    verbose("Big_trans = " + str(Big_trans))


    ## even_sublattice_of_bilinear_gram_matrix




    ## Deal with the remaining form
    pp1 = prime_divisors(max_ed)
    pp2 = prime_divisors(ed[-2])      ## This is ok, since n>=2 here.
    ##pp2 = [p  for p in prime_divisors(ed[-2])  if p != 2]      ## QUICK FIX TO
 AVOID SAGE ERROR WHEN P=2 -- CHANGE THIS!

    verbose("pp1 = " + str(pp1))
    verbose("pp2 = " + str(pp2))

    B1 = matrix(ZZ, max_ed * B.inverse())     ## This is a fix to deal with inte
ger matrices
```



```
    D, T, S = B1.smith_form()

    ## These columns give a basis for the dual lattice L# mod L.
    dual_columns = B1 * S

    ## Loop through all primes to find a maximal isotropic space for each.
    T_huge = matrix(ZZ,n,n,max_ed)
    for p in pp2:

        ## DIAGNSOTIC
        verbose("p = " + str(p))

        cofacp = max_ed/p
        d_p = n
        for i in range(n):
            if D[i,i] % p == 0:
                d_p = i         ## Dim of L#/L
                break


        dp_cols_small = dual_columns.matrix_from_columns(range(d_p))
        small_gram = matrix(ZZ,dp_cols_small.transpose() * B * dp_cols_small / m
ax_ed)
        ## Divide out by max_ed only once, because we'll need this later.
        ## print "small_gram = ", small_gram
        ## print "small_gram_det = ", small_gram.det()
        ## print "small_gram_ed = ", small_gram.elementary_divisors()
        Tp = find_basis_of_maximal_isotropic_subspace(matrix(GF(p), small_gram))

        ## DIAGNOSTIC
        verbose("Finished finding maximal isotropic subspace at prime " + str(p))

        TZ = dp_cols_small * matrix(ZZ,Tp).transpose()
        T_huge = T_huge.augment(cofacp*TZ)


    ## Return a basis for the maximal form.
    ## print "pre-lll"
    ## print "T_huge has ", T_huge.nrows(), " rows and ", T_huge.ncols(), "colum
ns."
    ## print type(T_huge), T_huge.parent()
    ## print T_huge.rows()
    nr = T_huge.ncols()  ## after LLL the last rows form a basis, the first ones
 are 0
    T_lll = T_huge.transpose().LLL().matrix_from_rows(range(nr-n,nr)).transpose(
)
    ## print "post-lll"
    Gram_of_max_lat = matrix(ZZ, T_lll.transpose() * B * T_lll / (max_ed* max_ed
))


    Gram_new, T_new = even_neighbor_of_bilinear_gram_matrix(Gram_of_max_lat)
    T_newnew = Big_trans * T_lll * T_new / max_ed



    ## Return the appropriate result
    Q_maximal = deepcopy(self)
    Q_maximal.__init__(Gram_new)
    if return_transformation:
        return Q_maximal, T_newnew
    else:
        return Q_maximal



##########################
## COMMENTS -- TO DELETE ##
```

```
################################################################################
###############

    Gram_even,trans_even,is_even = even_sublattice_of_bilinear_gram_matrix(Gram_
of_max_lat)
    if is_even:
        return Gram_of_max_lat, Big_trans * T_lll / max_ed  ## we already have f
ound an even lattice
################################################################################
###############
    ## better way
    # Diagonalise the  Gram_of_max_lat modulo 2 (if this is odd, otherwise we ar
e fine anyhow)
    # Then either the even sublattice is already the maximal even overlattice
    # or this is generated by the even sublattice and 1/2 of the sum of all (n o
r (n-1))
    # basisvectors of odd norm

    Gram_diag,trans_diagmod2,numberofodd = diagonalise_mod_2(Gram_of_max_lat)

        ## should return a unimodular integral matrix s.t. Gram_diag = trans_dia
gmod2.transpose()*Gram_of_max_lat*trans_diagmod2
        ## is a diagonal matrix modulo 2
        ## and an integer numberofodd, such that the first numberofodd vectors i
n trans_diagmod2 have odd norm
        ## the others have even norm (so if det = 2 then numberoofodd = n-1 else
 n )
    oddity = 0
    for i in range(numberofodd):
         oddity += Gram_diag[i,i]

    if oddity % 8 == 0:
        halfcolunms = vector([1  for j in range(numberofodd)] + [0  for j in ran
ge(numberofodd, n)])
        newtrans =  matrix(ZZ,(2*trans_diagmod2).augment(halfcolumns.transpose()
)).transpose()
        newtrans_lll = newtrans.LLL().matrix_from_rows(range(1,n+1)).transpose()


        ## Make the new matrix and transformation for non-zero oddity (mod 8)
        new_A = newtrans_lll.transpose() * Gram_even * newtrans_lll / 4
        new_T = Big_trans * T_lll * trans_even * newtrans_lll / (2* max_ed)
        print "new_A = \n", new_A
        print "new_T = \n", new_T
        return matrix(ZZ, new_A), new_T

    return Gram_even,  Big_trans * T_lll * trans_even / max_ed  ## if oddity is
not zero mod 8 then the even sublattice is already maximal even


################################################################################
###############
################################################################################
###############
```

```
from sage.rings.all import GF
from sage.misc.functional import sqrt

from sage.structure.element import is_Vector

from sage.modules.free_module_element import vector
from sage.rings.all import ZZ, QQ
from copy import deepcopy

from sage.matrix.constructor import Matrix

from sage.quadratic_forms.extras import extend_to_primitive
#from sage.quadratic_forms.maximal_extras import
#    even_sublattice_of_bilinear_gram_matrix, \
#    find_basis_of_maximal_isotropic_subspace, \
#    diagonalise_mod_2, \
#    even_neighbor_of_bilinear_gram_matrix

from sage.matrix.constructor import matrix
#from sage.quadratic_forms.quadratic_form import QuadraticForm    ## This create
s a circular import! =(

from sage.rings.arith import prime_divisors

from sage.quadratic_forms.projective_iterators import normalized_finite_projecti
ve_space_generator









#def genus_representatives(self,  exhaust_neighbor_lattice):
#    """
#    Compute a list of globally inequavalent representatives for the genus of Q.
#
#    THIS SHOULD ONLY
#    """
#    ## Sanity Check:  Base ring = ZZ only!
#    if self.base_ring() != ZZ:
#        raise NotImplementedError, "The genus representatives routine is only i
mplemented over the base ring ZZ."
#
#
#    ## Extend the known representatives with p-neighbor graphs until
#    ## we exhaust the genus (either check that all adjacency row sums
#    ## are of maximal degree, or use the mass formula).
#    pass



def p_neighbors_up_to_isometry(self, p, use_code="find_graph"):
    """
    Returns a list of p-neighbors up to isometry (i.e. integral equivalence).

    QUESTIONS:  Which primes p are allowed?

    INPUT:
        p -- a prime number (with some restrictions???)

    OUTPUT:
        a list of quadratic forms

    EXAMPLES:
```



```
    """
    ## Determine the algorithm to use
    if use_code == "mass_check":
        return self.p_neighbors_up_to_isometry__with_mass_check(p)     ## Defaul
t
    elif use_code == "find_graph":
        return self.p_neighbor_graph(p)[0]
    else:
        raise TypeError, " You entered an invalid use_code string.  Try using 'mass_check' (default) or 'fi
nd_graph' (somewhat slower)."




def p_neighbors_up_to_isometry__with_mass_check(self, p, known_distinct_represen
tatives=[]):
    """
    Compute a graph of the isometry classes of p-neighbors with edges
    given by the p-neighbor relation, allowing loops and multiplicities.


    TO DO:  Fix the mass to give the spinor mass or the sum of to spinor masses!


    INPUT:
        p -- a prime number (currently we require that p not divide the Hessian determinant -- see below.)
        known_distinct_reprsentatives -- these are required to be in different classes (i.e., pairwise globally inequivalent!
).

    OUTPUT:
        a pair (L, A) where L is a list of globally inequivalent quadratic forms,
        and A is the p-neighbor (symmetrix) adjacency matrix (with integer coeffs >= 0).

    EXAMPLES:
        sage: Q = DiagonalQuadraticForm(ZZ, [1, 1, 1])
        sage: Q.p_neighbors_up_to_isometry__with_mass_check(3)
        [Quadratic form in 3 variables over Integer Ring with coefficients:
        [ 1 0 0 ]
        [ * 1 0 ]
        [ * * 1 ]
        ]

    """
    ## Setup the initial neighbor list
    if known_distinct_representatives == []:
        Neighbor_list = [self]
    else:
        Neighbor_list = deepcopy(known_distinct_representatives)     ## Note: We
 allow known representatives for genus and spinor genus enumeration.
    N_len = len(Neighbor_list)
    n = self.dim()


    ## Setup Inital Mass info (TO DO: Presently we are incorrectly assuming that
 the total_neighbor_mass is the genus mass)
    total_neighbor_mass = self.conway_mass()
    auto_list = [ZZ(1) / Q.number_of_automorphisms()  for Q in Neighbor_list]
    partial_neighbor_mass = sum(auto_list)


    ## Setup the initial p-neighbor multiplicity
    if self.det() % p != 0:
        p_neighbor_multiplicity = ZZ(p**(n-1) - 1) / ZZ(p-1)
    else:
        raise NotImplementedError, "We need to find the p_neighbor_multiplicity from the dimension
of the maximal non-degernrate subspace of Q over F_p."
        p_neighbor_multiplicity = -1    ## TO DO: WE NEED TO SET THIS!!!!!!!!!!
```

```
!!!!!!!!!!!!!!



    ## Run through all neighbors of all lattices until we're done!
    i = 0
    while i < len(Neighbor_list):

        ## Set the starting form and run through all of its neighbors
        Q_start = Neighbor_list[i]
        num_of_i_neighbors_found = 0

        ## If there are no more neighbors to find, then move to the next form to
 look for neighbors
        if True:
            for v in normalized_finite_projective_space_generator(Q_start.dim(),
 p):
                try:
                    Q_neighbor = Q_start.compute_p_neighbor_from_vector(p, v)

                    ## Run through all neighbors looking for an isometry
                    new_flag = True
                    num_of_i_neighbors_found += 1
                    for j in range(len(Neighbor_list)):

                        ## Record any isometries with known forms  (in the upper
 triangular entries of A)
                        if Q_neighbor.is_globally_equivalent_to(Neighbor_list[j]
):
                            new_flag = False
                            break              ## Don't test past one successful
 isometry

                    ## If this form is new, record it and augment and increment
the adjacency matrix (only upper-triangular entries)
                    if new_flag:
                        Neighbor_list.append(Q_neighbor)
                        auto_list.append(ZZ(1) / Q_neighbor.number_of_automorphi
sms())
                        partial_neighbor_mass = sum(auto_list)

                    ## If there are no more neighbors to find, then move to the
next form to look for neighbors
                    if num_of_i_neighbors_found == p_neighbor_multiplicity:
                        break

                    ## If we have all representatives of this neighbor graph, th
en return with the representatives
                    if total_neighbor_mass == partial_neighbor_mass:
                        return Neighbor_list
                    elif (total_neighbor_mass < partial_neighbor_mass):
                        raise RuntimeError, "There is a problem with the mass formula -- we ha
ve more than the allowed mass!"

                ## Move to the next vector if we don't have a neighbor from v
                except AttributeError:
                    pass


        ## Increment the starting form
        i += 1
        #print " len(Neighbor_list) = ", len(Neighbor_list)
        #print "===============  Finishing i = ", i, " =================="



    ## Return the list of neighbors and the adjacency matrix
```



```
    return Neighbor_list








def p_neighbor_graph(self, p, known_distinct_representatives=[], ignore_multipli
city_check=False):
    """
  Compute a graph of the isometry classes of p-neighbors with edges
  given by the p-neighbor relation, allowing loops and multiplicities.

  INPUT:
    p -- a prime number (currently we require that p not divide the Hessian determinant -- see below.)
    known_distinct_reprsentatives -- these are required to be in different classes (i.e., pairwise globally inequivalent!
).

  OUTPUT:
    a pair (L, A) where L is a list of globally inequivalent quadratic forms,
    and A is the p-neighbor (symmetrix) adjacency matrix (with integer coeffs >= 0).

  EXAMPLES:
    sage: Q = DiagonalQuadraticForm(ZZ, [1, 1, 1])
    sage: Q.p_neighbor_graph(3)
    ([Quadratic form in 3 variables over Integer Ring with coefficients:
    [ 1 0 0 ]
    [ * 1 0 ]
    [ * * 1 ]
    ],
     [4])

    sage: Q = DiagonalQuadraticForm(ZZ, [1, 48, 144])
    sage: N5, A5 = Q.p_neighbor_graph(5,  ignore_multiplicity_check=True)
    sage: len(N5)
    4
    sage: A5
    [0 2 4 0]
    [2 0 0 4]
    [4 0 0 2]
    [0 4 2 0]
    sage: N7, A7 = Q.p_neighbor_graph(7,  ignore_multiplicity_check=True)
    sage: len(N7)
    2
    sage: A7
    [4 4]
    [4 4]

  """
    ## Setup the initial neighbor list and adjacency matrix
    if known_distinct_representatives == []:
        Neighbor_list = [self]
    else:
        Neighbor_list = deepcopy(known_distinct_representatives)      ## Note: We
 allow known representatives for genus and spinor genus enumeration.
    N_len = len(Neighbor_list)
    A = Matrix(ZZ, N_len, N_len, 0)            ## Adjacency_matrix
    small_zero_matrix = Matrix(ZZ, 1, 1, 0)


    ## Find the p-neighbor multiplicity (if we use it)
    n = self.dim()
    if not ignore_multiplicity_check:
        if self.det() % p != 0:
            p_neighbor_multiplicity = ZZ(p**(n-1) - 1) / ZZ(p-1)
        else:
```

```
            raise NotImplementedError, "We need to find the p_neighbor_multiplicity from the dime
nsion of the maximal non-degernrate subspace of Q over F_p."
            p_neighbor_multiplicity = -1    ## TO DO: WE NEED TO SET THIS!!!!!!!
!!!!!!!!!!!!!!!!!


    ## Run through all neighbors of all lattices until we're done!
    i = 0
    while i < len(Neighbor_list):

        ## Set the starting form and run through all of its neighbors
        Q_start = Neighbor_list[i]
        if not ignore_multiplicity_check:
            i_multiplicity_sum = sum([A[i, s]  for s in range(A.ncols())])
            #print "p_neighbor_mutliplicity = ", p_neighbor_multiplicity
            #print "i_mutliplicity_sum = ", i_multiplicity_sum


        ## If there are no more neighbors to find, then move to the next form to
 look for neighbors
        if ignore_multiplicity_check or ((not ignore_multiplicity_check) and (i_
multiplicity_sum != p_neighbor_multiplicity)):
            for v in normalized_finite_projective_space_generator(Q_start.dim(),
 p):
                try:
                    Q_neighbor = Q_start.compute_p_neighbor_from_vector(p, v)

                    ## Run through all neighbors looking for an isometry
                    new_flag = True
                    for j in range(len(Neighbor_list)):

                        print "Q_neighbor = " + str(Q_neighbor)
                        print "Neighbor_list[j] = " + str(Neighbor_list[j])
                        print "\n\n"

                        ## Record any isometries with known forms  (in the upper
 triangular entries of A)
                        if Q_neighbor.is_globally_equivalent_to(Neighbor_list[j]
):
                            new_flag = False
                            if i <= j:
                                if not ignore_multiplicity_check:
                                    i_multiplicity_sum += 1          ## Count on
ly the new neighbor edges
                                A[i, j] += 1
                            break              ## Don't test past one successful
 isometry

                    ## If this form is new, record it and augment and increment
the adjacency matrix (only upper-triangular entries)
                    if new_flag:
                        Neighbor_list.append(Q_neighbor)
                        A = A.block_sum(small_zero_matrix)
                        if not ignore_multiplicity_check:
                            i_multiplicity_sum += 1
                        A[i, -1] += 1


                    if not ignore_multiplicity_check:
                        #print "p_neighbor_mutliplicity = ", p_neighbor_multipli
city
                        #print "i_mutliplicity_sum = ", i_multiplicity_sum
                        pass


                    ## If there are no more neighbors to find, then move to the
next form to look for neighbors
                    if (not ignore_multiplicity_check) and (i_multiplicity_sum =
= p_neighbor_multiplicity):
```



```
                        break

                ## Move to the next vector if we don't have a neighbor from v
                except AttributeError:
                    pass


        ## Increment the starting form
        i += 1
        #print " len(Neighbor_list) = ", len(Neighbor_list)
        #print "===============  Finishing i = ", i, " ================="


    ## Symmetrize the (presently upper-triangular) adjacency matrix
    for i in range(1, A.nrows()):
        for j in range(i):
            A[i,j] = A[j,i]


    ## Return the list of neighbors and the adjacency matrix
    return Neighbor_list, A




def compute_p_neighbor_once(self, p):
    """
  Compute exactly one p-neighbor of the given quadratic form.

  RESTRICTIONS ON p???

  """
    ## TO DO: ADD SANITY CHECKS FOR THE PRIME p.


    ## Look for the first p-neighbor form
    for v in normalized_finite_projective_space_generator(self.dim(), p):

        ## Try to compute a p-neighboring form from the vector v
        try:
            Q_neighbor = self.compute_p_neighbor_from_vector(p, v)

        ## Move to the next vector if we don't have a neighbor from v
        except AttributeError:
            Q_neighbor = None

        ## Return the p-neighbor if we have one
        if Q_neighbor != None:
            return Q_neighbor




## ---------------------------------------------------------------------------
----------------------------------------


## Routines from the paper of Scharlau/Hempkemeier for finding neighbors.
```

```
def compute_p_neighbors(self, p, return_vectors=False):
    """
  Compute a list of all the (p^(n-1) - 1) p-neighbors of the
  current quadratic form.

  return_vectors -- boolean -- returns pairs [Q', v'] where v' is the normalized projective vector giving the p-neighb
or Q' of Q.


  TO DO: If a neighbor_index >0 and < p^n is passed, then the
  p-neighbor associated to the staring with the vector

  EXAMPLES:
  """
    ## Make a list of p-neighbors naively, excluding those vectors which produce
 bad p-neighbor lattices.
    Neighbor_list = []
    for v in normalized_finite_projective_space_generator(self.dim(), p):
        try:
            Q_neighbor = self.compute_p_neighbor_from_vector(p, v)

            if return_vectors:
                Neighbor_list.append([Q_neighbor, v])
            else:
                Neighbor_list.append(Q_neighbor)
        except AttributeError:
            pass

    return Neighbor_list




def compute_p_neighbor_from_vector(self, p, v, return_transformation=False):
    """
  Compute the p-neighbor of the current quadratic form associated to
  the vector v, assuming that:

    - p^2 divides Q(v)
    - v is not in p * (dual lattice of Z^n w.r.t. the Hessian bilinear form)

  (Note: A stronger condition is that p doesn't divide the Hessian determinant of Q)


  INPUT:
    return_transformation -- boolean -- returns the linear transformation.


  TO DO: If a neighbor_index >0 and < p^n is passed, then the
  p-neighbor associated to the staring with the vector

  EXAMPLES:
  """
    Q = self
    n = Q.dim()
    M = Q.matrix()


    ## Sanity Checks:
    ## --------------
    #print
    #print "Checking the vector v = ", v
    #print "Q(v) = ", Q(v)

    ## TO DO: Perhaps check for primitivity or rescale the form?
```



```
    ## Check if Q(v) = 0 (mod p) and <v,.> is not identically zero (mod p)
    ## (i.e., it is non-singular on the associated projective conic)
    zero_vec = vector(n * [ZZ(0)])
    if (Q(v) % p == 0) and ((vector(v) * M) % p != zero_vec):
        #print "Found Q(v) = 0 (mod p) which is non-singular on the conic"
        pass
    else:
        raise AttributeError, "The vector v isn't non-singular on the conic mod p."


    ######################
    ## This isn't required -- we can arrange for this (when p > 2?).
    ###############################################################################
##############################
    ## Check that Q(v) = 0 (mod p^2), which is required for
    ## integrality of the p-neighbor lattice.
    #if Q(v) % p**2 == 0:
    #    print "Q(v) = 0 (mod p^2), so the associated p-neighbor will be integer
-valued."
    #else:
    #    raise AttributeError, "Q(v) = 0 (mod p^2), so the associated p-neighbor
 won't be integer-valued."
    ###############################################################################
##############################




    ## STEP 1: Adjust the vector v to be isotropic mod p^2
    ## ---------------------------------------------------

    ## Find the index v_ind of the first non-zero coefficient of v (which should
 be 1 by our normalization convention)
    for i in range(n):
        if v[i] % p != 0:
            if v[i] != 1:           ## This is just a safety check -- not needed
at all.
                raise RuntimeError, "Something is very wrong with the normalized vector "+ str(
v) + \
                      " -- it's first non-zero coeff should be 1."
            v_ind = i
            break

    ## Find a basis vector e_m which has non-zero pairing with v (mod p)
    v_pairing_vec = vector(v) * M
    for i in range(n):
        if (i != v_ind) and (v_pairing_vec[i] % p != 0):
            v_pairing_non_singular_index = i
            v_pairing_non_zero_basis_vec = vector([0  for j in range(i)] + [1] +
 [0  for j in range(i+1, n)])
            vb_coeff = v_pairing_vec[i]
            break

    #print "v_pairing_vec = ", v_pairing_vec
    #print "v_pairing_non_singular_index = ", v_pairing_non_singular_index
    #print "v_pairing_non_zero_basis_vec = ", v_pairing_non_zero_basis_vec
    #print "vb_coeff = ", vb_coeff




    ## Adjust v (to a new vector v1) so that Q(v1) = 0 (mod p^2)
    ## by adding a multiple of an independent basis vector
    ## whose bilinear pairing with v doesn't vanish
    c = -Q(v) / (p * vb_coeff) % p
    v1 = v + p * c * v_pairing_non_zero_basis_vec
```

```
    #print "v1 = ", v1
    #print "Q(v1) = ", Q(v1)


    ## Check that Q(v1) = 0 (mod p^2)
    if Q(v1) % p*p != 0:
        raise RuntimeError, "The adjusted vector v1 is not isotropic mod p^2!"


    ## STEP 2: Compute the associated p-neighbor
    ## -----------------------------------------

    ## Find the associated (non-integral) change of basis matrix
    X = matrix(self.base_ring().fraction_field(), n, n, 1)
    ns_ind = v_pairing_non_singular_index
    for i in range(n):
        if i == v_ind:
            X.set_column(i, v1/p)     ## Scale v1 by 1/p
        elif i == ns_ind:
            X[i,i] = p                ## Scale the dual basis vector to v1 by p
        else:
            X[ns_ind,i] = -v_pairing_vec[i] / v_pairing_vec[ns_ind] % p     ## A
rrange for an orthogonal basis to v (mod p) at most vectors.

    #print "X = ", X


    ## Change basis to find the assiciated p-neighbor
    #Q1 = Q((p * X).inverse())
    Q1 = Q(p * X)
    Q2 = Q1.scale_by_factor(1/(p*p))      ## TO DO:  Simplify this code when the
 coefficient_ring is separated from the equivalence_ring
    #print
    #print "Found the neighbor: ", Q2
    #print


    ## Return the p-neighbor
    return Q2
```

```
"""
Quadratic Forms Overview

AUTHORS:

– Jon Hanke (2007–06–19)
– Anna Haensch (2010–07–01): Formatting and ReSTification
"""

#*****************************************************************************
#       Copyright (C) 2007 William Stein and Jonathan Hanke
#
#  Distributed under the terms of the GNU General Public License (GPL)
#
#    This code is distributed in the hope that it will be useful,
#    but WITHOUT ANY WARRANTY; without even the implied warranty of
#    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
#    General Public License for more details.
#
#  The full text of the GPL is available at:
#
#                  http://www.gnu.org/licenses/
#*****************************************************************************

from warnings import warn
from copy import deepcopy

from sage.matrix.constructor import matrix
from sage.matrix.matrix_space import MatrixSpace
from sage.matrix.matrix import Matrix, is_Matrix
from sage.rings.integer import is_Integer
from sage.rings.integer_ring import IntegerRing, ZZ
from sage.rings.rational_field import RationalField, QQ
from sage.rings.ring import Ring
from sage.misc.functional import ideal, denominator, is_even, is_field
from sage.rings.arith import GCD, LCM, valuation, kronecker_symbol
from sage.rings.fraction_field import FractionField
from sage.rings.principal_ideal_domain import is_PrincipalIdealDomain
from sage.rings.ring import is_Ring
from sage.matrix.matrix import is_Matrix
from sage.structure.element import is_Vector

from sage.quadratic_forms.quadratic_form__evaluate import QFEvaluateVector, QFEv
aluateMatrix



def QuadraticForm__constructor(R, n=None, entries=None):
    """
    Wrapper for the QuadraticForm class constructor.  This is meant
    for internal use within the QuadraticForm class code only.  You
    should instead directly call QuadraticForm().

    EXAMPLES::

        sage: from sage.quadratic_forms.quadratic_form import QuadraticForm__constructor
        sage: QuadraticForm__constructor(ZZ, 3)  ## Makes a generic quadratic form over the integers
        Quadratic form in 3 variables over Integer Ring with coefficients:
        [ 0 0 0 ]
        [ * 0 0 ]
        [ * * 0 ]

    """
    return QuadraticForm(R, n, entries)


def is_QuadraticForm(Q):
    """
    Determines if the object Q is an element of the QuadraticForm class.
```



```
    EXAMPLES::

        sage: Q = QuadraticForm(ZZ, 2, [1,2,3])
        sage: is_QuadraticForm(Q)  ##random -- deprecated
        True
        sage: is_QuadraticForm(2)  ##random -- deprecated
        False

    """
    return isinstance(Q, QuadraticForm)



class QuadraticForm():
    r"""
    The ''QuadraticForm'' class represents a quadratic form in n variables with
    coefficients in the ring R.

    INPUT:

    The constructor may be called in any of the following ways.

    #. ''QuadraticForm(R, n, entries)'', where

        – 'R' -- ring for which the quadratic form is defined
        – 'n' -- an integer >= 0
        – ''entries'' -- a list of 'n(n+1)/2' coefficients of the quadratic form
          in 'R' (given lexographically, or equivalently, by rows of the matrix)

    #. ''QuadraticForm(R, n)'', where

        – 'R' -- a ring
        – 'n' -- a symmetric 'n \times n' matrix with even diagonal (relative to
          'R')

    #. ''QuadraticForm(R)'', where

        – 'R' -- a symmetric 'n \times n' matrix with even diagonal (relative to
          its base ring)

    If the keyword argument ''unsafe_initialize'' is True, then the subsequent
    fields may by used to force the external initialization of various fields
    of the quadratic form. Currently the only fields which can be set are:

    – ''number_of_automorphisms''
    – ''determinant''


    OUTPUT:

    quadratic form

    EXAMPLES::

        sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
        sage: Q
        Quadratic form in 3 variables over Integer Ring with coefficients:
        [ 1 2 3 ]
        [ * 4 5 ]
        [ * * 6 ]

    ::

        sage: Q = QuadraticForm(QQ, 3, [1,2,3,4/3 ,5,6])
        sage: Q
        Quadratic form in 3 variables over Rational Field with coefficients:
        [ 1 2 3 ]
        [ * 4/3 5 ]
```

```
    [ * * 6 ]
    sage: Q[0,0]
    1
    sage: Q[0,0].parent()
    Rational Field

::

    sage: Q = QuadraticForm(QQ, 7, range(28))
    sage: Q
    Quadratic form in 7 variables over Rational Field with coefficients:
    [ 0 1 2 3 4 5 6 ]
    [ * 7 8 9 10 11 12 ]
    [ * * 13 14 15 16 17 ]
    [ * * * 18 19 20 21 ]
    [ * * * * 22 23 24 ]
    [ * * * * * 25 26 ]
    [ * * * * * * 27 ]

::

    sage: Q = QuadraticForm(QQ, 2, range(1,4))
    sage: A = Matrix(ZZ,2,2,[-1,0,0,1])
    sage: Q(A)
    Quadratic form in 2 variables over Rational Field with coefficients:
    [ 1 -2 ]
    [ * 3 ]

::

    sage: m = matrix(2,2,[1,2,3,4])
    sage: m + m.transpose()
    [2 5]
    [5 8]
    sage: QuadraticForm(m + m.transpose())
    Quadratic form in 2 variables over Integer Ring with coefficients:
    [ 1 5 ]
    [ * 4 ]

::

    sage: QuadraticForm(ZZ, m + m.transpose())
    Quadratic form in 2 variables over Integer Ring with coefficients:
    [ 1 5 ]
    [ * 4 ]

::

    sage: QuadraticForm(QQ, m + m.transpose())
    Quadratic form in 2 variables over Rational Field with coefficients:
    [ 1 5 ]
    [ * 4 ]
"""

    ## Import specialized methods:
    ## ---------------------------

    ## Routines to compute the p-adic local normal form
    from sage.quadratic_forms.quadratic_form__local_normal_form import \
            find_entry_with_minimal_scale_at_prime, \
            local_normal_form, \
            jordan_blocks_by_scale_and_unimodular, \
            jordan_blocks_in_unimodular_list_by_scale_power

    ## Routines to perform elementary variable substitutions
    from sage.quadratic_forms.quadratic_form__variable_substitutions import \
            swap_variables, \
            multiply_variable, \
            divide_variable, \
            scale_by_factor, \
            extract_variables, \
            elementary_substitution, \
            add_symmetric

    ## Routines to compute p-adic field invariants
    from sage.quadratic_forms.quadratic_form__local_field_invariants import \
            rational_diagonal_form, \
            signature_vector, \
            signature, \
            hasse_invariant, \
            hasse_invariant__OMeara, \
            is_hyperbolic, \
            is_anisotropic, \
            is_isotropic, \
            anisotropic_primes, \
            compute_definiteness, \
            compute_definiteness_string_by_determinants, \
            is_positive_definite, \
            is_negative_definite, \
            is_indefinite, \
            is_definite

    ## Routines to compute local densities by the reduction procedure
    from sage.quadratic_forms.quadratic_form__local_density_congruence import \
            count_modp_solutions__by_Gauss_sum, \
            local_good_density_congruence_odd, \
            local_good_density_congruence_even, \
            local_good_density_congruence, \
            local_zero_density_congruence, \
            local_badI_density_congruence, \
            local_badII_density_congruence, \
            local_bad_density_congruence, \
            local_density_congruence, \
            local_primitive_density_congruence

    ## Routines to compute local densities by counting solutions of various type
s
    from sage.quadratic_forms.quadratic_form__count_local_2 import \
            count_congruence_solutions_as_vector, \
            count_congruence_solutions, \
            count_congruence_solutions__good_type, \
            count_congruence_solutions__zero_type, \
            count_congruence_solutions__bad_type, \
            count_congruence_solutions__bad_type_I, \
            count_congruence_solutions__bad_type_II

    ## Routines to be called by the user to compute local densities
    from sage.quadratic_forms.quadratic_form__local_density_interfaces import \
            local_density, \
            local_primitive_density

    ## Routines for computing with ternary forms
    from sage.quadratic_forms.quadratic_form__ternary_Tornaria import \
            disc, \
            content, \
            adjoint, \
            antiadjoint, \
            is_adjoint, \
            reciprocal, \
            omega, \
            delta, \
            level__Tornaria, \
            discrec, \
            hasse_conductor, \
            clifford_invariant, \
            clifford_conductor, \
            basiclemma, \
            basiclemmavec, \
```

```
            xi, \
            xi_rec, \
            lll, \
            representation_number_list, \
            representation_vector_list, \
            is_zero, \
            is_zero_nonsingular, \
            is_zero_singular

    ## Routines to compute the theta function
    from sage.quadratic_forms.quadratic_form__theta import \
            theta_series, \
            theta_series_degree_2, \
            theta_by_pari, \
            theta_by_cholesky

    ## Routines to compute the product of all local densities
    from sage.quadratic_forms.quadratic_form__siegel_product import \
            siegel_product


    ## Routines to compute genus and spinor genus representatives
    from sage.quadratic_forms.quadratic_form__genus_enumeration import \
            class_number, \
            genus_representatives, \
            spinor_genus_representatives


    ## Routines to compute p-neighbors
    from sage.quadratic_forms.quadratic_form__neighbors import \
            p_neighbors_up_to_isometry, \
            p_neighbors_up_to_isometry__with_mass_check, \
            p_neighbor_graph, \
            compute_p_neighbor_once, \
            compute_p_neighbors, \
            compute_p_neighbor_from_vector

    ## Routines to reduce a given quadratic form
    from sage.quadratic_forms.quadratic_form__reduction_theory import \
            LLL_reduced, \
            reduced_binary_form1, \
            reduced_ternary_form__Dickson, \
            reduced_binary_form, \
            minkowski_reduction, \
            minkowski_reduction_for_4vars__SP

    ## Wrappers for Conway-Sloane genus routines (in ./genera/)
    from sage.quadratic_forms.quadratic_form__genus import \
            global_genus_symbol, \
            local_genus_symbol, \
            CS_genus_symbol_list


    ## Routines to compute local masses for ZZ.
    from sage.quadratic_forms.quadratic_form__mass import \
            has_class_number_one, \
            shimura_mass__maximal, \
            GHY_mass__maximal
    from sage.quadratic_forms.quadratic_form__mass__Siegel_densities import \
            mass__by_Siegel_densities, \
            Pall_mass_density_at_odd_prime, \
            Watson_mass_at_2, \
            Kitaoka_mass_at_2, \
            mass_at_two_by_counting_mod_power
    from sage.quadratic_forms.quadratic_form__mass__Conway_Sloane_masses import
\
            parity, \
            is_even, \
            is_odd, \
```



```
            conway_species_list_at_odd_prime, \
            conway_species_list_at_2, \
            conway_octane_of_this_unimodular_Jordan_block_at_2, \
            conway_diagonal_factor, \
            conway_cross_product_doubled_power, \
            conway_type_factor, \
            conway_p_mass, \
            conway_standard_p_mass, \
            conway_standard_mass, \
            conway_mass
#           conway_generic_mass, \
#           conway_p_mass_adjustment

    ## Routines to compute maximally integral quadratic form
#    from sage.quadratic_forms.quadratic_form__maximal import \
#           is_maximal, \
#           maximal_form

    ## Routines to check local representability of numbers
    from sage.quadratic_forms.quadratic_form__local_representation_conditions im
port \
            local_representation_conditions, \
            is_locally_universal_at_prime, \
            is_locally_universal_at_all_primes, \
            is_locally_universal_at_all_places, \
            is_locally_represented_number_at_place, \
            is_locally_represented_number

    ## Routines to make a split local covering of the given quadratic form.
    from sage.quadratic_forms.quadratic_form__split_local_covering import \
            cholesky_decomposition, \
            vectors_by_length, \
            complementary_subform_to_vector, \
            split_local_cover

    ## Routines to make automorphisms of the given quadratic form.
    from sage.quadratic_forms.quadratic_form__automorphisms import \
            basis_of_short_vectors, \
            short_vector_list_up_to_length, \
            short_primitive_vector_list_up_to_length, \
            automorphisms, \
            number_of_automorphisms, \
            number_of_automorphisms__souvigner, \
            set_number_of_automorphisms

    ## Routines to test the local and global equivalence/isometry of two quadrat
ic forms.
    from sage.quadratic_forms.quadratic_form__equivalence_testing import \
            is_globally_equivalent__souvigner, \
            is_globally_equivalent_to, \
            is_locally_equivalent_to, \
            has_equivalent_Jordan_decomposition_at_prime




    def __init__(self, R, n=None, entries=None, unsafe_initialization=False, num
ber_of_automorphisms=None, \
                 determinant=None, init_from_gram_matrix=False):
        """
    EXAMPLES::

      sage: s = QuadraticForm(ZZ, 4, range(10))
      sage: s == loads(dumps(s))
      True


      sage: QuadraticForm(QQ, Matrix(ZZ, 2, 2, 1), init_from_gram_matrix=False)
      Quadratic form in 2 variables over Rational Field with coefficients:
```

```
[ 1/2 0 ]
[ * 1/2 ]

sage: QuadraticForm(QQ, Matrix(ZZ, 2, 2, 1), init_from_gram_matrix=True)
Quadratic form in 2 variables over Rational Field with coefficients:
[ 1 0 ]
[ * 1 ]

sage: QuadraticForm(GF(2), Matrix(ZZ, 2, 2, 1), init_from_gram_matrix=True)
Quadratic form in 2 variables over Finite Field of size 2 with coefficients:
[ 1 0 ]
[ * 1 ]

    """
        ## Deal with:  QuadraticForm(ring, matrix)
        matrix_init_flag = False
        if isinstance(R, Ring):
            if is_Matrix(n):
                ## Test if n is symmetric and has even diagonal
                if not init_from_gram_matrix and not self._is_even_symmetric_mat
rix_(n, R):
                    raise TypeError, "Oops!  The matrix is not a symmetric with even diagonal defin
ed over R."

                ## Rename the matrix and ring
                M = n
                M_ring = R
                matrix_init_flag = True


        ## Deal with:  QuadraticForm(matrix)
        if is_Matrix(R) and (n == None):

            ## Test if R is symmetric and has even diagonal (when appropriate)
            if not init_from_gram_matrix and not self._is_even_symmetric_matrix_
(R):
                raise TypeError, "Oops!  The matrix is not a symmetric with even diagonal."

            ## Rename the matrix and ring
            M = R
            M_ring = R.base_ring()
            matrix_init_flag = True

        ## Perform the quadratic form initialization
        if matrix_init_flag == True:
            self.__n = M.nrows()
            self.__base_ring = M_ring
            self.__coeffs = []


            ## Usual initialize from Hessian matrix
            if not init_from_gram_matrix:
                for i in range(M.nrows()):
                    for j in range(i, M.nrows()):
                        if (i == j):
                            self.__coeffs += [ M_ring(M[i,j] / 2) ]
                        else:
                            self.__coeffs += [ M_ring(M[i,j]) ]

            ## The occasional initialize from a Gram matrix
            else:
                for i in range(M.nrows()):
                    for j in range(i, M.nrows()):
                        if (i == j):
                            self.__coeffs += [ M_ring(M[i,j]) ]
                        else:
                            self.__coeffs += [ M_ring(M[i,j] * 2) ]

            return
```



```
        ## ---------------------------------------------------------

        ## Verify the size of the matrix is an integer >= 0
        try:
            n = int(n)
        except:
            raise TypeError, "Oops! The size " + str(n) + " must be an integer."
            if (n < 0):
                raise TypeError, "Oops! The size " + str(n) + " must be a non-negative integer
."

        ## TODO: Verify that R is a ring...

        ## Store the relevant variables
        N = int(n*(n+1))/2
        self.__n = int(n)
        self.__base_ring = R
        self.__coeffs = [self.__base_ring(0)  for i in range(N)]

        ## Check if entries is a list for the current size, and if so, write the
 upper-triangular matrix
        if isinstance(entries, list) and (len(entries) == N):
            for i in range(N):
                self.__coeffs[i] = self.__base_ring(entries[i])
        elif (entries != None):
            raise TypeError, "Oops! The entries " + str(entries) + "must be a list of size n(n+
1)/2."

        ## ---------------------------------------------------------

        ## Process possible forced initialization of various fields
        self._external_initialization_list = []
        if unsafe_initialization:

            ## Set the number of automorphisms
            if number_of_automorphisms != None:
                self.set_number_of_automorphisms(number_of_automorphisms)
                #self.__number_of_automorphisms = number_of_automorphisms
                #self.__external_initialization_list.append('number_of_automorph
isms')

            ## Set the determinant
            if determinant != None:
                self.__det = determinant
                self._external_initialization_list.append('determinant')


    def list_external_initializations(self):
        """
    Returns a list of the fields which were set externally at
    creation, and not created through the usual QuadraticForm
    methods.  These fields are as good as the external process
    that made them, and are thus not guaranteed to be correct.

    EXAMPLES::

      sage: Q = QuadraticForm(ZZ, 2, [1,0,5])
      sage: Q.list_external_initializations()
      []
      sage: T = Q.theta_series()
      sage: Q.list_external_initializations()
      []
      sage: Q = QuadraticForm(ZZ, 2, [1,0,5], unsafe_initialization=False, number_of_automorphisms=3, determinan
t=0)
      sage: Q.list_external_initializations()
      []

    ::
```

```
        sage: Q = QuadraticForm(ZZ, 2, [1,0,5], unsafe_initialization=False, number_of_automorphisms=3, determinan
t=0)
        sage: Q.list_external_initializations()
        []
        sage: Q = QuadraticForm(ZZ, 2, [1,0,5], unsafe_initialization=True, number_of_automorphisms=3, determinant
=0)
        sage: Q.list_external_initializations()
        ['number_of_automorphisms', 'determinant']
    """
        return deepcopy(self._external_initialization_list)


    def _pari_(self):
        """
    Return a pari-formatted Hessian matrix for Q.

    EXAMPLES::

        sage: Q = QuadraticForm(ZZ, 2, [1,0,5])
        sage: Q._pari_()
        [2, 0; 0, 10]

    """
        return self.matrix()._pari_()


    def __repr__(self):
        """
    Give a text representation for the quadratic form given as an upper-triangular matrix of coefficients.

    EXAMPLES::

        sage: QuadraticForm(ZZ, 2, [1,3,5])
        Quadratic form in 2 variables over Integer Ring with coefficients:
        [ 1 3 ]
        [ * 5 ]

    """
        n = self.dim()
        out_str = "Quadratic form in " + str(n) + " variables over " + str(self.base_ring()
) + " with coefficients: \n"
        for i in range(n):
            out_str += "[ "
            for j in range(n):
                if (i > j):
                    out_str += "* "
                else:
                    out_str += str(self[i,j]) + " "
            out_str += "]\n"
        return out_str


    def _latex_(self):
        """
    Give a LaTeX representation for the quadratic form given as an upper-triangular matrix of coefficients.

    EXAMPLES::

        sage: Q = QuadraticForm(ZZ, 2, [2,3,5])
        sage: Q._latex_()
        'Quadratic form in 2 variables over Integer Ring with coefficients: \\newline\\left[ \\begin{array}{cc}2 & 3 &
* & 5 & \\end{array} \\right]'

    """
        n = self.dim()
        out_str = ""
        out_str += "Quadratic form in " + str(n) + " variables over " + str(self.base_ring(
))
```



```
        out_str += " with coefficients: \\newline"
        out_str += "\\left[ \\begin{array}{" + n * "c" + "}"
        for i in range(n):
            for j in range(n):
                if (i > j):
                    out_str += " * & "
                else:
                    out_str += str(self[i,j]) + " & "
#            if i < (n-1):
#                out_str += "\\"
        out_str += "\\end{array} \\right]"
        return out_str



    def __getitem__(self, ij):
        """
    Return the coefficient 'a_{ij}' of 'x_i * x_j'.

    EXAMPLES::

        sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
        sage: matrix(ZZ, 3, 3, [Q[i,j]  for i in range(3) for j in range(3)])
        [1 2 3]
        [2 4 5]
        [3 5 6]

    """
        ## Unpack the list of indices
        i, j =  ij
        i = int(i)
        j = int(j)

        ## Ensure we're using upper-triangular coordinates
        if i > j:
            tmp = i
            i = j
            j = tmp

        return self.__coeffs[i*self.__n - i*(i-1)/2 + j - i]


    def __setitem__(self, ij, coeff):
        """
    Set the coefficient 'a_{ij}' in front of 'x_i * x_j'.

    EXAMPLES::

        sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
        sage: Q
        Quadratic form in 3 variables over Integer Ring with coefficients:
        [ 1 2 3 ]
        [ * 4 5 ]
        [ * * 6 ]
        sage: Q[2,1] = 17
        sage: Q
        Quadratic form in 3 variables over Integer Ring with coefficients:
        [ 1 2 3 ]
        [ * 4 17 ]
        [ * * 6 ]

    """
        ## Unpack the list of indices
        i, j =  ij
        i = int(i)
        j = int(j)

        ## TO DO:  Verify that 0 <= i, j <= (n-1)
```

```
        ## Ensure we're using upper-triangular coordinates
        if i > j:
            tmp = i
            i = j
            j = tmp

        ## Set the entry
        try:
            self.__coeffs[i*self.__n - i*(i-1)/2 + j -i] = self.__base_ring(coef
f)
        except:
            raise RuntimeError, "Oops!  This coefficient can't be coerced to an element of the base ring
 for the quadratic form."


    def __eq__(self, other):
        """
    Determines if two quadratic forms are equal.

    EXAMPLES::

      sage: Q = QuadraticForm(ZZ, 2, [1,4,10])
      sage: Q == Q
      True

      sage: Q1 = QuadraticForm(QQ, 2, [1,4,10])
      sage: Q == Q1
      False

      sage: Q2 = QuadraticForm(ZZ, 2, [1,4,-10])
      sage: Q == Q1
      False
      sage: Q == Q2
      False
      sage: Q1 == Q2
      False

    """
        if not isinstance(other, QuadraticForm):
            return False
        return (self.__base_ring == other.__base_ring) and (self.__coeffs == oth
er.__coeffs)



    def __ne__(self, other):
        """
    Determines if two quadratic forms are not equal.

    INPUT:
      other -- a quadratic form

    OUTPUT:
      boolean

    EXAMPLES:
      sage: Q = QuadraticForm(ZZ, 2, [1,4,10])
      sage: Q != Q
      False

      sage: Q1 = QuadraticForm(QQ, 2, [1,4,10])
      sage: Q != Q1
      True

      sage: Q2 = QuadraticForm(ZZ, 2, [1,4,-10])
      sage: Q != Q1
      True
      sage: Q != Q2
      True
```



```
      sage: Q1 != Q2
      True

    """
        return not self.__eq__(other)



####################################
# TO DO:    def __cmp__(self, other):
####################################


    def __cmp__(self, other):
        """
    This is the default comparison routine for <, <=, >, >= if
    no special comparison operator is defined.  These operations
    are not defined at present, and it is not clear what anything
    but equality would mean in this context, so we raise a
    NotImplementedError.

    INPUT:
      other -- a quadratic form

    OUTPUT:
      boolean

    EXAMPLES:
      sage: Q1 = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
      sage: Q2 = Q1(Matrix(ZZ, 3, 3, [1,2,3,0,2,4,0,0,3]))
      sage: Q1.__cmp__(Q2)
      Traceback (most recent call last):
      ...
      NotImplementedError: Warning: The comparison operation just used is not presently defined.
      sage: Q1 > Q2
      Traceback (most recent call last):
      ...
      NotImplementedError: Warning: The comparison operation just used is not presently defined.


    """
        raise NotImplementedError, "Warning: The comparison operation just used is not presently def
ined."



    def __add__(self, right):
        """
    Returns the direct sum of two quadratic forms.

    EXAMPLES::
      sage: Q = QuadraticForm(ZZ, 2, [1,4,10])
      sage: Q
      Quadratic form in 2 variables over Integer Ring with coefficients:
      [ 1 4 ]
      [ * 10 ]
      sage: Q2 = QuadraticForm(ZZ, 2, [1,4,-10])
      sage: Q + Q2
      Quadratic form in 4 variables over Integer Ring with coefficients:
      [ 1 4 0 0 ]
      [ * 10 0 0 ]
      [ * * 1 4 ]
      [ * * * -10 ]

    """
        if not isinstance(right, QuadraticForm):
            raise TypeError, "Oops!  Can't add these objects since they're not both quadratic forms.
```

```
=("
            elif (self.base_ring() != right.base_ring()):
                raise TypeError, "Oops! Can't add these since the quadratic forms don't have the same b
ase rings... =("
            else:
                Q = QuadraticForm(self.base_ring(), self.dim() + right.dim())
                n = self.dim()
                m = right.dim()

                for i in range(n):
                    for j in range(i,n):
                        Q[i,j] = self[i,j]

                for i in range(m):
                    for j in range(i,m):
                        Q[n+i,n+j] = right[i,j]

                return Q


    def sum_by_coefficients_with(self, right):
        """
        Returns the sum (on coefficients) of two quadratic forms of the same size.

        EXAMPLES::

            sage: Q = QuadraticForm(ZZ, 2, [1,4,10])
            sage: Q
            Quadratic form in 2 variables over Integer Ring with coefficients:
            [ 1 4 ]
            [ * 10 ]
            sage: Q+Q
            Quadratic form in 4 variables over Integer Ring with coefficients:
            [ 1 4 0 0 ]
            [ * 10 0 0 ]
            [ * * 1 4 ]
            [ * * * 10 ]

            sage: Q2 = QuadraticForm(ZZ, 2, [1,4,-10])
            sage: Q.sum_by_coefficients_with(Q2)
            Quadratic form in 2 variables over Integer Ring with coefficients:
            [ 2 8 ]
            [ * 0 ]

        """
            if not isinstance(right, QuadraticForm):
                raise TypeError, "Oops! Can't add these objects since they're not both quadratic forms.
=("
            elif (self.__n != right.__n):
                raise TypeError, "Oops! Can't add these since the quadratic forms don't have the same si
zes... =("
            elif (self.__base_ring != right.__base_ring):
                raise TypeError, "Oops! Can't add these since the quadratic forms don't have the same b
ase rings... =("
            else:
                return QuadraticForm(self.__base_ring, self.__n, [self.__coeffs[i]
 + right.__coeffs[i]  for i in range(len(self.__coeffs))])


## ======================  CHANGE THIS TO A TENSOR PRODUCT?!?  Even in Charact
eristic 2?!?  ======================
#    def __mul__(self, right):
#        """
#        Multiply (on the right) the quadratic form Q by an element of the ring
that Q is defined over.
#
#        EXAMPLES::
#            sage: Q = QuadraticForm(ZZ, 2, [1,4,10])
#            sage: Q*2
```



```
#            Quadratic form in 2 variables over Integer Ring with coefficients:
#            [ 2 8 ]
#            [ * 20 ]
#
#            sage: Q+Q == Q*2
#            True
#
#        """
#        try:
#            c = self.base_ring()(right)
#        except:
#            raise TypeError, "Oh no! The multiplier cannot be coerced into the
base ring of the quadratic form. =("
#
#        return QuadraticForm(self.base_ring(), self.dim(), [c * self.__coeffs[i
]  for i in range(len(self.__coeffs))])
# ============================================================================
==========================================



    def __call__(self, v):
        """
        Evaluate this quadratic form Q on a vector or matrix of elements
        coercible to the base ring of the quadratic form.  If a vector
        is given then the output will be the ring element Q('v'), but if a
        matrix is given then the output will be the quadratic form Q'
        which in matrix notation is given by:

        .. math::
            Q' = v^t * Q * v.


        EXAMPLES::

            ## Evaluate a quadratic form at a vector:
            ## ---------------------------------------
            sage: Q = QuadraticForm(QQ, 3, range(6))
            sage: Q
            Quadratic form in 3 variables over Rational Field with coefficients:
            [ 0 1 2 ]
            [ * 3 4 ]
            [ * * 5 ]
            sage: Q([1,2,3])
            89
            sage: Q([1,0,0])
            0
            sage: Q([1,1,1])
            15

        ::

            ## Evaluate a quadratic form using a column matrix:
            ## ------------------------------------------------
            sage: Q = QuadraticForm(QQ, 2, range(1,4))
            sage: A = Matrix(ZZ,2,2,[-1,0,0,1])
            sage: Q(A)
            Quadratic form in 2 variables over Rational Field with coefficients:
            [ 1 -2 ]
            [ * 3 ]
            sage: Q([1,0])
            1
            sage: type(Q([1,0]))
            <type 'sage.rings.rational.Rational'>
            sage: Q = QuadraticForm(QQ, 2, range(1,4))
            sage: Q(matrix(2, [1,0]))
            Quadratic form in 1 variables over Rational Field with coefficients:
            [ 1 ]
```

```
    ::

        ## Simple 2x2 change of variables:
        ## ---------------------------------
        sage: Q = QuadraticForm(ZZ, 2, [1,0,1])
        sage: Q
        Quadratic form in 2 variables over Integer Ring with coefficients:
        [ 1 0 ]
        [ * 1 ]
        sage: M = Matrix(ZZ, 2, 2, [1,1,0,1])
        sage: M
        [1 1]
        [0 1]
        sage: Q(M)
        Quadratic form in 2 variables over Integer Ring with coefficients:
        [ 1 2 ]
        [ * 2 ]

    ::

        ## Some more tests:
        ## ----------------
        sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
        sage: Q([1,2,3])
        14
        sage: v = vector([1,2,3])
        sage: Q(v)
        14
        sage: t = tuple([1,2,3])
        sage: Q(v)
        14
        sage: M = Matrix(ZZ, 3, [1,2,3])
        sage: Q(M)
        Quadratic form in 1 variables over Integer Ring with coefficients:
        [ 14 ]

    """
        ## If we are passed a matrix A, return the quadratic form Q(A(x))
        ## (In matrix notation: A^t * Q * A)
        n = self.dim()

        if is_Matrix(v):
            ## Check that v has the correct number of rows
            if v.nrows() != n:
                raise TypeError, "Oops!  The matrix must have " + str(n) + " rows. =("

            ## Create the new quadratic form
            m = v.ncols()
            Q2 = QuadraticForm(self.base_ring(), m)
            return QFEvaluateMatrix(self, v, Q2)

        elif (is_Vector(v) or isinstance(v, (list, tuple))):
            ## Check the vector/tuple/list has the correct length
            if not (len(v) == n):
                raise TypeError, "Oops!  Your vector needs to have length " + str(n) + "."

            ## TO DO:  Check that the elements can be coerced into the base ring
 of Q -- on first elt.
            if len(v) > 0:
                try:
                    x = self.base_ring()(v[0])
                except:
                    raise TypeError, "Oops!  Your vector is not coercible to the base ring of the qua
dratic form... =("

            ## Attempt to evaluate Q[v]
            return QFEvaluateVector(self, v)

        else:
```



```
            raise(TypeError, "Oops! Presently we can only evaluate a quadratic form on a list, tuple, vec
tor or matrix.")




## ============================================================================
========================

    def _is_even_symmetric_matrix_(self, A, R=None):
        """
    Tests if a matrix is symmetric, defined over R, and has even diagonal in R.

    INPUT:
      A -- matrix

      R -- ring

    EXAMPLES::

      sage: Q = QuadraticForm(ZZ, 2, [2,3,5])
      sage: A = Q.matrix()
      sage: A
      [ 4  3]
      [ 3 10]
      sage: Q._is_even_symmetric_matrix_(A)
      True
      sage: A[0,0] = 1
      sage: Q._is_even_symmetric_matrix_(A)
      False

    """
        if not is_Matrix(A):
            raise TypeError, "A is not a matrix."

        ring_coerce_test = True
        if R == None:             ## This allows us to omit the ring from the var
iables, and take it from the matrix
            R = A.base_ring()
            ring_coerce_test = False

        if not isinstance(R, Ring):
            raise TypeError, "R is not a ring."

        if not A.is_square():
            return False

        ## Test that the matrix is symmetric
        n = A.nrows()
        for i in range(n):
            for j in range(i+1, n):
                if A[i,j] != A[j,i]:
                    return False

        ## Test that all entries coerce to R
        if not ((A.base_ring() == R) or (ring_coerce_test == True)):
            try:
                for i in range(n):
                    for j in range(i, n):
                        x = R(A[i,j])
            except:
                return False

        ## Test that the diagonal is even (if 1/2 isn't in R)
        if not R(2).is_unit():
            for i in range(n):
                if not is_even(R(A[i,i])):
                    return False
```

```
        return True


## ============================================================================
=======================

    def matrix(self):
        """
    Returns the Hessian matrix A for which Q(X) =  '(1/2) * X^t * A * X'.

    EXAMPLES::

        sage: Q = QuadraticForm(ZZ, 3, range(6))
        sage: Q.matrix()
        [ 0  1  2]
        [ 1  6  4]
        [ 2  4 10]

    """
        return self.Hessian_matrix()


    def Hessian_matrix(self):
        """
    Returns the Hessian matrix A for which Q(X) = '(1/2) * X^t * A * X'.

    EXAMPLES::

        sage: Q = QuadraticForm(QQ, 2, range(1,4))
        sage: Q
        Quadratic form in 2 variables over Rational Field with coefficients:
        [ 1 2 ]
        [ * 3 ]
        sage: Q.Hessian_matrix()
        [2 2]
        [2 6]
        sage: Q.matrix().base_ring()
        Rational Field

    """
        mat_entries = []
        for i in range(self.dim()):
            for j in range(self.dim()):
                if (i == j):
                    mat_entries += [ 2 * self[i,j] ]
                else:
                    mat_entries += [ self[i,j] ]

        return matrix(self.base_ring(), self.dim(), self.dim(), mat_entries)


    def Gram_matrix_rational(self):
        """
    Returns a (symmetric) Gram matrix A for the quadratic form Q,
    meaning that

    .. math::

        Q(x) = x^t * A * x,

    defined over the fraction field of the base ring.

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,3,5,7])
        sage: A = Q.Gram_matrix_rational(); A
        [1 0 0 0]
        [0 3 0 0]
        [0 0 5 0]
```



```
        [0 0 0 7]
        sage: A.base_ring()
        Rational Field

    """
        R = self.base_ring()
        return (R(1) / R(2)) * self.Hessian_matrix()


    def Gram_matrix(self):
        """
    Returns a (symmetric) Gram matrix A for the quadratic form Q,
    meaning that

    .. math::

        Q(x) = x^t * A * x,

    defined over the base ring of Q.  If this is not possible,
    then a TypeError is raised.

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,3,5,7])
        sage: A = Q.Gram_matrix(); A
        [1 0 0 0]
        [0 3 0 0]
        [0 0 5 0]
        [0 0 0 7]
        sage: A.base_ring()
        Integer Ring

    """
        R = self.base_ring()
        A = self.Hessian_matrix() * (R(1) / R(2))
        n = self.dim()

        ## Test to see if it has an integral Gram matrix
        Int_flag = True
        for i in range(n):
            for j in range(i,n):
                Int_flag = Int_flag and (A[i,j] in R)   ## Note the single = (fo
r assignment) is correct here!

        ## Return the Gram matrix, or an error
        if Int_flag:
            return MatrixSpace(self.base_ring(), n, n)(A)
        else:
            raise TypeError, "Oops! This form does not have an integral Gram matrix. =("


    def has_integral_Gram_matrix(self):
        """
    Returns whether the quadratic form has an integral Gram matrix (with respect to its base ring).

    A warning is issued if the form is defined over a field, since in that case the return is trivially true.

    EXAMPLES::

        sage: Q = QuadraticForm(ZZ, 2, [7,8,9])
        sage: Q.has_integral_Gram_matrix()
        True

    ::

        sage: Q = QuadraticForm(ZZ, 2, [4,5,6])
        sage: Q.has_integral_Gram_matrix()
        False
```

```
    """
        ## Warning over fields
        if is_field(self.base_ring()):
            warn("Warning -- A quadratic form over a field always has integral Gram matrix.  Do you really wa
nt to do this?!?")

        ## Determine integrality of the Gram matrix
        flag = True
        try:
            self.Gram_matrix()
        except:
            flag = False

        return flag


    def gcd(self):
        """
    Returns the greatest common divisor of the coefficients of the
    quadratic form (as a polynomial).

    EXAMPLES::

      sage: Q = QuadraticForm(ZZ, 4, range(1, 21, 2))
      sage: Q.gcd()
      1

    ::

      sage: Q = QuadraticForm(ZZ, 4, range(0, 20, 2))
      sage: Q.gcd()
      2
    """
        if self.base_ring() != ZZ:
            raise TypeError, "Oops! The given quadratic form must be defined over ZZ."

        return GCD(self.coefficients())


    def is_primitive(self):
        """
    Determines if the given integer-valued form is primitive
    (i.e. not an integer (>1) multiple of another integer-valued
    quadratic form).

    EXAMPLES::

      sage: Q = QuadraticForm(ZZ, 2, [2,3,4])
      sage: Q.is_primitive()
      True
      sage: Q = QuadraticForm(ZZ, 2, [2,4,8])
      sage: Q.is_primitive()
      False

    """
        return (self.gcd() == 1)


    def primitive(self):
        """
    Returns a primitive version of an integer-valued quadratic form, defined over 'ZZ'.

    EXAMPLES::

      sage: Q = QuadraticForm(ZZ, 2, [2,3,4])
      sage: Q.primitive()
      Quadratic form in 2 variables over Integer Ring with coefficients:
      [ 2 3 ]
      [ * 4 ]
```



```
      sage: Q = QuadraticForm(ZZ, 2, [2,4,8])
      sage: Q.primitive()
      Quadratic form in 2 variables over Integer Ring with coefficients:
      [ 1 2 ]
      [ * 4 ]

    """
        if self.base_ring() != ZZ:
            raise TypeError, "Oops! The given quadratic form must be defined over ZZ."

        g = self.gcd()
        return QuadraticForm(self.base_ring(), self.dim(), [ZZ(x/g)  for x in se
lf.coefficients()])


    def adjoint_primitive(self):
        """
    Returns the primitive adjoint of the quadratic form, which is
    the smallest discriminant integer-valued quadratic form whose
    matrix is a scalar multiple of the inverse of the matrix of
    the given quadratic form.

    EXAMPLES::

      sage: Q = QuadraticForm(ZZ, 2, [1,2,3])
      sage: Q.adjoint_primitive()
      Quadratic form in 2 variables over Integer Ring with coefficients:
      [ 3 -2 ]
      [ *  1 ]

    """
        return QuadraticForm(self.Hessian_matrix().adjoint()).primitive()



    def dim(self):
        """
    Gives the number of variables of the quadratic form.

    EXAMPLES::

      sage: Q = QuadraticForm(ZZ, 2, [1,2,3])
      sage: Q.dim()
      2

    """
        return deepcopy(self.__n)


    def base_ring(self):
        """
    Gives the ring over which the quadratic form is defined.

    EXAMPLES::

      sage: Q = QuadraticForm(ZZ, 2, [1,2,3])
      sage: Q.base_ring()
      Integer Ring

    """
        return deepcopy(self.__base_ring)


    def coefficients(self):
        """
    Gives the matrix of upper triangular coefficients,
    by reading across the rows from the main diagonal.
```

EXAMPLES::

```
sage: Q = QuadraticForm(ZZ, 2, [1,2,3])
sage: Q.coefficients()
[1, 2, 3]
```

```
"""
        return deepcopy(self.__coeffs)


    def det(self):
        """
```

Gives the determinant of the Gram matrix of 2*Q, or
equivalently the determinant of the Hessian matrix of Q.

(Note: This is always defined over the same ring as the
quadratic form.)

EXAMPLES::

```
sage: Q = QuadraticForm(ZZ, 2, [1,2,3])
sage: Q.det()
8
```

```
"""
        try:
            return self.__det
        except:
            ## Compute the determinant
            if self.dim() == 0:
                new_det = self.base_ring()(1)
            else:
                new_det = self.matrix().det()

            ## Cache and return the determinant
            self.__det = new_det
            return new_det


    def Gram_det(self):
        """
```

Gives the determinant of the Gram matrix of Q.

(Note: This is defined over the fraction field of the ring of
the quadratic form, but is often not defined over the same
ring as the quadratic form.)

EXAMPLES::

```
sage: Q = QuadraticForm(ZZ, 2, [1,2,3])
sage: Q.Gram_det()
2
```

```
"""
        return self.det() / ZZ(2**self.dim())


    def base_change_to(self, R):
        """
```

Alters the quadratic form to have all coefficients
defined over the new base_ring R. Here R must be
coercible to from the current base ring.

Note: This is preferable to performing an explicit
coercion through the base_ring() method, which does
not affect the individual coefficients. This is
particularly useful for performing fast modular
arithmetic evaluations.


```
        INPUT:
            R -- a ring

        OUTPUT:
            quadratic form

        EXAMPLES::

            sage: Q = DiagonalQuadraticForm(ZZ,[1,1]); Q
            Quadratic form in 2 variables over Integer Ring with coefficients:
            [ 1 0 ]
            [ * 1 ]

        ::

            sage: Q1 = Q.base_change_to(IntegerModRing(5)); Q1
            Quadratic form in 2 variables over Ring of integers modulo 5 with coefficients:
            [ 1 0 ]
            [ * 1 ]

            sage: Q1([35,11])
            1

        """
            ## Check that a canonical coercion is possible
            if not is_Ring(R):
                raise TypeError, "Oops!  R is not a ring. =("
            if not R.has_coerce_map_from(self.__base_ring):
                raise TypeError, "Oops!  There is no canonical coercion from " + str(self.__base_
ring) + " to R."

            ## Return the coerced form
            return QuadraticForm(R, self.dim(), [R(x) for x in self.coefficients()])


        def level(self):
            r"""
```

Determines the level of the quadratic form over a PID, which is a
generator for the smallest ideal 'N' of 'R' such that N * (the matrix of
2*Q)^(-1) is in R with diagonal in 2*R.

Over '\ZZ' this returns a non-negative number.

(Caveat: This always returns the unit ideal when working over a field!)

EXAMPLES::

```
            sage: Q = QuadraticForm(ZZ, 2, range(1,4))
            sage: Q.level()
            8

            sage: Q1 = QuadraticForm(QQ, 2, range(1,4))
            sage: Q1.level()      # random
            UserWarning: Warning -- The level of a quadratic form over a field is always 1.  Do you really want to do this
?!?
            1

            sage: Q = DiagonalQuadraticForm(ZZ, [1,3,5,7])
            sage: Q.level()
            420

        """
            ## Try to return the cached level
            try:
                return deepcopy(self.__level)
            except:

                ## Check that the base ring is a PID
```

```
            if not is_PrincipalIdealDomain(self.base_ring()):
                raise TypeError, "Oops!  The level (as a number) is only defined over a Principal Ideal
Domain.  Try using level_ideal()."


            ## Warn the user if the form is defined over a field!
            if self.base_ring().is_field():
                warn("Warning -- The level of a quadratic form over a field is always 1.  Do you really want
to do this?!?")
                #raise RuntimeError, "Warning -- The level of a quadratic form o
ver a field is always 1.  Do you really want to do this?!?"


            ## Check invertibility and find the inverse
            try:
                mat_inv = self.matrix()**(-1)
            except:
                raise TypeError, "Oops!  The quadratic form is degenerate (i.e. det = 0). =("

            ## Compute the level
            inv_denoms = []
            for i in range(self.dim()):
                for j in range(i, self.dim()):
                    if (i == j):
                        inv_denoms += [denominator(mat_inv[i,j] / 2)]
                    else:
                        inv_denoms += [denominator(mat_inv[i,j])]
            lvl = LCM(inv_denoms)
            lvl = ideal(self.base_ring()(lvl)).gen()
            ################################################################
            ## To do this properly, the level should be the inverse of the
            ## fractional ideal (over R) generated by the entries whose
            ## denominators we take above. =)
            ################################################################

            ## Normalize the result over ZZ
            if self.base_ring() == IntegerRing():
                lvl = abs(lvl)

            ## Cache and return the level
            self.__level = lvl
            return deepcopy(lvl)



    def level_ideal(self):
        """
    Determines the level of the quadratic form (over R), which is the
    smallest ideal N of R such that N * (the matrix of 2*Q)^(-1) is
    in R with diagonal in 2*R.
    (Caveat: This always returns the principal ideal when working over a field!)

    WARNING:  THIS ONLY WORKS OVER A PID RING OF INTEGERS FOR NOW!
        (Waiting for Sage fractional ideal support.)

    EXAMPLES::

        sage: Q = QuadraticForm(ZZ, 2, range(1,4))
        sage: Q.level_ideal()
        Principal ideal (8) of Integer Ring

    ::

        sage: Q1 = QuadraticForm(QQ, 2, range(1,4))
        sage: Q1.level_ideal()
        Principal ideal (1) of Rational Field

    ::
```



```
        sage: Q = DiagonalQuadraticForm(ZZ, [1,3,5,7])
        sage: Q.level_ideal()
        Principal ideal (420) of Integer Ring

    """
        ################################################################
        ## To do this properly, the level should be the inverse of the
        ## fractional ideal (over R) generated by the entries whose
        ## denominators we take above. =)
        ################################################################

        return ideal(self.base_ring()(self.level()))



## ============================================================================
=======================

def DiagonalQuadraticForm(R, diag):
    """
    Returns a quadratic form over 'R' which is a sum of squares.

    INPUT:

    - 'R' -- ring
    - ''diag'' -- list/tuple of elements coercible to R

    OUTPUT:

    quadratic form

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,3,5,7])
        sage: Q
        Quadratic form in 4 variables over Integer Ring with coefficients:
        [ 1 0 0 0 ]
        [ * 3 0 0 ]
        [ * * 5 0 ]
        [ * * * 7 ]

    """
    Q = QuadraticForm(R, len(diag))
    for i in range(len(diag)):
        Q[i,i] = diag[i]
    return Q
```

```
"""
Reduction Theory
"""
from copy import deepcopy
from sage.matrix.constructor import matrix
from sage.functions.all import floor
from sage.misc.mrange import mrange
from sage.modules.free_module_element import vector
from sage.rings.integer_ring import ZZ

def LLL_reduced(self, give_transformation=False):
        """
    Returns a globally equivalent LLL-reduced quadratic form.
    If give_transformation=True then we additionally return the
    transformation (on column vectors) used to produce the new basis.
    """
        H = self.Hessian_matrix()
        U = H.LLL_gram()
        if give_transformation:
            return self(U), U
        else:
            return self(U)



def reduced_binary_form1(self):
    r"""
  Reduce the form 'ax^2 + bxy+cy^2' to satisfy the reduced condition 'lb| \le
  a \le c', with 'b \ge 0' if 'a = c'. This reduction occurs within the
  proper class, so all transformations are taken to have determinant 1.

  EXAMPLES::

    sage: QuadraticForm(ZZ,2,[5,5,2]).reduced_binary_form1()
    (Quadratic form in 2 variables over Integer Ring with coefficients:
    [ 2 -1 ]
    [ * 2 ]
    ,
    [ 0 -1]
    [ 1  1])

  """
    if self.dim() != 2:
        raise TypeError, "This must be a binary form for now..."

    R = self.base_ring()
    interior_reduced_flag = False
    Q = deepcopy(self)
    M = matrix(R, 2, 2, [1,0,0,1])

    while (interior_reduced_flag == False):
        interior_reduced_flag = True

        ## Arrange for a <= c
        if Q[0,0] > Q[1,1]:
            M_new = matrix(R,2,2,[0, -1, 1, 0])
            Q = Q(M_new)
            M = M * M_new
            interior_reduced_flag = False
            #print "A"

        ## Arrange for |b| <= a
        if abs(Q[0,1]) > Q[0,0]:
            r = R(floor(round(Q[0,1]/(2*Q[0,0]))))
            M_new = matrix(R,2,2,[1, -r, 0, 1])
            Q = Q(M_new)
            M = M * M_new
            interior_reduced_flag = False
```



```
            #print "B"

    return Q, M



def reduced_ternary_form__Dickson(self):
    """
  Find the unique reduced ternary form according to the conditions
  of Dickson's "Studies in the Theory of Numbers", pp164-171.

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1, 1, 1])
    sage: Q.reduced_ternary_form__Dickson()
    Traceback (most recent call last):
    ...
    NotImplementedError: TO DO

  """
    raise NotImplementedError, "TO DO"



def reduced_binary_form(self):
    """
  Find a form which is reduced in the sense that no further binary
  form reductions can be done to reduce the original form.

  EXAMPLES::

    sage: QuadraticForm(ZZ,2,[5,5,2]).reduced_binary_form()
    (Quadratic form in 2 variables over Integer Ring with coefficients:
    [ 2 -1 ]
    [ * 2 ]
    ,
    [ 0 -1]
    [ 1  1])

  """
    R = self.base_ring()
    n = self.dim()
    interior_reduced_flag = False
    Q = deepcopy(self)
    M = matrix(R, n, n)
    for i in range(n):
        M[i,i] = 1


    while (interior_reduced_flag == False):
        interior_reduced_flag = True

        #print Q

        ## Arrange for (weakly) increasing diagonal entries
        for i in range(n):
            for j in range(i+1,n):
                if Q[i,i] > Q[j,j]:
                    M_new = matrix(R,n,n)
                    for k in range(n):
                        M_new[k,k] = 1
                    M_new[i,j] = -1
                    M_new[j,i] = 1
                    M_new[i,i] = 0
                    M_new[j,j] = 1

                    Q = Q(M_new)
                    M = M * M_new
```

```
                    interior_reduced_flag = False
                    #print "A"

                ## Arrange for |b| <= a
                if abs(Q[i,j]) > Q[i,i]:
                    r = R(floor(round(Q[i,j]/(2*Q[i,i]))))

                    M_new = matrix(R,n,n)
                    for k in range(n):
                        M_new[k,k] = 1
                    M_new[i,j] = -r

                    Q = Q(M_new)
                    M = M * M_new
                    interior_reduced_flag = False
                    #print "B"

    return Q, M


def minkowski_reduction(self):
    """
  Find a Minkowski–reduced form equivalent to the given one.
  This means that

  .. math::

      Q(v_k) <= Q(s_1 * v_1 + ... + s_n * v_n)

  for all 's_i' where GCD'(s_k, ... s_n) = 1'.

  Note: When Q has dim <= 4 we can take all 's_i' in {1, 0, –1}.

  References:
    Schulze–Pillot's paper on "An algorithm for computing genera
            of ternary and quaternary quadratic forms", p138.
    Donaldson's 1979 paper "Minkowski Reduction of Integral
            Matrices", p203.

  EXAMPLES::

    sage: Q = QuadraticForm(ZZ,4,[30,17,11,12,29,25,62,64,25,110])
    sage: Q
    Quadratic form in 4 variables over Integer Ring with coefficients:
    [ 30 17 11 12 ]
    [ * 29 25 62 ]
    [ * * 64 25 ]
    [ * * * 110 ]
    sage: Q.minkowski_reduction()
    (Quadratic form in 4 variables over Integer Ring with coefficients:
    [ 30 17 11 –5 ]
    [ * 29 25 4 ]
    [ * * 64 0 ]
    [ * * * 77 ]
    ,
    [ 1  0  0  0]
    [ 0  1  0 –1]
    [ 0  0  1  0]
    [ 0  0  0  1])

  """
    R = self.base_ring()
    n = self.dim()
    interior_reduced_flag = False
    Q = deepcopy(self)
    M = matrix(R, n, n)
    for i in range(n):
        M[i,i] = 1
```



```
    ## Begin the reduction
    done_flag = False
    while done_flag == False:

        ## Loop through possible shorted vectors until
        done_flag = True
        #print " j_range = ", range(n-1, -1, -1)
        for j in range(n-1, -1, -1):
            for a_first in mrange([2  for i in range(j)]):
                y = [x-1 for x in a_first] + [1] + [0 for k in range(n-1-j)]
                e_j = [0  for k in range(n)]
                e_j[j] = 1
                #print "j = ", j

                ## Reduce if a shorter vector is found
                #print "y = ", y, "   e_j = ", e_j, "\n"
                if Q(y) < Q(e_j):

                    ## Create the transformation matrix
                    M_new = matrix(R, n, n)
                    for k in range(n):
                        M_new[k,k] = 1
                    for k in range(n):
                        M_new[k,j] = y[k]

                    ## Perform the reduction and restart the loop
                    #print "Q_before = ", Q
                    Q = Q(M_new)
                    M = M * M_new
                    done_flag = False

                    ## DIAGNOSTIC
                    #print "Q(y) = ", Q(y)
                    #print "Q(e_j) = ", Q(e_j)
                    #print "M_new = ", M_new
                    #print "Q_after = ", Q
                    #print

                if not done_flag:
                    break

            if not done_flag:
                break

    ## Return the results
    return Q, M



def minkowski_reduction_for_4vars__SP(self):
    """
  Find a Minkowski–reduced form equivalent to the given one.
  This means that

    Q('v_k') <= Q('s_1 * v_1 + ... + s_n * v_n')

  for all 's_i' where GCD('s_k, ... s_n') = 1.

  Note: When Q has dim <= 4 we can take all 's_i' in {1, 0, –1}.

  References:
    Schulze–Pillot's paper on "An algorithm for computing genera
            of ternary and quaternary quadratic forms", p138.
    Donaldson's 1979 paper "Minkowski Reduction of Integral
            Matrices", p203.

  EXAMPLES::
```

```
    sage: Q = QuadraticForm(ZZ,4,[30,17,11,12,29,25,62,64,25,110])
    sage: Q
    Quadratic form in 4 variables over Integer Ring with coefficients:
    [ 30 17 11 12 ]
    [ * 29 25 62 ]
    [ * * 64 25 ]
    [ * * * 110 ]
    sage: Q.minkowski_reduction_for_4vars__SP()
    (Quadratic form in 4 variables over Integer Ring with coefficients:
    [ 29 -17 25 4 ]
    [ * 30 -11 5 ]
    [ * * 64 0 ]
    [ * * * 77 ]
    ,
    [ 0  1  0  0]
    [ 1  0  0 -1]
    [ 0  0  1  0]
    [ 0  0  0  1])

    """
        R = self.base_ring()
        n = self.dim()
        interior_reduced_flag = False
        Q = deepcopy(self)
        M = matrix(R, n, n)
        for i in range(n):
            M[i,i] = 1

        ## Only allow 4-variable forms
        if n != 4:
            raise TypeError, "Oops!  The given quadratic form has " + str(n) +  \
                    " != 4 variables. =|"


        ## Step 1: Begin the reduction
        done_flag = False
        while done_flag == False:

            ## Loop through possible shorter vectors
            done_flag = True
            #print " j_range = ", range(n-1, -1, -1)
            for j in range(n-1, -1, -1):
                for a_first in mrange([2  for i in range(j)]):
                    y = [x-1 for x in a_first] + [1] + [0 for k in range(n-1-j)]
                    e_j = [0  for k in range(n)]
                    e_j[j] = 1
                    #print "j = ", j

                    ## Reduce if a shorter vector is found
                    #print "y = ", y, "   e_j = ", e_j, "\n"
                    if Q(y) < Q(e_j):

                        ## Further n=4 computations
                        B_y_vec = Q.matrix() * vector(ZZ, y)
                            ## SP's B = our self.matrix()/2
                            ## SP's A = coeff matrix of his B
                            ## Here we compute the double of both and compare.
                        B_sum = sum([abs(B_y_vec[i])  for i in range(4)  if i != j])
                        A_sum = sum([abs(Q[i,j])  for i in range(4)  if i != j])
                        B_max = max([abs(B_y_vec[i])  for i in range(4)  if i != j])
                        A_max = max([abs(Q[i,j])  for i in range(4)  if i != j])

                        if (B_sum < A_sum) or ((B_sum == A_sum) and (B_max < A_max))
:

                            ## Create the transformation matrix
                            M_new = matrix(R, n, n)
                            for k in range(n):
```



```
                                M_new[k,k] = 1
                            for k in range(n):
                                M_new[k,j] = y[k]

                            ## Perform the reduction and restart the loop
                            #print "Q_before = ", Q
                            Q = Q(M_new)
                            M = M * M_new
                            done_flag = False

                            ## DIAGNOSTIC
                            #print "Q(y) = ", Q(y)
                            #print "Q(e_j) = ", Q(e_j)
                            #print "M_new = ", M_new
                            #print "Q_after = ", Q
                            #print

                    if not done_flag:
                        break

                if not done_flag:
                    break

        ## Step 2: Order A by certain criteria
        for i in range(4):
            for j in range(i+1,4):

                ## Condition (a)
                if (Q[i,i] > Q[j,j]):
                    Q.swap_variables(i,j,in_place=True)
                    M_new = matrix(R,n,n)
                    M_new[i,j] = -1
                    M_new[j,i] = 1
                    for r in range(4):
                        if (r == i) or (r == j):
                            M_new[r,r] = 0
                        else:
                            M_new[r,r] = 1
                    M = M * M_new

                elif (Q[i,i] == Q[j,j]):
                    i_sum = sum([abs(Q[i,k])  for k in range(4)  if k != i])
                    j_sum = sum([abs(Q[j,k])  for k in range(4)  if k != j])

                    ## Condition (b)
                    if (i_sum > j_sum):
                        Q.swap_variables(i,j,in_place=True)
                        M_new = matrix(R,n,n)
                        M_new[i,j] = -1
                        M_new[j,i] = 1
                        for r in range(4):
                            if (r == i) or (r == j):
                                M_new[r,r] = 0
                            else:
                                M_new[r,r] = 1
                        M = M * M_new

                    elif (i_sum == j_sum):
                        for k in [2,1,0]:   ## TO DO: These steps are a little redun
dant...
                            Q1 = Q.matrix()

                            c_flag = True
                            for l in range(k+1,4):
                                c_flag = c_flag and (abs(Q1[i,l]) == abs(Q1[j,l]))

                            ## Condition (c)
                            if c_flag and (abs(Q1[i,k]) > abs(Q1[j,k])):
                                Q.swap_variables(i,j,in_place=True)
```

```
                            M_new = matrix(R,n,n)
                            M_new[i,j] = -1
                            M_new[j,i] = 1
                            for r in range(4):
                                if (r == i) or (r == j):
                                    M_new[r,r] = 0
                                else:
                                    M_new[r,r] = 1
                            M = M * M_new


    ## Step 3: Order the signs
    for i in range(4):
        if Q[i,3] < 0:
            Q.multiply_variable(-1, i, in_place=True)
            M_new = matrix(R,n,n)
            for r in range(4):
                if r == i:
                    M_new[r,r] = -1
                else:
                    M_new[r,r] = 1
            M = M * M_new

    for i in range(4):
        j = 3
        while (Q[i,j] == 0):
            j += -1
        if (Q[i,j] < 0):
            Q.multiply_variable(-1, i, in_place=True)
            M_new = matrix(R,n,n)
            for r in range(4):
                if r == i:
                    M_new[r,r] = -1
                else:
                    M_new[r,r] = 1
            M = M * M_new

    if Q[1,2] < 0:
        ## Test a row 1 sign change
        if (Q[1,3] <= 0 and \
            ((Q[1,3] < 0) or (Q[1,3] == 0 and Q[1,2] < 0)  \
                or (Q[1,3] == 0 and Q[1,2] == 0 and Q[1,1] < 0))):
            Q.multiply_variable(-1, i, in_place=True)
            M_new = matrix(R,n,n)
            for r in range(4):
                if r == i:
                    M_new[r,r] = -1
                else:
                    M_new[r,r] = 1
            M = M * M_new

        elif (Q[2,3] <= 0 and \
            ((Q[2,3] < 0) or (Q[2,3] == 0 and Q[2,2] < 0)  \
                or (Q[2,3] == 0 and Q[2,2] == 0 and Q[2,1] < 0))):
            Q.multiply_variable(-1, i, in_place=True)
            M_new = matrix(R,n,n)
            for r in range(4):
                if r == i:
                    M_new[r,r] = -1
                else:
                    M_new[r,r] = 1
            M = M * M_new


    ## Return the results
    return Q, M
```

```
"""
Siegel Products
"""
from sage.rings.arith import fundamental_discriminant
from sage.rings.integer_ring import ZZ
from sage.rings.rational_field import QQ
from sage.rings.arith import kronecker_symbol, bernoulli, prime_divisors
#from sage.combinat.combinat import bernoulli_polynomial
from sage.functions.all import sqrt
#from sage.rings.polynomial.polynomial_ring_constructor import PolynomialRing
from sage.misc.functional import ideal
from sage.quadratic_forms.special_values import QuadraticBernoulliNumber

from sage.misc.misc import verbose



#/*!  \brief Computes the product of all local densities for comparison with ind
ependently computed Eisenstein coefficients.
# *
# *  \todo We fixed the generic factors to compensate for using the matrix of 2Q
, but we need to document this better! =)
# */

#////////////////////////////////////////////////////////////////
#///
#////////////////////////////////////////////////////////////////

#mpq_class Matrix_mpz::siegel_product(mpz_class u) const {

def siegel_product(self, u):
    """
    Computes the infinite product of local densities of the quadratic
    form for the number 'u'.

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1])
        sage: Q.theta_series(11)
        1 + 8*q + 24*q^2 + 32*q^3 + 24*q^4 + 48*q^5 + 96*q^6 + 64*q^7 + 24*q^8 + 104*q^9 + 144*q^10 + O(q^11)

        sage: Q.siegel_product(1)
        8
        sage: Q.siegel_product(2)      ## This one is wrong -- expect 24, and the higher powers of 2 don't work... =(
        24
        sage: Q.siegel_product(3)
        32
        sage: Q.siegel_product(5)
        48
        sage: Q.siegel_product(6)
        96
        sage: Q.siegel_product(7)
        64
        sage: Q.siegel_product(9)
        104

        sage: Q.local_density(2,1)    ## This is ok
        1
        sage: M = 8; len([v  for v in mrange([M,M,M,M])  if Q(v) % M == 1]) / M^3
        1
        sage: M = 16; len([v  for v in mrange([M,M,M,M])  if Q(v) % M == 1]) / M^3     ## long time
        1

        sage: Q.local_density(2,2)    ## This is ok now. =)
        3/2
        sage: M = 8; len([v  for v in mrange([M,M,M,M])  if Q(v) % M == 2]) / M^3
        3/2
        sage: M = 16; len([v  for v in mrange([M,M,M,M])  if Q(v) % M == 2]) / M^3     ## long time
        3/2
```



```
        sage: [1] + [Q.siegel_product(ZZ(a))  for a in range(1,11)] == Q.theta_series(11).list()
        True

    """
    ## Protect u (since it fails often if it's an just an int!)
    u = ZZ(u)

    n = self.dim()
    d = self.det()       ## ??? Warning: This is a factor of 2^n larger than it
should be!

    ## DIAGNOSTIC
    verbose("n = " + str(n))
    verbose("d = " + str(d))
    verbose("In siegel_product:  d = ", d, "\n");


    ## Product of "bad" places to omit
    S = 2 * d * u

    ## DIAGNOSTIC
    verbose("siegel_product Break 1.\n")
    verbose(" u = ", u, "\n")


    ## Make the odd generic factors
    if ((n % 2) == 1):
        m = (n-1) / 2
        d1 = fundamental_discriminant(((-1)**m) * 2*d * u)      ## Replaced d by
2d here to compensate for the determinant
        f = abs(d1)                                             ## gaining an odd
 power of 2 by using the matrix of 2Q instead
                                                                ## of the matrix
of Q.
                                                                ##  --> Old d1 =
CoreDiscriminant((mpz_class(-1)^m) * d * u);

        ## Make the ratio of factorials factor: [(2m)! / m!] * prod_{i=1}^m (2*i
-1)
        factor1 = 1
        for i in range(1, m+1):
            factor1 *= 2*i - 1
        for i in range(m+1, 2*m + 1):
            factor1 *= i

        genericfactor = factor1 * ((u / f) ** m) \
            * QQ(sqrt((2 ** n) *  f) / (u * d)) \
            * abs(QuadraticBernoulliNumber(m, d1) / bernoulli(2*m))



    ## DIAGNOSTIC
    verbose("siegel_product Break 2.\n")


    ## Make the even generic factor
    if ((n % 2) == 0):
        m = n / 2
        d1 = fundamental_discriminant(((-1)**m) * d)
        f = abs(d1)

        ## DIAGNOSTIC
        #cout << " mpz_class(-1)^m = " << (mpz_class(-1)^m) << " and d = " << d
<< endl;
        #cout << " f = " << f << " and d1 = " << d1 << endl;


        genericfactor = m / QQ(sqrt(f*d)) \
```

```
            * ((u/2) ** (m-1)) * (f ** m) \
            / abs(QuadraticBernoulliNumber(m, d1)) \
            * (2 ** m)                                          ## This las
t factor compensates for using the matrix of 2*Q


    ##return genericfactor


    ## Omit the generic factors in S and compute them separately
    omit = 1
    include = 1

    S_divisors = prime_divisors(S)

    ## DIAGNOSTIC
    #cout << "\n S is " << S << endl;
    #cout << " The Prime divisors of S are :";
    #PrintV(S_divisors);


    for p in S_divisors:
        Q_normal = self.local_normal_form(p)


        ## DIAGNOSTIC
        verbose(" p = " + str(p) + " and its Kronecker symbol (d1/p) = (" + str(d1) + "/" + st
r(p) + ") is " + str(kronecker_symbol(d1, p)) + "\n")

        omit *= 1 / (1 - (kronecker_symbol(d1, p) / (p**m)))


        ## DIAGNOSTIC
        verbose(" omit = " + str(omit) + "\n")
        verbose(" Q_normal is \n" + str(Q_normal) + "\n")
        verbose(" Q_normal = \n" + str(Q_normal))
        verbose(" p = " + str(p) + "\n")
        verbose(" u = " +str(u) + "\n")
        verbose(" include = " + str(include) + "\n")


        include *= Q_normal.local_density(p, u)


        ## DIAGNOSTIC
        #cout << " Including the p = " << p << " factor: " << local_density(Q_no
rmal, p, u) << endl;

        ## DIAGNSOTIC
        verbose("  --- Exiting loop \n")




    #// ***************  Important ******************
    #// Additional fix (only included for n=4) to deal
    #// with the power of 2 introduced at the real place
    #// by working with Q instead of 2*Q.  This needs to
    #// be done for all other n as well...
    #/*
    #if (n==4)
    #  genericfactor = 4 * genericfactor;
    #*/


    ## DIAGNSOTIC
    #cout << endl;
    #cout << " generic factor = " << genericfactor << endl;
    #cout << " omit = " << omit << endl;
```



```
    #cout << " include = " << include << endl;
    #cout << endl;


    ## DIAGNSOTIC
    #//  cout << "siegel_product Break 3. " << endl;


    ## Return the final factor (and divide by 2 if n=2)
    if (n == 2):
        return (genericfactor * omit * include / 2)
    else:
        return (genericfactor * omit * include)
```

```
"""
Split Local Covering
"""
#############################################################################
## Routines that look for a split local covering for a given quadratic ##
## form in 4 variables.                                                    ##
#############################################################################

from copy import deepcopy

from sage.quadratic_forms.extras import extend_to_primitive
from sage.quadratic_forms.quadratic_form import QuadraticForm__constructor, is_Q
uadraticForm

from sage.rings.real_mpfr import RealField_class, RealField
from sage.rings.real_double import RDF
from sage.matrix.matrix_space import MatrixSpace
#from sage.matrix.matrix import Matrix
from sage.matrix.constructor import matrix
from sage.functions.all import sqrt, floor, ceil
from sage.rings.integer_ring import ZZ
from sage.rings.rational_field import QQ
from sage.misc.functional import round
from sage.rings.arith import GCD


def cholesky_decomposition(self, bit_prec = 53):
    """
  Give the Cholesky decomposition of this quadratic form 'Q' as a real matrix
  of precision ''bit_prec''.

  RESTRICTIONS:
    Q must be given as a QuadraticForm defined over '\ZZ', '\QQ', or some
    real field. If it is over some real field, then an error is raised if
    the precision given is not less than the defined precision of the real
    field defining the quadratic form!

  REFERENCE:
    From Cohen's "A Course in Computational Algebraic Number Theory" book,
    p 103.

  INPUT:
    ''bit_prec'' -- a natural number (default 53).

  OUTPUT:
    an upper triangular real matrix of precision ''bit_prec''.


  TO DO:
    If we only care about working over the real double field (RDF), then we
    can use the ''cholesky()'' method present for square matrices over that.

  .. note::

    There is a note in the original code reading

    ::

      ##////////////////////////////////////////////////////////////////////////////////////
      ##// Finds the Cholesky decomposition of a quadratic form -- as an upper-triangular matrix!
      ##// (It's assumed to be global, hence twice the form it refers to.)  <-- Python revision asks:  Is this true?!? =|
      ##////////////////////////////////////////////////////////////////////////////////////


  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1])
    sage: Q.cholesky_decomposition()
    [ 1.00000000000000 0.000000000000000 0.000000000000000]
```



```
    [0.000000000000000  1.00000000000000 0.000000000000000]
    [0.000000000000000 0.000000000000000  1.00000000000000]

  ::

    sage: Q = QuadraticForm(QQ, 3, range(1,7)); Q
    Quadratic form in 3 variables over Rational Field with coefficients:
    [ 1 2 3 ]
    [ * 4 5 ]
    [ * * 6 ]
    sage: Q.cholesky_decomposition()
    [ 1.00000000000000  1.00000000000000  1.50000000000000]
    [0.000000000000000  3.00000000000000 0.333333333333333]
    [0.000000000000000 0.000000000000000  3.41666666666667]

  """

    ## Check that the precision passed is allowed.
    if isinstance(self.base_ring(), RealField_class) and (self.base_ring().prec(
) < bit_prec):
        raise RuntimeError, "Oops! The precision requested is greater than that of the given quadratic for
m!"

    ## 1. Initialization
    n = self.dim()
    R = RealField(bit_prec)
    MS = MatrixSpace(R, n, n)
    Q = MS(R(0.5)) * MS(self.matrix())               ## Initialize the real symm
etric matrix A with the matrix for Q(x) = x^t * A * x

    ## DIAGNOSTIC
    #print "After 1:  Q is \n" + str(Q)

    ## 2. Loop on i
    for i in range(n):
        for j in range(i+1, n):
            Q[j,i] = Q[i,j]             ## Is this line redundant?
            Q[i,j] = Q[i,j] / Q[i,i]

        ## 3. Main Loop
        for k in range(i+1, n):
            for l in range(k, n):
                Q[k,l] = Q[k,l] - Q[k,i] * Q[i,l]

    ## 4. Zero out the strictly lower-triangular entries
    for i in range(n):
        for j in range(i):
            Q[i,j] = 0

    return Q



def vectors_by_length(self, bound):
    """
  Returns a list of short vectors together with their values.

  This is a naive algorithm which uses the Cholesky decomposition,
  but does not use the LLL-reduction algorithm.

  INPUT:
    bound -- an integer >= 0

  OUTPUT:
    A list L of length (bound + 1) whose entry L '[i]' is a list of
    all vectors of length 'i'.

  Reference: This is a slightly modified version of Cohn's Algorithm
  2.7.5 in "A Course in Computational Number Theory", with the
```

increment step moved around and slightly re-indexed to allow clean looping.

Note: We could speed this up for very skew matrices by using LLL first, and then changing coordinates back, but for our purposes the simpler method is efficient enough. =)

EXAMPLES::

```
    sage: Q = DiagonalQuadraticForm(ZZ, [1,1])
    sage: Q.vectors_by_length(5)
    [[[0, 0]],
     [[0, -1], [-1, 0]],
     [[-1, -1], [1, -1]],
     [],
     [[0, -2], [-2, 0]],
     [[-1, -2], [1, -2], [-2, -1], [2, -1]]]

::

    sage: Q1 = DiagonalQuadraticForm(ZZ, [1,3,5,7])
    sage: Q1.vectors_by_length(5)
    [[[0, 0, 0, 0]],
     [[-1, 0, 0, 0]],
     [],
     [[0, -1, 0, 0]],
     [[-1, -1, 0, 0], [1, -1, 0, 0], [-2, 0, 0, 0]],
     [[0, 0, -1, 0]]]

::

    sage: Q = QuadraticForm(ZZ, 4, [1,1,1,1, 1,0,0, 1,0, 1])
    sage: map(len, Q.vectors_by_length(2))
    [1, 12, 12]

::

    sage: Q = QuadraticForm(ZZ, 4, [1,-1,-1,-1, 1,0,0, 4,-3, 4])
    sage: map(len, Q.vectors_by_length(3))
    [1, 3, 0, 3]
"""
    # pari uses eps = 1e-6 ; nothing bad should happen if eps is too big
    # but if eps is too small, roundoff errors may knock off some
    # vectors of norm = bound (see #7100)
    eps = RDF(1e-6)
    bound = ZZ(floor(max(bound, 0)))
    Theta_Precision = bound + eps
    n = self.dim()

    ## Make the vector of vectors which have a given value
    ## (So theta_vec[i] will have all vectors v with Q(v) = i.)
    theta_vec = [[] for i in range(bound + 1)]

    ## Initialize Q with zeros and Copy the Cholesky array into Q
    Q = self.cholesky_decomposition()


    ## 1. Initialize
    T = n * [RDF(0)]    ## Note: We index the entries as 0 --> n-1
    U = n * [RDF(0)]
    i = n-1
    T[i] = RDF(Theta_Precision)
    U[i] = RDF(0)

    L = n * [0]
    x = n * [0]
    Z = RDF(0)

    ## 2. Compute bounds
```



```
    Z = (T[i] / Q[i][i]).sqrt(extend=False)
    L[i] = ( Z - U[i]).floor()
    x[i] = (-Z - U[i]).ceil()

    done_flag = False
    Q_val_double = RDF(0)
    Q_val = 0 ## WARNING: Still need a good way of checking overflow for this va
lue...

    ## Big loop which runs through all vectors
    while not done_flag:

        ## 3b. Main loop -- try to generate a complete vector x (when i=0)
        while (i > 0):
            #print " i = ", i
            #print " T[i] = ", T[i]
            #print " Q[i][i] = ", Q[i][i]
            #print " x[i] = ", x[i]
            #print " U[i] = ", U[i]
            #print " x[i] + U[i] = ", (x[i] + U[i])
            #print " T[i-1] = ", T[i-1]

            T[i-1] = T[i] - Q[i][i] * (x[i] + U[i]) * (x[i] + U[i])

            #print " T[i-1] = ",  T[i-1]
            #print " x = ", x
            #print

            i = i - 1
            U[i] = 0
            for j in range(i+1, n):
                U[i] = U[i] + Q[i][j] * x[j]

            ## Now go back and compute the bounds...
            ## 2. Compute bounds
            Z = (T[i] / Q[i][i]).sqrt(extend=False)
            L[i] = ( Z - U[i]).floor()
            x[i] = (-Z - U[i]).ceil()

            # carry if we go out of bounds -- when Z is so small that
            # there aren't any integral vectors between the bounds
            # Note: this ensures T[i-1] >= 0 in the next iteration
            while (x[i] > L[i]):
                i += 1
                x[i] += 1

        ## 4. Solution found (This happens when i = 0)
        #print "-- Solution found! --"
        #print " x = ", x
        #print " Q_val = Q(x) = ", Q_val
        Q_val_double = Theta_Precision - T[0] + Q[0][0] * (x[0] + U[0]) * (x[0]
+ U[0])
        Q_val = Q_val_double.round()

        ## SANITY CHECK: Roundoff Error is < 0.001
        if abs(Q_val_double -  Q_val) > 0.001:
            print " x = ", x
            print " Float = ", Q_val_double, "  Long = ", Q_val
            raise RuntimeError, "The roundoff error is bigger than 0.001, so we should use more precis
ion somewhere..."

        #print " Float = ", Q_val_double, "   Long = ", Q_val, "  XX "
        #print " The float value is ", Q_val_double
        #print " The associated long value is ", Q_val

        if (Q_val <= bound):
            #print " Have vector ",  x, " with value ", Q_val
            theta_vec[Q_val].append(deepcopy(x))
```

```
        ## 5. Check if x = 0, for exit condition. =)
        j = 0
        done_flag = True
        while (j < n):
            if (x[j] != 0):
                done_flag = False
            j += 1


        ## 3a. Increment (and carry if we go out of bounds)
        x[i] += 1
        while (x[i] > L[i]) and (i < n-1):
            i += 1
            x[i] += 1


    #print " Leaving ThetaVectors()"
    return theta_vec



def complementary_subform_to_vector(self, v):
    """
    Finds the '(n-1)'-dim'l quadratic form orthogonal to the vector 'v'.

    Note: This is usually not a direct summand!

    Technical Notes: There is a minor difference in the cancellation
    code here (form the C++ version) since the notation Q '[i,j]' indexes
    coefficients of the quadratic polynomial here, not the symmetric
    matrix.  Also, it produces a better splitting now, for the full
    lattice (as opposed to a sublattice in the C++ code) since we
    now extend 'v' to a unimodular matrix.

    INPUT:
      'v' -- a list of self.dim() integers

    OUTPUT:
      a QuadraticForm over 'ZZ'


    EXAMPLES::

      sage: Q1 = DiagonalQuadraticForm(ZZ, [1,3,5,7])
      sage: Q1.complementary_subform_to_vector([1,0,0,0])
      Quadratic form in 3 variables over Integer Ring with coefficients:
      [ 3 0 0 ]
      [ * 5 0 ]
      [ * * 7 ]

    ::

      sage: Q1.complementary_subform_to_vector([1,1,0,0])
      Quadratic form in 3 variables over Integer Ring with coefficients:
      [ 12 0 0 ]
      [ * 5 0 ]
      [ * * 7 ]

    ::

      sage: Q1.complementary_subform_to_vector([1,1,1,1])
      Quadratic form in 3 variables over Integer Ring with coefficients:
      [ 624 -480 -672 ]
      [ * 880 -1120 ]
      [ * * 1008 ]

    """
```



```
    n = self.dim()

    ## Copy the quadratic form
    Q = deepcopy(self)

    ## Find the first non-zero component of v, and call it nz  (Note: 0 <= nz < n)
    nz = 0
    while (nz < n) and (v[nz] == 0):
        nz += 1

    ## Abort if v is the zero vector
    if nz == n:
        raise TypeError, "Oops, v cannot be the zero vector! =("

    ## Make the change of basis matrix
    new_basis = extend_to_primitive(matrix(ZZ,n,1,v))

    ## Change Q (to Q1) to have v as its nz-th basis vector
    Q1 = Q(new_basis)

    ## Pick out the value Q(v) of the vector
    d = Q1[0, 0]

    #print Q1

    ## For each row/column, perform elementary operations to cancel them out.
    for i in range(1,n):

        ## Check if the (i,0)-entry is divisible by d,
        ## and stretch its row/column if not.
        if Q1[i,0] % d != 0:
            Q1 = Q1.multiply_variable(d / GCD(d, Q1[i, 0]//2), i)

        #print "After scaling at i =", i
        #print Q1

        ## Now perform the (symmetric) elementary operations to cancel out the (i,0) entries/
        Q1 = Q1.add_symmetric(-(Q1[i,0]/2) / (GCD(d, Q1[i,0]//2)), i, 0)

        #print "After cancelling at i =", i
        #print Q1

    ## Check that we're done!
    done_flag = True
    for i in range(1, n):
        if Q1[0,i] != 0:
            done_flag = False

    if done_flag == False:
        raise RuntimeError, "There is a problem cancelling out the matrix entries! =O"


    ## Return the complementary matrix
    return Q1.extract_variables(range(1,n))



def split_local_cover(self):
    """
    Tries to find subform of the given (positive definite quaternary)
    quadratic form Q of the form

    .. math::

      d*x^2 + T(y,z,w)

    where 'd > 0' is as small as possible.
```

This is done by exhaustive search on small vectors, and then
comparing the local conditions of its sum with it's complementary
lattice and the original quadratic form Q.

INPUT:
none

OUTPUT:
a QuadraticForm over ZZ

EXAMPLES::

```
sage: Q1 = DiagonalQuadraticForm(ZZ, [7,5,3])
sage: Q1.split_local_cover()
Quadratic form in 3 variables over Integer Ring with coefficients:
[ 3 0 0 ]
[ * 7 0 ]
[ * * 5 ]
```

```
    """
        ## 0. If a split local cover already exists, then return it.
        if hasattr(self, "__split_local_cover"):
            if is_QuadraticForm(self.__split_local_cover):  ## Here the computation
has been done.
                return self.__split_local_cover
            elif isinstance(self.__split_local_cover, Integer):    ## Here it indexe
s the values already tried!
                current_length = self.__split_local_cover + 1
                Length_Max = current_length + 5
        else:
            current_length = 1
            Length_Max = 6

        ## 1. Find a range of new vectors
        all_vectors = self.vectors_by_length(Length_Max)
        current_vectors = all_vectors[current_length]

        ## Loop until we find a split local cover...
        while True:

            ## 2. Check if any of the primitive ones produce a split local cover
            for v in current_vectors:
                #print "current length = ", current_length
                #print "v = ", v
                Q = QuadraticForm__constructor(ZZ, 1, [current_length]) + self.compl
ementary_subform_to_vector(v)
                #print Q
                if Q.local_representation_conditions() == self.local_representation_
conditions():
                    self.__split_local_cover = Q
                    return Q

            ## 3. Save what we have checked and get more vectors.
            self.__split_local_cover = current_length
            current_length += 1
            if current_length >= len(all_vectors):
                Length_Max += 5
                all_vectors = self.vectors_by_length(Length_Max)
            current_vectors = all_vectors[current_length]
```

```
"""
Tornaria Methods for Computing with Quadratic Forms

"""

#################################################################
## Routines from Gonzalo Tornaria (7/9/07)
## for computing with ternary quadratic forms.
#################################################################


#from sage.rings.rational_field import QQ

from sage.rings.integer_ring import ZZ
from sage.misc.functional import is_odd
from sage.rings.power_series_ring import PowerSeriesRing

from sage.libs.pari.all import pari
from sage.misc.misc import prod
from sage.rings.arith import factor, gcd, prime_to_m_part, CRT_list, CRT_vectors
from sage.rings.arith import hilbert_symbol, kronecker_symbol

from sage.quadratic_forms.quadratic_form import QuadraticForm__constructor as Qu
adraticForm
from sage.modules.free_module import FreeModule
from sage.modules.free_module_element import vector


## TO DO -- Add second argument
#  def __call__(self,v,w=None):
#    if w==None:
#        return half(v * self._matrix_() * v)
#    else:
#      return v * self._matrix_() * w



def disc(self):
    r"""
  Returns the discriminant of the quadratic form, defined as

  – '(–1)^n {\rm det}(B)' for even dimension '2n'
  – '{\rm det}(B)/2' for odd dimension

  where '2Q(x) = x^t B x'.

  This agrees with the usual discriminant for binary and ternary quadratic forms.

  EXAMPLES::

    sage: DiagonalQuadraticForm(ZZ, [1]).disc()
    1
    sage: DiagonalQuadraticForm(ZZ, [1,1]).disc()
    –4
    sage: DiagonalQuadraticForm(ZZ, [1,1,1]).disc()
    4
    sage: DiagonalQuadraticForm(ZZ, [1,1,1,1]).disc()
    16

  """
    if is_odd(self.dim()):
      return  self.base_ring()(self.det() / 2)      ## This is not so good for c
haracteristic 2.
    else:
      return (-1)**(self.dim()/2) * self.det()


def content(self):
    """
```



```
  Returns the GCD of the coefficients of the quadratic form.

  .. warning::

    Only works over Euclidean domains (probably just '\ZZ').

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1, 1])
    sage: Q.matrix().gcd()
    2
    sage: Q.content()
    1
    sage: DiagonalQuadraticForm(ZZ, [1, 1]).is_primitive()
    True
    sage: DiagonalQuadraticForm(ZZ, [2, 4]).is_primitive()
    False
    sage: DiagonalQuadraticForm(ZZ, [2, 4]).primitive()
    Quadratic form in 2 variables over Integer Ring with coefficients:
    [ 1 0 ]
    [ * 2 ]
  """
    return self.gcd()


## in quadratic_form.py
#def is_primitive(self):
#    """
#    Checks if the form is a multiple of another form... only over ZZ for now.
#    """
#    return self.content() == 1



## in quadratic_form.py
#def primitive(self):
#    """
#    Returns a primitive quadratic forms in the similarity class of the given fo
rm.
#
#    This only works when we have GCDs... so over ZZ.
#    """
#    c=self.content()
#    new_coeffs = [self.base_ring()(a/c)  for a in self.__coeffs]
#    return QuadraticForm(self.base_ring(), self.dim(), new_coeffs)



def adjoint(self):
    """
  This gives the adjoint (integral) quadratic form associated to the
  given form, essentially defined by taking the adjoint of the matrix.

  EXAMPLES::

    sage: Q = QuadraticForm(ZZ, 2, [1,2,5])
    sage: Q.adjoint()
    Quadratic form in 2 variables over Integer Ring with coefficients:
    [ 5 –2 ]
    [ * 1 ]

  ::

    sage: Q = QuadraticForm(ZZ, 3, [1, 0, –1, 2, –1, 5])
    sage: Q.adjoint()
    Quadratic form in 3 variables over Integer Ring with coefficients:
    [ 39 2 8 ]
    [ * 19 4 ]
    [ * * 8 ]
```

```
    """
        if is_odd(self.dim()):
            return QuadraticForm(self.matrix().adjoint()*2)
        else:
            return QuadraticForm(self.matrix().adjoint())


    def antiadjoint(self):
        """
    This gives an (integral) form such that its adjoint is the given form.

    EXAMPLES::

        sage: Q = QuadraticForm(ZZ, 3, [1, 0, -1, 2, -1, 5])
        sage: Q.adjoint().antiadjoint()
        Quadratic form in 3 variables over Integer Ring with coefficients:
        [ 1 0 -1 ]
        [ * 2 -1 ]
        [ * * 5 ]
        sage: Q.antiadjoint()
        Traceback (most recent call last):
        ...
        ValueError: not an adjoint

    """
        try:
            n = self.dim()
            R = self.base_ring()
            d = R(self.disc()**(ZZ(1)/(n-1)))
            if is_odd(n):
                return self.adjoint().scale_by_factor( R(1) / 4 / d**(n-2) )
            else:
                return self.adjoint().scale_by_factor( R(1) / d**(n-2) )
        except TypeError:
            raise ValueError, "not an adjoint"


    def is_adjoint(self):
        """
    Determines if the given form is the adjoint of another form

    EXAMPLES::

        sage: Q = QuadraticForm(ZZ, 3, [1, 0, -1, 2, -1, 5])
        sage: Q.is_adjoint()
        False
        sage: Q.adjoint().is_adjoint()
        True

    """
        try:
            self.antiadjoint()
        except ValueError:
            return False
        return True


    def reciprocal(self):
        """
    This gives the reciprocal quadratic form associated to the given form.
    This is defined as the multiple of the primitive adjoint with the same
    content as the given form.

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,1,37])
        sage: Q.reciprocal()
        Quadratic form in 3 variables over Integer Ring with coefficients:
```



```
        [ 37 0 0 ]
        [ * 37 0 ]
        [ * * 1 ]
        sage: Q.reciprocal().reciprocal()
        Quadratic form in 3 variables over Integer Ring with coefficients:
        [ 1 0 0 ]
        [ * 1 0 ]
        [ * * 37 ]
        sage: Q.reciprocal().reciprocal() == Q
        True

    """
        return self.adjoint().primitive() . scale_by_factor( self.content() )


    def omega(self):
        """
    This is the content of the adjoint of the primitive associated quadratic form.

    Ref: See Dickson's "Studies in Number Theory".

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,1,37])
        sage: Q.omega()
        4

    """
        return self.primitive().adjoint().content()

    def delta(self):
        """
    This is the omega of the adjoint form,
    which is the same as the omega of the reciprocal form.

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,1,37])
        sage: Q.delta()
        148

    """
        return self.adjoint().omega()


    def level__Tornaria(self):
        """
    Returns the level of the quadratic form,
    defined as

        level(B)    for even dimension
        level(B)/4  for odd dimension

    where 2Q(`x`) `= x^t * B * x`.

    This agrees with the usual level for even dimension...

    EXAMPLES::

        sage: DiagonalQuadraticForm(ZZ, [1]).level__Tornaria()
        1
        sage: DiagonalQuadraticForm(ZZ, [1,1]).level__Tornaria()
        4
        sage: DiagonalQuadraticForm(ZZ, [1,1,1]).level__Tornaria()
        1
        sage: DiagonalQuadraticForm(ZZ, [1,1,1,1]).level__Tornaria()
        4

    """
```

```
    return self.base_ring()(abs(self.disc())/self.omega()/self.content()**self.d
im())


def discrec(self):
    """
  Returns the discriminant of the reciprocal form.

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,37])
    sage: Q.disc()
    148
    sage: Q.discrec()
    5476
    sage: [4 * 37, 4 * 37^2]
    [148, 5476]

  """
    return self.reciprocal().disc()



### Rational equivalence

def hasse_conductor(self):
    """
  This is the product of all primes where the Hasse invariant equals −1

  EXAMPLES::

    sage: Q = QuadraticForm(ZZ, 3, [1, 0, −1, 2, −1, 5])
    sage: Q.hasse_invariant(2)
    −1
    sage: Q.hasse_invariant(37)
    −1
    sage: Q.hasse_conductor()
    74

  ::

    sage: DiagonalQuadraticForm(ZZ, [1, 1, 1]).hasse_conductor()
    1
    sage: QuadraticForm(ZZ, 3, [2, −2, 0, 2, 0, 5]).hasse_conductor()
    10
  """
    D = self.disc()
    return prod(filter(lambda(p):self.hasse_invariant(p)==-1, \
             map(lambda(x):x[0],factor(2*self.level()))))

def clifford_invariant(self, p):
    """
  This is the Clifford invariant, i.e. the class in the Brauer group of the
  Clifford algebra for even dimension, of the even Clifford Algebra for odd dimension.

  See Lam (AMS GSM 67) p. 117 for the definition, and p. 119 for the formula
  relating it to the Hasse invariant.

  EXAMPLES:

  For hyperbolic spaces, the clifford invariant is +1::

    sage: H = QuadraticForm(ZZ, 2, [0, 1, 0])
    sage: H.clifford_invariant(2)
    1
    sage: (H + H).clifford_invariant(2)
    1
    sage: (H + H + H).clifford_invariant(2)
```



```
    1
    sage: (H + H + H + H).clifford_invariant(2)
    1

  """
    n = self.dim() % 8
    if  n == 1 or n == 2:
        s = 1
    elif n == 3 or n == 4:
        s = hilbert_symbol(-1, -self.disc(), p)
    elif n == 5 or n == 6:
        s = hilbert_symbol(-1, -1, p)
    elif n == 7 or n == 0:
        s = hilbert_symbol(-1, self.disc(), p)
    return s * self.hasse_invariant(p)

def clifford_conductor(self):
    """
  This is the product of all primes where the Clifford invariant is −1

  Note: For ternary forms, this is the discriminant of the
  quaternion algebra associated to the quadratic space
  (i.e. the even Clifford algebra)

  EXAMPLES::

    sage: Q = QuadraticForm(ZZ, 3, [1, 0, −1, 2, −1, 5])
    sage: Q.clifford_invariant(2)
    1
    sage: Q.clifford_invariant(37)
    −1
    sage: Q.clifford_conductor()
    37

  ::

    sage: DiagonalQuadraticForm(ZZ, [1, 1, 1]).clifford_conductor()
    2
    sage: QuadraticForm(ZZ, 3, [2, −2, 0, 2, 0, 5]).clifford_conductor()
    30

  For hyperbolic spaces, the clifford conductor is 1::

    sage: H = QuadraticForm(ZZ, 2, [0, 1, 0])
    sage: H.clifford_conductor()
    1
    sage: (H + H).clifford_conductor()
    1
    sage: (H + H + H).clifford_conductor()
    1
    sage: (H + H + H + H).clifford_conductor()
    1

  """
    D = self.disc()
    return prod(filter(lambda(p):self.clifford_invariant(p)==-1, \
             map(lambda(x):x[0],factor(2*self.level()))))


### Genus theory

def basiclemma(self,M):
    """
  Finds a number represented by self and coprime to M.

  EXAMPLES::

    sage: Q = QuadraticForm(ZZ, 2, [2, 1, 3])
    sage: Q.basiclemma(6)
```

```
    71

  """
    a=self(self.basiclemmavec(M))
    assert gcd(a,M) == 1
    return a

def basiclemmavec(self,M):
    """
  Finds a vector where the value of the quadratic form is coprime to M.

  EXAMPLES::

    sage: Q = QuadraticForm(ZZ, 2, [2, 1, 5])
    sage: Q.basiclemmavec(10)
    (6, 5)
    sage: Q(_)
    227

  """
    V=FreeModule(self.base_ring(),self.dim())
    mat = self.matrix()
    vec = []
    mod = []
    M0 = abs(M)
    if M0 == 1:
        return V(0)

    for i in range(self.dim()):
        M1 = prime_to_m_part(M0, self[i,i])
        if M1 != 1:
            vec.append(V.gen(i))
            mod.append(M1)
        M0 = M0/M1
        if M0 == 1:
            return tuple(CRT_vectors(vec,mod))

    for i in range(self.dim()):
        for j in range(i):
            M1 = prime_to_m_part(M0, self[i,j])
            if M1 != 1:
                vec.append(V.i + V.j)
                mod.append(M1)
            M0 = M0/M1
            if M0 == 1:
                return __crt_list(vec,mod)

    raise ValueError, "not primitive form"


### FIXME: get the rules for validity of characters straight...
### p=2 might be bad!!!
def xi(self,p):
    """
  Return the value of the genus characters Xi_p... which may be missing one character.
  We allow -1 as a prime.

  Reference: Dickson's "Studies in the Theory of Numbers"

  EXAMPLES::

    sage: Q1 = QuadraticForm(ZZ, 3, [1, 1, 1, 14, 3, 14])
    sage: Q2 = QuadraticForm(ZZ, 3, [2, -1, 0, 2, 0, 50])
    sage: [Q1.omega(), Q2.omega()]
    [5, 5]
    sage: [Q1.hasse_invariant(5), Q2.hasse_invariant(5)]    # equivalent over Q_5
    [1, 1]
    sage: [Q1.xi(5), Q2.xi(5)]                    # not equivalent over Z_5
    [1, -1]
```



```
  """
    if self.dim() == 2 and self.disc() % p:
        raise ValueError, "not a valid character"
    if self.dim() >= 3 and self.omega() % p:
        raise ValueError, "not a valid character"
    if (p == -1) or (p == 2):
        return kronecker_symbol(p, self.basiclemma(2))
    return kronecker_symbol(self.basiclemma(p), p)


def xi_rec(self,p):
    """
  Returns Xi(`p`) for the reciprocal form.

  EXAMPLES::

    sage: Q1 = QuadraticForm(ZZ, 3, [1, 1, 1, 14, 3, 14])
    sage: Q2 = QuadraticForm(ZZ, 3, [2, -1, 0, 2, 0, 50])
    sage: [Q1.clifford_conductor(), Q2.clifford_conductor()]   # equivalent over Q
    [3, 3]
    sage: Q1.is_locally_equivalent_to(Q2)               # not in the same genus
    False
    sage: [Q1.delta(), Q2.delta()]
    [480, 480]
    sage: factor(480)
    2^5 * 3 * 5
    sage: map(Q1.xi_rec, [-1,2,3,5])
    [-1, -1, -1, 1]
    sage: map(Q2.xi_rec, [-1,2,3,5])
    [-1, -1, -1, -1]

  """
    return self.reciprocal().xi(p)


def lll(self):
    """
  Returns an LLL-reduced form of Q (using Pari).

  EXAMPLES::

    sage: Q = QuadraticForm(ZZ, 4, range(1,11))
    sage: Q.is_definite()
    True
    sage: Q.lll()
    Quadratic form in 4 variables over Integer Ring with coefficients:
    [ 1 0 1 0 ]
    [ * 4 3 3 ]
    [ * * 6 3 ]
    [ * * * 6 ]

  """
    return self(self.matrix().LLL_gram())


def representation_number_list(self, B):
    """
  Returns the vector of representation numbers < B.

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ,[1,1,1,1,1,1,1,1])
    sage: Q.representation_number_list(10)
    [1, 16, 112, 448, 1136, 2016, 3136, 5504, 9328, 12112]

  """
    ans = pari(1).concat(self._pari_().qfrep(B-1, 1) * 2)
    return ans._sage_()
```

```
def representation_vector_list(self, B, maxvectors = 10**8):
    """
  Find all vectors v where Q(v) < B.

  EXAMPLES::

    sage: Q = DiagonalQuadraticForm(ZZ, [1, 1])
    sage: Q.representation_vector_list(10)
    [[(0, 0)],
     [(0, 1), (0, -1), (1, 0), (-1, 0)],
     [(1, 1), (-1, -1), (-1, 1), (1, -1)],
     [],
     [(0, 2), (0, -2), (2, 0), (-2, 0)],
     [(1, 2), (-1, -2), (-1, 2), (1, -2), (2, 1), (-2, -1), (-2, 1), (2, -1)],
     [],
     [],
     [(2, 2), (-2, -2), (-2, 2), (2, -2)],
     [(0, 3), (0, -3), (3, 0), (-3, 0)]]
    sage: map(len, _)
    [1, 4, 4, 0, 4, 8, 0, 0, 4, 4]
    sage: Q.representation_number_list(10)
    [1, 4, 4, 0, 4, 8, 0, 0, 4, 4]

  """
    n, m, vs = self._pari_().qfminim(2*(B-1), maxvectors)
    if n != 2 * len(vs):
        raise RuntimeError("insufficient number of vectors")
    ms = [[] for _ in xrange(B)]
    ms[0] = [vector([0] * self.dim())]
    for v in vs._sage_().columns():
        ms[int(self(v))] += [v, -v]
    return ms


### zeros

def is_zero(self, v, p=0):
    """
  Determines if the vector v is on the conic Q(x) = 0 (mod p).

  EXAMPLES::

    sage: Q1 = QuadraticForm(ZZ, 3, [1, 0, -1, 2, -1, 5])
    sage: Q1.is_zero([0,1,0], 2)
    True
    sage: Q1.is_zero([1,1,1], 2)
    True
    sage: Q1.is_zero([1,1,0], 2)
    False

  """
    norm = self(v)
    if p != 0:
        norm = norm % p
    return  norm == 0

def is_zero_nonsingular(self, v, p=0):
    """
  Determines if the vector 'v' is on the conic Q('x') = 0 (mod 'p'),
  and that this point is non-singular point of the conic.

  EXAMPLES::

    sage: Q1 = QuadraticForm(ZZ, 3, [1, 0, -1, 2, -1, 5])
    sage: Q1.is_zero_nonsingular([1,1,1], 2)
    True
    sage: Q1.is_zero([1, 19, 2], 37)
```



```
    True
    sage: Q1.is_zero_nonsingular([1, 19, 2], 37)
    False

  """
    if not self.is_zero(v, p):
        return False
    vm = vector(self.base_ring(), v) * self.matrix()
    if p != 0:
        vm = vm % p
    return (vm != 0)

def is_zero_singular(self, v, p=0):
    """
  Determines if the vector 'v' is on the conic Q('x') = 0 (mod 'p'),
  and that this point is singular point of the conic.

  EXAMPLES::

    sage: Q1 = QuadraticForm(ZZ, 3, [1, 0, -1, 2, -1, 5])
    sage: Q1.is_zero([1,1,1], 2)
    True
    sage: Q1.is_zero_singular([1,1,1], 2)
    False
    sage: Q1.is_zero_singular([1, 19, 2], 37)
    True

  """
    if not self.is_zero(v, p):
        return False
    vm = vector(self.base_ring(), v) * self.matrix()
    if p != 0:
        vm = vm % p
    return (vm == 0)
```

```
"""
Theta Series of Quadratic Forms

AUTHORS:

– Jonathan Hanke: initial code, theta series of degree 1

– Gonzalo Tornaria (2009–02–22): fixes and doctests

– Gonzalo Tornaria (2010–03–23): theta series of degree 2

"""

from copy import deepcopy

from sage.rings.real_mpfr import RealField
from sage.matrix.matrix_space import MatrixSpace
from sage.rings.power_series_ring import PowerSeriesRing
from sage.rings.integer_ring import IntegerRing, ZZ
from sage.functions.all import sqrt, floor, ceil

from sage.interfaces.gp import gp

from sage.modular.dims import sturm_bound

from sage.misc.misc import cputime, verbose


def theta_series(self, Max=10, var_str='q', safe_flag=True):
    """
    Compute the theta series as a power series in the variable given
    in var_str (which defaults to ''q''), up to the specified precision
    'O(q^max)'.

    This uses the PARI/GP function qfrep, wrapped by the
    theta_by_pari() method.  This caches the result for future
    computations.

    The safe_flag allows us to select whether we want a copy of the
    output, or the original output.  It is only meaningful when a
    vector is returned, otherwise a copy is automatically made in
    creating the power series.  By default safe_flag = True, so we
    return a copy of the cached information.  If this is set to False,
    then the routine is much faster but the return values are
    vulnerable to being corrupted by the user.

    TO DO: Allow the option Max='mod_form' to give enough coefficients
    to ensure we determine the theta series as a modular form.  This
    is related to the Sturm bound, but we'll need to be careful about
    this (particularly for half–integral weights!).

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,3,5,7])
        sage: Q.theta_series()
        1 + 2*q + 2*q^3 + 6*q^4 + 2*q^5 + 4*q^6 + 6*q^7 + 8*q^8 + 14*q^9 + O(q^10)

        sage: Q.theta_series(25)
        1 + 2*q + 2*q^3 + 6*q^4 + 2*q^5 + 4*q^6 + 6*q^7 + 8*q^8 + 14*q^9 + 4*q^10 + 12*q^11 + 18*q^12 + 12*q^13
    + 12*q^14 + 8*q^15 + 34*q^16 + 12*q^17 + 8*q^18 + 32*q^19 + 10*q^20 + 28*q^21 + 16*q^23 + 44*q^24 + O(q^2
    5)

    """
        ## Sanity Check: Max is an integer or an allowed string:
        try:
            M = ZZ(Max)
        except:
            M = -1
```



```
        if (Max not in ['mod_form']) and (not M >= 0):
            print Max
            raise TypeError, "Oops!  Max is not an integer >= 0 or an allowed string."

        if Max == 'mod_form':
            raise NotImplementedError, "Oops!  We have to figure out the correct number of Fourier coeffi
    cients to use..."
            #return self.theta_by_pari(sturm_bound(self.level(), self.dim() / ZZ(2))
     + 1, var_str, safe_flag)
        else:
            return self.theta_by_pari(M, var_str, safe_flag)



    ## -------------  Compute the theta function by using the PARI/GP routine qfrep
     ------------

    def theta_by_pari(self, Max, var_str='q', safe_flag=True):
        """
        Use PARI/GP to compute the theta function as a power series (or
        vector) up to the precision 'O(q^Max)'.  This also caches the result
        for future computations.

        If var_str = '', then we return a vector 'v' where 'v[i]' counts the
        number of vectors of length 'i'.

        The safe_flag allows us to select whether we want a copy of the
        output, or the original output.  It is only meaningful when a
        vector is returned, otherwise a copy is automatically made in
        creating the power series.  By default safe_flag = True, so we
        return a copy of the cached information.  If this is set to False,
        then the routine is much faster but the return values are
        vulnerable to being corrupted by the user.


        INPUT:
            Max -– an integer >=0
            var_str -– a string

        OUTPUT:
            a power series or a vector

        EXAMPLES::

            sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1])
            sage: Prec = 100
            sage: compute = Q.theta_by_pari(Prec, '')
            sage: exact = [1] + [8 * sum([d  for d in divisors(i)  if d % 4 != 0])  for i in range(1, Prec)]
            sage: compute == exact
            True

        """
            ## Try to use the cached result if it's enough precision
            if hasattr(self, '__theta_vec') and len(self.__theta_vec) >= Max:
                theta_vec = self.__theta_vec[:Max]
            else:
                theta_vec = self.representation_number_list(Max)
                ## Cache the theta vector
                self.__theta_vec = theta_vec

            ## Return the answer
            if var_str == '':
                if safe_flag:
                    return deepcopy(theta_vec)          ## We must make a copy here to in
    sure the integrity of the cached version!
                else:
                    return theta_vec
            else:
                return PowerSeriesRing(ZZ, var_str)(theta_vec, Max)
```

```
## -------------  Compute the theta function by using an explicit Cholesky decom
position ------------


##########################################################################
## Routines to compute the Fourier expansion of the theta function of Q ##
## (to a given precision) via a Cholesky decomposition over RR.         ##
##                                                                      ##
## The Cholesky code was taken from:                                    ##
## ~/Documents/290_Project/C/Ver13.2__3-5-2007/Matrix_mpz/Matrix_mpz.cc ##
##########################################################################



def theta_by_cholesky(self, q_prec):
    r"""
  Uses the real Cholesky decomposition to compute (the 'q'-expansion of) the
  theta function of the quadratic form as a power series in 'q' with terms
  correct up to the power 'q^{\text{q\_prec}}'. (So its error is 'O(q^
  {\text{q\_prec} + 1})'.)

  REFERENCE:
    From Cohen's "A Course in Computational Algebraic Number Theory" book,
    p 102.

  EXAMPLES::

    ## Check the sum of 4 squares form against Jacobi's formula
    sage: Q = DiagonalQuadraticForm(ZZ, [1,1,1,1])
    sage: Theta = Q.theta_by_cholesky(10)
    sage: Theta
    1 + 8*q + 24*q^2 + 32*q^3 + 24*q^4 + 48*q^5 + 96*q^6 + 64*q^7 + 24*q^8 + 104*q^9 + 144*q^10
    sage: Expected =  [1] + [8*sum([d for d in divisors(n) if d%4 != 0])  for n in range(1,11)]
    sage: Expected
    [1, 8, 24, 32, 24, 48, 96, 64, 24, 104, 144]
    sage: Theta.list() == Expected
    True

  ::

    ## Check the form x^2 + 3y^2 + 5z^2 + 7w^2 represents everything except 2 and 22.
    sage: Q = DiagonalQuadraticForm(ZZ, [1,3,5,7])
    sage: Theta = Q.theta_by_cholesky(50)
    sage: Theta_list = Theta.list()
    sage: [m  for m in range(len(Theta_list))  if Theta_list[m] == 0]
    [2, 22]

  """
    ## RAISE AN ERROR -- This routine is deprecated!
    #raise NotImplementedError, "This routine is deprecated.  Try theta_series()
, which uses theta_by_pari()."


    n = self.dim()
    theta = [0 for i in range(q_prec+1)]
    PS = PowerSeriesRing(ZZ, 'q')

    bit_prec = 53                                          ## TO DO: Set this preci
sion to reflect the appropriate roundoff
    Cholesky = self.cholesky_decomposition(bit_prec)     ## error estimate, to b
e confident through our desired q-precision.
    Q = Cholesky      ##  <----  REDUNDANT!!!
    R = RealField(bit_prec)
    half = R(0.5)
```



```
    ## 1. Initialize
    i = n - 1
    T = [R(0)  for j in range(n)]
    U = [R(0)  for j in range(n)]
    T[i] = R(q_prec)
    U[i] = 0
    L = [0 for j in range (n)]
    x = [0 for j in range (n)]


    ## 2. Compute bounds
    #Z = sqrt(T[i] / Q[i,i])      ## IMPORTANT NOTE: sqrt preserves the precisio
n of the real number it's given... which is not so good... =|
    #L[i] = floor(Z - U[i])       ## Note: This is a Sage Integer
    #x[i] = ceil(-Z - U[i]) - 1   ## Note: This is a Sage Integer too


    done_flag = False
    from_step4_flag = False
    from_step3_flag = True        ## We start by pretending this, since then we
get to run through 2 and 3a once. =)

    #double Q_val_double;
    #unsigned long Q_val;                  // WARNING: Still need a good way of c
hecking overflow for this value...



    ## Big loop which runs through all vectors
    while (done_flag == False):

        ## Loop through until we get to i=1 (so we defined a vector x)
        while from_step3_flag or from_step4_flag:              ## IMPORTANT WARN
ING:  This replaces a do...while loop, so it may have to be adjusted!

            ## Go to directly to step 3 if we're coming from step 4, otherwise p
erform step 2.
            if from_step4_flag:
                from_step4_flag = False
            else:
                ## 2. Compute bounds
                from_step3_flag = False
                Z = sqrt(T[i] / Q[i,i])
                L[i] = floor(Z - U[i])
                x[i] = ceil(-Z - U[i]) - 1



            ## 3a. Main loop

            ## DIAGNOSTIC
            #print
            #print "  L = ", L
            #print "  x = ", x

            x[i] += 1
            while (x[i] > L[i]):

                ## DIAGNOSTIC
                #print "  x = ", x

                i += 1
                x[i] += 1


            ## 3b. Main loop
            if (i > 0):
                from_step3_flag = True
```

```
                ## DIAGNOSTIC
                #print " i = " + str(i)
                #print " T[i] = " + str(T[i])
                #print " Q[i,i] = " + str(Q[i,i])
                #print " x[i] = " + str(x[i])
                #print " U[i] = " + str(U[i])
                #print " x[i] + U[i] = " + str(x[i] + U[i])
                #print " T[i-1] = " + str(T[i-1])

                T[i-1] = T[i] - Q[i,i] * (x[i] + U[i]) * (x[i] + U[i])

                # DIAGNOSTIC
                #print " T[i-1] = " + str(T[i-1])
                #print

                i += - 1
                U[i] = 0
                for j in range(i+1, n):
                    U[i] += Q[i,j] * x[j]



        ## 4. Solution found (This happens when i=0)
        from_step4_flag = True
        Q_val_double = q_prec - T[0] + Q[0,0] * (x[0] + U[0]) * (x[0] + U[0])
        Q_val = floor(Q_val_double + half)        ## Note: This rounds the value
up, since the "round" function returns a float, but floor returns integer.



        ## DIAGNOSTIC
        #print " Q_val_double = ",  Q_val_double
        #print " Q_val = ",  Q_val
        #raise RuntimeError


        ## OPTIONAL SAFETY CHECK:
        eps = 0.000000001
        if (abs(Q_val_double - Q_val) > eps):
            raise RuntimeError, "Oh No!  We have a problem with the floating point precision... \n"  \
                + " Q_val_double = " + str(Q_val_double) + "\n"  \
                + " Q_val = " + str(Q_val) + "\n"  \
                + " x = " + str(x) + "\n"


        ## DIAGNOSTIC
        #print " The float value is " + str(Q_val_double)
        #print " The associated long value is " + str(Q_val)
        #print

        if (Q_val <= q_prec):
            theta[Q_val] += 2

        ## 5. Check if x = 0, for exit condition. =)
        done_flag = True
        for j in range(n):
            if (x[j] != 0):
                done_flag = False


    ## Set the value: theta[0] = 1
    theta[0] = 1

    ## DIAGNOSTIC
    #print "Leaving ComputeTheta \n"


    ## Return the series, truncated to the desired q-precision
```



```
    return PS(theta)


def theta_series_degree_2(Q, prec):
    r"""
    Compute the theta series of degree 2 for the quadratic form Q.

    INPUT:

    – ''prec'' –– an integer.

    OUTPUT:

    dictionary, where:

    – keys are '{\rm GL}_2(\ZZ)'–reduced binary quadratic forms (given as triples of
      coefficients)
    – values are coefficients

    EXAMPLES::

      sage: Q2 = QuadraticForm(ZZ, 4, [1,1,1,1, 1,0,0, 1,0, 1])
      sage: S = Q2.theta_series_degree_2(10)
      sage: S[(0,0,2)]
      24
      sage: S[(1,0,1)]
      144
      sage: S[(1,1,1)]
      192

    AUTHORS:

    – Gonzalo Tornaria (2010–03–23)

    REFERENCE:

    – Raum, Ryan, Skoruppa, Tornaria, 'On Formal Siegel Modular Forms'
      (preprint)
    """
    if Q.base_ring() != ZZ:
        raise TypeError, "The quadratic form must be integral"
    if not Q.is_positive_definite():
        raise ValueError, "The quadratic form must be positive definite"
    try:
        X = ZZ(prec-1)    # maximum discriminant
    except:
        raise TypeError, "prec is not an integer"

    if X < -1:
        raise ValueError, "prec must be >= 0"

    if X == -1:
        return {}

    V = ZZ ** Q.dim()
    H = Q.Hessian_matrix()

    t = cputime()
    max = int(floor((X+1)/4))
    v_list = (Q.vectors_by_length(max))        # assume a>0
    v_list = map(lambda(vs):map(V,vs), v_list) # coerce vectors into V
    verbose("Computed vectors_by_length" , t)

    # Deal with the singular part
    coeffs = {(0,0,0):1}
    for i in range(1,max+1):
        coeffs[(0,0,i)] = 2 * len(v_list[i])

    # Now deal with the non-singular part
```

```
    a_max = int(floor(sqrt(X/3)))
    for a in range(1, a_max + 1):
        t = cputime()
        c_max = int(floor((a*a + X)/(4*a)))
        for c in range(a, c_max + 1):
            for v1 in v_list[a]:
                v1_H = v1 * H
                def B_v1(v):
                    return v1_H * v2
                for v2 in v_list[c]:
                    b = abs(B_v1(v2))
                    if b <= a and 4*a*c-b*b <= X:
                        qf = (a,b,c)
                        count = 4 if b == 0 else 2
                        coeffs[qf] = coeffs.get(qf, ZZ(0)) + count
        verbose("done a = %d" % a, t)

    return coeffs
```

```
"""
Variable Substitution, Multiplication, Division, Scaling

"""
#*****************************************************************************
#       Copyright (C) 2007 William Stein and Jonathan Hanke
#
#  Distributed under the terms of the GNU General Public License (GPL)
#
#    This code is distributed in the hope that it will be useful,
#    but WITHOUT ANY WARRANTY; without even the implied warranty of
#    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
#    General Public License for more details.
#
#  The full text of the GPL is available at:
#
#                  http://www.gnu.org/licenses/
#*****************************************************************************

import copy


def swap_variables(self, r, s, in_place = False):
    """
    Switch the variables 'x_r' and 'x_s' in the quadratic form
    (replacing the original form if the in_place flag is True).

    INPUT:
      'r', 's' -- integers >= 0

    OUTPUT:
      a QuadraticForm (by default, otherwise none)

    EXAMPLES::

      sage: Q = QuadraticForm(ZZ, 4, range(1,11))
      sage: Q
      Quadratic form in 4 variables over Integer Ring with coefficients:
      [ 1 2 3 4 ]
      [ * 5 6 7 ]
      [ * * 8 9 ]
      [ * * * 10 ]


      sage: Q.swap_variables(0,2)
      Quadratic form in 4 variables over Integer Ring with coefficients:
      [ 8 6 3 9 ]
      [ * 5 2 7 ]
      [ * * 1 4 ]
      [ * * * 10 ]


      sage: Q.swap_variables(0,2).swap_variables(0,2)
      Quadratic form in 4 variables over Integer Ring with coefficients:
      [ 1 2 3 4 ]
      [ * 5 6 7 ]
      [ * * 8 9 ]
      [ * * * 10 ]

    """
    if (in_place == False):
        Q = copy.deepcopy(self)
        Q.__init__(self.base_ring(), self.dim(), self.coefficients())
        Q.swap_variables(r,s,in_place=True)
        return Q

    else:
        ## Switch diagonal elements
        tmp = self[r,r]
        self[r,r] = self[s,s]
        self[s,s] = tmp

        ## Switch off-diagonal elements
        for i in range(self.dim()):
            if (i != r) and (i != s):
                tmp = self[r,i]
                self[r,i] = self[s,i]
                self[s,i] = tmp


def multiply_variable(self, c, i, in_place = False):
    """
    Replace the variables 'x_i' by 'c*x_i' in the quadratic form
    (replacing the original form if the in_place flag is True).

    Here 'c' must be an element of the base_ring defining the
    quadratic form.

    INPUT:
      'c' -- an element of Q.base_ring()

      'i' -- an integer >= 0

    OUTPUT:
      a QuadraticForm (by default, otherwise none)

    EXAMPLES::

      sage: Q = DiagonalQuadraticForm(ZZ, [1,9,5,7])
      sage: Q.multiply_variable(5,0)
      Quadratic form in 4 variables over Integer Ring with coefficients:
      [ 25 0 0 0 ]
      [ * 9 0 0 ]
      [ * * 5 0 ]
      [ * * * 7 ]

    """
    if (in_place == False):
        Q = copy.deepcopy(self)
        Q.__init__(self.base_ring(), self.dim(), self.coefficients())
        Q.multiply_variable(c,i,in_place=True)
        return Q

    else:
        ## Stretch the diagonal element
        tmp = c * c * self[i,i]
        self[i,i] = tmp

        ## Switch off-diagonal elements
        for k in range(self.dim()):
            if (k != i):
                tmp = c * self[k,i]
                self[k,i] = tmp


def divide_variable(self, c, i, in_place = False):
    """
    Replace the variables 'x_i' by '(x_i)/c' in the quadratic form
    (replacing the original form if the in_place flag is True).

    Here 'c' must be an element of the base_ring defining the
    quadratic form, and the division must be defined in the base
    ring.

    INPUT:
      'c' -- an element of Q.base_ring()

      'i' -- an integer >= 0
```

```
    OUTPUT:
        a QuadraticForm (by default, otherwise none)

    EXAMPLES::

        sage: Q = DiagonalQuadraticForm(ZZ, [1,9,5,7])
        sage: Q.divide_variable(3,1)
        Quadratic form in 4 variables over Integer Ring with coefficients:
        [ 1 0 0 0 ]
        [ * 1 0 0 ]
        [ * * 5 0 ]
        [ * * * 7 ]

    """
        if (in_place == False):
            Q = copy.deepcopy(self)
            Q.__init__(self.base_ring(), self.dim(), self.coefficients())
            Q.divide_variable(c,i,in_place=True)
            return Q

        else:
            ## Stretch the diagonal element
            tmp = self[i,i] / (c*c)
            self[i,i] = tmp

            ## Switch off-diagonal elements
            for k in range(self.dim()):
                if (k != i):
                    tmp = self[k,i] / c
                    self[k,i] = tmp


    def scale_by_factor(self, c, change_value_ring_flag=False):
        """
        Scale the values of the quadratic form by the number 'c', if
        this is possible while still being defined over its base ring.

        If the flag is set to true, then this will alter the value ring
        to be the field of fractions of the original ring (if necessary).

        INPUT:
            'c' -- a scalar in the fraction field of the value ring of the form.

        OUTPUT:
            A quadratic form of the same dimension

        EXAMPLES::

            sage: Q = DiagonalQuadraticForm(ZZ, [3,9,18,27])
            sage: Q.scale_by_factor(3)
            Quadratic form in 4 variables over Integer Ring with coefficients:
            [ 9 0 0 0 ]
            [ * 27 0 0 ]
            [ * * 54 0 ]
            [ * * * 81 ]

            sage: Q.scale_by_factor(1/3)
            Quadratic form in 4 variables over Integer Ring with coefficients:
            [ 1 0 0 0 ]
            [ * 3 0 0 ]
            [ * * 6 0 ]
            [ * * * 9 ]

        """
            ## Try to scale the coefficients while staying in the ring of values.
            new_coeff_list = [x*c  for x in self.coefficients()]

            ## Check if we can preserve the value ring and return result. -- USE THE BAS
```



```
E_RING FOR NOW...
            R = self.base_ring()
            try:
                list2 = [R(x)  for x in new_coeff_list]
                # This is a hack: we would like to use QuadraticForm here, but
                # it doesn't work by scoping reasons.
                Q = self.__class__(R, self.dim(), list2)
                return Q
            except:
                if (change_value_ring_flag == False):
                    raise TypeError, "Oops! We could not rescale the lattice in this way and preserve its definin
g ring."
                else:
                    raise UntestedCode, "This code is not tested by current doctests!"
                    F = R.fraction_field()
                    list2 = [F(x)  for x in new_coeff_list]
                    Q = copy.deepcopy(self)
                    Q.__init__(self.dim(), F, list2, R)  ## DEFINE THIS!  IT WANTS TO SE
T THE EQUIVALENCE RING TO R, BUT WITH COEFFS IN F.
                    #Q.set_equivalence_ring(R)
                    return Q


    def extract_variables(self, var_indices):
        """
        Extract the variables (in order) whose indices are listed in
        var_indices, to give a new quadratic form.

        INPUT:
            var_indices -- a list of integers >= 0

        OUTPUT:
            a QuadraticForm

        EXAMPLES::

            sage: Q = QuadraticForm(ZZ, 4, range(10)); Q
            Quadratic form in 4 variables over Integer Ring with coefficients:
            [ 0 1 2 3 ]
            [ * 4 5 6 ]
            [ * * 7 8 ]
            [ * * * 9 ]
            sage: Q.extract_variables([1,3])
            Quadratic form in 2 variables over Integer Ring with coefficients:
            [ 4 6 ]
            [ * 9 ]

        """
            m = len(var_indices)
            Q = copy.deepcopy(self)
            Q.__init__(self.base_ring(), m)
            for i in range(m):
                for j in range(i, m):
                    Q[i,j] = self[ var_indices[i], var_indices[j] ]

            return Q



    def elementary_substitution(self, c, i, j, in_place = False):     ## CHECK THIS!
!!
        """
        Perform the substitution 'x_i --> x_i + c*x_j' (replacing the
        original form if the in_place flag is True).

        INPUT:
            'c' -- an element of Q.base_ring()

            'i', 'j' -- integers >= 0
```

```
    OUTPUT:
        a QuadraticForm (by default, otherwise none)

    EXAMPLES::

        sage: Q = QuadraticForm(ZZ, 4, range(1,11))
        sage: Q
        Quadratic form in 4 variables over Integer Ring with coefficients:
        [ 1 2 3 4 ]
        [ * 5 6 7 ]
        [ * * 8 9 ]
        [ * * * 10 ]

        sage: Q.elementary_substitution(c=1, i=0, j=3)
        Quadratic form in 4 variables over Integer Ring with coefficients:
        [ 1 2 3 6 ]
        [ * 5 6 9 ]
        [ * * 8 12 ]
        [ * * * 15 ]

    ::

        sage: R = QuadraticForm(ZZ, 4, range(1,11))
        sage: R
        Quadratic form in 4 variables over Integer Ring with coefficients:
        [ 1 2 3 4 ]
        [ * 5 6 7 ]
        [ * * 8 9 ]
        [ * * * 10 ]

    ::

        sage: M = Matrix(ZZ, 4, 4, [1,0,0,1,0,1,0,0,0,0,1,0,0,0,0,1])
        sage: M
        [1 0 0 1]
        [0 1 0 0]
        [0 0 1 0]
        [0 0 0 1]
        sage: R(M)
        Quadratic form in 4 variables over Integer Ring with coefficients:
        [ 1 2 3 6 ]
        [ * 5 6 9 ]
        [ * * 8 12 ]
        [ * * * 15 ]

    """
        if (in_place == False):
            Q = copy.deepcopy(self)
            Q.__init__(self.base_ring(), self.dim(), self.coefficients())
            Q.elementary_substitution(c, i, j, True)
            return Q

        else:
            ## Adjust the a_{k,j} coefficients
            ij_old = self[i,j]    ## Store this since it's overwritten, but used in
    the a_{j,j} computation!
            for k in range(self.dim()):
                if (k != i) and (k != j):
                    ans = self[j,k] + c*self[i,k]
                    self.__setitem__((j,k), ans)
                elif (k == j):
                    ans = self[j,k] + c*ij_old + c*c*self[i,i]
                    self[j,k] = ans
                else:
                    ans = self[j,k] + 2*c*self[i,k]
                    self[j,k] = ans
```



```
def add_symmetric(self, c, i, j, in_place = False):
        """
    Performs the substitution 'x_j --> x_j + c*x_i', which has the
    effect (on associated matrices) of symmetrically adding
    'c * j'-th row/column to the 'i'-th row/column.

    NOTE: This is meant for compatibility with previous code,
    which implemented a matrix model for this class.  It is used
    in the local_normal_form() method.


    INPUT:
        'c' -- an element of Q.base_ring()

        'i', 'j' -- integers >= 0

    OUTPUT:
        a QuadraticForm (by default, otherwise none)

    EXAMPLES::

        sage: Q = QuadraticForm(ZZ, 3, range(1,7)); Q
        Quadratic form in 3 variables over Integer Ring with coefficients:
        [ 1 2 3 ]
        [ * 4 5 ]
        [ * * 6 ]
        sage: Q.add_symmetric(-1, 1, 0)
        Quadratic form in 3 variables over Integer Ring with coefficients:
        [ 1 0 3 ]
        [ * 3 2 ]
        [ * * 6 ]
        sage: Q.add_symmetric(-3/2, 2, 0)     ## ERROR: -3/2 isn't in the base ring ZZ
        Traceback (most recent call last):
        ...
        RuntimeError: Oops!  This coefficient can't be coerced to an element of the base ring for the quadratic form.

    ::

        sage: Q = QuadraticForm(QQ, 3, range(1,7)); Q
        Quadratic form in 3 variables over Rational Field with coefficients:
        [ 1 2 3 ]
        [ * 4 5 ]
        [ * * 6 ]
        sage: Q.add_symmetric(-3/2, 2, 0)
        Quadratic form in 3 variables over Rational Field with coefficients:
        [ 1 2 0 ]
        [ * 4 2 ]
        [ * * 15/4 ]

    """
        return self.elementary_substitution(c, j, i, in_place)
```

```
## Routines that allow one to compute with quadratic lattices (i.e., a lattice i
n a quadratic space)



from sage.rings.all import ZZ
from sage.modules.free_module import FreeModule
from sage.rings.principal_ideal_domain import is_PrincipalIdealDomain
from sage.quadratic_forms.quadratic_form import QuadraticForm
from sage.matrix.constructor import Matrix
from copy import deepcopy


class QuadraticLattice():
    """

  This class represents a (possibly not full rank) lattice in a
  fixed quadratic space (V, Q), which is regarded as a discrete
  additive subgroup of V which we may also require to be an R--module
  where the quotient field of R has a natural inclusion to the base
  field of V.  Here we do not distinguish the basis describing L as
  ordered, though we do store an internal set of generators, which
  we require to be a basis when the integers of the base field are
  not free.

  """



    def __init__(self, QS, list_of_lattice_generators=None, base_ring=ZZ):
        """

    If no list of lattice generators is passed, then use the
    standard basis by default.



    TO DO:
      Add support for a matrix (of column vectors) whose span defines the lattice.



    INPUT:
      QS -- a quadratic space
      lattice_info -- a list of vectors whose span defines the lattice
      base_ring -- the ring over which our lattice is a module

    OUTPUT:
      a quadratic lattice

    EXAMPLES:
      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(ZZ, [1,1]))
      sage: L = QuadraticLattice(QS, [[1,1], [1,-1]]); L
      Quadratic Lattice in Quadratic space defined by the Quadratic form in 2 variables over Rational Field with coeff
icients:
      [ 1 0 ]
      [ * 1 ]
      spanned by ((1, 1), (0, 2)).

    """
        ## Check that the base ring is compatible with the quadratic space
        if base_ring != ZZ:
            raise NotmplementedError, "Presently we only support the base-ring ZZ."

        ## Store the ambient quadratic space and base ring
        self.__quadratic_space = QS
        self.__base_ring = base_ring
```



```
        ## Setup and store an ambient free module, and another one corresponding
 to the lattice.
        FM = FreeModule(ZZ, QS.dim())
        self.__ambient_free_module = FM
        if list_of_lattice_generators == None:
            LM = FM.span(Matrix(ZZ, QS.dim(), QS.dim(), 1).columns())   ## Use t
he standard basis by default
        else:
            LM = FM.span(list_of_lattice_generators)
        self.__lattice_free_module = LM


        ## ADD THIS SOMEHOW...  to allow us to choose the basis for a quadratic
lattice (as with SBSs already)!
        #self.__ambient_bilinear_space = V
        #self.__lattice_module = FreeModule_submodule_with_basis_pid(V.vector_sp
ace(), basis)
        #self.__base_ring = V.base_field().ring_of_integers()









    def __repr__(self):
        """
    Print a string describing the quadratic lattice.

    INPUT:
      None

    OUTPUT:
      a string

    EXAMPLES:
      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(ZZ, [1,1]))
      sage: L = QuadraticLattice(QS, [[1,1], [1,-1]]); L
      Quadratic Lattice in Quadratic space defined by the Quadratic form in 2 variables over Rational Field with coeff
icients:
      [ 1 0 ]
      [ * 1 ]
      spanned by ((1, 1), (0, 2)).

      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(ZZ, [1,2]))
      sage: L = QuadraticLattice(QS); L
      Quadratic Lattice in Quadratic space defined by the Quadratic form in 2 variables over Rational Field with coeff
icients:
      [ 1 0 ]
      [ * 2 ]
      spanned by ((1, 0), (0, 1)).

      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(ZZ, []))
      sage: L = QuadraticLattice(QS); L
      Quadratic Lattice in Quadratic space defined by the Quadratic form in 0 variables over Rational Field with coeff
icients:
      spanned by ().

    """
        return "Quadratic Lattice in " + str(self.quadratic_space()) + "spanned by " + str
(self.__lattice_free_module.gens()) + "."



    def __call__(self, v):
        """
    Evaluate the value of the ambient quadratic form on the given
    vector in the quadratic space.
```

```
    Note: This is evaluated as an element of the base field of the
    quadratic space, since the vectormay not be integer-valued.


    INPUT:
      a vector, tuple or list of n numbers in the base_field

    OUTPUT:
      a number in the base field

    EXAMPLES:
      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(ZZ, [1,2]))
      sage: L = QuadraticLattice(QS); L
      Quadratic Lattice in Quadratic space defined by the Quadratic form in 2 variables over Rational Field with coeff
icients:
      [ 1 0 ]
      [ * 2 ]
      spanned by ((1, 0), (0, 1)).
      sage: L([1,1]) == 3
      True

      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(ZZ, []))
      sage: L = QuadraticLattice(QS); L
      Quadratic Lattice in Quadratic space defined by the Quadratic form in 0 variables over Rational Field with coeff
icients:
      spanned by ().
      sage: L([]) == 0
      True

    """
        return self.__quadratic_space(v)


    def __eq__(self, other):
        """

    Determine if two quadratic lattices are equal (meaning that
    they have the same underlying quadratic space V and give the
    same subset of V).


    INPUT:
      other -- a quadratic lattice

    OUTPUT:
      boolean

    EXAMPLES:
      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(ZZ, [1,2]))
      sage: L1 = QuadraticLattice(QS)
      sage: L2 = QuadraticLattice(QS, [[3, 4], [3,3], [1,0], [0,0]])
      sage: L2 == L1
      True
      sage: L1 == L2
      True

      sage: QS_new = QuadraticSpace(QQ, DiagonalQuadraticForm(ZZ, [2,1]))
      sage: L_new = QuadraticLattice(QS_new)
      sage: L1 == L_new
      False
      sage: L2 == L_new
      False
      sage: L_new == L1
      False
      sage: L_new == L2
      False

    """
```



```
        ## Check that ambient quadratic spaces are equal
        if self.__quadratic_space != other.__quadratic_space:
            return False

        ## Check that the lattices have the same span in the quadratic space
        return self.__lattice_free_module == other.__lattice_free_module




    def base_ring(self):
        """
    Returns the base ring over which we consider the quadratic
    lattice as a module.

    INPUT:
      None

    OUTPUT:
      a ring

    EXAMPLES:
      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(ZZ, [1,2]))
      sage: L = QuadraticLattice(QS)
      sage: L.base_ring()
      Integer Ring

      #sage: QS = QuadraticSpace(FiniteField(3), DiagonalQuadraticForm(ZZ, [1, 2, 3, 4, 5]))
      #sage: L = QuadraticLattice(QS)
      #sage: L.base_ring()
      #Integer Ring            ## We would like this to be 'Finite Field of size 3', but we'll need to think about how to s
ay the image of ZZ in a given field...


    """
        return deepcopy(self.__base_ring)


    def quadratic_space(self):
        """
    Returns the ambient quadratic space.

    INPUT:
      None

    OUTPUT:
      a quadratic space

    EXAMPLES:
      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(ZZ, [1, 5, 0, 1]))
      sage: L = QuadraticLattice(QS)
      sage: L.quadratic_space()
      Quadratic space defined by the Quadratic form in 4 variables over Rational Field with coefficients:
      [ 1 0 0 0 ]
      [ * 5 0 0 ]
      [ * * 0 0 ]
      [ * * * 1 ]

    """
        return deepcopy(self.__quadratic_space)


    def lattice_free_module(self):
        """
    Returns the free module giving the lattice.
```

TO DO: Fix this!

INPUT:
None

OUTPUT:
The free module representing the lattice.

EXAMPLES:
sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(ZZ, [1, 5]))
sage: L = QuadraticLattice(QS, [[1,1], [0, −1]])
sage: L.lattice_free_module()
Free module of degree 2 and rank 2 over Integer Ring
Echelon basis matrix:
[1 0]
[0 1]

```
    """
        return self.__lattice_free_module            ## Note:  This doesn't need
(and can't use) deepcopy because freemodules are immutable! =)
```

```
    def quadratic_form__integral(self):
        """
```

Returns a quadratic form associated to the quadratic lattice
(if it is free) in the given basis for the lattice, which is
the restriction of the quadratic form on the ambient quadratic
space. If it is not free, then we raise an exception.

This quadratic form is defined over the base field of the
quadratic space, since there are no guarantees that it is
integer−valued.

TO DO: We would really like to think of the associated
quadratic form as being rational−valued, with the equivalence
for it defined as integral equivalence. Unfortunately,
support for this kind of quadratic form has not been added.

INPUT:
None

OUTPUT:
A quadratic form over a field.

EXAMPLES:
sage: QS = QuadraticSpace(QQ, QuadraticForm(ZZ, 3, range(1,7)))
sage: L = QuadraticLattice(QS)
sage: L.quadratic_form__integral()
Quadratic form in 3 variables over Integer Ring with coefficients:
[1 2 3]
[* 4 5]
[* * 6]

```
    """
        ## Check that the lattice is free
        if not self.is_free():
            return NotImplementedError, "Presently support for testing freeness is not supported for
non−PID's."

        ## Make a quadratic form using the Gram inner product on the underlying
quadratic space.
        Basis = self.basis()
        r = len(Basis)
        new_QF = QuadraticForm(self.base_ring(), r)
        for i in range(r):
            for j in range(i, r):
                if i == j:
                    new_QF[i,j] = self.__quadratic_space.inner_product__gram(Bas
is[i], Basis[j])
                else:
                    new_QF[i,j] = 2 * self.__quadratic_space.inner_product__gram
(Basis[i], Basis[j])

        ## Return the quadratic form
        return new_QF
```



```
    def quadratic_form__rational(self):
        """
```

Returns a quadratic form associated to the quadratic lattice
(if it is free) in the given basis for the lattice, which is
the restriction of the quadratic form on the ambient quadratic
space. If it is not free, then we raise an exception.

This quadratic form is defined over the base field of the
quadratic space, since there are no guarantees that it is
integer−valued.

TO DO: We would really like to think of the associated
quadratic form as being rational−valued, with the equivalence
for it defined as integral equivalence. Unfortunately,
support for this kind of quadratic form has not been added.

INPUT:
None

OUTPUT:
A quadratic form over a field.

EXAMPLES:
sage: QS = QuadraticSpace(QQ, QuadraticForm(ZZ, 3, range(1,7)))
sage: L = QuadraticLattice(QS)
sage: L.quadratic_form__rational()
Quadratic form in 3 variables over Rational Field with coefficients:
[1 2 3]
[* 4 5]
[* * 6]

```
    """
        ## Check that the lattice is free
        if not self.is_free():
            return NotImplementedError, "Presently support for testing freeness is not supported for
non−PID's."

        ## Make a quadratic form using the Gram inner product on the underlying
quadratic space.
        Basis = self.basis()
        r = len(Basis)
        new_QF = QuadraticForm(self.__quadratic_space.base_field(), r)
        for i in range(r):
            for j in range(i, r):
                if i == j:
                    new_QF[i,j] = self.__quadratic_space.inner_product__gram(Bas
is[i], Basis[j])
                else:
                    new_QF[i,j] = 2 * self.__quadratic_space.inner_product__gram
(Basis[i], Basis[j])

        ## Return the quadratic form
        return new_QF
```

```
    def is_free(self):
        """
    Determines if the given lattice is a free over its base
    ring.

    TO DO:  IMPLEMENT THIS IN ANY NON-TRIVIAL CASE -- LIKE ORDERS IN NUMBER FIELDS!!!


    INPUT:
        None

    OUTPUT:
        boolean

    EXAMPLES:
        sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1,1]))
        sage: L = QuadraticLattice(QS)
        sage: L.is_free()
        True

        #sage: x = polygen(QQ, 'x')
        #sage: F.<a> = NumberField(x^2 + 5)
        #sage: R = F.ring_of_integers()
        #sage: QS = QuadraticSpace(F, DiagonalQuadraticForm(QQ, [1,1]))
        #sage: L1 = QuadraticLattice(QS)
        #sage: L1.is_free()
        #True
        #
        #sage: L2 = QuadraticLattice(QS, [[1,0], [0, 1 + a]])
        #sage: L2.is_free()
        #False

    """
        ## Check if the base ring is a PID
        if is_PrincipalIdealDomain(self.__base_ring) == True:
            return True

        ## IMPLEMENT THIS!!!
        return NotImplementedError, "Presently support for testing freeness is not supported for non-P
ID's."



    def is_full_rank(self):
        """
    Determines if the lattice has full rank in the ambient
    quadratic space.

    INPUT:
        None

    OUTPUT:
        boolean

    EXAMPLES:
        sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1,1]))

        sage: L = QuadraticLattice(QS)
        sage: L.is_full_rank()
        True

        sage: L = QuadraticLattice(QS, [])
        sage: L.is_full_rank()
        False

        sage: L = QuadraticLattice(QS, [[1,1]])
        sage: L.is_full_rank()
        False
```



```
        sage: L = QuadraticLattice(QS, [[1,1], [2,2]])
        sage: L.is_full_rank()
        False

        sage: L = QuadraticLattice(QS, [[1,1], [1,0]])
        sage: L.is_full_rank()
        True

    """
        if self.__base_ring == ZZ:
            return self.__lattice_free_module.rank() == self.__quadratic_space.d
im()
        else:
            raise NotImplementedError, "Rank finding is not supported unless we're over ZZ."



    def basis(self):
        """
    Returns a basis for the quadratic lattice if it is free, and
    an error otherwise.

    INPUT:
        None

    OUTPUT:
        A list of vectors

    EXAMPLES:
        sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1,1]))

        sage: L = QuadraticLattice(QS, [[1,1], [2,2]])
        sage: L.basis()
        [
        (1, 1)
        ]

        sage: L = QuadraticLattice(QS, [[1,1], [1,0]])
        sage: L.basis()
        [
        (1, 0),
        (0, 1)
        ]

    """
        if self.is_free():
            return self.__lattice_free_module.basis()
        else:
            raise NotImplementedError, "Basis finding is not supported unless we're over ZZ."



    def generators(self):
        """
    Returns a set of generators for the quadratic lattice, as a
    module over its base ring.

    TO DO: Add some non-trivial example where the basis and
    generators are distinct! (I.e. a non-free module!)


    INPUT:
        None

    OUTPUT:
        A list of vectors
```

```
EXAMPLES:
    sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1,1]))

    sage: L = QuadraticLattice(QS, [[1,1], [2,2]])
    sage: L.generators()
    ((1, 1),)

    sage: L = QuadraticLattice(QS, [[1,1], [1,0]])
    sage: L.generators()
    ((1, 0), (0, 1))

"""
        return self.__lattice_free_module.gens()



    def has_same_quadratic_space(self, other):
        """
Determines if self and other are lattices in the same
quadratic space (as an equality, not just an isomorphism).

INPUT:
    other -- a quadratic lattice

OUTPUT:
    boolean

EXAMPLES:
    sage: QS1 = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1,1]))
    sage: QS2 = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1,4]))

    sage: L1 = QuadraticLattice(QS1)
    sage: L2 = QuadraticLattice(QS2)
    sage: L3 = QuadraticLattice(QS1, [[1,1]])

    sage: L1.has_same_quadratic_space(L1)
    True

    sage: L1.has_same_quadratic_space(L2)
    False
    sage: L2.has_same_quadratic_space(L1)
    False

    sage: L1.has_same_quadratic_space(L3)
    True

    sage: L2.has_same_quadratic_space(L3)
    False

"""
        return self.__quadratic_space == other.__quadratic_space



    def intersect_with(self, other):
        """
Returns the quadratic lattice defined as the intersection of
the two given lattices (which must be defined on the same
ambient quadratic space, or an error is raised).

TO DO:  Fix this to use initialization of quadratic spaces from free modules when this is supported!


INPUT:
    other -- a quadratic lattice on the same (equal) quadratic space

OUTPUT:
```



```
    a quadratic lattice

EXAMPLES:
    sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1,1]))
    sage: L1 = QuadraticLattice(QS, [[1, 1]])
    sage: L2 = QuadraticLattice(QS, [[5, 0], [0, 3]])

    sage: L3 = L1.intersect_with(L2)
    sage: L3 == L2.intersect_with(L1)
    True

    sage: L3
    Quadratic Lattice in Quadratic space defined by the Quadratic form in 2 variables over Rational Field with coeff
icients:
    [ 1 0 ]
    [ * 1 ]
    spanned by ((15, 15),).

"""
        ## Check that both lattices live on the same quadratic space
        if not self.has_same_quadratic_space(other):
            raise TypeError, "The two quadratic lattices live on different quadratic spaces!"

        ## Return the intersection of the two quadratic latties
        return QuadraticLattice(self.quadratic_space(), self.__lattice_free_modu
le.intersection(other.__lattice_free_module).basis())



    def sum_with(self, other):
        """
    Returns the quadratic lattice defined as the sum of the two
    given lattices (which must be defined on the same ambient
    quadratic space, or an error is raised).

    INPUT:
        other -- a quadratic lattice on the same ambient quadratic space

    OUTPUT:
        a quadratic lattice on the same quadratic space

    EXAMPLES:
        sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1,1]))

        sage: L1 = QuadraticLattice(QS, [[1, 1]])
        sage: L2 = QuadraticLattice(QS, [[1, -1]])
        sage: L3 = L1.sum_with(L2); L3
        Quadratic Lattice in Quadratic space defined by the Quadratic form in 2 variables over Rational Field with coeff
icients:
        [ 1 0 ]
        [ * 1 ]
        spanned by ((1, 1), (0, 2)).

        sage: L3 == L2.sum_with(L1)
        True

        sage: L4 = QuadraticLattice(QS, [[3, 3]])
        sage: L5 = QuadraticLattice(QS, [[5, 5]])
        sage: L6 = L4.sum_with(L5); L6
        Quadratic Lattice in Quadratic space defined by the Quadratic form in 2 variables over Rational Field with coeff
icients:
        [ 1 0 ]
        [ * 1 ]
        spanned by ((1, 1),).

    """
        ## Do the sum with another quadratic lattice
        if isinstance(other, QuadraticLattice):
```

```
            ## Check that both lattices live on the same quadratic space
            if not self.has_same_quadratic_space(other):
                raise TypeError, "The two quadratic lattices live on different quadratic spaces!"

            ## Return the lattice generated by generators of both lattices
            return QuadraticLattice(self.quadratic_space(), self.generators() +
other.generators())

        ## Otherwise do the sum with a list of vectors (taken to generate some o
ther possibly non full rank lattice) we're passing in
        elif isinstance(other, list):

            ## Return the lattice generated by generators of both lattices
            return QuadraticLattice(self.quadratic_space(), self.generators() +
other)

        ## Otherwise raise an error
        else:
            raise TypeError, "You must sum with eithe a QuadraticLattice or a list of vectors coercible t
o the ambient quadratic space."



    def is_integer_valued(self):
        """
    Determines if the ambient quadratic space assumes only integer
    values (by which we mean values in the base_ring) on the
    quadratic lattice.

    INPUT:
      None

    OUTPUT:
      boolean

    EXAMPLES:
      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1,4]))
      sage: L1 = QuadraticLattice(QS, [[1, 1]])
      sage: L1.is_integer_valued()
      True

      sage: L2 = QuadraticLattice(QS)
      sage: L2.is_integer_valued()
      True

      sage: L3 = QuadraticLattice(QS, [[0, 1/2]])
      sage: L3.is_integer_valued()
      True

      sage: L4 = QuadraticLattice(QS, [[1/2, 0]])
      sage: L4.is_integer_valued()
      False


    """
        try:
            self.quadratic_form__integral()
            return True
        except:
            return False



    def apply_linear_transformation_on_left(self, T):
        """
    Return the lattice given by applying the linear transformation
    T to the given lattice (where T acts on column vectors by
    left-multiplication!).  Here T can also be a scalar.
```



```
      INPUT:
        T -- a matrix giving a linear transformation on
             the underlying quadratic space, or a scalar.

      OUTPUT:
        a quadratic lattice


      EXAMPLES:
        sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1,1]))
        sage: L = QuadraticLattice(QS, [[1, 1]])
        sage: T1 = Matrix(ZZ, 2, 2, [1,2,3,4])
        sage: L1 = L.apply_linear_transformation_on_left(T1); L1
        Quadratic Lattice in Quadratic space defined by the Quadratic form in 2 variables over Rational Field with coeff
icients:
        [ 1 0 ]
        [ * 1 ]
        spanned by ((3, 7),).

        sage: T2 = Matrix(ZZ, 2, 2, -1)
        sage: L2 = L.apply_linear_transformation_on_left(T2); L2
        Quadratic Lattice in Quadratic space defined by the Quadratic form in 2 variables over Rational Field with coeff
icients:
        [ 1 0 ]
        [ * 1 ]
        spanned by ((1, 1),).

        sage: L2 == L
        True

      """
          gens = self.generators()
          new_L = QuadraticLattice(self.quadratic_space(), [T * gens[i]  for i in
range(len(gens))])
          return new_L



      def apply_linear_transformation_on_right(self, T):
          """
      Return the lattice given by applying the linear transformation
      T to the given lattice (where T acts on column vectors by
      right-multiplication!).  Here T can also be a scalar.

      INPUT:
        T -- a matrix giving a linear transformation on
             the underlying quadratic space, or a scalar.

      OUTPUT:
        a quadratic lattice


      EXAMPLES:
        sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1,1]))
        sage: L = QuadraticLattice(QS, [[1, 1]])
        sage: T1 = Matrix(ZZ, 2, 2, [1,2,3,4])
        sage: L1 = L.apply_linear_transformation_on_right(T1); L1
        Quadratic Lattice in Quadratic space defined by the Quadratic form in 2 variables over Rational Field with coeff
icients:
        [ 1 0 ]
        [ * 1 ]
        spanned by ((4, 6),).

        sage: T2 = Matrix(ZZ, 2, 2, -1)
        sage: L2 = L.apply_linear_transformation_on_right(T2); L2
        Quadratic Lattice in Quadratic space defined by the Quadratic form in 2 variables over Rational Field with coeff
icients:
        [ 1 0 ]
        [ * 1 ]
```

```
        spanned by ((1, 1),).

        sage: L2 == L
        True


        sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(ZZ, [1,1,1]))
        sage: M = Matrix(ZZ, 3, 3, range(9)); M
        [0 1 2]
        [3 4 5]
        [6 7 8]
        sage: QL =  QuadraticLattice(QS, [M.column(i)  for i in range(2)]); QL
        Quadratic Lattice in Quadratic space defined by the Quadratic form in 3 variables over Rational Field with coeff
icients:
        [ 1 0 0 ]
        [ * 1 0 ]
        [ * * 1 ]
        spanned by ((1, 1, 1), (0, 3, 6)).
        sage: QL.apply_linear_transformation_on_right(M[:2, :3])
        Quadratic Lattice in Quadratic space defined by the Quadratic form in 3 variables over Rational Field with coeff
icients:
        [ 1 0 0 ]
        [ * 1 0 ]
        [ * * 1 ]
        spanned by ((1, 4, 7), (0, 9, 18)).

    """
        ## Check that the linear transformation has the correct size!

        ## Perform the linear transformation
        gens = self.generators()
        M = Matrix(self.quadratic_space().base_field(), len(gens), self.quadrati
c_space().dim(), gens).transpose()  ## Makes the column matrix
        new_L = QuadraticLattice(self.quadratic_space(), (M * T).columns())
        return new_L

        #gens = self.generators()
        #T_transpose = T.transpose()
        #new_L = QuadraticLattice(self.quadratic_space(), [T_transpose * gens[i]
  for i in range(len(gens))])
        #return new_L




    def inner_product__hessian(self, v, w):
        """
    Compute the (Hessian) inner product of the given vectors in
    the quadratic space.  This inner product is also used for
    computing the dual lattice.

    INPUT:
      v, w -- vectors, lists or tuples defining a vector in the ambient quadratic space of the quadratic lattice.

    OUTPUT:
      an element of the base field of the quadratic space

    EXAMPLES:
      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1,-1]))
      sage: L = QuadraticLattice(QS)
      sage: L.inner_product__hessian((1,0), (1,0))
      2
      sage: L.inner_product__hessian((1,0), (0,1))
      0

    """
        return self.__quadratic_space.inner_product__hessian(v, w)
```



```
    def Hessian_matrix(self, rational_matrix=False):
        """
    Compute the Hessian matrix w.r.t the basis of this quadratic lattice.
    """
        B = self.basis()
        n = len(B)
        if rational_matrix:
            F = self.quadratic_space().base_field()
            return Matrix(F, n, n, [self.inner_product__hessian(B[i], B[j])  for
 i in range(n)  for j in range(n)])
        else:
            R = self.base_ring()
            return Matrix(R, n, n, [self.inner_product__hessian(B[i], B[j])  for
 i in range(n)  for j in range(n)])


    def dual_lattice(self):
        """
    Compute the dual lattice with respect to the Hessian bilinear
    form associated to the quadratic form on the underlying
    quadratic space.

    INPUT:
      None

    OUTPUT:
      a quadratic lattice in the same ambient quadratic space.

    EXAMPLES:
      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1,1]))
      sage: L = QuadraticLattice(QS)
      sage: L_dual = L.dual_lattice(); L_dual
      Quadratic Lattice in Quadratic space defined by the Quadratic form in 2 variables over Rational Field with coeff
icients:
      [ 1 0 ]
      [ * 1 ]
      spanned by ((1/2, 0), (0, 1/2)).

      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [2,45/11,1/12]))
      sage: L = QS.integral_lattice()
      sage: L.dual_lattice()
      Quadratic Lattice in Quadratic space defined by the Quadratic form in 3 variables over Rational Field with coeff
icients:
      [ 2 0 0 ]
      [ * 45/11 0 ]
      [ * * 1/12 ]
      spanned by ((1/4, 0, 0), (0, 1/90, 0), (0, 0, 1)).

    """
        ## Check that the lattice is free -- useful for now, but can be easily c
ircumvented by dealing with generators instead!
        if not self.is_free():
            raise NotImplementedError, "We currently don't have an implementation for non-free la
ttices..."

        ## Compute the dual basis w.r.t. the Hessian bilinear form.
        H = self.__quadratic_space.hessian_matrix()      ## Hessian matrix in st
d basis
        A = self.__lattice_free_module.basis_matrix()       ## Matrix of basis f
or L, as row vectors.

        ## Solve A * H * B = Id to get a basis of the dual lattice
        B = (A * H).inverse()

        return QuadraticLattice(self.quadratic_space(), B.columns())
```

```
    #def LLL_reduced(self, give_transformation=False):
    #    """
    #    Returns the same quadratic lattice in an LLL-reduced basis.
    #    If return_transformation=True then we additionally return the
    #    transformation (on column vectors) used to produce the new basis.
    #
    #    """
    #    H = self.Hessian_matrix()
    #    U = H.LLL_gram()
    #    if give_transformation:
    #        return self.apply_linear_transformation_on_right(U), U
    #    else:
    #        return self.apply_linear_transformation_on_right(U)


################################################################################
#############
################################################################################
#############
################################################################################
#############
####  BELOW THIS POINT WE IGNORE THE CODE!
################################################################################
#############
################################################################################
#############
################################################################################
#############

def Watson_superlattice(self):
    """

  Find the Watson superlattice of the given lattice, defined by the
  formula


  Note: Presenty this code comes from the
  QuadraticForm.maximal_form() method.

  """

    ## Compute the exponent of L#/L
    B = self.matrix()

    ## Precompute the exponent/index data once
    ed = B.elementary_divisors()
    max_ed = ed[-1]
    f = max_ed.squarefree_part()
    fa = sqrt(f * max_ed)
    a = fa / f

    #print
    #print "f = ", f
    #print "a = ", a
    #print "fa = ", fa

    ## Try to find the Watson superlattice
    while a != 1:
        B_inv = B.inverse()
        generator_mat = matrix(ZZ, max_ed * B_inv)
        X = generator_mat.augment(matrix(ZZ, n, n, a))
        gen_lll = X.transpose().LLL().transpose().matrix_from_columns(range(n,2*
n))
        ##assume that the last rows returned by LLL constitue the basis of the l
attice

        B = matrix(ZZ, gen_lll.transpose() * B * gen_lll / (a*a))
```



```
        ## Save the transformation matrix
        Big_trans = Big_trans * gen_lll/a

        ## Recompute the index/exponent data for L#/L
        ed = B.elementary_divisors()
        max_ed = ed[-1]

        f = max_ed.squarefree_part()

        ## DIAGNOSTIC
        #print "X = ", X
        #print "gen_lll = ", gen_lll
        verbose("B = " + str(B))
        verbose("")
        verbose("f = " + str(f))

        fa = sqrt(f * max_ed)
        a = fa / f

        ## DIAGNOSTIC
        verbose("a = " + str(a))
        verbose("fa = " + str(fa))



    ## Deal with the 1-dimensional case:
    if n == 1:
        new_coeff = squarefree_part(self[0,0])
        Q_maximal = deepcopy(self)
        Q_maximal.__init__(ZZ, 1, [new_coeff])
        if return_transformation:
            return Q_maximal, Matrix(QQ, 1, 1, [sqrt(self[0,0]/new_coeff)])
        else:
            return Q_maximal




    ## Return the Watson form (for a superlattice)
    verbose("Watson lattice B = " + str(B))
    verbose("Big_trans = " + str(Big_trans))




## ============================================================================
=======================================


#    def maximal_integer_valued_superlattice(self):
#        """
#        Produces some integer-valued superlattice of the given
#        quadratic lattice, assuming that the given lattice is
#        integer-valued.  If not, then an exception/error is raised.
#
#        """
#        pass



## ============================================================================
=======================================


#    def neighbors_of_index(self, m):
#        """
```

```
#         Returns the set of m-neighbors of the given quadratic lattice.
#
#         """
#         pass



#     def discriminant_group(self):
#         """
#
#         Returns the discriminant group, which is a module with an
#         induced (QQ/ZZ)-valued quadratic form.
#
#         """
#         pass
```

```
## Required structures:
## --------------------



from sage.quadratic_forms.square_classes import SquareClass, local_squareclass_r
epresentatives_list, local_squareclass_radius_val
from sage.quadratic_forms.weak_approx import weak_approx_for_numbers_over_QQ, \
                                             weak_approx_for_squareclasses_over_
QQ, \
                                             strong_approx_for_squareclasses_by_
QQ_except_at_one_prime

from sage.quadratic_forms.localization import Qv

from sage.quadratic_forms.quadratic_lattice import QuadraticLattice



from sage.rings.arith import hilbert_symbol, legendre_symbol, valuation, is_squa
re, is_prime, prime_divisors, is_squarefree
from sage.rings.integer_ring import ZZ
from sage.rings.finite_rings.constructor import GF


from sage.functions.other import floor, sqrt


from sage.misc.functional import squarefree_part, is_even, is_odd
from sage.misc.misc import prod, verbose
from sage.misc.mrange import mrange


from sage.quadratic_forms.quadratic_form import QuadraticForm, DiagonalQuadratic
Form
from sage.quadratic_forms.extras import least_quadratic_nonresidue
from sage.functions.generalized import sgn
from sage.matrix.all import is_Matrix
from sage.matrix.constructor import matrix, Matrix

from sage.rings.field import Field


from sage.rings.rational_field import is_RationalField, QQ, RationalField
from sage.rings.real_mpfr import is_RealField, RealField
from sage.rings.all import is_ComplexField, is_pAdicField, is_FiniteField



from sage.rings.infinity import Infinity
from sage.rings.padics.factory import Qp

from sage.rings.polynomial.polynomial_ring_constructor import PolynomialRing
from sage.modules.free_module_element import vector


from copy import deepcopy


#from sage.rings.arith import is_square, is_prime, valuation, legendre_symbol

from sage.quadratic_forms.symmetric_bilinear import SymmetricBilinearSpace
```



```
################################################
## Create a QuadraticSpace class which defines a ##
## quadratic space over a local or global field. ##
################################################

class QuadraticSpace():
    """
  Defines a quadratic space, by which we mean a diagonal quadratic form over a field.
  """


    def __init__(self, K, coeffs=None, matrix_type='Gram'):
        """
      Initializes a quadratic space over a given field from:
        1) a symmetric matrix (either Gram or Hessian)
        2) a list of diagonal entries
        ##3) local invariants (for a local field)

      If the coefficients are not elements of the base field, then
      they must automatically coerce into it or a RuntimeError will
      be raised.

      Valid syntax possibilities:
        QuadraticSpace(Q) -- where Q is a quadratic form defined over a field
        QuadraticSpace(K, Q) -- where Q is a quadratic form/space with coefficients coercible to the field K
        QuadraticSpace(K, [a_1, a_2, ..., a_n]) -- where K is a field and the elements a_1, ..., a_n are coercible to elem
ents of K.

      The option 'matrix_type' is only used if we are respectively
      given a matrix or quadratic form as input.  The setting
      matrix_type describes the type of matrix that is passed in.
      If matrix_type is 'Hessian', then we create a quadratic space
      whose Hessian matrix agrees with this form (by taking half of
      this matrix first).

      CONVENTION/BIG ASSUMPTION: The matrix associated to a
      quadratic space is always its Gram matrix, and we assume that
      the field is of characteristic not 2 (since this is the
      customary usage for most people!).


      INPUT:
        K -- a local or global field, or possibly a symmetric matrix or a quadratic form defined over a field.
        coeffs -- either a list of diagonal coefficients, a quadratic form or a symmetric matrix
        matrix_type -- either 'Gram' or 'Hessian'

      OUTPUT:
        none

      INTERNAL VARIABLES:
        self.__quadratic_form -- the underlying quadratic form, used for printing or evaluation in the standard basis
        self.__diagonal_form -- used for internal computations of invariants
        self.__gram_matrix -- used for computing inner products

      EXAMPLES:
        sage: QS = QuadraticSpace(DiagonalQuadraticForm(QQ, [1,2,3])); QS
        Quadratic space defined by the Quadratic form in 3 variables over Rational Field with coefficients:
        [ 1 0 0 ]
        [ * 2 0 ]
        [ * * 3 ]

        sage: QS = QuadraticSpace(Qp(5), DiagonalQuadraticForm(QQ, [1,2,3])); QS
        Quadratic space defined by the Quadratic form in 3 variables over 5-adic Field with capped relative precision 2
0 with coefficients:
        [ 1 + O(5^20) 0 0 ]
        [ * 2 + O(5^20) 0 ]
```

```
        [ * * 3 + O(5^20) ]

        sage: QS = QuadraticSpace(QQ, DiagonalMatrix(QQ,[1,2,3]), matrix_type="Gram"); QS
        Quadratic space defined by the Quadratic form in 3 variables over Rational Field with coefficients:
        [ 1 0 0 ]
        [ * 2 0 ]
        [ * * 3 ]
        sage: QS.gram_matrix()
        [1 0 0]
        [0 2 0]
        [0 0 3]


        sage: A = matrix(QQ, 2, 2, [3,1,1,3])
        sage: QS = QuadraticSpace(QQ, A, matrix_type="Gram"); QS
        Quadratic space defined by the Quadratic form in 2 variables over Rational Field with coefficients:
        [ 3 2 ]
        [ * 3 ]
        sage: QS.gram_matrix() == A
        True

        sage: QS = QuadraticSpace(QQ, A, matrix_type="Hessian"); QS
        Quadratic space defined by the Quadratic form in 2 variables over Rational Field with coefficients:
        [ 3/2 1 ]
        [ * 3/2 ]
        sage: QS.hessian_matrix() == A
        True

    """
        ## Validate the input:
        ## -------------------
        if not isinstance(K, (Field, QuadraticForm)):
            raise TypeError, "The first argument must be either a field or a quadratic form!"

        if (coeffs != None) and not (is_Matrix(coeffs) or isinstance(coeffs, (list, QuadraticForm, QuadraticSpace))):
            raise TypeError, "The second argument entry must be either a list of coefficients, a quadratic form, a quadratic space or a (symmetric) matrix!"


        ## INPUT #1:  Check for the syntax QuadraticSpace(Q) where Q is a QuadraticForm defined over a base_ring which is a field.
        if is_Matrix(K) or isinstance(K, QuadraticForm):

            ## Sanity Check: Verify that coeffs has not been set redundantly.
            if (coeffs != None):
                raise TypeError, "Invalid Syntax -- only one argument is allowed when the first argument is a matrix or a quadratic form."

            coeffs = K

            ## Check that the quadratic form is defined over a field.
            base_field = coeffs.base_ring()
            if not isinstance(base_field, Field):
                raise TypeError, "The syntax QuadraticSpace(Q) requires that the QuadraticForm (or QuadraticSpace or Matrix) Q is defined over a field -- it is defined over " + str(Q.base_ring()) + "."

        ## INPUT #2  Check for the syntax QuadraticSpace(K, Q) where K is a field and Q is a quadratic form or a quadratic space
        else:
            base_field = K

            ## Test for fields of characteristic 2      -- This is only partially supported for now!  self.anisotropic_dim() should work! =)
            #if K.characteristic() == 2:
            #    raise NotImplementedError, "Fields of characteristic 2 are not currently supported."
```



```
        ## Initialize the quadratic space:
        ## -------------------------------

        ## Initialize from a QuadraticSpace
        if isinstance(coeffs, QuadraticSpace):
            self.__quadratic_form = coeffs.defining_quadratic_form().base_change_to(base_field)
            self.__hessian_bilinear_space = SymmetricBilinearSpace(base_field, self.__quadratic_form.Hessian_matrix())
            #self.__gram_matrix = self.__quadratic_form.Gram_matrix()
            self.__diagonal_form = self.__quadratic_form.rational_diagonal_form()
            self.__diagonal_squareclass_list = [SquareClass(self.base_field(), self.__diagonal_form[i,i])  for i in range(self.dim())]

        ## Initialize from a list of coefficients
        elif isinstance(coeffs, list):
            self.__quadratic_form = DiagonalQuadraticForm(base_field, coeffs)
            self.__hessian_bilinear_space = SymmetricBilinearSpace(base_field, self.__quadratic_form.Hessian_matrix())
            #self.__gram_matrix = self.__quadratic_form.Gram_matrix()
            self.__diagonal_form = self.__quadratic_form
            self.__diagonal_squareclass_list = [SquareClass(self.base_field(), self.__diagonal_form[i,i])  for i in range(self.dim())]

        ## Initialize from a Matrix (by converting it to a quadratic form)
        elif is_Matrix(coeffs):
            if not coeffs.is_symmetric():
                raise TypeError, "The given input matrix \n" + str(coeffs) + "\nmust be symmetric."

            ## Create the associated quadratic form Q for this matrix (so that the Gram Matrix of Q is the quadratic form on the ambient quadratic space.)
            if matrix_type == "Gram":
                self.__quadratic_form = QuadraticForm(base_field, coeffs, init_from_gram_matrix=True)
                #self.__quadratic_form = QuadraticForm(base_field, coeffs).scale_by_factor(ZZ(2))
                self.__hessian_bilinear_space = SymmetricBilinearSpace(base_field, self.__quadratic_form.Hessian_matrix())
                #self.__gram_matrix = self.__quadratic_form.Gram_matrix()
                self.__diagonal_form = self.__quadratic_form.rational_diagonal_form()
                self.__diagonal_squareclass_list = [SquareClass(self.base_field(), self.__diagonal_form[i,i])  for i in range(self.dim())]
            elif matrix_type == "Hessian":
                try:
                    self.__quadratic_form = QuadraticForm(base_field, coeffs)
                    self.__hessian_bilinear_space = SymmetricBilinearSpace(base_field, self.__quadratic_form.Hessian_matrix())
                    #self.__gram_matrix = self.__quadratic_form.Gram_matrix()
                    self.__diagonal_form = self.__quadratic_form.rational_diagonal_form()
                    self.__diagonal_squareclass_list = [SquareClass(self.base_field(), self.__diagonal_form[i,i])  for i in range(self.dim())]
                except:
                    raise RuntimeError, "There is a problem with division by 2 in your field -- does it have characteristic 2?"
            else:
                raise TypeError, "The matrix_type must be either 'Gram' or 'Hessian'."

        ## Initialize from a Quadratic Form
        elif isinstance(coeffs, QuadraticForm):
            self.__quadratic_form = coeffs.base_change_to(base_field)
            self.__hessian_bilinear_space = SymmetricBilinearSpace(base_field, self.__quadratic_form.Hessian_matrix())
            #self.__gram_matrix = self.__quadratic_form.Gram_matrix()
            self.__diagonal_form = self.__quadratic_form.rational_diagonal_form(
```

```
)
            self.__diagonal_squareclass_list = [SquareClass(self.base_field(), s
elf.__diagonal_form[i,i])  for i in range(self.dim())]



    def __repr__(self):
        """
    Print a string describing the quadratic space.

    INPUT:
      None

    OUTPUT:
      a string

    EXAMPLES:
      sage: QS = QuadraticSpace(DiagonalQuadraticForm(QQ, [1/2, 11]))
      sage: QS.__repr__()
      'Quadratic space defined by the Quadratic form in 2 variables over Rational Field with coefficients: \n[ 1/2 0 ]\n
[ * 11 ]\n'
    """
        return "Quadratic space defined by the " + str(self.__quadratic_form)



    def anisotropic_dim(self, QQ_place=None):
        """

    Return the dimension of the maximal anisotropic subspace of
    the quadratic space.  If the quadratic form is defined over
    QQ, then an additional place may be passed to compute the
    anisptropic dimension of the localization of that quadratic
    space at that place.

    Since the computation of the anisotropic dimension depends on
    the base field, currently the only the finite fields, real and
    complex fields, p-adic fields (Qp only), and QQ are supported.

    INPUT:
      QQ_place -- a prime number or Infinity
        (an optional argument which only makes sense if the
        base field is QQ).

    OUTPUT:
      an integer >= 0

    EXAMPLES:
      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1, -1, 2, 3]))
      sage: QS.anisotropic_dim()
      2

      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1, -1, 1, -1]))
      sage: QS.anisotropic_dim()
      0

      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1, 1, 1, 1, 1, 1]))
      sage: QS.anisotropic_dim()
      6

      sage: S2 = SymmetricBilinearSpace(GF(2), DiagonalMatrix(ZZ, [1,1,1,1]))
      sage: Q2 = QuadraticSpace(S2.base_field(), S2.gram_matrix(), matrix_type="Gram"); Q2
      sage: Q2
      Quadratic space defined by the Quadratic form in 4 variables over Finite Field of size 2 with coefficients:
      [ 1 0 0 0 ]
      [ * 1 0 0 ]
      [ * * 1 0 ]
      [ * * * 1 ]
      sage: Q2.anisotropic_dim()
```



```
    1

    """
        ## Deal with (Hessian) degenerate quadratic spaces!
        if self.is_degenerate():
            (R, C) = self.__hessian_bilinear_space.find_basis_of_radical_subspac
e()
            #print "self = " + str(self)
            #print "R = " + str(R)
            #print "C = " + str(C)
            F = self.base_field()
            if C != []:
                M_nondeg = Matrix(F, C).transpose()          ## Matrix of column
s giving a basis for a maximal (Hessian) non-degenerate quadratic space
                nondeg_QS = QuadraticSpace(self.base_field(), self.__quadratic_f
orm(M_nondeg))
                nondeg_aniso_dim = nondeg_QS.anisotropic_dim()
            else:
                nondeg_aniso_dim = 0    ## If there are no matrix entries, then
there is no anisotripic vector!

            ## In characteristic 2, adjust by the dimension of the span of the d
iagonal square-classes (See Kitaoka's book, Thrm 1.2.1 on pp4-5.)
            if F.characteristic() == 2:
                if R != []:
                    M_deg = Matrix(F, R).transpose()          ## Matrix of colum
ns giving a basis for the maximal (Hessian) degenerate quadratic space
                    deg_QF = self.__quadratic_form(M_deg)
                    if is_FiniteField(F):
                        t = 0
                        for i in range(deg_QF.dim()):
                            if deg_QF[i,i] != 0:
                                t = 1
                                break
                    else:
                        raise NotImplementedError, \
                            "Still need to write the anisotropic dim over non-finite fields " + \
                            "-- here we want t to be the dim of span of the squareclasses of diagonal ele
ments!"
                else:
                    t = 0      ## If there are no matrix entries, then there is
no anisotripic vector!

                return nondeg_aniso_dim + t

            else:
                return nondeg_aniso_dim


            ## TO DO:  We can simplify the above code by allowing a 0x0 matrix t
ransformation to always return the empty QF, and never raise an error!
            ## (We could even allow a flag that permits this behavior, if we don
't want to allow it in general.)



        ## Deal with non-degenerate quadratic spaces:
        ## ------------------------------------------
        F = self.base_field()
        n = self.dim()

        ## Case 1: Finite Fields (with char > 2 or char == 2)
        if is_FiniteField(F):
            if n % 2 == 1:
                return 1
            else:

                ## Perform Square-testing to see if the last binary space splits
                if F.characteristic() == 2:
```

```
                    return 0                    ## All elements are squares in cha
racteristic 2
                else:
                    d = self.determinant().representative()
                    d_aniso = d * (-1)**((n-2)/2)

                    if (-d_aniso)**((F.order() - 1) / 2) == 1:   ## Use the Lege
ndre symbol test for characteristic > 2
                        return 0
                    else:
                        return 2

        ## Case 2: p-adic fields Q_p
        elif is_pAdicField(F) or (is_RationalField(F) and (QQ_place != None) and
 is_prime(QQ_place)):
            d = self.determinant().representative()
            p = self.base_field().prime()
            if not is_prime(p):
                raise RunTimeError, "We only support the p-adic fields Q_p where p is a prime for n
ow."
            return local_quadratic_space_anisotropic_dimension_by_invariants(p,
n, d, self.hasse_invariant(p))

        ## Case 3: Real Fields
        elif is_RealField(F) or (is_RationalField(F) and (QQ_place == Infinity))
:
            return abs(self.__quadratic_form.signature())

        ## Case 4: Complex Fields
        elif is_ComplexField(F):
            return n % 2

        ## Case 5: The Rational Numbers QQ:
        ## --------------------------------
        elif is_RationalField(F):

            ## Find the anisotropic dimension at all "bad" places
            d = self.determinant().normalized_representative()
            bad_prime_list = prime_divisors(2*d)
            real_aniso_dim = abs(self.__quadratic_form.signature())
            aniso_dim_list = [local_quadratic_space_anisotropic_dimension_by_inv
ariants(p, n, d, self.hasse_invariant(p))  for p in bad_prime_list] + [real_anis
o_dim]

            ## Compute the generic local anisotropic dimension
            if n % 2 != 0:
                generic_aniso_dim = 1
            else:
                d_aniso = d * (-1)**((n-2)/2)
                ## Check if the 2-dim'l is split over QQ, which determines the e
ven-dim'l generic anisotropic dimension
                if is_square(-d_aniso):
                    generic_aniso_dim = 0
                else:
                    generic_aniso_dim = 2

            ## Compute the actual anisotropic dimension
            aniso_QQ_dim = max(aniso_dim_list + [generic_aniso_dim])
            return aniso_QQ_dim


        ## Case 6: UNSUPPORTED
        else:
            raise NotImplementedError, "The anisotropic dimension calculation is not supported yet f
or this basefield."
```



```
    def anisotropic_det(self):
        """
Return the determinant (squareclass) of the maximal anisotropic subspace of the quadratic space.

INPUT:
  None

OUTPUT:
  a non-zero squareclass

EXAMPLES:
  sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1, -1, 2, 3]))
  sage: QS.anisotropic_det()
  The squareclass represented by 6 over Rational Field

  sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1, -1, 1, -1]))
  sage: QS.anisotropic_det()
  The squareclass represented by 1 over Rational Field

  sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1, 1, 1, 1, 1, 1]))
  sage: QS.anisotropic_det()
  The squareclass represented by 1 over Rational Field

"""
        aniso_dim = self.anisotropic_dim()
        return self.determinant() * (-1)**((self.dim() - aniso_dim) / 2)



    def base_field(self):
        """
Returns the base field of scalars for the quadratic space as a vectorspace.

INPUT:
  None

OUTPUT:
  a field

EXAMPLES:
  sage: QF = DiagonalQuadraticForm(ZZ, [1, 1])

  sage: QS = QuadraticSpace(QQ, QF)
  sage: QS.base_field()
  Rational Field

  sage: QS = QuadraticSpace(Qp(17), QF)
  sage: QS.base_field()
  17-adic Field with capped relative precision 20

  sage: QS = QuadraticSpace(RR, QF)
  sage: QS.base_field()
  Real Field with 53 bits of precision

  sage: QS = QuadraticSpace(CC, QF)
  sage: QS.base_field()
  Complex Field with 53 bits of precision

  sage: QS = QuadraticSpace(FiniteField(11), QF)
  sage: QS.base_field()
  Finite Field of size 11

"""
        return self.__quadratic_form.base_ring()
```

```
    def defining_quadratic_form(self):
        """
    Returns the defining quadratic form (i.e. the quadratic form
    of the quadratic space in the standard basis) for the
    quadratic space, coerced to the base_field of the quadratic space.

    TO DO: Change the name to 'quadratic_form()', since eventually
    this will not be canonical when we are working with non-free
    lattices!

    INPUT:
        None

    OUTPUT:
        a quadratic form

    EXAMPLES:
        sage: QF = QuadraticForm(ZZ, 3, range(6))

        sage: QS = QuadraticSpace(QQ, QF)
        sage: QS.defining_quadratic_form()
        Quadratic form in 3 variables over Rational Field with coefficients:
        [ 0 1 2 ]
        [ * 3 4 ]
        [ * * 5 ]

        sage: QS = QuadraticSpace(CC, QF)
        sage: QS.defining_quadratic_form()
        Quadratic form in 3 variables over Complex Field with 53 bits of precision with coefficients:
        [ 0 1.00000000000000 2.00000000000000 ]
        [ * 3.00000000000000 4.00000000000000 ]
        [ * * 5.00000000000000 ]

        sage: QS = QuadraticSpace(FiniteField(5), QF)
        sage: QS.defining_quadratic_form()
        Quadratic form in 3 variables over Finite Field of size 5 with coefficients:
        [ 0 1 2 ]
        [ * 3 4 ]
        [ * * 0 ]

    """
        return deepcopy(self.__quadratic_form)


    def dim(self):
        """
    Returns the dimension (as a vectorspace over the base field)
    of the quadratic space.

    INPUT:
        None

    OUTPUT:
        an integer >= 0

    EXAMPLES:
        sage: QF = QuadraticForm(ZZ, 3, range(6))
        sage: QS = QuadraticSpace(QQ, QF)
        sage: QS.dim()
        3

    """
        return self.__quadratic_form.dim()



    def det_Gram(self):
```



```
        """
    Returns the determinant of the Gram matrix of the defining
    quadratic form for the quadratic space (in the standard basis).

    INPUT:
        None

    OUTPUT:
        an element of the base field

    EXAMPLES:
        sage: QF = QuadraticForm(ZZ, 3, range(6))
        sage: QS = QuadraticSpace(QQ, QF)
        sage: QS.det_Gram()
        -9/4

    """
        return self.__quadratic_form.Gram_det()


    def det_Hessian(self):
        """
    Returns the determinant of the Hessian matrix of the defining
    quadratic form for the quadratic space (in the standard basis).

    INPUT:
        None

    OUTPUT:
        an element of the base field

    EXAMPLES:
        sage: QF = QuadraticForm(ZZ, 3, range(6))
        sage: QS = QuadraticSpace(QQ, QF)
        sage: QS.det_Hessian()
        -18

    """
        return self.__quadratic_form.det()      ## TO DO: This should be changed
to Hessian_det() in QF, and not only be the default.



    def determinant(self):
        """
    Returns the squareclass of the Gram determinant of the given
    quadratic form.  This determinant is the product of the
    diagonal entries when the form is diagonal (which it can
    always be arranged to be).

    TO DO:  CHANGE THIS TO 'det_squareclass()', SO THE USER DOESN'T EXPECT A NUMBER!

    INPUT:
        None

    OUTPUT:
        an element of the base field

    EXAMPLES:
        sage: QF = DiagonalQuadraticForm(ZZ, [1, 1, 1])
        sage: QS = QuadraticSpace(QQ, QF)
        sage: QS.determinant()
        The squareclass represented by 1 over Rational Field

        sage: QS.det_Hessian()
        8

        sage: QS.det_Gram()
        1
```

```
    """
        return SquareClass(self.base_field(), self.det_Gram())


    def determinant_signed(self):
        """
    Returns the squareclass of the signed Gram determinant of the
    given quadratic form.  This determinant is the product of the
    diagonal entries when the form is diagonal (which it can
    always be arranged to be), multiplied by (−1)^(n*(n−1)/2)
    where n is the dimension of the quadratic space.

    TO DO: Change the name to 'det_signed_Gram()' to be consistent
    with the other method notation!

    REFERENCE:
        This is defined in Lam's book section II.2 on p30.

    INPUT:
        None

    OUTPUT:
        an element of the base field

    EXAMPLES:
        sage: QF = DiagonalQuadraticForm(ZZ, [1, 1, 1])
        sage: QS = QuadraticSpace(QQ, QF)
        sage: QS.determinant_signed()
        −1

    """
        n = self.dim()
        return self.det_Gram() * (-1)**(n*(n-1)/2)



    def diagonal_squareclass_list(self):
        """
    Returns a list of the self.dim() squareclasses defined by some
    (usually not unique) diagonalization of the quadratic form on
    this quadratic space.

    INPUT:
        None

    OUTPUT:
        a list of squareclasses defined over the base field

    EXAMPLES:
        sage: QF = DiagonalQuadraticForm(QQ, [1, 2, 1/18])
        sage: QS = QuadraticSpace(QQ, QF)
        sage: QS.diagonal_squareclass_list()
        [The squareclass represented by 1 over Rational Field, The squareclass represented by 2 over Rational Field, Th
e squareclass represented by 1/18 over Rational Field]

    """
        return deepcopy(self.__diagonal_squareclass_list)



    def gram_matrix(self):
        """
    Returns the gram matrix (which defines all Gram inner
    products) for this quadratic space.

    """
        return self.__quadratic_form.matrix() * ZZ(1)/ZZ(2)
```



```
    def hasse_invariant(self, QQ_place=None):
        """
    Returns the Hasse invariant of the quadratic form, which is
    defined as the product of the Hilbert symbols (a_i, a_j) where
    i < j.  This is defined for any field of characteristic not 2.

    If the quadratic space is defined over QQ, then we must pass in
    QQ_place as either a prime p > 0 or Infinity.  In this case we
    will return the local Hasse invariant over the associated
    local field.

    INPUT:
        QQ_place −− a prime number or Infinity
            (an argument required if the base field is QQ).

    OUTPUT:
        an integer >= 0

    EXAMPLES:
        sage: QS = QuadraticSpace(Qp(5), DiagonalQuadraticForm(QQ, [1, −1, 2, 3]))
        sage: QS.hasse_invariant()
        1

        sage: QS = QuadraticSpace(RR, DiagonalQuadraticForm(QQ, [1, −1, 1, −1]))
        sage: QS.hasse_invariant()
        −1

        sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1, 1, 1, 1, −1, −1]))
        sage: QS.hasse_invariant(Infinity)
        −1
        sage: QS.hasse_invariant(2)
        −1
        sage: QS.hasse_invariant(3)
        1
        sage: QS.hasse_invariant(5)
        1

    """
        hasse_temp = 1
        n = self.dim()
        DSL = self.__diagonal_squareclass_list

        for j in range(n-1):
            for k in range(j+1, n):
                hasse_temp = hasse_temp * DSL[j].hilbert_symbol(DSL[k], QQ_place
)

        return hasse_temp



    def hessian_matrix(self):
        """
    Returns the Hessian matrix (which defines all Hessian inner
    products) for this quadratic space.

    """
        return self.__quadratic_form.matrix()



    def scale_by_factor(self, c):
        """
    Gives the current quadratic space scaled by the constant factor c.

    Note: Think about coercing c into a global element first.
```

```
    Also check the creation conventions to make sure we're
    compatibile with them.

    INPUT:
      c -- a number coercible to the base field

    OUTPUT:
      a quadratic space

    EXAMPLES:
      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1, -2, 3]))  ## Scaling a 3-dim'l form
      sage: QS3 = QS.scale_by_factor(3)
      sage: QS.det_Hessian() * 3**QS.dim() == QS3.det_Hessian()
      True

      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, []))  ## Try scaling a 0-dim'l form
      sage: QS3 = QS.scale_by_factor(3)
      sage: QS.det_Hessian() == QS3.det_Hessian()
      True

    """
        ## Validate the constant c -- TO DO!
        try:
            c1 = self.base_field()(c)
        except:
            raise RuntimeError, "The scaling factor " + str(c) + " is not coercible to the base fie
ld " + str(self.base_field()) + "."

        ## Return the scaled quadratic space
        return QuadraticSpace(self.__quadratic_form.scale_by_factor(c))




    def hasse_primes_of_QQ(self):
        """
    Give a list of the finitely many primes p where the quadratic space
    has Hasse invariant c_p = -1.

    NOTE: HERE THE QUADRATIC FORM MUST BE DEFINED OVER QQ!

    INPUT:
      None

    OUTPUT:
      a list of prime numbers

    EXAMPLES:
      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1, 3, 3]))
      sage: QS.hasse_primes_of_QQ()
      [2, 3]

    """
        ## Check that we're over QQ
        if self.base_field() != QQ:
            raise TypeError, "This method only applies to quadratic forms over global fields, and only Q
Q for now."

        ## List all primes dividing the discriminant (and also add p = 2)
        possible_prime_list = [p  for p in prime_divisors(2 * self.normalized_sp
ace().det_Gram())]

        ## Check which primes have some exceptional behavior
        prime_list = []
        d = self.determinant()
```



```
        for p in possible_prime_list:
            if (self.hasse_invariant(p) == -1):
                prime_list.append(p)

        ## Return the list of primes
        return prime_list




    def local_characteristic_primes_of_QQ(self):
        """
    Give a list of the finitely many characteristic (bad) primes
    of a quadratic space defined over QQ where either the
    discriminant squareclass is not unit, or the Hasse invariant
    is -1.

    NOTE: HERE THE QUADRATIC FORM MUST BE DEFINED OVER QQ!

    INPUT:
      None

    OUTPUT:
      a list of prime numbers

    EXAMPLES:
      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1, -2, 3]))
      sage: QS.local_characteristic_primes_of_QQ()
      [2, 3]

    """
        ## Check that we're over QQ
        if self.base_field() != QQ:
            raise TypeError, "This method only applies to quadratic forms over global fields, and only Q
Q for now."

        ## List all primes dividing the discriminant (and also add p = 2)
        possible_prime_list = [p  for p in prime_divisors(2 * self.normalized_sp
ace().det_Gram())]

        ## Check which primes have some exceptional behavior
        prime_list = []
        d = self.determinant()
        for p in possible_prime_list:
            if (self.hasse_invariant(p) == -1) or (d.valuation(p) == 1):
                prime_list.append(p)

        ## Return the list of primes
        return prime_list




    def local_characteristic_places_of_QQ(self):
        """
    Give a list of the finitely many characteristic (bad) places
    (including Infinity first) of a quadratic space defined over
    QQ where either the discriminant squareclass is not unit, or
    the Hasse invariant is -1, or the place is archimedean.

    NOTE: HERE THE QUADRATIC FORM MUST BE DEFINED OVER QQ!

    INPUT:
      None

    OUTPUT:
```

```
        a list of places (i.e. Infinity and prime numbers) starting with Infinity

    EXAMPLES:
        sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1, -2, 3]))
        sage: QS.local_characteristic_places_of_QQ()
        [+Infinity, 2, 3]

    """
        ## Add Infinity and return the list of places
        return [Infinity] + self.local_characteristic_primes_of_QQ()



    def local_characteristic_space_list(self):
        """
    Give a list of local quadratic spaces at the characteristic
    (i.e. bad) local places of the given quadratic space over QQ.
    These local spaces will determine the global (rational) space
    up to isomorphism by the Hasse-Minkowski Theorem.

    NOTE: HERE THE QUADRATIC FORM MUST BE DEFINED OVER QQ!

    (TO DO: PERHAPS CHANGE THE NAME TO BE CONSISTENT WITH
    'local_characteristic_places_of_QQ()'?)

    INPUT:
        None

    OUTPUT:
        a list of places (i.e. Infinity and prime numbers) starting with Infinity

    EXAMPLES:
        sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1, -2, 3]))
        sage: QS.local_characteristic_space_list()
        [Quadratic space defined by the Quadratic form in 3 variables over Real Field with 53 bits of precision with coe
fficients:
        [ 1.00000000000000 0.000000000000000 0.000000000000000 ]
        [ * -2.00000000000000 0.000000000000000 ]
        [ * * 3.00000000000000 ]
        , Quadratic space defined by the Quadratic form in 3 variables over 2-adic Field with capped relative precision
20 with coefficients:
        [ 1 + O(2^20) 0 0 ]
        [ * 2 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7 + 2^8 + 2^9 + 2^10 + 2^11 + 2^12 + 2^13 + 2^14 + 2^15 + 2^16 + 2
^17 + 2^18 + 2^19 + 2^20 + O(2^21) 0 ]
        [ * * 1 + 2 + O(2^20) ]
        , Quadratic space defined by the Quadratic form in 3 variables over 3-adic Field with capped relative precision
20 with coefficients:
        [ 1 + O(3^20) 0 0 ]
        [ * 1 + 2*3 + 2*3^2 + 2*3^3 + 2*3^4 + 2*3^5 + 2*3^6 + 2*3^7 + 2*3^8 + 2*3^9 + 2*3^10 + 2*3^11 + 2*3^12
 + 2*3^13 + 2*3^14 + 2*3^15 + 2*3^16 + 2*3^17 + 2*3^18 + 2*3^19 + O(3^20) 0 ]
        [ * * 3 + O(3^21) ]
        ]

    """
        ## Check that we're over QQ
        if self.base_field() != QQ:
            raise TypeError, "This method only applies to quadratic forms over global fields, and only Q
Q for now."

        ## Make the list by localizing at the characteristic places
        local_space_list = []
        for v in self.local_characteristic_places_of_QQ():
            local_space_list.append(self.localize_at_place(v))

        return local_space_list



    def localize_at_place(self, v):
```



```
        """
    Return the localization of the current quadratic space at the
    place v.

    INPUT:
        v -- a place of the basefield (currently only the base
        field QQ is supported, so in this case v is either a prime
        number or Infinity).

    OUTPUT:
        a quadratic space over the localization of the base field at v.

    EXAMPLES:
        sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1, -2, 3]))
        sage: QS.localize_at_place(Infinity)
        Quadratic space defined by the Quadratic form in 3 variables over Real Field with 53 bits of precision with coef
ficients:
        [ 1.00000000000000 0.000000000000000 0.000000000000000 ]
        [ * -2.00000000000000 0.000000000000000 ]
        [ * * 3.00000000000000 ]

        sage: QS.localize_at_place(7)
        Quadratic space defined by the Quadratic form in 3 variables over 7-adic Field with capped relative precision 2
0 with coefficients:
        [ 1 + O(7^20) 0 0 ]
        [ * 5 + 6*7 + 6*7^2 + 6*7^3 + 6*7^4 + 6*7^5 + 6*7^6 + 6*7^7 + 6*7^8 + 6*7^9 + 6*7^10 + 6*7^11 + 6*7^12
 + 6*7^13 + 6*7^14 + 6*7^15 + 6*7^16 + 6*7^17 + 6*7^18 + 6*7^19 + O(7^20) 0 ]
        [ * * 3 + O(7^20) ]

    """
        ## Check that we're over QQ
        if self.base_field() != QQ:
            raise TypeError, "This method only applies to quadratic forms over global fields, and only Q
Q for now."

        ## Construct the local field from the place
        if v == Infinity:
            F = RealField()
        elif is_prime(v):
            F = Qp(v)
        else:
            raise RuntimeError, "The place " + str(v) + " you passed is not recognized."

        ## Return the localized quadratic space
        return QuadraticSpace(F, self)



    def inner_product__gram(self, x, y):
        """
    Compute the Gram inner product of two vectors, which is the
    inner product <.,.> satisfying Q(x) = <x,x> for all vectors x.

    Note:  This loop could be sped up by more careful coding.

    """
        G = self.gram_matrix()

        ## Check that x and y are vectors of the appropriate length, and defined
 over the base_field.


        ## Compute the (Gram) inner product
        n = self.dim()
        tmp_sum = self.__quadratic_form.base_ring()(0)
        for i in range(n):
            for j in range(n):
                tmp_sum += x[i] * G[i, j] * y[j]
```

```
        ## Return the inner product
        return tmp_sum



    def inner_product__hessian(self, x, y):
        """
    Compute the Hessian inner product of two vectors, which is the
    inner product <.,.> satisfying 2*Q(x) = <x,x> for all vectors x.

    Note:  This loop could be sped up by more careful coding.

    """
        H = self.hessian_matrix()

        ## Check that x and y are vectors of the appropriate length, and defined
over the base_field.


        ## Compute the (Hessian) inner product
        n = self.dim()
        tmp_sum = self.__quadratic_form.base_ring()(0)
        for i in range(n):
            for j in range(n):
                tmp_sum += x[i] * (H[i, j]) * y[j]

        ## Return the inner product
        return tmp_sum




    def __call__(self, x):
        """
    Evaluate the underlying quadratic form on the given vector x.
    See also QuadraticForm.__call__() for more details.

    INPUT:
        x -- a vector, list, ot tuple of numbers coercible to the base field

    OUTPUT:
        a number in the base field

    EXAMPLES:
        sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1, -2, 3]))
        sage: QS.__call__([1,0,0])
        1
        sage: QS.__call__([0,1,0])
        -2
        sage: QS.__call__([1,1,1])
        2

        sage: QS.__call__([1,1,1]) == QS([1,1,1])
        True

    """
        return self.__quadratic_form(x)



    def __eq__(self, other):
        """
    Perform equality testing, which means that the base_field,
    coefficient_field, dimension, and defining coefficients are
    all equal.  (Note: This is much stronger than being rationally
    equivalent!)
```



```
    INPUT:
        other -- a quadratic space

    OUTPUT:
        boolean

    EXAMPLES:
        sage: Q1 = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
        sage: Q2 = Q1(Matrix(ZZ, 3, 3, [1,2,3,0,2,4,0,0,3]))
        sage: QS1 = QuadraticSpace(QQ, Q1)
        sage: QS2 = QuadraticSpace(QQ, Q2)
        sage: QS1 == QS1
        True
        sage: QS1 == QS2
        False

    """
        ## Check that it's another quadratic space
        if not isinstance(other, QuadraticSpace):
            return False

        ## Check that the two defining quadratic forms are equal (which includes
their base fields being equal!)
        if (self.__quadratic_form != other.__quadratic_form):
            return False

        ## All Tests Passed -- they're equal!
        return True



    def __ne__(self, other):
        """
    Checks if the two quadratic spaces are not equal (see self.__eq__ for more details).

    INPUT:
        other -- a quadratic space

    OUTPUT:
        boolean

    EXAMPLES:
        sage: Q1 = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
        sage: Q2 = Q1(Matrix(ZZ, 3, 3, [1,2,3,0,2,4,0,0,3]))
        sage: QS1 = QuadraticSpace(QQ, Q1)
        sage: QS2 = QuadraticSpace(QQ, Q2)
        sage: QS1 != QS1
        False
        sage: QS1 != QS2
        True
    """
        return not self.__eq__(other)



    def __cmp__(self, other):
        """
    This is the default comparison routine for <, <=, ==, >, >= if
    no special comparison operator is defined.  These operations
    are not defined at present, and it is not clear what anything
    but equality would mean in this context, so we raise a
    NotImplementedError.

    INPUT:
        other -- a quadratic space

    OUTPUT:
        boolean
```

```
    EXAMPLES:
        sage: Q1 = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
        sage: Q2 = Q1(Matrix(ZZ, 3, 3, [1,2,3,0,2,4,0,0,3]))
        sage: QS1 = QuadraticSpace(QQ, Q1)
        sage: QS2 = QuadraticSpace(QQ, Q2)
        sage: QS1.__cmp__(QS2)
        Traceback (most recent call last):
        ...
        NotImplementedError: Warning: The comparison operation just used is not presently defined.
        sage: QS1 > QS2
        Traceback (most recent call last):
        ...
        NotImplementedError: Warning: The comparison operation just used is not presently defined.


    """
        raise NotImplementedError, "Warning: The comparison operation just used is not presently def
ined."




    def hessian_bilinear_space(self):
        """
    Return the underlying Hessian bilinear space for this quadratic latice.
    """
        ## We should be able to use this -- but we can't due a characteristic 2
vector space bug!  (TRAC #)
        #return deepcopy(self.__hessian_bilinear_space)

        ## Here is our workaround!
        return SymmetricBilinearSpace(self.__hessian_bilinear_space.base_field()
, self.__hessian_bilinear_space.gram_matrix())



## ------------------------------------  Field-specific Routines  --------------
------------------------------



    def orthogonal_basis(self):
        """
    Return an orthogonal basis for the quadratic space.

    TO DO:  MAKE THIS FIELD-INDEPENDENT!!!

    INPUT:
        None

    OUTPUT:
        A list of vectors

    EXAMPLES:
        sage: QS = QuadraticSpace(QQ, QuadraticForm(QQ, 3, [2, 11/3, 5,  45/11, 1,  1/12]))
        sage: B = QS.orthogonal_basis()
        sage: M =Matrix(QQ, 3, 3, [QS.inner_product__hessian(B[i], B[j])  for i in range(3)  for j in range(3)])
        sage: M.is_diagonal()
        True
    """
        return self.__hessian_bilinear_space.orthogonal_basis()



    def integral_lattice(self):
        """
    Return a Z-valued quadratic lattice on this quadratic space (by scaling all of
```



```
    the lattice generators to be Z-valued).

    TO DO: Modify this to allow it to return I-valued forms for any ideal I over a number field.

    INPUT:
        None

    OUTPUT:
        a quadratic lattice over ZZ

    EXAMPLES:
        sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [2,45/11,1/12]))
        sage: QS.integral_lattice()
        Quadratic Lattice in Quadratic space defined by the Quadratic form in 3 variables over Rational Field with coeff
icients:
        [ 2 0 0 ]
        [ * 45/11 0 ]
        [ * * 1/12 ]
        spanned by ((1, 0, 0), (0, 11, 0), (0, 0, 6)).

        sage: QS = QuadraticSpace(QQ, QuadraticForm(QQ, 3, [2, 11/3, 5,  45/11, 1,  1/12]))
        sage: L = QS.integral_lattice(); L
        Quadratic Lattice in Quadratic space defined by the Quadratic form in 3 variables over Rational Field with coeff
icients:
        [ 2 11/3 5 ]
        [ * 45/11 1 ]
        [ * * 1/12 ]
        spanned by ((1, 0, 0), (0, 66, 11454), (0, 0, 22908)).
        sage: Matrix(QQ, 3, 3, [L.inner_product__hessian(L.basis()[i], L.basis()[j])  for i in range(3)  for j in range(3)])
in MatrixSpace(ZZ, 3, 3)
        True
    """
        new_gen_list = []

        ## Scale all vectors in an orthogonal basis to make them integer-valued
        for v in self.orthogonal_basis():
            d = self(v).denominator()
            scale_factor = sqrt(d * squarefree_part(d))
            new_gen_list.append(v * scale_factor)

        return QuadraticLattice(self, new_gen_list)    ## Warning: This operatio
n does *not* preserve the given basis!



    def find_isotropic_vector(self):
        """
    Returns a non-zero vector v in the quadratic space with Q(v) = 0,
    and returns False if there is no such vector.

    INPUT:
        None

    OUTPUT:
        a vector over self.base_field()

    EXAMPLES:
        sage: A = matrix(ZZ, 2, 2, [3,1,1,3])
        sage: QS= QuadraticSpace(GF(3), A)
        sage: v = QS.find_isotropic_vector()
        sage: QS(v) == 0
        True
        sage: v.parent()
        Vector space of dimension 2 over Finite Field of size 3

    """
        ## Use the symmetric bilinear space when the characteristic is not 2
        if self.base_field().characteristic() != 2:
            return self.__hessian_bilinear_space.find_isotropic_vector()
```

```
        ## When the characteristic is 2, do this.....  FIX THIS!!!
        ## -------------------------------------------------------
        d = self.dim()
        F = self.base_field()
        char_p = F.characteristic()


        ## Deal with dimension <=1 forms -- (very easy)
        if (d == 0) or (d == 1):
            return False

        ## Deal with anistropic forms -- (uses invariants, if they exist!)
        if self.is_anisotropic():
            return False


        ## NOT IMPLEMENTED: Check that the space is non-degenerate
        if self.is_degenerate():
            raise NotImplementedError, "For now we need a non-degenerate quadratic space."



        ## Deal with isotropic forms (so we have isotropic vectors)!
        PR = PolynomialRing(F, 'y')
        y = PR.gen()

        while True:                                                    ## This mus
t terminate since n >= 3
            ## Choose a random (non-degenerate) linear polynomial vector, giving
 a general line in our space
            v1 = vector([PR(F.random_element())  for i in range(d)])
            while v1 == v1.parent().zero_vector():
                v1 = vector([PR(F.random_element())  for i in range(d)])
            v2 = vector([PR(F.random_element())  for i in range(d)])
            while v2 == v1.parent().zero_vector():
                v2 = vector([PR(F.random_element())  for i in range(d)])
            v = v1 + y * v2
            #print "v = ", v

            ## Find its value (as a quadratic polynomial in y)  --  this could b
e sped up by not re-copying the matrix G every time, and evaluatng instead!
            G = self.hessian_matrix()
            G1 = matrix(PR, G)
            m1 = (v * G1 * v.transpose())[0]
            #print "G = ", G
            #print "G1 = ", G1
            #print "m1 = ", m1

            ## Deal with every vector being isotropic
            if F == 0:
                return vector(F, v1)

            ## Otherwise find roots, and return an isotropic vector
            m1_roots = m1.roots()
            if len(m1_roots) != 0:
                a = m1_roots[0][0]  ## Take the first root over F_p
                new_v = v1 + a*v2
                #print "m1_roots = ", m1_roots
                #print "new_v = ", new_v
                if new_v != new_v.parent().zero_vector():
                    return vector(F, new_v)



    def orthogonal_subspace_to_vector(self, subspace):
        """
```



```
        Find the subspace of the quadratic space orthogonal to the given vector.
        """
            ## Check if we have a vector
            if is_Vector(v):
                return self.__hessian_bilinear_space.orthogonal_subspace_to_vector(v
)




    def find_basis_of_maximal_isotropic_subspace(self):
        """
        Find a basis of a maximal isotropic subspace of the quadratic space, as a matrix of row vectors.

        INPUT:
          None

        OUTPUT:
          a matrix of row vectors

        EXAMPLES:
          sage: from sage.quadratic_forms.maximal_extras import find_basis_of_maximal_isotropic_subspace

          sage: MM = matrix(ZZ, 6, 6, [0, 0, 1, 2, 2, 2, 0, 0, 0, 1, 0, 1, 1, 0, 2, 2, 3, 0, 2, 1, 2, 3, 1, 2, 2, 0, 3, 1, 1, 0, 2, 1, 0,
2, 0, 1])
          sage: QS = QuadraticSpace(GF(5), MM)
          sage: QS.find_basis_of_maximal_isotropic_subspace()  ## random
          [2 0 0 3 0 2]
          [0 3 0 1 3 1]
          [0 0 2 2 2 3]

        """
            ## Use the symmetric bilinear space when the characteristic is not 2
            if self.base_field().characteristic() != 2:
                return self.__hessian_bilinear_space.find_basis_of_maximal_isotropic
_subspace()



            ## When the characteristic is 2, do this.....  FIX THIS!!!
            ## -------------------------------------------------------
            G = self.hessian_matrix()
            F = self.base_field()
            n = G.nrows()
            p = G.parent().base_ring().characteristic()

            ## Make the transformation matrix (of rows!!!)
            T = matrix(F, 0, n, [])

            ## Find one isotropic vector
            v = self.find_isotropic_vector_at_prime()

            ## Check if we're done.
            if v == False:
                return T

            ## Find a basis for v^\perp
            K = self.orthogonal_subspace_to_vector(v).basis_matrix()    ## Note: Row
 vectors here, in reduced row echelon form!

            ## DIAGNOSTIC
            verbose("v = " + str(v))
            verbose("K = " + str(K))

            ## Find the first non-zero entry of v, to use to decide which kernel bas
is vector to replace with v.
            for i in range(n):
                if v[i] != 0:
```

```
                v_nz_index = i
                break
        ## Find the associated basis vector (using heavily the row echelon form
of the output)
        for i in range(K.nrows()):
            if K[i, v_nz_index] != 0:
                K_nz_index = i
                break

        ## DIAGNOSIC
        verbose("v_nz_index = " + str(v_nz_index))
        verbose("K_nz_index = " + str(K_nz_index))

        ## Extract the kernel basis excluding v
        K1 = K.matrix_from_rows([j for j in range(K.nrows())  if j != K_nz_index
])
        G1 = K1 * G * K1.transpose()

        ## Perform the recursion
        Q1 = QuadraticSpace(self.base_field(), G1)
        T1 = Q1.find_basis_of_maximal_isotropic_subspace
        T_last = T1 * K1
        T_new = (T_last.transpose().augment(v.transpose())).transpose()   ## Aug
ment T_last by adding the row v

        ## DIAGNOSTIC
        verbose("Found T_new of dimension " + str(T_new.nrows()))

        return T_new




    def maximal_quadratic_lattice(self):
        """
    Find a quadratic lattice equivalent to a maximal lattice in the given quadratic space.

    TO DO: Add support for (a)–maximal lattices too.


    INPUT:
      None

    OUTPUT:
      a quadratic lattice over ZZ

    EXAMPLES:
      sage: QS = QuadraticSpace(QQ, QuadraticForm(QQ, 3, [2, 11/3, 5,  45/11, 1,  1/12]))


    """
        F = self.base_field()
        n = self.dim()

        ## Check that we're working over QQ
        if self.base_field() != QQ:
            raise NotImplementedError, "Presently only the base field of QQ is supported."


        #print " self.__hessian_bilinear_space() = " + str( self.__hessian_bilin
ear_space)


        ## Find a maximal lattice for the associated Hessian bilinear space
        Hessian_maximal_lattice = self.__hessian_bilinear_space.maximal_bilinear
_lattice()

        #print "Hessian_maximal_lattice = " + str(Hessian_maximal_lattice)
```



```
        ## Find the even sublattice, and look for a maximal even superlattice of
 it
        Even_sublattice = Hessian_maximal_lattice.even_sublattice()

        #print "Even_sublattice = " + str(Even_sublattice)

        Pn_Z2 = mrange(n*[2])[1:]   ## This runs over all non-zero vectors of P^
n(GF(2))
        ## Loop through all index 2 superlattices, looking for an even one
        for v in Pn_Z2:
            #print "v = " + str(v)
            #print "Even_sublattice.basis_matrix_of_rows() = " + str(Even_sublat
tice.basis_matrix_of_rows())
            #print "Matrix(F, vector(F, v) * (ZZ(1)/ZZ(2))) = " + str(Matrix(F,
vector(F, v) * (ZZ(1)/ZZ(2))))
            v1_list = (Matrix(F, vector(F, v) * (ZZ(1)/ZZ(2))) * Even_sublattice
.basis_matrix_of_rows()).rows()
            L = Even_sublattice.sum_with(v1_list)
            if L.is_even():
                return QuadraticLattice(self, L.basis())

        ## If there are no even maximal superlattices, then return the even latt
ice!
        return QuadraticLattice(self, Even_sublattice.basis())




        ## ====================================================================
========
        ## ====================================================================
========
        ## ====================================================================
========
        ##
        ## Nothing to see here but old code... to be deleted!
        ##
        ## ====================================================================
========
        ## ====================================================================
========
        ## ====================================================================
========



        ## Find a maximal Z-valued quadratic lattice:
        ## ----------------------------------------
        Gram_new, T_new = even_neighbor_of_bilinear_gram_matrix(Gram_of_max_lat)

        ## Return a maximal form
        #return L

        return False

        ## ====================================================================
========
        ## ====================================================================
========
        ## ====================================================================
========


        ## Find an integral lattice in our quadratic space
        L = self.integral_lattice()

        ## DIAGNOTSIC
        verbose("\n L = " + str(L))
```

```
        verbose("\n ====================== ")

        ## Enlarge the integral quadratic lattice so its discriminant module L^#
/L is a product of quadratic spaces
        H = L.Hessian_matrix()
        max_ed = H.elementary_divisors()[-1]
        while not is_squarefree(max_ed):    ## Check if the largest elementary d
ivisor is not squarefree

            ## Enlarge the lattice with the scaled dual lattice
            big_sq_factor = sqrt(max_ed * squarefree_part(max_ed))  ## This is t
he amount we scale the dual lattice by, doing all primes at once!
            L = L.sum_with(L.dual_lattice().apply_linear_transformation_on_right
(big_sq_factor))

            ## DIAGNOSTIC
            verbose("\n big_sq_factor = " + str(big_sq_factor))
            verbose("\n L = " + str(L))
            verbose("\n ====================== ")

            ## Prepare to check if we're done
            H = L.Hessian_matrix()
            max_ed = H.elementary_divisors()[-1]



        ## Find a maximal Z-valued Hessian bilinear lattice:
        ## ------------------------------------------------
        L_dual = L.dual_lattice()
        A = Matrix(ZZ, L_dual.Hessian_matrix(rational_matrix=True).inverse())
  ## This matrix describes L in the given basis of L^#
        D, U, V = A.smith_form()
        iso_eligible_primes = prime_divisors(D[-2,-2])        ## The list of pri
mes which have at most a 2-dim'l subspace in L^#/L
        L1_dual = L_dual.apply_linear_transformation_on_right(U)        ## This
is the

        ## DIAGNOSTIC
        verbose("\n \n\nStart looking for a maximal Z-valued Hessian bilinear lattice.")
        verbose("\n A = " + str(A))
        verbose("\n D = " + str(D))
        verbose("\n U = " + str(U))
        verbose("\n V = " + str(V))
        verbose("\n iso_eligible_primes = " + str(iso_eligible_primes))
        verbose("\n L_dual = " + str(L_dual))
        verbose("\n L1_dual = " + str(L1_dual))
        verbose("\n ==================================== ")

        ## Loop through all primes to find a maximal isotropic space for each.
        T_huge = matrix(ZZ,n,n,1)    ## This will hold the final list of generat
ors for the isotropic submodule (of L^#)
        for p in iso_eligible_primes:

            ## Compute the dimension d_p of L^#/L at p
            d_p = n
            for i in range(n):
                if D[i,i] % p == 0:
                    d_p = i         ## Dim of L#/L
                    break

            ## Create the Gram matrix for the smaller subspace
            small_Hessian_gram = L1_dual.Hessian_matrix(rational_matrix=True)[:d
_p, :d_p]

            ## Create the quadratic lattice in the basis
            small_QS = QuadraticSpace(GF(p), p * small_Hessian_gram)  ## This mu
lt-by-p makes the gram matrix p-integral
            Tp = small_QS.find_basis_of_maximal_isotropic_subspace()
```



```
            ## DIAGNOSTIC
            verbose("\n p = " + str(p))
            verbose("\n d_p = " + str(d_p))
            verbose("\n small_Hessian_gram = " + str(small_Hessian_gram))
            verbose("\n small_QS = " + str(small_QS))
            verbose("\n T_p = " + str(Tp))
            verbose("\n Finished finding maximal isotropic subspace at prime " + str(p))

            ## Add lifts of this subspace to our matrix of generators
            dp_cols_small = U.matrix_from_columns(d_p)    ## These columns are t
he basis of L^# we used to find the maximal iso subspace.
            TZ = dp_cols_small * matrix(ZZ,Tp).transpose()  ## Add (a lift to ZZ
 of) these vectors to a (column) matrix of generators.
            T_huge = T_huge.augment(TZ)


        ## Return a basis for the maximal form.
        verbose("\n Status -- Pre-LLL")
        verbose("\n T_huge has " + str(T_huge.nrows()) + " rows and " + str(T_huge.ncol
s()) + " columns.")
        verbose("\n " + str(type(T_huge)) + " " + str(T_huge.parent()))
        verbose("\n " + str(T_huge.rows()))

        nr = T_huge.ncols()  ## after LLL the last rows form a basis, the first
ones are 0
        T_lll = T_huge.transpose().LLL().matrix_from_rows(range(nr-n,nr)).transp
ose()

        verbose("\n Status -- Post-LLL")
        #Gram_of_max_lat = matrix(ZZ, T_lll.transpose() * B * T_lll / (max_ed* m
ax_ed))




        ## Find a maximal Z-valued quadratic lattice:
        ## ------------------------------------------
        #Gram_new, T_new = even_neighbor_of_bilinear_gram_matrix(Gram_of_max_lat
)

        ## Return a maximal form
        #return L
```

```
    def normalized_space(self):
        """
    Returns a normalized (diagonal) version of this quadratic space using
    normalized representatives for each squareclass (assuming that the base
    field has squareclass normalization support).

    INPUT:
      None

    OUTPUT:
      a (diagonal) quadratic space over the same basefield, equivalent to self.

    EXAMPLES:
      sage: QF = DiagonalQuadraticForm(QQ, [1, 3, 9, 1/27])
      sage: QS = QuadraticSpace(QQ, QF)
      sage: QS.normalized_space()
      Quadratic space defined by the Quadratic form in 4 variables over Rational Field with coefficients:
      [ 1 0 0 0 ]
      [ * 3 0 0 ]
      [ * * 1 0 ]
      [ * * * 3 ]
```

```
    """
        normalized_diag_list = [s.normalized_representative()  for s in self.dia
gonal_squareclass_list()]
        return QuadraticSpace(self.base_field(), DiagonalQuadraticForm(self.base
_field(), normalized_diag_list))



    def is_isotropic(self):
        """
    Determines if the quadratic space is isotropic over its base field.

    INPUT:
      None

    OUTPUT:
      boolean

    EXAMPLES:
      sage: QF = DiagonalQuadraticForm(ZZ, [1,1,1,1])

      sage: QS = QuadraticSpace(QQ, QF)
      sage: QS.is_isotropic()
      False

      sage: QS = QuadraticSpace(RR, QF)
      sage: QS.is_isotropic()
      False

      sage: QS = QuadraticSpace(Qp(2), QF)
      sage: QS.is_isotropic()
      False

      sage: QS = QuadraticSpace(Qp(3), QF)
      sage: QS.is_isotropic()
      True

      sage: QS = QuadraticSpace(FiniteField(11), QF)
      sage: QS.is_isotropic()
      True

    """
        return self.anisotropic_dim() != self.dim()



    def is_anisotropic(self):
        """
    Determines if the quadratic space is anisotropic over its base field.

    INPUT:
      None

    OUTPUT:
      boolean

    EXAMPLES:
      sage: QF = DiagonalQuadraticForm(ZZ, [1,1,1,1])

      sage: QS = QuadraticSpace(QQ, QF)
      sage: QS.is_anisotropic()
      True

      sage: QS = QuadraticSpace(RR, QF)
      sage: QS.is_anisotropic()
      True
```



```
      sage: QS = QuadraticSpace(Qp(2), QF)
      sage: QS.is_anisotropic()
      True

      sage: QS = QuadraticSpace(Qp(3), QF)
      sage: QS.is_anisotropic()
      False

      sage: QS = QuadraticSpace(FiniteField(11), QF)
      sage: QS.is_anisotropic()
      False

    """
        return not self.is_isotropic()


    def is_hyperbolic_space(self):
        """
    Returns if this space is a direct sum of hyperbolic planes.

    TO DO: Deal with Characteristic 2 fields also!
    """
        char_p = self.base_field().characteristic()

        ## Check that the base field does not have characteristic 2
        if char_p == 2:
            raise NotImplementedError, "We need to deal with fields of characteristic 2 also!"

        ## Check hyperbolicness with the anisotropic dimension
        return self.anisotropic_dim() == 0


    def is_degenerate(self):
        """
    Determines if the quadratic space is degenerate (i.e. it has
    some non-zero vector orthogonal to the entire space).

    INPUT:
      None

    OUTPUT:
      boolean

    EXAMPLES:
      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(ZZ, [1,1,1,1]))
      sage: QS.is_degenerate()
      False

      sage: QS = QuadraticSpace(FiniteField(9,x), DiagonalQuadraticForm(ZZ, [1,3,5,7]))
      sage: QS.is_degenerate()
      True

      sage: QS = QuadraticSpace(RR, DiagonalQuadraticForm(ZZ, []))   ## The zero-dim'l for is non-degenerate by
def'n!
      sage: QS.is_degenerate()
      False

    """
        return self.__hessian_bilinear_space.is_degenerate()


    def is_nondegenerate(self):
        """
    Determines if the quadratic space is non-degenerate (i.e. it
    has no non-zero vectors orthogonal to the entire space).

    INPUT:
      None
```

```
    OUTPUT:
      boolean

    EXAMPLES:
      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(ZZ, [1,1,1,1]))
      sage: QS.is_nondegenerate()
      True

      sage: QS = QuadraticSpace(FiniteField(9,x), DiagonalQuadraticForm(ZZ, [1,3,5,7]))
      sage: QS.is_nondegenerate()
      False

      sage: QS = QuadraticSpace(RR, DiagonalQuadraticForm(ZZ, []))  ## The zero-dim'l for is non-degenerate by
def'n!
      sage: QS.is_nondegenerate()
      True
    """
        return not self.is_degenerate()



    def represents_the_space(self, V):
        """
    Determine if the number or quadratic space V is represented by
    the current quadratic space (i.e. whether self represents V).
    Note: The representation here is allowed to be degenerate.

    INPUT:
      V -- a quadratic space

    OUTPUT:
      boolean

    EXAMPLES:
      sage: QS4 = QuadraticSpace(QQ, DiagonalQuadraticForm(ZZ, [1,1,1,1]))
      sage: QS1 = QuadraticSpace(QQ, DiagonalQuadraticForm(ZZ, [1,1]))
      sage: QS4.represents_the_space(QS1)
      Traceback (most recent call last):
      ....
      NotImplementedError: This function is not implemented yet!

    """
        raise NotImplementedError, "This function is not implemented yet!"



    def is_represented_by_the_space(self, V):
        """
    Determine if the number or quadratic space V is represented by
    the current quadratic space (i.e. whether self represents V).
    Note: The representation here is allowed to be degenerate.

    INPUT:
      V -- a quadratic space

    OUTPUT:
      boolean

    EXAMPLES:
      sage: QS4 = QuadraticSpace(QQ, DiagonalQuadraticForm(ZZ, [1,1,1,1]))
      sage: QS1 = QuadraticSpace(QQ, DiagonalQuadraticForm(ZZ, [1,1]))
      sage: QS1.is_represented_by_the_space(QS4)
      Traceback (most recent call last):
      ....
      NotImplementedError: This function is not implemented yet!
```



```
        """
        if isinstance(V, QuadraticSpace):
            return V.represents_the_space(self)
        else:
            raise TypeError, "The argument must also be a quadratic space!"



    def is_isomorphic_to(self, other, comparison_field=None):
        """
    Determine if the two quadratic spaces are rationally
    isomorphic over comparison_field.  If no comparison field is
    specified, then we test over the base field of the two
    quadratic spaces (which we assume to be the same, and raise an
    error otherwise).

    TO DO: Implement anisotropic_dim() and many field-specific isomorphism tests!!

    INPUT:
      other -- a quadratic space
      comparison_field -- an optional field to which both
        quadratic spaces can be extended to test isomorphism.

    OUTPUT:
      boolean

    EXAMPLES:
      sage: Q1 = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
      sage: Q2 = Q1(Matrix(ZZ, 3, 3, [1,2,3,0,2,4,0,0,3]))
      sage: QS1 = QuadraticSpace(QQ, Q1)
      sage: QS2 = QuadraticSpace(QQ, Q2)
      sage: QS1 == QS1
      True
      sage: QS1 == QS2
      False
      sage: QS1.is_isomorphic_to(QS2)
      True

    """
        ## Sanity Check: Check that it's another quadratic space
        if not isinstance(other, QuadraticSpace):
            raise TypeError, "Oops!  Both spaces must be of the same type to test equality!"


        ## Check their base fields agree
        if (self.base_field() != other.base_field()):                ## SERIOUS
WARNING:  HERE WE ARE TESTING FIELD EQUALITY -- NOT FIELD ISOMORPHISM!  Be *espe
cially* careful about precisions for inexact/real fields!
            return False

        ## Check their total and anisotropic dimensions agree
        if (self.dim() != other.dim())  or (self.anisotropic_dim() != other.anis
otropic_dim()):
            return False

        ## Check if their determinants lie in the same squareclass
        if (self.determinant() != other.determinant()):
            return False


        ## Field-Specific Isomorphism Testing:
        ## -----------------------------------

        ## Test for QQ:
        if self.base_field() == RationalField():

            ## Check that the Hasse invariants agree at all places where c_p = -
1
```

```
            if self.hasse_primes_of_QQ() != other.hasse_primes_of_QQ():
                return False

            ## Check that the real signatures agree also
            return self.__quadratic_form.signature() == other.__quadratic_form.s
ignature()


        ## Test for Finite Fields:
        elif is_FiniteField(self.base_field()):
            return True

        ## Test for CC:
        elif is_ComplexField(self.base_field()):
            return True

        ## Test for RR:
        elif is_RealField(self.base_field()):
            self.__quadratic_form.signature() == other.quadratic_form.signature(
)

        ## Test for p-adic Fields:
        elif is_pAdicFields(elf.base_field()):

            ## Test that we're over Q_p
            p = self.base_field().prime()
            if not is_prime(p):
                raise RuntimeError, "Only p-adic fields Q_p with p prime are supported currently."

            ## Finish the isomorphism test
            return (self.hasse_invariant() == other.hasse_invariant())


        ## Tests for Number Fields:
        elif isinstance(self.base_field(), NumberField):
            ### Over a NumberFieldLocalization, compare local invariants
            #if isinstance(self.base_field(), NumberFieldLocalization):
            #    return (self.hasse_invariant == other.hasse_invariant) and (sel
f.determinant() == other.determinant())
            #
            ### Check their determinants are not zero
            #if self.determinant() == 0 or other.determinant() == 0:
            #    raise NotImplementedError, "Sorry, we don't know how to deal wi
th singular forms yet! =("
            raise NotImplementedError, "Isomorphism testing for has not been implemented yet for n
umber fields."


        ## If we're here, then the base_field has no isomorphism test yet!
        else:
            raise NotImplementedError, "Isomorphism testing for has not been implemented yet for t
he base field " + str(self.base_field()) + "."




################################################################################
################################################
#################################################  Routines for Local Quadrati
c Spaces  ###############################################
################################################################################
################################################


def local_quadratic_space_anisotropic_dimension_by_invariants(p, n, d, c):
    """
    Computes the anisotropic dimension of the quadratic space over Q_p
    of dimension n, Gram determinant d, and Hasse invariant c (with
    the i<j definition).

    Here we are using the direct sum formula for invariants, the fact
    that the hyperbolic plane has d=-1, c=1, and the classification of
    invariants of anisotropic spaces by local invariants.

    INPUT:
        p -- a prime number > 0
        n -- an integer >=0
        d -- an integer (or perhaps a rational number?)
        c -- 1 or -1

    OUTPUT:
        0, 1, 2, 3, or 4.

    EXAMPLES:
        sage: from sage.quadratic_forms.quadratic_space import local_quadratic_space_anisotropic_dimension_by_invari
ants

        sage: Q = DiagonalQuadraticForm(ZZ, [1, -2, 5])  ## Anisotoropic at p=2 and p=5
        sage: Q.anisotropic_primes()
        [2, 5]
        sage: Q.Gram_det()
        -10
        sage: Q.hasse_invariant(2)
        -1
        sage: Q.hasse_invariant(5)
        -1
        sage: Q.hasse_conductor()
        10
        sage: local_quadratic_space_anisotropic_dimension_by_invariants(2, 3, -10, -1)
        3
        sage: local_quadratic_space_anisotropic_dimension_by_invariants(3, 3, -10, 1)
        1
        sage: local_quadratic_space_anisotropic_dimension_by_invariants(5, 3, -10, -1)
        3
        sage: [local_quadratic_space_anisotropic_dimension_by_invariants(p, Q.dim(), Q.Gram_det(), Q.hasse_invariant(
p)) for p in primes_first_n(20)] == [3,1,3] + 17*[1]
        True

        sage: Q = DiagonalQuadraticForm(ZZ, [1, -2, 5, -10])  ## Anisotoropic at p=2 and p=5
        sage: [local_quadratic_space_anisotropic_dimension_by_invariants(p, Q.dim(), Q.Gram_det(), Q.hasse_invariant(
p)) for p in primes_first_n(20)] == [4, 0, 4] + 17*[0]
        True

        sage: local_quadratic_space_anisotropic_dimension_by_invariants(2, 9, -2, -1)
        3

        sage: local_quadratic_space_anisotropic_dimension_by_invariants(2, 4, 1, 1)
        4

    """
    ## Handle even dim'l forms
    if n % 2 == 0:
        m = n/2
        if SquareClass(Qp(p), d) != SquareClass(Qp(p), (-1)**m):   ## This chara
cterizes the non-extremal even dim'l cases
            return 2
        else:
            ## Compute the hasse invariant c_m of the hyperbolic plane of dimens
ion 2m
            if (p != 2) or (m % 4 <= 1):    ## m == 0 or 1 mod 4 here
                c_m = 1
            else:
                c_m = -1

            ## Compute the even anisotropic dimension
            if c == c_m:    ## This characterizes the Hyperbolic plane
```

```
                return 0
            else:
                return 4

    ## Handle odd dim'l forms
    else:
        m = (n-3)/2
        d_m = (-1)**m
        ## Compute the hasse invariant c_m of the hyperbolic plane of dimension
2m
        if (p != 2) or (m % 4 <= 1):    ## m == 0 or 1 mod 4 here
            c_m = 1
        else:
            c_m = -1

        ## Compute the invariants associated with the (possibly anisotropic) ter
nary space
        d1 = d * d_m
        c1 = c * c_m * hilbert_symbol(d1, d_m, p)

        ## Compute the odd anisotropic dimension
        if c1 == hilbert_symbol(-1, -d1, p):   # Check if the 3-dim'l space is i
sotropic
            return 1
        else:
            return 3




def local_quadratic_space_core_invariants_from_invariants(p, n, d, c):
    """
  Returns a triple of local invariants (n', d', c') describing the
  core (maximal anisotropic) subspace of the quadratic space over
  Q_p with invariants (n, d, c) of dimension n, Gram determinant d,
  and Hasse invariant c (with the i<j definition).

  INPUT:
    p -- a prime number > 0
    n -- an integer >=0
    d -- an integer (or perhaps a rational number?)
    c -- 1 or -1

  OUTPUT:
    a triple (n, d, c) as above.

  EXAMPLES:
    sage: from sage.quadratic_forms.quadratic_space import local_quadratic_space_core_invariants_from_invariants

    sage: local_quadratic_space_core_invariants_from_invariants(2, 4, 1, 1)  ## This sa a 2-dim'l core subspace
    (4, 1, 1)
    sage: local_quadratic_space_core_invariants_from_invariants(2, 4, 1, -1)  ## This is anisotropic at p=2
    (0, 1, 1)
    sage: local_quadratic_space_core_invariants_from_invariants(3, 4, 1, 1)
    (0, 1, 1)
    sage: local_quadratic_space_core_invariants_from_invariants(5, 4, 1, -1)  ## This is anisptropic at p=5
    (4, 1, -1)

  """
    ## Find the dimension of a maximal anisotropic subspace
    a = local_quadratic_space_anisotropic_dimension_by_invariants(p, n, d, c)
    m = (n - a) / 2

    ## Compute the Hilbert symbol for the hyperbolic plane of dimension 2m
    d_m = (-1)**m
    if (p != 2) or (m % 4 <= 1):    ## m == 0 or 1 mod 4 here

        c_m = 1
```



```
    else:
        c_m = -1

    ## Compute and return the invariants
    d1 = d * d_m
    c1 = c_m * c * hilbert_symbol(d_m, d1, p)
    return (a, d1, c1)          ## TO DO:  We could add a sanity check here to be
 sure our invariants correspond to an anisotropic space!




def local_quadratic_space_by_invariants(n, d, c):
    """
  Returns a (diagonal) quadratic space over Q_p of dimension n, Gram
  determinant d, and Hasse invariant c (with the i<j definition).

  INPUT:
    n -- an integer >=0
    d -- a non-zero squareclass over Qp
    c -- 1 or -1

  OUTPUT:
    a diagonal quadratic space over Q_p

  EXAMPLES:
    sage: from sage.quadratic_forms.quadratic_space import local_quadratic_space_by_invariants

    sage: QS1 = local_quadratic_space_by_invariants(4, SquareClass(Qp(2), 1), 1)
    sage: QS1.base_field().prime() == 2
    True
    sage: QS1.dim() == 4
    True
    sage: QS1.determinant() == SquareClass(Qp(2), 1)
    True
    sage: QS1.hasse_invariant() == 1
    True

  """
    p = d.base_field().prime()

    #print "Entering local_quadratic_space_by_invariants with variables:"
    #print "p = ", p
    #print "n = ", n
    #print "d = ", d
    #print "c = ", c
    #print



    ## Find the invariants of the core subspace
    n1, d1, c1 = local_quadratic_space_core_invariants_from_invariants(p, n, d.n
ormalized_representative(), c)
    #print "The core subspace invariants at p = " + str(p) + " are:"
    #print "n1 = ", n1
    #print "d1 = ", d1
    #print "c1 = ", c1
    #print


    ## Construct the anisotropic subspace
    if n1 == 0:
        Q1 = QuadraticForm(Qp(p), 0, [])

    elif n1 == 1:
        Q1 = QuadraticForm(Qp(p), 1, [d1])
```

```
    elif n1 == 2:
        if c1 == 1:
            Q1 = QuadraticForm(Qp(p), 2, [1, 0, d1])
        else:
            for a in local_squareclass_representatives_list(p):
                if hilbert_symbol(a, a*d1, p) == -1:
                    Q1 = QuadraticForm(Qp(p), 2, [a, 0, a*d1])


    elif n1 == 3:
        ## Find the appropriate unit squareclass to normalize the determinant
        u1 = d1 / p**valuation(d1, p)

        ## Find a non-square unit in Z_p (with special conditions when p = 2)
        if p != 2:
            v = least_quadratic_nonresidue(p)
        else:
            for v in [3,5,7]:
                if hilbert_symbol(-p*v*u1, -v, p) == c1:
                    break

        ## Return the anisotropic ternary space
        if valuation(d1, p) % 2 == 1:
            Q1 = DiagonalQuadraticForm(Qp(p), [1, -v, -p*v*u1])
        else:
            Q1 = DiagonalQuadraticForm(Qp(p), [-v*u1, p, -p*v])    ## This works
 because c is scale invariant for odd dimensions.


    elif n1 == 4:
        ## Find a non-square unit in Z_p
        if p != 2:
            u = least_quadratic_nonresidue(p)
            Q1 = DiagonalQuadraticForm(Qp(p), [1, -u, p, -p*u])
        else:
            for u in [3, 5, 7]:
                Q1 = DiagonalQuadraticForm(Qp(p), [1, -u, p, -p*u])
                if Q1.hasse_invariant(2) == 1:        ## The Hasse invariant mus
t be 1 for the 4-dim'l anisotropic form at p=2
                    break
    else:
        raise RuntimeError, "There is a problem, since we got n1 = " + str(n1) + ", but the anisotr
opic dimension must be <= 4 for Q_p."


    ## Append hyperbolic planes to get the correct dimension
    m = (n - n1)/2
    #return Q1 + HyperbolicPlane_quadratic_form(Qp(p), m)
    Q_final = Q1 + DiagonalQuadraticForm(Qp(p), m*[1, -1])    ## This returns a
diagonal form. =)


    #print "p = ", p
    #print "Q_final =", Q_final


    ## Sanity Check: Test the invariants are what we asked for!
    if Q_final.dim() != n:
        raise RuntimeError, "The dimension " + str(Q_final.dim()) + \
              " of the local space we constructed doesn't match the desired dimension " + str(n) + " over Q
_" + str(p) + "."
    if SquareClass(Qp(p), Q_final.Gram_det()) != SquareClass(Qp(p), d):
        raise RuntimeError, "The determinant squareclass of " + str(Q_final.Gram_det()) +
 \
              " of the local space we constructed doesn't match the desired squareclass of " + str(d) + " ov
er Q_" + str(p) + "." \
              + "\n" + str(Q_final)
    if Q_final.hasse_invariant(p) != c:
```



```
        raise RuntimeError, "The Hasse invariant " + str(Q_final.hasse_invariant(p)) +
 \
              " of the local space we constructed doesn't match the desired invariant " + str(c) + " over Q_
" + str(p) + "." \
              + "\n" + str(Q_final)


    ## Return the local quadratic space
    return QuadraticSpace(Q_final)




def local_quadratic_space_GHY_to_Standard_invariants(n, delta, w):
    """
    Translates a triple (n, delta, w) of GHY local invariants (see
    pp116–7 of [GHY]) for a local quadratic space over Q_p to the
    usual local invariants (n,d,c) (given in [Ca] on p13, p55, and
    p403, and in [OM] on pp86–7).

    INPUTS:
      n –– integer >= 0
      delta –– a non–zero squareclass over Qp
      w –– 1 or –1

    OUTPUTS:
      n –– integer >= 0
      d –– a non–zero (Gram determinant) squareclass over Qp
      w –– 1 or –1  (the Hasse invariant)

    REFERENCES:
      [Ca]  Cassels, "Rational Quadratic forns", Book
      [GHY] Gan, Hanke, Yu, "On an exact mass formula of Shimura", paper
      [OM] O'Meara "Introduction to Quadratic Forms", Book
      (See my 2/1/2006  and 9/16/2006 notes for details)

    EXAMPLES:
      sage: from sage.quadratic_forms.quadratic_space import local_quadratic_space_GHY_to_Standard_invariants

      ## Odd, w=1, p=2 –– checked
      sage: local_quadratic_space_GHY_to_Standard_invariants(3, SquareClass(Qp(2), 1), 1)
       (3, The squareclass represented by 1 + 2 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7 + 2^8 + 2^9 + 2^10 + 2^11 + 2^12
+ 2^13 + 2^14 + 2^15 + 2^16 + 2^17 + 2^18 + 2^19 + O(2^20) over 2–adic Field with capped relative precision 20, 1)
      sage: local_quadratic_space_GHY_to_Standard_invariants(3, SquareClass(Qp(2), –1), 1)
       (3, The squareclass represented by 1 + O(2^20) over 2–adic Field with capped relative precision 20, –1)
      sage: local_quadratic_space_GHY_to_Standard_invariants(3, SquareClass(Qp(2), 5) ,1)
       (3, The squareclass represented by 1 + 2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7 + 2^8 + 2^9 + 2^10 + 2^11 + 2^12 + 2^1
3 + 2^14 + 2^15 + 2^16 + 2^17 + 2^18 + 2^19 + O(2^20) over 2–adic Field with capped relative precision 20, 1)
      sage: local_quadratic_space_GHY_to_Standard_invariants(3,SquareClass(Qp(2), –5) ,1)
       (3, The squareclass represented by 1 + 2^2 + O(2^20) over 2–adic Field with capped relative precision 20, –1)

      ## Odd, w=–1, p=2 –– checked
      sage: local_quadratic_space_GHY_to_Standard_invariants(3, SquareClass(Qp(2), 1), –1)
       (3, The squareclass represented by 1 + 2 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7 + 2^8 + 2^9 + 2^10 + 2^11 + 2^12
+ 2^13 + 2^14 + 2^15 + 2^16 + 2^17 + 2^18 + 2^19 + O(2^20) over 2–adic Field with capped relative precision 20, –1
)
      sage: local_quadratic_space_GHY_to_Standard_invariants(3, SquareClass(Qp(2), –1), –1)
       (3, The squareclass represented by 1 + O(2^20) over 2–adic Field with capped relative precision 20, 1)
      sage: local_quadratic_space_GHY_to_Standard_invariants(3, SquareClass(Qp(2), 5), –1)
       (3, The squareclass represented by 1 + 2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7 + 2^8 + 2^9 + 2^10 + 2^11 + 2^12 + 2^1
3 + 2^14 + 2^15 + 2^16 + 2^17 + 2^18 + 2^19 + O(2^20) over 2–adic Field with capped relative precision 20, –1)
      sage: local_quadratic_space_GHY_to_Standard_invariants(3, SquareClass(Qp(2), –5), –1)
       (3, The squareclass represented by 1 + 2^2 + O(2^20) over 2–adic Field with capped relative precision 20, 1)
```

```
  """
    p = delta.base_field().prime()


    ## Sanity checks
    if not( n in ZZ  and n>=1  and  w in ZZ and abs(w) == 1  and  isinstance(del
ta, SquareClass)  and delta.is_nonzero()  and is_prime(p)):
        raise TypeError, "Oops!  There is a problem with your input data (n, delta, w, p) = (" \
            + str(n) + "," + str(delta) + "," + str(w) + "," + str(p) + ")!"


    ## Compute the new invariants:
    ## ---------------------------
    if is_even(n):

        ## Case 1: V_(d,1) = H^r  OR  Case 2: V_(d,-1) = H^(r-2) + D  --  delta
= square
        if delta.is_unit_squares():
            r = int(n/2)
            d = SquareClass(Qp(p), (-1)**r)
            c = w * (hilbert_symbol(-1, -1, p)**(floor(r/2)))

            ## Return the results
            return int(n), d, c

        ## Case 2: V = H^(r-1) + 2-dim'l space
        else:
            r = int(n/2)
            d = delta * (-1)**r
            c = w * hilbert_symbol(-1, -1, p)**floor((r-1)/2) * (delta * (-1)).h
ilbert_symbol(SquareClass(Qp(p), (-1)**(r-1)))

            ## Return the results
            return int(n), d, c

    else:
        ## Case 3: V = H^r + delta * x^2
        if (w == 1):
            r = int((n-1)/2)
            d = delta * (-1)**r
            c = delta.hilbert_symbol(SquareClass(Qp(p), (-1)**r))  * hilbert_sym
bol(-1, -1, p)**floor(r/2)

            ## Return the results
            return int(n), d, c

        ## Case 4: V = H^(r-1) + 3-dim'l space
        else:
            r = int((n-1)/2)
            d = delta * (-1)**r
            c1 = hilbert_symbol(-1, -1, p)**floor((r-1)/2)
            c2 = (delta * (-1)).hilbert_symbol(SquareClass(Qp(p), (-1)**(r-1)))
            c = -delta.hilbert_symbol(SquareClass(Qp(p), -1)) * c1 * c2

            ## Return the results
            return int(n), d, c




def local_quadratic_space_Standard_to_GHY_invariants(n, d, c):
    """
  ********************** UNTESTED **********************

  Translates a triple (n,delta,w) of GHY local invariants (see
  pp116–7 of [GHY]) for a local quadratic space over Q_p to the
  usual local invariants (n,d,c) (given in [Ca] on p13, p55, and
  p403, and in [OM] on pp86–7).
```



```
  INPUTS:
    p -- prime number
    n -- integer
    delta -- non-zero integer representing a squareclass
    w -- 1 or -1

  OUTPUTS:
    p -- prime number
    n -- integer
    d -- non-zero integer representing the determinant/discriminant squareclass
    w -- 1 or -1  (the Hasse invariant)

  REFERENCES:
    [Ca] Cassels, "Rational Quadratic forns", Book
    [GHY] Gan, Hanke, Yu, "On an exact mass formula of Shimura", paper
    [OM] O'Meara "Introduction to Quadratic Forms", Book
    (See my 2/1/2006  and 9/16/2006 notes for details)

  EXAMPLES:
    sage: from sage.quadratic_forms.quadratic_space import local_quadratic_space_GHY_to_Standard_invariants
    sage: from sage.quadratic_forms.quadratic_space import local_quadratic_space_Standard_to_GHY_invariants

    #sage: for x in range(3): \
    #     print x


    ## Check it is compatible with its inverse routine
    sage: for p in prime_range(22):
    ...     for n in range(1, 9):
    ...         for d in local_squareclass_representatives_list(p):
    ...             for c in [1, -1]:
    ...                 if (n, SquareClass(Qp(p), d), c) != local_quadratic_space_GHY_to_Standard_invariants(*local_quadr
atic_space_Standard_to_GHY_invariants(n, SquareClass(Qp(p), d), c)):
    ...                     raise RuntimeError, "There was a problem with Std -> GHY -> Std conversio
n for (p, n, d, c) = (" + str(p) + ", " + str(n) + ", " + str(d) + ", " + str(c) + ")."


  """
    p = d.base_field().prime()


    ## Sanity checks
    if not( n in ZZ  and n>=1  and  c in ZZ and abs(c) == 1  and  isinstance(d,
SquareClass)  and d.is_nonzero()  and is_prime(p)):
        raise TypeError, "Oops!  There is a problem with your input data (p, n, d, c) = (" \
            + str(p) + "," + str(n) + "," + str(d) + "," + str(c) + ")!"



    ## Compute the anisotropic/core invariants
    aniso_dim, aniso_det, aniso_hasse = local_quadratic_space_core_invariants_fr
om_invariants(p, n, d.normalized_representative(), c)


    ## Compute the new invariants:
    ## ---------------------------
    if is_even(n):

        ## Case 1: V_(d,1) = H^r  OR  Case 2: V_(d,-1) = H^(r-2) + D  --  delta
= square
        if aniso_dim == 0:
            return n, SquareClass(Qp(p), 1), 1     ## delta = 1, w = 1
        elif aniso_dim == 4:
            return n, SquareClass(Qp(p), 1), -1    ## delta = 1, w = -1
        ## Case 2: V = H^(r-1) + 2-dim'l space
        elif aniso_dim == 2:
            return n, SquareClass(Qp(p), -aniso_det), aniso_hasse    ## See 9/28
/09 Notes, page 2
        else:
```

```
            raise RuntimeError, "There is a problem with the anisotropic dimension " + str(aniso
_dim) + " here... it must be 0, 2, or 4."

    else:

        ## Case 3: V = H^r + delta * x^2
        if aniso_dim == 1:
            return n, SquareClass(Qp(p), aniso_det), 1
        ## Case 4: V = H^(r-1) + 3-dim'l space
        elif aniso_dim == 3:
            return n, SquareClass(Qp(p), -aniso_det), -1
        else:
            raise RuntimeError, "There is a problem with the anisotropic dimension " + str(aniso
_dim) + " here... it must be 1 or 3."




def find_locally_represented_number(n, d, c, p):
    """
    Finds a number represented by the quadratic form with local
    invariants (n,d,c) at the prime p.

    TO DO: FIX THE ORDER OF THIS QUADRUPLE TO MAKE IT AGREE WITH THE
       OTHER ROUTINES ABOVE (i.e. put p first!).

    INPUT:
       n -- integer > 0
       d -- integer (or squareclass over Q_p??)
       c -- 1 or -1
       p -- prime integer (or prime ideal)

    OUTPUT:
       integer (or number field element)

    EXAMPLES:
       sage: from sage.quadratic_forms.quadratic_space import find_locally_represented_number
       sage: find_locally_represented_number(1, 1, 1, 5)
       1

       sage: find_locally_represented_number(3, 41, -1, 41)
       3

    """
    ## Deal with d if it's a squareclass
    if isinstance(d, SquareClass):
        d = d.representative()


    ## Sanity Checks
    if not (n in ZZ  and n > 0):
        raise TypeError, "Oops!  The dimension " + str(n) + " must be a positive integer!"

    if not ((d in ZZ and d != 0) or isinstance(d, SquareClass)):
        raise TypeError, "Oops!  The determinant is not a non-zero number/squareclass!"

    if not (c in ZZ and abs(c) == 1):
        raise TypeError, "Oops!  The Hasse invariant " + str(c) +" must be 1 or -1."

    if not (p in ZZ and is_prime(p)):
        raise TypeError, "Oops! " + str(p) + " must be a positive prime integer!"

    ## Check that the local invariants are compatible for 1 and 2-dim'l forms
    if (n == 1 and c != 1)  or  (n == 2  and c != 1 and SquareClass(Qp(p), -d).i
s_unit_squares()):
        raise ValueError, "Oops! The local invariants (n,d,c,p) = " + str((n,d,c,p)) + " are inc
ompatible! =("
```



```


    ## Compute a represented number
    if n == 1:
        return d

    elif n == 2:
        for b in local_squareclass_representatives_list(p):
            if hilbert_symbol(b, -d, p) == c:
                return b

    elif n == 3:
        ## TO DO: Could speed this up since we only need to check at
        ## most 2 squareclasses, and we already know 1 and p. =)
        d1 = SquareClass(Qp(p), d)
        for b in local_squareclass_representatives_list(p):
            if SquareClass(Qp(p), -b) != d1:
                return b

    elif n >= 4:  ## It's either isotropic => universal or its anisotropic and k
nown to be universal.
        return 1




def rational_quadratic_space_from_local_space_list(local_quadratic_list):
    """
    Find a rational quadratic space over QQ which realizes the given
    local spaces, and has unit discriminant and Hasse invariant 1 at
    all other places.

    Note: Over a number field F we will need to specify which
    places/primes of F are associated to which p-adic spaces.

    TO DO:  ADD POSITIVE DEFINITE ABILITY!!!

    INPUT:
       local_quadratic_list -- a list of quadratic spaces defined
          over localizations of a global field (currently only QQ is
          supported)

    OUTPUT:
       a quadratic space over a global field (currently only QQ is supported)

    EXAMPLES:
       sage: from sage.quadratic_forms.quadratic_space import rational_quadratic_space_from_local_space_list
       sage: Q1 = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
       sage: QS1 = QuadraticSpace(QQ, Q1)
       sage: QS1_2 = QS1.localize_at_place(2)
       sage: QS1_41 = QS1.localize_at_place(41)
       sage: QS1_Infty = QS1.localize_at_place(Infinity)
       sage: N = rational_quadratic_space_from_local_space_list([QS1_Infty, QS1_2, QS1_41])
       sage: QS1.is_isomorphic_to(N)
       True
       sage: N
       Quadratic space defined by the Quadratic form in 3 variables over Rational Field with coefficients:
       [ 249 0 0 ]
       [ * 95203 0 ]
       [ * * 41/23705547 ]

    """
    ## Set some local variables
    F = RationalField()
    n = local_quadratic_list[0].dim()
```

```
    ## Check that all spaces are of the same dimension and are localization of t
he given number field F
    pass

    ## Check that the determinant (square class) is realized by some global fiel
d element.
    pass

    ## Check that no localization appears twice.
    pass

    ## Check that the product of the Hasse invariants is 1.
    for entry in local_quadratic_list:
        hasse_prod = 1
        for entry in local_quadratic_list:
            hasse_prod *= entry.hasse_invariant()
        if hasse_prod != 1:
            raise ValueError, " The product of the Hasse invariants is not 1. =("

    ## WARNING: Only do over Q to avoid units and the class group...
    if F.degree() > 1:
        raise NotImplementedError, " Sorry, we only do local-global over the rational numbers for no
w... =("


    ############################################################################
#####
    ## Step 1: Find a common rational determinant, and deal with 1-dimensional f
orms
    ############################################################################
#####

    ## Find a common rational determinant squareclass compatible with the local
ones
    try:
        d = weak_approx_for_squareclasses_over_QQ([entry.determinant()  for entr
y in local_quadratic_list])
    except:
        raise ValueError, " There was a problem finding a rational squareclass to represent the determinan
t. =("


    #print "local_quadratic_list = \n", local_quadratic_list
    #print "weak input = \n", [entry.determinant()  for entry in local_quadratic
_list]
    #print "d = ", d, type(d)


    ## Deal with the case n=1
    if n == 1:
        return QuadraticSpace(F, [d])


    ##################################################
    ## Step 2: Setup lists to keep the local invariants
    ##################################################

    ## Make two lists describing the desired local behavior (for the rationals o
nly!!!)
    tmp_n = n
    tmp_d = d

    tmp_P_list = []
    tmp_C_list = []
    tmp_num_list = []
    for entry in local_quadratic_list:
        ## p-adic Fields
        if is_pAdicField(entry.base_field()):
            tmp_P_list.append(entry.base_field().prime())
```



```
            tmp_C_list.append(entry.hasse_invariant())
            tmp_num_list.append(find_locally_represented_number(n, d, \
                                                entry.hasse_invariant(), entry.base_fie
ld().prime()))
        ## archimedean (i.e. real) fields
        else:
            tmp_signature = entry.defining_quadratic_form().signature()
            tmp_P_list.append(Infinity)
            tmp_C_list.append(tmp_signature)

            ## Determine some value represented by a real quadratic form
            if tmp_signature == -tmp_n:
                tmp_num_list.append(-1)
            else:
                tmp_num_list.append(1)


    ## Be sure that p=2 is in there as well! =)
    if not 2 in tmp_P_list:
        tmp_P_list.append(2)
        tmp_C_list.append(1)
        tmp_num_list.append(find_locally_represented_number(n, d, 1, 2))  ## TOD
O: This never changes, so over Q we can replace it with a number! =)


    #print
    #print "tmp_n = ", tmp_n
    #print "tmp_d = ", tmp_d
    #print "tmp_P_list = ", tmp_P_list
    #print "tmp_C_list = ", tmp_C_list
    #print "tmp_num_list = ", tmp_num_list
    #print


    ############################################################
    ## Step 3: Reduce to the binary case by splitting off lines
    ############################################################
    splitting_diagonal = []
    #prime_flag = False

    while tmp_n > 2:
    #for i in range(n-1):

        #print "n-1 = ", n-1
        #print "i = ", i
        #print

        ## Set an additional prime_flag when i = n-1, to pass into the weak appr
ox routine! =)
        #if i == (n - 2):
        #    prime_flag = True

        ## Setup for the weak approximation to find the splitting number t
        approx_list = []
        for j in range(len(tmp_P_list)):
            p = tmp_P_list[j]
            approx_list.append([p, local_squareclass_radius_val(p), tmp_num_list
[j]])        ## This deals with Infinity as well (i.e. radius is 1)! =)

        #print "approx_list = ", approx_list


        ## Compute a rational splitting number t, and save it
        t = weak_approx_for_numbers_over_QQ(approx_list)
        splitting_diagonal.append(QQ(t))

        ## Recompute the 2 lists, and lower the dimension:
        ## ----------------------------------------------
```

```
        ## Extend the current lists by adding new (possibly bad) primes
        t_primes = prime_divisors(t)
        big_P_list = tmp_P_list + [p  for p in t_primes  if not p in tmp_P_list]
        big_C_list = tmp_C_list + [1  for p in t_primes  if not p in tmp_P_list]
        tmp_P_list = []
        tmp_C_list = []
        tmp_num_list = []
        n_new = tmp_n - 1
        d_new = tmp_d * t     ## Note: This is a SquareClass over QQ

        #print "-->  using t = ", t

        ###########################################################################
########

        ## Compute the new invariant lists (for the summand)
        for j in range(len(big_P_list)):
            p = big_P_list[j]

            ## p-adic Fields
            if p != Infinity:
                c_new = big_C_list[j] * d_new.hilbert_symbol(SquareClass(QQ, t),
 p)  ## Find the Hasse invariant of the reduced form
                if is_odd(d_new.valuation(p)) or (c_new == -1) or (p == 2):
                    tmp_P_list.append(p)
                    tmp_C_list.append(c_new)

                    #print "p = ", p
                    #print "c_new = ", c_new

                    tmp_num_list.append(find_locally_represented_number(n_new, d
_new, c_new, p))

            ## archimedean (i.e. real) fields
            else:
                tmp_P_list.append(Infinity)
                tmp_signature = big_C_list[j] - sgn(t)
                tmp_C_list.append(tmp_signature)

                ## Determine some value represented by a real quadratic form
                if tmp_signature == -n_new:
                    tmp_num_list.append(-1)
                else:
                    tmp_num_list.append(1)


        ## Update the dimension and determinant
        tmp_n = n_new
        tmp_d = d_new   ## Adjust this to be squarefree?


        ###########################################################################
######


        #print
        #print "splitting_diagonal = ", splitting_diagonal
        #print "tmp_n = ", tmp_n
        #print "tmp_d = ", tmp_d
        #print "tmp_P_list = ", tmp_P_list
        #print "tmp_C_list = ", tmp_C_list
        #print "tmp_num_list = ", tmp_num_list
        #print


        #print "tmp_C_list = ", tmp_C_list
        #print "big_C_list = ", big_C_list
        #print
```



```
    ###########################################################################
    ## Step 4: Construct the rational binary form, by Dirichlet's theorem
    ###########################################################################
#    print "d-type = ", type(d)
#    print "t-type = ", type(splitting_diagonal[0])


    ## Make a list of local squareclasses to use
    tmp_sq_list = [SquareClass(Qv(tmp_P_list[i]), tmp_num_list[i])  for i in ran
ge(len(tmp_P_list)) \
                   if (tmp_C_list[i] == -1) or (tmp_d.valuation(tmp_P_list[i]) =
= 1) \
                   or (tmp_P_list[i] == 2) or (tmp_P_list[i] == Infinity)]


    #print " tmp_sq_list = ", tmp_sq_list


    ## Find the number locally represented by the binary space, divisible by pri
mes from only one good place.
    a = strong_approx_for_squareclasses_by_QQ_except_at_one_prime(tmp_sq_list, r
eturn_integral_representative=True)

    #print "a = ", a

    ## Add the binary form to the splitting diagonal, and return the (diagonal)
rational quadratic space
    splitting_diagonal.append(a)
    splitting_diagonal.append( d.representative() / prod(splitting_diagonal) )
    return QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, splitting_diagonal))
```

```
"""
Creating A Random Quadratic Form
"""
from sage.quadratic_forms.quadratic_form import QuadraticForm
from sage.rings.ring import is_Ring

###############################################
## Routines to create a random quadratic form ##
###############################################

def random_quadraticform(R, n, rand_arg_list=[]):
    """
    Create a random quadratic form in ‘n‘ variables defined over the ring ‘R‘.

    The last (and optional) argument ‘‘rand_arg_list‘‘ is a list of at most 3
    elements which is passed (as at most 3 separate variables) into the method
    ‘‘R.random_element()‘‘.

    INPUT:

    – ‘R‘ –– a ring.
    – ‘n‘ –– an integer ‘\ge 0‘
    – ‘‘rand_arg_list‘‘ –– a list of at most 3 arguments which can be taken by
      ‘‘R.random_element()‘‘.

    OUTPUT:

    A quadratic form over the ring ‘R‘.

    EXAMPLES::

        sage: random_quadraticform(ZZ, 3, [1,5])    ## RANDOM
        Quadratic form in 3 variables over Integer Ring with coefficients:
        [ 3 2 3 ]
        [ * 1 4 ]
        [ * * 3 ]

    ::

        sage: random_quadraticform(ZZ, 3, [–5,5])    ## RANDOM
        Quadratic form in 3 variables over Integer Ring with coefficients:
        [ 3 2 –5 ]
        [ * 2 –2 ]
        [ * * –5 ]

    ::

        sage: random_quadraticform(ZZ, 3, [–50,50])    ## RANDOM
        Quadratic form in 3 variables over Integer Ring with coefficients:
        [ 1 8 –23 ]
        [ * 0 0 ]
        [ * * 6 ]
    """
    ## Sanity Checks: We have a ring and there are at most 3 parameters for rand
omness!
    if len(rand_arg_list) > 3:
        raise TypeError, "Oops!  The list of randomness arguments can have at most 3 elements."
    if not is_Ring(R):
        raise TypeError, "Oops!  The first argument must be a ring."

    ## Create a list of upper-triangular entries for the quadratic form
    L = len(rand_arg_list)
    nn = int(n*(n+1)/2)
    if L == 0:
        rand_list = [R.random_element()  for _ in range(nn)]
    elif L == 1:
        rand_list = [R.random_element(rand_arg_list[0])  for _ in range(nn)]
    elif L == 2:
        rand_list = [R.random_element(rand_arg_list[0], rand_arg_list[1])  for _
```



```
 in range(nn)]
    elif L == 3:
        rand_list = [R.random_element(rand_arg_list[0], rand_arg_list[1], rand_a
rg_list[2])  for _ in range(nn)]

    ## Return  the Quadratic Form
    return QuadraticForm(R, n, rand_list)


def random_quadraticform_with_conditions(R, n, condition_list=[], rand_arg_list=
[]):
    """
    Create a random quadratic form in ‘n‘ variables defined over the ring ‘R‘
    satisfying a list of boolean (i.e. True/False) conditions.

    The conditions ‘c‘ appearing in the list must be boolean functions which
    can be called either as ‘‘Q.c()‘‘ or ‘‘c(Q)‘‘, where ‘‘Q‘‘ is the random
    quadratic form.

    The last (and optional) argument ‘‘rand_arg_list‘‘ is a list of at most 3
    elements which is passed (as at most 3 separate variables) into the method
    ‘‘R.random_element()‘‘.

    EXAMPLES::

        sage: Q = random_quadraticform_with_conditions(ZZ, 3, [QuadraticForm.is_positive_definite], [–5, 5])
        sage: Q   ## RANDOM
        Quadratic form in 3 variables over Integer Ring with coefficients:
        [ 3 –2 –5 ]
        [ * 2 2 ]
        [ * * 3 ]

    """
    Q = random_quadraticform(R, n, rand_arg_list)
    Done_Flag = True

    ## Check that all conditions are satisfied
    while Done_Flag:
        Done_Flag = False
        for c in condition_list:

            ## Check if condition c is satisfied
            try:
                bool_ans = Q.c()
            except:
                bool_ans = c(Q)

            ## Create a new quadratic form if a condition fails
            if (bool_ans == False):
                Q = random_quadraticform(R, n, rand_arg_list)
                Done_Flag = True
                break

    ## Return the quadratic form
    return Q
```

```
"""
Routines for computing special values of L-functions
"""

from sage.rings.polynomial.polynomial_ring_constructor import PolynomialRing
from sage.rings.arith import kronecker_symbol, bernoulli, factorial, fundamental
_discriminant
from sage.rings.all import RealField
from sage.combinat.combinat import bernoulli_polynomial
from sage.rings.rational_field import QQ
from sage.rings.integer_ring import ZZ
from sage.symbolic.constants import pi
from sage.symbolic.pynac import I
from sage.rings.real_mpfr import is_RealField
from sage.misc.functional import denominator, numerator
from sage.rings.infinity import infinity


## ---------------- The Gamma Function  ------------------

def gamma__exact(n):
    """
  Evaluates the exact value of the gamma function at an integer or
  half-integer argument.

  EXAMPLES::

    sage: gamma__exact(4)
    6
    sage: gamma__exact(3)
    2
    sage: gamma__exact(2)
    1
    sage: gamma__exact(1)
    1

    sage: gamma__exact(1/2)
    sqrt(pi)
    sage: gamma__exact(3/2)
    1/2*sqrt(pi)
    sage: gamma__exact(5/2)
    3/4*sqrt(pi)
    sage: gamma__exact(7/2)
    15/8*sqrt(pi)

    sage: gamma__exact(-1/2)
    -2*sqrt(pi)
    sage: gamma__exact(-3/2)
    4/3*sqrt(pi)
    sage: gamma__exact(-5/2)
    -8/15*sqrt(pi)
    sage: gamma__exact(-7/2)
    16/105*sqrt(pi)

  """
    from sage.all import sqrt
    ## SANITY CHECK
    if (not n in QQ) or (denominator(n) > 2):
        raise TypeError, "Oops!  You much give an integer or half-integer argument."

    if (denominator(n) == 1):
        if n <= 0:
            return infinity
        if n > 0:
            return factorial(n-1)
    else:
        ans = QQ(1)
        while (n != QQ(1)/2):
            if (n<0):
                ans *= QQ(1)/n
                n = n + 1
            elif (n>0):
                n = n - 1
                ans *= n

        ans *= sqrt(pi)
        return ans



## ------------- The Riemann Zeta Function  --------------

def zeta__exact(n):
    r"""
  Returns the exact value of the Riemann Zeta function

  References:

  - Iwasawa's "Lectures on p-adic L-functions", p13, "Special value of '\zeta(2k)'"
  - Ireland and Rosen's "A Classical Introduction to Modern Number Theory"
  - Washington's "Cyclotomic Fields"

  EXAMPLES::

    sage: ## Testing the accuracy of the negative special values
    sage: RR = RealField(100)
    sage: for i in range(1,10):
    ...       print "zeta(" + str(1-2*i) + "): ", RR(zeta__exact(1-2*i)) - zeta(RR(1-2*i))
    zeta(-1):  0.00000000000000000000000000000
    zeta(-3):  0.00000000000000000000000000000
    zeta(-5):  0.00000000000000000000000000000
    zeta(-7):  0.00000000000000000000000000000
    zeta(-9):  0.00000000000000000000000000000
    zeta(-11):  0.00000000000000000000000000000
    zeta(-13):  0.00000000000000000000000000000
    zeta(-15):  0.00000000000000000000000000000
    zeta(-17):  0.00000000000000000000000000000

    sage: RR = RealField(100)
    sage: for i in range(1,10):
    ...       print "zeta(" + str(1-2*i) + "): ", RR(zeta__exact(1-2*i)) - zeta(RR(1-2*i))
    zeta(-1):  0.00000000000000000000000000000
    zeta(-3):  0.00000000000000000000000000000
    zeta(-5):  0.00000000000000000000000000000
    zeta(-7):  0.00000000000000000000000000000
    zeta(-9):  0.00000000000000000000000000000
    zeta(-11):  0.00000000000000000000000000000
    zeta(-13):  0.00000000000000000000000000000
    zeta(-15):  0.00000000000000000000000000000
    zeta(-17):  0.00000000000000000000000000000

  """
    if n<0:
        k = 1-n
        return -bernoulli(k)/k
    elif n>1:
        if (n % 2 == 0):
            return ZZ(-1)**(n/2 + 1) * ZZ(2)**(n-1) * pi**n * bernoulli(n) / fac
torial(n)
        else:
            raise TypeError, "n must be a critical value! (I.e. even > 0 or odd < 0.)"
    elif n==1:
        return infinity
    elif n==0:
        return -1/2

## ---------- Dirichlet L-functions with quadratic characters ----------
```

```
def QuadraticBernoulliNumber(k, d):
    r"""
  Compute k–th Bernoulli number for the primitive
  quadratic character associated to '\chi(x) = \left(\frac{d}{x}\right)'.

  Reference: Iwasawa's "Lectures on p-adic L-functions", pp7–16.

  EXAMPLES::

    sage: ## Makes a set of odd fund discriminants < –3
    sage: Fund_odd_test_set = [D  for D in range(–163, –3, 4)  if is_fundamental_discriminant(D)]

    sage: ## In general, we have B_{1, \chi_d} = –2h/w  for odd fund disc < 0
    sage: for D in Fund_odd_test_set:
    ...     if len(BinaryQF_reduced_representatives(D)) != –QuadraticBernoulliNumber(1, D):
    ...       print "Oops!  There is an error at D = ", D
  """
    ## Ensure the character is primitive
    d1 = fundamental_discriminant(d)
    f = abs(d1)

    ## Make the (usual) k–th Bernoulli polynomial
    x =  PolynomialRing(QQ, 'x').gen()
    bp = bernoulli_polynomial(x, k)

    ## Make the k–th quadratic Bernoulli number
    total = sum([kronecker_symbol(d1, i) * bp(i/f)  for i in range(f)])
    total *= (f ** (k–1))

    return total


def quadratic_L_function__exact(n, d):
    r"""
  Returns the exact value of a quadratic twist of the Riemann Zeta function
  by '\chi_d(x) = \left(\frac{d}{x}\right)'.

  References:

  – Iwasawa's "Lectures on p-adic L-functions", p16–17, "Special values of
      'L(1–n, \chi)' and 'L(n, \chi)'
    – Ireland and Rosen's "A Classical Introduction to Modern Number Theory"
  – Washington's "Cyclotomic Fields"

    EXAMPLES::

        sage: bool(quadratic_L_function__exact(1, –4) == pi/4)
        True

    """
  from sage.all import SR, sqrt
  if n<=0:
    k = 1–n
    return –QuadraticBernoulliNumber(k,d)/k
  elif n>=1:
    ## Compute the kind of critical values (p10)
    if kronecker_symbol(fundamental_discriminant(d), –1) == 1:
      delta = 0
    else:
      delta = 1

    ## Compute the positive special values (p17)
    if ((n – delta) % 2 == 0):
      f = abs(fundamental_discriminant(d))
      if delta == 0:
        GS = sqrt(f)
      else:
        GS = I * sqrt(f)
      ans = SR(ZZ(–1)**(1+(n–delta)/2))
```



```
      ans *= (2*pi/f)**n
      ans *= GS     ## Evaluate the Gauss sum here! =0
      ans *= 1/(2 * I**delta)
      ans *= QuadraticBernoulliNumber(n,d)/factorial(n)
      return ans
    else:
      if delta == 0:
        raise TypeError, "n must be a critical value!\n" + "(I.e. even > 0 or odd < 0.)"
      if delta == 1:
        raise TypeError, "n must be a critical value!\n" + "(I.e. odd > 0 or even <= 0.)
"



def quadratic_L_function__numerical(n, d, num_terms=1000):
  """
    Evaluate the Dirichlet L-function (for quadratic character) numerically
    (in a very naive way).

    EXAMPLES::

        sage:  ## Test several values for a given character
        sage: RR = RealField(100)
        sage: for i in range(5):
        ...       print "L(" + str(1+2*i) + ",(–4/.)): ", RR(quadratic_L_function__e
xact(1+2*i, –4)) – quadratic_L_function__numerical(RR(1+2*i),–4, 10000)
        L(1, (–4/.)):  0.00004999999950000024999996962707
        L(3, (–4/.)):  4.99999970000003...e–13
        L(5, (–4/.)):  4.99999922759382...e–21
        L(7, (–4/.)):  ...e–29
        L(9, (–4/.)):  ...e–29

        sage: ## Testing the accuracy of the negative special values
        sage: ## ---- THIS FAILS SINCE THE DIRICHLET SERIES DOESN'T CONVERGE HER
E! ----

        sage: ## Test several characters agree with the exact value, to a given
accuracy.
        sage: for d in range(–20,0):
        ...       if abs(RR(quadratic_L_function__numerical(1, d, 10000) – quadr
atic_L_function__exact(1, d))) > 0.001:
        ...           print "Oops! We have a problem at d = ", d, "  exact = ", RR(quadrati
c_L_function__exact(1, d)), "  numerical = ", RR(quadratic_L_function__numerical(1,
 d))
        ...

    """
  ## Set the correct precision if it's given (for n).
  if is_RealField(n.parent()):
    R = n.parent()
  else:
    R = RealField()

  d1 = fundamental_discriminant(d)
  ans = R(0)
  for i in range(1,num_terms):
    ans += R(kronecker_symbol(d1,i) / R(i)**n)
  return ans
```

```
from sage.rings.arith import is_square, is_prime, valuation, legendre_symbol, hi
lbert_symbol
from sage.rings.infinity import Infinity
from sage.rings.padics.factory import Qp


from sage.rings.rational_field import is_RationalField, QQ
from sage.rings.real_mpfr import is_RealField, RealField

from sage.rings.all import is_ComplexField, is_pAdicField, is_FiniteField


from sage.misc.functional import squarefree_part
from sage.quadratic_forms.extras import least_quadratic_nonresidue

from sage.functions.generalized import sgn




def local_squareclass_representatives_list(v):
    """
    Returns a list of representatives (in ZZ) for the non-zero
    squareclasses in the local field Q_v, where v is either a prime
    number or Infinity.

    INPUT:
        v -- a prime number or Infinity

    OUTPUT:
        a list of integers

    EXAMPLES:
        sage: local_squareclass_representatives_list(Infinity)
        [1, -1]
        sage: local_squareclass_representatives_list(2)
        [1, 3, 5, 7, 2, 6, 10, 14]
        sage: local_squareclass_representatives_list(5)
        [1, 2, 5, 10]

    """
    ## Sanity Check: v is a prime or Infinity
    if not ((v == Infinity) or is_prime(v)):
        raise TypeError, "You must pass in either a prime number or Infinity."

    ## Return the list of squareclass representatives
    if v == Infinity:
        return [1, -1]
    elif v == 2:
        return [1, 3, 5, 7, 2, 6, 10, 14]
    else:
        nr = least_quadratic_nonresidue(v)
        return [1, nr, v, v*nr]



def local_squareclass_radius_val(v):
    """

    Returns the valuation for the p-adic/real radius of a (non-zero)
    squareclass in the local field Qv.  When v = Infinity, by
    convention we return 1 (for the sign determining the squareclass).
    When p=2 this returns 3 (since we need to look mod 8 to determine
    the squareclass), and for other primes it is 1 (since the
    squareclass is determined mod p).
```



```
    INPUT:
        v -- a prime number or Infinity

    OUTPUT:
        1 or 3

    EXAMPLES:
        sage: from sage.quadratic_forms.square_classes import local_squareclass_radius_val
        sage: local_squareclass_radius_val(Infinity)
        1
        sage: local_squareclass_radius_val(2)
        3
        sage: local_squareclass_radius_val(3)
        1
        sage: local_squareclass_radius_val(5)
        1

    """
    ## TO DO: Validate the input

    ## Return the valuation of the modulus needed to define a squareclass over Q
v
    if v == 2:
        return 3
    else:
        return 1



def is_SquareClass(x):
    """
    Decides if x is a squareclass (i.e. is an instance of the SquareClass class).

    INPUT:
        None

    OUTPUT:
        boolean

    EXAMPLES:
        sage: from sage.quadratic_forms.square_classes import is_SquareClass, SquareClass

        sage: S = SquareClass(QQ, 17)
        sage: is_SquareClass(S)
        True

        sage: is_SquareClass(3)
        False

    """
    return isinstance(x, SquareClass)



################################################################################
## Defines squareclasses for QQ, Q_p, RR, or F_q.
################################################################################
class SquareClass:
    """
    Defines a squareclass over QQ, Q_p, RR, or finite fields F_q.  Later
    this will also include number fields or one of its localizations
    or residue fields.

    """
```

```
    def __init__(self, F, x, normalize_element=False):
        """
    Creates the squareclass over the field F represented by all
    non-zero square multiples of x of F of nonzero_elt (which could
    be a number coercible to F or a squareclass defined by such a
    number).


    INTERNAL VARIABLES:
       self._base_field -- the field defining the squareclass
       self._representative_elt -- the element that is passed to create
          the non-zero squareclass
       self._normalized_flag -- the flag which determines
          whether the current representative is normalized.

    EXAMPLES:
       sage: S1 = SquareClass(QQ, 1); S1
       The squareclass represented by 1 over Rational Field
       sage: S3 = SquareClass(QQ, 3); S3
       The squareclass represented by 3 over Rational Field

       sage: S3*S3
       The squareclass represented by 9 over Rational Field
       sage: S1 == S3*S3
       True

       sage: SquareClass(QQ, S3)
       The squareclass represented by 3 over Rational Field
       sage: SquareClass(Qp(5), S3)
       The squareclass represented by 3 + O(5^20) over 5-adic Field with capped relative precision 20
    """



        ## Allow a square_class to be passed in
        if is_SquareClass(x):
            self.__init__(F, x.representative(), normalize_element)


        ## Deal with a representative being passed in
        else:

            ## Check that F is of the allowed type (QQ, Q_p, RR, or F_q)
            if not (is_RationalField(F) or is_RealField(F) or is_pAdicField(F) o
r is_FiniteField(F) or is_ComplexField(F)):
                raise TypeError, "The field F must be QQ, Q_p, RR, F_q or CC."

            ## Check that the element can be coerced into the field F.
            try:
                F_elt = F(x)
            except:
                raise TypeError, "The element cannot be coerced into the defining field F."

            ## Check that our local p-adic field has enough precision to determi
ne a squareclass
            if (is_pAdicField(F) and (F.prime() == 2) and (F_elt.precision_relat
ive() < 3)):
                raise RuntimeError, "The 2-adic relative precision (" + str(F_elt.precisio
n_relative()) + ") of the number " + str(F_elt) + " is not sufficient (i.e. < 3) to determine the squarecl
ass!"



            ## Store the non-zero element defining the squareclass (which may or
 may not be in F, but is coercible to F).
            self._base_field = F
            self._representative_elt = F_elt
            self._normalized_flag = False
```



```
            ## Normalize the squareclass, if desired.
            if normalize_element:
                self._representative_elt = self.normalized_representative()
                self._normalized_flag = True




    def __cmp__(self, other):
        """
    This catches unimplemented comparison methods and raise an
     error message.  (For us the unimplemented comparison mentods
     should be <, <=, >, >=, which don't make sense in general for
     squareclasses).  The methods for == and != are handled in
     __eq__() and __ne__() separately.

    INPUT:
       other -- a squareclass.

    OUTPUT:
       error message -- since we should never be comparing squareclasses using this method.

    EXAMPLES:
       sage: S1 = SquareClass(QQ, 1)
       sage: S1.__cmp__(S1)
       Traceback (most recent call last):
       ...
       NotImplementedError: The comparison operation you tried isn't implemented for squareclasses.  Try using ==
or != instead.

    """
        raise NotImplementedError, "The comparison operation you tried isn't implemented for square
classes.  Try using == or != instead."


    def __ne__(self, other):
        """
    Tests if the two square classes are not equivalent.  See
    __eq__() for more details.

    INPUT:
       other -- a squareclass

    OUTPUT:
       boolean

    EXAMPLES:
       sage: S1 = SquareClass(QQ, 1)
       sage: S2 = SquareClass(QQ, 2)
       sage: S4 = SquareClass(QQ, 4)

       sage: S1 != S2
       True

       sage: S1 != S2
       True

       sage: S1 != S4
       False

    """
        return not self.__eq__(other)


    def __eq__(self, other):
        """
    Tests if the two square classes are equivalent (meaning they
```

are equal as sets, so we don't care about the choice of representative, only that the underlying sets are equal!).

This tests equality of square−classes by first testing that the base fields are equal (meaning that they are exactly the same model of the field, including precision), and then checking that they give the same squareclass by the following priority scheme:

1) Check if their normalized representatives are the same (if this makes sense)

2) Check if the ratio of their representatives (which are always given in the base field) is a square.

INPUT:
other −− a squareclass

OUTPUT:
boolean

EXAMPLES:
sage: S1 = SquareClass(QQ, 1)
sage: S2 = SquareClass(QQ, 2)
sage: S4 = SquareClass(QQ, 4)

sage: S1 == S2
False

sage: S1 == S2
False

sage: S1 == S4
True

sage: S1 = SquareClass(Qp(3), 1)
sage: S1 = SquareClass(Qp(3), 2)

sage: S1 == S1
True

sage: S2 == S2
True

sage: S1 == S2
False

sage: S2 == S1
False

```
    """
        ## Compare base fields
        if self.base_field() != other.base_field():  ## TO CHECK: What about rea
l fields of different precision, or different p-dic models?
            return False

        ## Try to compare the normalized representatives
        if self.is_normalized() and other.is_normalized():
            return self.representative() == other.representative()

        ## Otherwise check the ratios of their representatives (in a field-speci
fic way).
        else:
            ## Deal with the zero squareclasses (on either side)
            if other.is_zero():
                if self.is_zero():
                    return True
```



```
                else:
                    return False

            ## Deal with unit squareclasses (here "other" is known to be a unit)
            return self.is_square_element(self.representative() / other.represen
tative())
```

```
    def __mul__(self, other, normalize_elt=False):
        """
```

Returns the product of two squareclasses, or the product of a squareclass and a number coercible to the basefield of the squareclass.

INPUT:
other −− a squareclass

OUTPUT:
a squareclass over the same basefield

EXAMPLES:
sage: A = SquareClass(QQ, 3)
sage: B = SquareClass(QQ, 4)
sage: A*B
The squareclass represented by 12 over Rational Field

```
    """
        ## Check if other is a squareclass (or a number)
        if not is_SquareClass(other):

            ## Try to coerce the element other into the basefield, and multiply
them.
            try:
                F = self.base_field()
                b = self.base_field()(other)
                return SquareClass(self.base_field(), self.representative() * b,
 normalize_elt)
            except:
                raise TypeError, "Oops!  The second object is neither a squareclass nor is it coercible i
nto the basefield of the first squareclass!  It's a " + str(type(other)) + "."

        ## Check that both squareclasses have the same base fields
        if self.base_field() != other.base_field():
            raise TypeError, "Oops!  These two squareclasses don't have the same base field!"

        ## Return their product squareclass
        return SquareClass(self.base_field(), self.representative() * other.repr
esentative(), normalize_elt)
```

```
    def __div__(self, other, normalize_elt=False):
        """
```

Returns the quotient of two squareclasses, or the quotient of a squareclass by a number coercible into the basefield of the squareclass.

INPUT:
other −− a non−zero squareclass or non−zero number.

OUTPUT:
a squareclass over the same basefield

EXAMPLES:
sage: A = SquareClass(QQ, 3)
sage: B = SquareClass(QQ, 4)

```
    sage: A / B
    The squareclass represented by 3/4 over Rational Field

    """
        ## Check if other is a squareclass (or a number)
        if not is_SquareClass(other):

            ## Try to coerce the element other into the basefield, and divide th
e representatives.
            try:
                F = self.base_field()
                b = self.base_field()(other)

                ## Check that the second squareclass isn't zero
                if b == 0:
                    raise RuntimeError, "Oops!  The squareclass we're dividing by must be non-ze
ro!"

                ## Return the new quotient squareclass
                return SquareClass(self.base_field(), self.representative() / b,
 normalize_elt)
            except:
                raise TypeError, "Oops!  The second object is neither a squareclass nor is it coercible i
nto the basefield of the first squareclass!  It's a " + str(type(other)) + "."

        ## Check that both squareclasses have the same base fields
        if self.base_field() != other.base_field():
            raise TypeError, "Oops!  These two squareclasses don't have the same base field!"

        ## Check that the second squareclass isn't zero
        if self.is_zero():
            raise RuntimeError, "Oops!  The squareclass we're dividing by must be non-zero!"

        ## Return their quotient squareclass
        return SquareClass(self.base_field(), self.representative() / other.repr
esentative(), normalize_elt)




    def __repr__(self):
        """
    Returns a string describing the squareclass.

    INPUT:
      None

    OUTPUT:
      string

    EXAMPLES:
      sage: S1 = SquareClass(QQ, 1)
      sage: S1.__repr__()
      'The squareclass represented by 1 over Rational Field'

    """
        return "The squareclass represented by " + str(self.representative()) + " over " + s
tr(self.base_field())


    def base_field(self):
        """
    Returns the basefield of the squareclass.

    INPUT:
      None

    OUTPUT:
```



```
      a field

    EXAMPLES:
      sage: S1 = SquareClass(QQ, 1)
      sage: S1.base_field()
      Rational Field

    """
        return self._base_field



    def representative(self):
        """
    Returns a representative for the given squareclass.

    INPUT:
      None

    OUTPUT:
      an element of the basefield.

    EXAMPLES:
      sage: S1 = SquareClass(QQ, 1)
      sage: S1.representative()
      1

    """
        return self._representative_elt



    def is_normalized(self):
        """
    Determines if the current squareclass representative has been
    certified to be a normalized representative.

    INPUT:
      None

    OUTPUT:
      boolean

    EXAMPLES:
      sage: S1 = SquareClass(QQ, 1)
      sage: S1.representative()
      1
      sage: S1.is_normalized()
      False

      sage: S1 = SquareClass(QQ, 1, True)
      sage: S1.representative()
      1
      sage: S1.is_normalized()
      True
    """
        return self._normalized_flag


    def normalized_squareclass(self):
        """
    Returns a normalized squareclass equivalent to the current
    one, assuming we have an explicit normalization algorithm
    defined in self.normalized_representative().

    INPUT:
      None

    OUTPUT:
```

```
        a squareclass over the same base field which has a representative (certified as a normalized representative).

    EXAMPLES:
        sage: S1 = SquareClass(QQ, 1)
        sage: S1.representative()
        1
        sage: S1.is_normalized()
        False
        sage: S1_norm = S1.normalized_squareclass()
        sage: S1_norm.representative()
        1
        sage: S1_norm.is_normalized()
        True

        sage: S4 = SquareClass(QQ, 4)
        sage: S4.representative()
        4
        sage: S4.is_normalized()
        False
        sage: S4_norm = S1.normalized_squareclass()
        sage: S4_norm.representative()
        1
        sage: S4_norm.is_normalized()
        True

    """
        return SquareClass(self.base_field(), self.representative(), normalize_e
lement=True)



    def normalized_representative(self):
        """
    Returns a normalized representative for the (non-zero)
    squareclass over the special fields CC, RR, QQ, Q_p and F_p
    when p is prime.

    INPUT:
        None

    OUTPUT:
        a number in the basefield.

    EXAMPLES:
        sage: S4 = SquareClass(QQ, 4)
        sage: S4.representative()
        4
        sage: S4.is_normalized()
        False
        sage: S4.normalized_representative()
        1

        sage: S = SquareClass(RR, -22/3)
        sage: S.normalized_representative()
        -1

        sage: S = SquareClass(CC, -22/3)
        sage: S.normalized_representative()
        1

        sage: S = SquareClass(FiniteField(5), -1)
        sage: S.normalized_representative()
        1

        sage: S = SquareClass(FiniteField(7), -1)
        sage: S.normalized_representative()
        3

        #sage: F49 = FiniteField(49, x)
```



```
        #sage: S = SquareClass(F49, -1)
        #sage: S.normalized_representative()
        #3

        sage: S = SquareClass(Qp(5), 5/3)
        sage: S.normalized_representative()
        10

    """
        ## Set some convenient local variables
        F = self.base_field()
        elt = self.representative()

        ## Return a normalized element if it exists
        if self.is_normalized():
            return elt

        ## Deal with the zero squareclass
        if self.is_zero():
            return elt


        ## Compute normalized representatives for each of the supported fields
        if is_ComplexField(F):
            return 1
        elif is_RealField(F):
            return sgn(elt)    ## Use the sign 1 or -1 for RR
        elif is_RationalField(F):
            return squarefree_part(elt)    ## Use a squarefree integer for QQ
        elif is_pAdicField(F):
            p = F.prime()
            if not is_prime(p):
                raise TypeError, "Only normaliztions af element for Q_p where p is prime is currently
supported."

            ## Separate out the p-part and unit parts
            v , unit_part = elt.val_unit()
            new_p_part = p**(v % 2)


            ## Normalize the unit part
            if p != 2:
                if legendre_symbol(unit_part.lift() % p, p) == 1:
    ## TO FIX: This is .lift() because of the bug in Ticket #7016
                    new_unit_part = 1
                else:
                    new_unit_part = least_quadratic_nonresidue(p)
            else:
                new_unit_part = unit_part.lift() % 8
    ## TO FIX: This is .lift() because of the bug in Ticket #7016

            ## Set the normalized representative
            return new_p_part * new_unit_part


        elif is_FiniteField(F):
            p = F.order()
            if not is_prime(p):
                raise TypeError, "Normalized elements are only supported for finite fields with prime
numbers of elements."

            ## Set the normalized elements over finite fields.  (This uses the a
ssumption that the squareclass is non-zero)
            if F(elt).is_square():
                return 1
            else:
                return least_quadratic_nonresidue(p)
```

```
        else:
            raise TypeError, "Normalized squareclass representatives for the base field " + str(F)
+ " isn't currently supported."



    def localize_at_place(self, v):
        """
    Return the localization of the current squareclass (which must
    be defined over QQ) at the place v.

    INPUT:
      v -- either a prime number or Infinity

    OUTPUT:
      a squareclass over the (p-adic or real local field) QQ_v

    EXAMPLES:
      sage: S = SquareClass(QQ, -22)
      sage: S.base_field()
      Rational Field

      sage: S0 = S.localize_at_place(Infinity)
      sage: S0.base_field()
      Real Field with 53 bits of precision
      sage: S0
      The squareclass represented by -22.0000000000000 over Real Field with 53 bits of precision

      sage: S11 = S.localize_at_place(11)
      sage: S11.base_field()
      11-adic Field with capped relative precision 20
      sage: S11
      The squareclass represented by 9*11 + 10*11^2 + 10*11^3 + 10*11^4 + 10*11^5 + 10*11^6 + 10*11^7 + 10*1
1^8 + 10*11^9 + 10*11^10 + 10*11^11 + 10*11^12 + 10*11^13 + 10*11^14 + 10*11^15 + 10*11^16 + 10*11^17 + 1
0*11^18 + 10*11^19 + 10*11^20 + O(11^21) over 11-adic Field with capped relative precision 20

    """
        ## Check that we're over QQ
        if self.base_field() != QQ:
            raise TypeError, "This method only applies to square classes over global fields, and only QQ
 for now."

        ## Construct the local field from the place
        if v == Infinity:
            F = RealField()
        elif is_prime(v):
            F = Qp(v)
        else:
            raise RuntimeError, "The place " + str(v) + " you passed is not recognized."

        ## Return the localized quadratic space
        return SquareClass(F, self)



    def valuation(self, v=None):
        """
    Returns the (parity of the) valuation of the squareclass
    relative to the valuation structure on the base field (which
    is either the integer 0 or 1).  Over the real/complex numbers
    or over a finite field we return 0 always.

    If the squareclass is defined over QQ, then we determine the
    valuation at the localization of the squareclass at the given
    place v.

    INPUT:
      None
```



```
    OUTPUT:
      an integer

    EXAMPLES:
      sage: S1 = SquareClass(QQ, -125/3)
      sage: S1.localize_at_place(Infinity).valuation()
      0
      sage: S1.localize_at_place(2).valuation()
      0
      sage: S1.localize_at_place(3).valuation()
      1
      sage: S1.localize_at_place(5).valuation()
      1
      sage: S1.localize_at_place(7).valuation()
      0


      sage: S1 = SquareClass(QQ, -125/3)
      sage: S1.valuation(Infinity)
      0
      sage: S1.valuation(2)
      0
      sage: S1.valuation(3)
      1
      sage: S1.valuation(5)
      1
      sage: S1.valuation(7)
      0

    """
        ## Set some convenient local variables
        F = self.base_field()

        ## Return the valuation for a local or finite field
        if is_pAdicField(F):
            return self.representative().valuation() % 2
        elif is_ComplexField(F) or is_RealField(F) or is_FiniteField(F):
            return 0

        ## Return the valuation at a place of QQ
        elif is_RationalField(F):
            if (v == None) or ((v != Infinity) and not is_prime(v)):
                raise TypeError, "The squareclasses over the rational numbers do not have a unique na
tural valuation structure!"
            else:
                if v == Infinity:
                    return 0
                else:
                    return self.representative().valuation(v) % 2

        ## Raise an error otherwise
        else:
            raise NotImplementedError, "There is no supported valuation structure for squareclasses
over the field " + str(F) + "."



    def valuation_free_part(self):
        """
    Return the squareclass given by removing the prime
    contributing to the valuation.  This only makes sense
    for a squareclass defined over a p-adic field.

    INPUT:
      None

    OUTPUT:
```

```
    a squareclass

EXAMPLES:
    sage: S1 = SquareClass(QQ, -125/3)
    sage: S1_3 = S1.localize_at_place(3)
    sage: S1_3.valuation()
    1
    sage: S1_3.valuation_free_part().valuation()
    0

"""
        if self.valuation() == 0:
            return self
        else:
            return self * self.base_field().prime()




    def is_zero(self):
        """
Determines if this squareclass is the zero squareclass.

INPUT:
    None

OUTPUT:
    boolean

EXAMPLES:
    sage: S = SquareClass(QQ, -125/3)
    sage: S.is_zero()
    False

    sage: S0 = SquareClass(QQ, 0)
    sage: S0.is_zero()
    True

    sage: (S * S0).is_zero()
    True

"""
        return self.representative() == self.base_field()(0)


    def is_nonzero(self):
        """
Determines if this squareclass is not the zero squareclass.

INPUT:
    None

OUTPUT:
    boolean

EXAMPLES:
    sage: S = SquareClass(QQ, -125/3)
    sage: S.is_nonzero()
    True

    sage: S0 = SquareClass(QQ, 0)
    sage: S0.is_nonzero()
    False

    sage: (S * S0).is_nonzero()
    False

"""
```



```
        return not self.is_zero()


    def is_unit_squares(self):
        """
    THIS SHOULD BE RENAMED is_squareclass_of_nonzero_squares()!

    Determines if this squareclass is the unit squareclass, which
    is the squareclass of all unit squares.  This uses the method
    self.is_square_element() to decide if the representative is a
    square.

    INPUT:
        None

    OUTPUT:
        boolean

    EXAMPLES:
        sage: S = SquareClass(QQ, -125/3)
        sage: S.is_unit_squares()
        False

        sage: S1 = SquareClass(QQ, 1)
        sage: S1.is_unit_squares()
        True

        sage: (S * S1).is_unit_squares()
        False

        sage: (S * S).is_unit_squares()
        True

    """
        return self.is_nonzero() and self.is_square_element(self.representative(
))



    def is_square_element(self, x):
        """
    Decide if the element x is a square in the base field of this
    squareclass.  Currently only the fields CC, RR, QQ, F_q, Q_p
    are supported.

    INPUT:
        x -- a number in the basefield.

    OUTPUT:
        boolean

    EXAMPLES:
        sage: S = SquareClass(QQ, -125/3)
        sage: S.is_square_element(-125/3)
        False
        sage: S.is_square_element(1)
        True
        sage: S.is_square_element(4)
        True
        sage: S.is_square_element(0)
        True
        sage: S.is_square_element(-1)
        False

        sage: S = SquareClass(Qp(5), 2)
        sage: S.is_square_element(1)
        True
        sage: S.is_square_element(4)
        True
```

```
        sage: S.is_square_element(0)
        True
        sage: S.is_square_element(2)
        False
        sage: S.is_square_element(6)
        True


    """
        F = self.base_field()

        ## Deal with zero
        if F(x) == 0:
            return True

        ## Try to use the base field is_square method (if it exists)
        try:
            return F.is_square(x)
        except:
            pass

        ## Deal with elements in a field-specific way:
        ## ------------------------------------------
        if is_ComplexField(F):
            return True           ## All elements are squares in CC

        elif is_RealField(F):
            return x>0            ## All positive elements are squares in RR

        elif is_RationalField(F):
            return is_square(x)   ## Use the is_square() method in sage/rings/ar
ith.py

        elif is_FiniteField(F):
            if F.characteristic() == 2:
                return True                  ## All elements are squares in char
acteristic 2
            else:
                return x**((F.order() - 1) / 2) == 1     ## Use the Legendre sym
bol test otherwise

        elif is_pAdicField(F):
            p = F.prime()
            if not is_prime(p):
                raise TypeError, "Only square-testing element for Q_p where p is prime is currently s
upported."

            ## Separate out the p-part and unit parts
            v, unit_part = F(x).val_unit()
            #v = F(x).valuation()
            #new_p_part = QQ(p**(v % 2))
            #unit_part = x / (p**v)

            ## Deal with elements of odd valuation
            if v % 2 != 0:
                return False

            ## Check if the unit part is a square
            if p == 2:
                return (unit_part.lift() % 8) == 1                      ## BUG -
- Ticket 7016: We should be able to just say "unit_part % 8" here!
            else:
                return legendre_symbol(unit_part, p) == 1

        else:
            raise NotImplementedError, "We haven't implemented square-testing for the field " +
str(F) + "."
```



```
    def hilbert_symbol(self, other, QQ_place=None):
        """
Computes the Hilbert symbol for the squareclasses self and
other, which is defined by whether the number 1 is in the sum
of the two squareclasses.

If one base_field is QQ and the other is a local field, then
the Hilbert symbol is evaluated over the local field.  If both
fields are local, then they mest be the same.  If both are QQ,
then the local field is specified by QQ_place, which is either
a prime number or Infinity.

INPUT:
    other -- a squareclass
    QQ_place -- a prime number or Infinity (required only if the base_field of both squareclasses is global)

OUTPUT:
    1 or -1

EXAMPLES:
    sage: S1 = SquareClass(QQ, 1)
    sage: N1 = SquareClass(QQ, -1)
    sage: S2 = SquareClass(QQ, 2)
    sage: S3 = SquareClass(QQ, 3)
    sage: S5 = SquareClass(QQ, 5)

    sage: S3.hilbert_symbol(S3, 3)
    -1
    sage: S3.hilbert_symbol(S3.localize_at_place(3))
    -1
    sage: S3.localize_at_place(3).hilbert_symbol(S3)
    -1
    sage: S3.localize_at_place(3).hilbert_symbol(S3.localize_at_place(3))
    -1

    sage: N1.hilbert_symbol(N1, Infinity)
    -1
    sage: N1.hilbert_symbol(N1, 2)
    -1
    sage: N1.hilbert_symbol(N1, 3)
    1
    sage: N1.hilbert_symbol(N1, 5)
    1


    sage: N1.hilbert_symbol(N1, 2)
    -1

    sage: S3.hilbert_symbol(S5, Infinity)
    1

    sage: S3.hilbert_symbol(S5, 3)
    -1

    sage: R1 = SquareClass(RR, 1)
    sage: R2 = SquareClass(RR, -1)
    sage: R1.hilbert_symbol(R1)
    1
    sage: R1.hilbert_symbol(R2)
    1
    sage: R2.hilbert_symbol(R1)
    1
    sage: R2.hilbert_symbol(R2)
    -1

    """
        ## Check that other is a squareclass
```

```
        ## Check that neither squareclass is zero

        ## Check that only allowed basefields have been passed


        ## Case 1: Both basefields are QQ
        if (self.base_field() == QQ) and (other.base_field() == QQ):

            ## Case 1a: Compute the Hilbert symbol over QQ by the Strong Hasse P
rinciple
            if QQ_place == None:
                raise NotImplementedError, "The computation of Hilbert symbols over QQ is not c
urrently implemented!"

            ## Case 1b: Compute the Hilbert symbol over some localization
            elif QQ_place == Infinity:
                if (self.representative() < 0) and (other.representative() < 0):
                    return -1
                else:
                    return 1
            elif is_prime(QQ_place):
                return hilbert_symbol(self.representative(), other.representativ
e(), QQ_place)

        ## Case 2: Exactly one of the basefields is QQ
        if (self.base_field() == QQ) or (other.base_field() == QQ):

            ## Label the squareclasses so the first one S1 is over QQ
            if (self.base_field() == QQ):
                S1 = self    ## over QQ
                S2 = other
            else:
                S1 = other   ## over QQ
                S2 = self

            ## Compute the appropriate Hilbert symbol
            if is_RealField(S2.base_field()):
                return (S1.representative() < 0) and (S2.representative() < 0)
            else:
                try:
                    p = S2.base_field().prime()
                    S2rep = S2.representative()

                    ## Check that p is prime
                    if not is_prime(p):
                        raise RuntimeError, "The p-adic field must be Q_p for some prime p."

                    ## Compute the symbol
                    return hilbert_symbol(S1.representative(), S2.representative
().lift(), p)
                except:
                    raise RuntimeError, "There was a problem computing the Hilbert symbol..."

        ## Case 3: Both fields are (the same) localfields
        if (self.base_field() != other.base_field()):
            raise TypeError, "The local fields must be the same to compute the Hilbert symbol."

        if is_ComplexField(self.base_field()):
            return 1
        elif is_RealField(self.base_field()):
                if (self.representative() < 0) and (other.representative() < 0):
                    return -1
                else:
                    return 1
        else:
            p = self.base_field().prime()

            ## Check that p is prime
            if not is_prime(p):
```



```
                raise RuntimeError, "The p-adic field must be Q_p for some prime p."

            return hilbert_symbol(self.representative().lift(), other.representa
tive().lift(), p)



        ## Raise an error if we're here
        raise RuntimeError, "Something is wrong..."
```

```
## Required structures:
## --------------------

#from sage.quadratic_forms.symmetric_bilinear_lattice import SymmetricBilinearLa
ttice


from sage.quadratic_forms.square_classes import SquareClass, local_squareclass_r
epresentatives_list, local_squareclass_radius_val
from sage.quadratic_forms.weak_approx import weak_approx_for_numbers_over_QQ, \
                                             weak_approx_for_squareclasses_over_
QQ, \
                                             strong_approx_for_squareclasses_by_
QQ_except_at_one_prime

from sage.quadratic_forms.localization import Qv


#from sage.quadratic_forms.quadratic_space import QuadraticSpace



from sage.rings.arith import hilbert_symbol, legendre_symbol, valuation, is_squa
re, is_prime, prime_divisors, is_squarefree, GCD
from sage.rings.integer_ring import ZZ
from sage.rings.finite_rings.constructor import GF


from sage.functions.other import floor, sqrt

from sage.misc.functional import squarefree_part, is_even, is_odd
from sage.misc.misc import prod, verbose
from sage.misc.mrange import mrange


from sage.structure.element import is_Vector


from sage.quadratic_forms.quadratic_form import QuadraticForm, DiagonalQuadratic
Form
from sage.quadratic_forms.extras import least_quadratic_nonresidue
from sage.functions.generalized import sgn
from sage.matrix.all import is_Matrix, MatrixSpace
from sage.matrix.constructor import matrix, Matrix
#from sage.matrix.matrix import Matrix, is_Matrix

from sage.rings.field import Field


from sage.rings.rational_field import is_RationalField, QQ, RationalField
from sage.rings.real_mpfr import RealField
from sage.rings.all import is_ComplexField, is_pAdicField, is_FiniteField



from sage.rings.infinity import Infinity
from sage.rings.padics.factory import Qp

from sage.rings.polynomial.polynomial_ring_constructor import PolynomialRing
from sage.modules.free_module_element import vector
from sage.modules.free_module import VectorSpace
from sage.modules.all import is_VectorSpace

from copy import deepcopy
```



```
#from sage.rings.arith import is_square, is_prime, valuation, legendre_symbol



def DiagonalMatrix(base_ring, diag_list):
    """
  Create a diagonal matrix from a list of entries.

  INPUT:
    a list of elements coercible to base_ring

  OUTPUT:
    a matrix defined over base_ring

  EXAMPLES:
    sage: DiagonalMatrix(QQ, [1,2,3])
    [1 0 0]
    [0 2 0]
    [0 0 3]
    sage: DiagonalMatrix(QQ, [])
    []
  """
    n = len(diag_list)
    B = Matrix(base_ring, n, n)
    for i in range(n):
        B[i,i] = diag_list[i]
    return B




############################################################
## Create a SymmetricBilinearSpace class which defines a  ##
## symmetric bilinear space over a local or global field. ##
############################################################

#class SymmetricBilinearSpace(FreeQuadraticModule):
class SymmetricBilinearSpace():
    """
  Defines a symmetric bilinear space, by which we mean a diagonal quadratic form over a field.
  """


    def __init__(self, K, coeffs=None):
        """
      Initializes a symmetric bilinear space over a given field from:
        1) a symmetric matrix
        2) a list of diagonal entries
        3) a vectorspace
        4) a symmetric bilinear form

      If the coefficients are not elements of the base field, then
      they must automatically coerce into it or a RuntimeError will
      be raised.

      Valid syntax possibilities:
        SymmetricBilinearSpace(B) -- where B is a symmetric matrix defined over a field
        SymmetricBilinearSpace(K, B) -- where B is a symmetric matrix with coefficients coercible to the field K
        SymmetricBilinearSpace(K, [a_1, a_2, ..., a_n]) -- where K is a field and the elements a_1, ..., a_n are coercibl
e to elements of K
        SymmetricBilinearSpace(V) -- where V is a vectorspace (that has its own internal inner product)
        SymmetricBilinearSpace(K, V) -- where V is a vectorspace and its inner product matrix is coercible to the fiel
d K.
        SymmetricBilinearSpace(SBS) -- where SBS is a symmetric bilinear space
        SymmetricBilinearSpace(K, SBS) -- where SBS is a symmetric bilinear space and its inner product matrix is c
oercible to the field K.
```

```
    INPUT:
      K -- a local or global field, or possibly a symmetric matrix or a symmetric bilinear form defined over a field.
      coeffs -- either a list of diagonal coefficients, a symmetric bilinear form or a symmetric matrix

    OUTPUT:
      none

    INTERNAL VARIABLES:
      self.__vector_space -- the underlying vector space, equipped with its own inner product.

    EXAMPLES:
      sage: SymmetricBilinearSpace(DiagonalMatrix(QQ, [1,2,3]))
      Symmetric bilinear space over Rational Field of dimension 3 defined by the Gram matrix
      [1 0 0]
      [0 2 0]
      [0 0 3]

      sage: SymmetricBilinearSpace(Qp(5), DiagonalMatrix(QQ, [1,2,3]))
      Symmetric bilinear space over 5-adic Field with capped relative precision 20 of dimension 3 defined by the Gram matrix
      [1 + O(5^20)         0         0]
      [        0 2 + O(5^20)         0]
      [        0         0 3 + O(5^20)]

      sage: SymmetricBilinearSpace(QQ, DiagonalMatrix(QQ, [1,2,3]))
      Symmetric bilinear space over Rational Field of dimension 3 defined by the Gram matrix
      [1 0 0]
      [0 2 0]
      [0 0 3]

      sage: SymmetricBilinearSpace(QQ, [1,3,5])
      Symmetric bilinear space over Rational Field of dimension 3 defined by the Gram matrix
      [1 0 0]
      [0 3 0]
      [0 0 5]

      sage: SymmetricBilinearSpace(VectorSpace(GF(3), 2))
      Symmetric bilinear space over Finite Field of size 3 of dimension 2 defined by the Gram matrix
      [1 0]
      [0 1]

      sage: SymmetricBilinearSpace(GF(9,'x'), VectorSpace(GF(3), 2))
      Symmetric bilinear space over Finite Field in x of size 3^2 of dimension 2 defined by the Gram matrix
      [1 0]
      [0 1]

      sage: SBS = SymmetricBilinearSpace(VectorSpace(QQ, 3))
      sage: SymmetricBilinearSpace(SBS)
      Symmetric bilinear space over Rational Field of dimension 3 defined by the Gram matrix
      [1 0 0]
      [0 1 0]
      [0 0 1]
      sage: SymmetricBilinearSpace(RR, SBS)
      Symmetric bilinear space over Real Field with 53 bits of precision of dimension 3 defined by the Gram matrix
      [ 1.00000000000000 0.000000000000000 0.000000000000000]
      [0.000000000000000  1.00000000000000 0.000000000000000]
      [0.000000000000000 0.000000000000000  1.00000000000000]

    """
        ## Validate the input:
        ## -------------------
        if not (isinstance(K, (Field, SymmetricBilinearSpace)) or is_Matrix(K) or is_VectorSpace(K)):
            raise TypeError, "The first argument must be either a field, a vectorspace, a symmetric matrix, or a symmetric bilinear space!"

        if (coeffs != None) and not (isinstance(coeffs, (list, SymmetricBilinearSpace)) or is_Matrix(coeffs) or is_VectorSpace(coeffs)):
            raise TypeError, "The second argument entry must be either a list of coefficients, a vectorspace, a symmetric matrix, or a symmetric bilinear space!"


        ## INPUT #1:  Check for the syntax SymmetricBilinearSpace(K, .) where K is a field.
        if isinstance(K, Field):

            ## Sanity Check: Verify the second argument has been set.
            if (coeffs == None):
                raise TypeError, "Invalid Syntax -- two arguments are needed when the first argument is a field."

            base_field = K
            init_object = coeffs

        ## INPUT #2  Check for the syntax SymmetricBilinearSpace(.).
        else:
            init_object = K
            try:
                base_field = init_object.base_field()
            except:
                base_field = init_object.base_ring()

                ## Sanity Check: Verify that the abjectpassed is defined over a field
                if not isinstance(base_field, Field):
                    raise TypeError, "The object given is not defined over a field!"



        ## Initialize the symmetric bilinear space:
        ## ----------------------------------------
#        FreeQuadraticModule.__init__(base_field, len(init_object), )



        ## Initialize from a VectorSpace
        if is_VectorSpace(init_object):
            self.__vector_space = init_object.base_extend(base_field)

        ## Initialize from a list of coefficients
        elif isinstance(init_object, list):
            self.__vector_space = VectorSpace(base_field, len(init_object), inner_product_matrix=DiagonalMatrix(base_field, init_object))

        ## Initialize from a Symmetric Matrix
        elif is_Matrix(init_object):
            if not init_object.is_symmetric():
                raise TypeError, "The given input matrix \n" + str(init_object) + "\nmust be symmetric."

            self.__vector_space = VectorSpace(base_field, init_object.nrows(), inner_product_matrix=init_object)

        ## Initialize from a Symmetric Bilinear Space
        elif isinstance(init_object, SymmetricBilinearSpace):
            self.__vector_space = VectorSpace(base_field, init_object.dim(), inner_product_matrix=init_object.gram_matrix())




    def __repr__(self):
        """
        Print a string describing the symmetric bilinear space.

        INPUT:
```

```
        None

    OUTPUT:
        a string

    EXAMPLES:
        sage: SBS = SymmetricBilinearSpace(QQ, [1,2,-1])
        sage: SBS.__repr__()
        'Symmetric bilinear space over Rational Field of dimension 3 defined by the Gram matrix\n[ 1  0  0]\n[ 0  2  0]\
n[ 0  0 -1]'

    """
        return "Symmetric bilinear space over " + str(self.base_field()) + " of dimension " +
 str(self.dim()) \
               + " defined by the Gram matrix\n" + str(self.__vector_space.inner_product_ma
trix())




    def base_field(self):
        """
    Returns the base field of scalars for the quadratic space as a vectorspace.

    INPUT:
        None

    OUTPUT:
        a field

    EXAMPLES:
        sage: SBS = SymmetricBilinearSpace(QQ, [1,2,-1])
        sage: SBS.base_field()
        Rational Field

        sage: SBS = SymmetricBilinearSpace(Qp(3), [1,3,5])
        sage: SBS.base_field()
        3-adic Field with capped relative precision 20

        sage: SBS = SymmetricBilinearSpace(RR, [])
        sage: SBS.base_field()
        Real Field with 53 bits of precision

        sage: SBS = SymmetricBilinearSpace(GF(5), [5])
        sage: SBS.base_field()
        Finite Field of size 5
        """
        return self.__vector_space.base_field()



    def vector_space(self):
        """
    Returns the defining vector space (with inner product).

    INPUT:
        None

    OUTPUT:
        a vector space

    EXAMPLES:
        sage: SBS = SymmetricBilinearSpace(GF(5), [5])
        sage: SBS.vector_space()
        Ambient quadratic space of dimension 1 over Finite Field of size 5
        Inner product matrix:
        [0]
```



```
        #sage: SBS = SymmetricBilinearSpace(RR, [])
        #sage: SBS.vector_space()                ## THIS CRASHES!?!  WHY???     DEBUG THIS!!!

        sage: SBS = SymmetricBilinearSpace(QQ, [])
        sage: SBS.vector_space()                 ## BUT THIS IS FINE. =)
        Ambient quadratic space of dimension 0 over Rational Field
        Inner product matrix:
        []

    """
        #V = deepcopy(self.__vector_space)        -- This fails due to a GF(2) ve
ctorspace bug!  -- See Trac #

        ## Here is our temporary fix:
        V = VectorSpace(self.__vector_space.base_field(), self.__vector_space.di
mension(), inner_product_matrix=self.__vector_space.inner_product_matrix())
        return V


    def dim(self):
        """
    Returns the dimension (as a vectorspace over the base field)
    of the quadratic space.

    INPUT:
        None

    OUTPUT:
        an integer >= 0

    EXAMPLES:
        sage: SBS = SymmetricBilinearSpace(QQ, [])
        sage: SBS.dim()
        0

        sage: SBS = SymmetricBilinearSpace(RR, [])
        sage: SBS.dim()
        0

        sage: SBS = SymmetricBilinearSpace(RR, range(6))
        sage: SBS.dim()
        6
    """
        return self.__vector_space.dimension()



    def det(self):
        """
    Returns the determinant of the Gram matrix of the defining
    quadratic form for the quadratic space (in the standard basis).

    INPUT:
        None

    OUTPUT:
        an element of the base field

    EXAMPLES:
        sage: SBS = SymmetricBilinearSpace(RR, range(6))
        sage: SBS.det()                          ## This Crashes -- immutable matrix error! =(  WIERD!

    """
        return self.__vector_space.inner_product_matrix().determinant()
```

```
    def determinant_squareclass(self):
        """
    Returns the squareclass of the Gram determinant of the given
    quadratic form.  This determinant is the product of the
    diagonal entries when the form is diagonal (which it can
    always be arranged to be).

    TO DO:  CHANGE THIS TO 'det_squareclass()', SO THE USER DOESN'T EXPECT A NUMBER!

    INPUT:
      None

    OUTPUT:
      an element of the base field

    EXAMPLES:
      sage: QF = DiagonalQuadraticForm(ZZ, [1, 1, 1])
      sage: QS = QuadraticSpace(QQ, QF)
      sage: QS.determinant()
      The squareclass represented by 1 over Rational Field

      sage: QS.det_Hessian()
      8

      sage: QS.det_Gram()
      1

    """
        return SquareClass(self.base_field(), self.det())




    def gram_matrix(self):
        """
    Returns the gram matrix (which defines all Gram inner
    products) for this quadratic space.

    INPUT:
      None

    OUTPUT:
      a symmetric matrix defined over the base_field

    EXAMPLES:
      sage: SBS = SymmetricBilinearSpace(RR, range(6))
      sage: SBS.gram_matrix()                    ## THIS GIVES ANOTHER WIERD DEEPCOPY ERROR!?! =(
      ...
      TypeError: __deepcopy__() takes no arguments (1 given)

      sage: SBS = SymmetricBilinearSpace(QQ, range(6))
      sage: SBS.gram_matrix()
      [0 0 0 0 0 0]
      [0 1 0 0 0 0]
      [0 0 2 0 0 0]
      [0 0 0 3 0 0]
      [0 0 0 0 4 0]
      [0 0 0 0 0 5]

    """
        ## This should work -- but it fails due to a deepcopy bug -- see Trac Ti
cket #10606!
        #return deepcopy(self.__vector_space.inner_product_matrix())

        ## This is our workaround for now
        return self.gram_matrix_for_vectors(self.basis())
```



```
    def gram_matrix_for_vectors(self, vec_list):
        """
    Returns the gram matrix for the given list of vectors in the
    symmetric bilinear space.

    INPUT:
      None

    OUTPUT:
      a symmetric matrix defined over the base_field

    EXAMPLES:
      sage: SBS = SymmetricBilinearSpace(RR, range(6))
      sage: SBS.gram_matrix()                    ## THIS GIVES ANOTHER WIERD DEEPCOPY ERROR!?! =(

    """
        n = len(vec_list)
        G = Matrix(self.base_field(), n, n)
        for i in range(n):
            for j in range(n):
                G[i,j] = self.inner_product(vec_list[i], vec_list[j])
        return G




    def scale_by_factor(self, c):
        """
    Returns the current symmetric bilinear space with its matrix scaled by the constant factor c.

    Note: Think about coercing c into a global element first.
    Also check the creation conventions to make sure we're
    compatibile with them.

    INPUT:
      c -- a number coercible to the base field

    OUTPUT:
      a symmetric bilinear space

    EXAMPLES:
      sage: SBS = SymmetricBilinearSpace(QQ, range(1,3)); SBS
      Symmetric bilinear space over Rational Field of dimension 2 defined by the Gram matrix
      [1 0]
      [0 2]
      sage: SBS.scale_by_factor(3)
      Symmetric bilinear space over Rational Field of dimension 2 defined by the Gram matrix
      [3 0]
      [0 6]

    """
        ## Validate the constant c -- TO DO!
        try:
            c1 = self.base_field()(c)
        except:
            raise RuntimeError, "The scaling factor " + str(c) + " is not coercible to the base fie
ld " + str(self.base_field()) + "."

        ## Return the scaled quadratic space
        return SymmetricBilinearSpace(c1 * self.gram_matrix())
```

```
    def localize_at_place(self, v):
        """
    Return the localization of the current symmetric bilinear space at the
    place v.

    INPUT:
      v -- a place of the basefield (currently only the base
      field QQ is supported, so in this case v is either a prime
      number or Infinity).

    OUTPUT:
      a symmetric bilinear space over the localization of the base field at v.

    EXAMPLES:
      sage: SBS = SymmetricBilinearSpace(QQ, [1, 2])
      sage: SBS.localize_at_place(Infinity)
      Symmetric bilinear space over Real Field with 53 bits of precision of dimension 2 defined by the Gram matrix
      [ 1.00000000000000 0.000000000000000]
      [0.000000000000000  2.00000000000000]

      sage: SBS.localize_at_place(7)
      Symmetric bilinear space over 7-adic Field with capped relative precision 20 of dimension 2 defined by the Gra
m matrix
      [1 + O(7^20)         0]
      [        0 2 + O(7^20)]
    """
        ## Check that we're over QQ
        if self.base_field() != QQ:
            raise TypeError, "This method only applies to quadratic forms over global fields, and only Q
Q for now."

        ## Construct the local field from the place
        if v == Infinity:
            F = RealField()
        elif is_prime(v):
            F = Qp(v)
        else:
            raise RuntimeError, "The place " + str(v) + " you passed is not recognized."

        ## Return the localized symmetric bilinear space
        return SymmetricBilinearSpace(F, self)



    def inner_product(self, x, y):
        """
    Compute the inner product of two vectors, considered as elements of the symmetric bilinear space.

    INPUT:
      two vectors coercible to elements of this vector space

    OUTPUT:
      an element of the base_field

    EXAMPLES:
      sage: SBS = SymmetricBilinearSpace(QQ, [1, 2])
      sage: e1 = (1,0)
      sage: e2 = (0,1)

      sage: SBS.inner_product(e1, e1)
      1
      sage: SBS.inner_product(e1, e2)
      0
      sage: SBS.inner_product(e2, e2)
      2
```



```
      sage: v1 = (1,1)
      sage: SBS.inner_product(e1, v1)
      1
      sage: SBS.inner_product(v1, e1)
      1
      sage: SBS.inner_product(v1, e2)
      2
      sage: SBS.inner_product(e2, v1)
      2
      """
          ## Return the inner product
          return self(x,y)




    def __call__(self, x, y):
        """
    Evaluate the underlying quadratic form on the given vector x.
    See also QuadraticForm.__call__() for more details.

    INPUT:
      x -- a vector, list, ot tuple of numbers coercible to the base field

    OUTPUT:
      a number in the base field

    EXAMPLES:
      sage: SBS = SymmetricBilinearSpace(GF(5), Matrix(QQ, 2, 2, [1, 2, 2, 3])); SBS
      Symmetric bilinear space over Finite Field of size 5 of dimension 2 defined by the Gram matrix
      [1 2]
      [2 3]
      sage: e1 = (1,0)
      sage: e2 = (0,1)
      sage: SBS(e1, e1)
      1
      sage: SBS(e1, e2)
      2
      sage: SBS(e2, e1)
      2
      sage: SBS(e2, e2)
      3

    """
        ## Check that x and y are vectors of the appropriate length, and defined
over the base_field.
        pass

        ## Coerce the vectors to live in the ambient inner product space
        V = self.__vector_space
        x1 = V(x)
        y1 = V(y)

        ##  Evaluate the inner product by the vector space inner product method
        return x1.inner_product(y1)



    def __eq__(self, other):
        """
    Perform equality testing, which means that the base_field,
    dimension, and symmetric inner product matrix coefficients are
    all equal.  (Note: This is much stronger than being rationally
    equivalent!)
```

INPUT:
other –– a symmetric bilinear space

OUTPUT:
boolean

EXAMPLES:
sage: SBS1 = SymmetricBilinearSpace(QQ, Matrix(QQ, 2, 2, [1,3,3,–2]))
sage: SBS2 = SymmetricBilinearSpace(QQ, Matrix(QQ, 2, 2, [–1,3,3, –2]))
sage: SBS1 == SBS1
True
sage: SBS1 == SBS2
False
sage: SBS2 == SBS2
True
sage: SBS3 = SymmetricBilinearSpace(Qp(3), Matrix(QQ, 2, 2, [1,3,3,–2]))
sage: SBS1 == SBS3
False
sage: SBS2 == SBS3
False
sage: SBS3 == SBS3 ## THIS GIVES A WIERD DEEPCOPY ERROR ALSO! =(FIX THIS!!!
True

```
"""
     ## Check that it's another symmetric bilinear space
     if not isinstance(other, SymmetricBilinearSpace):
          return False

     ## Check that the two dimensions are equal
     if (self.dim() != other.dim()):
          return False

     ## Check that the two base fields are equal
     if (self.base_field() != other.base_field()):
          return False

     ## Check that the two inner product matrices are equal
     if (self.gram_matrix() != other.gram_matrix()):
          return False

     ## All Tests Passed -- they're equal!
     return True



 def __ne__(self, other):
     """
```

Checks if the two symmetric bilinear spaces are not equal (see self.__eq__ for more details).

INPUT:
other –– a symmetric bilinear space

OUTPUT:
boolean

EXAMPLES:

```
"""
     return not self.__eq__(other)



 def __cmp__(self, other):
     """
```

This is the default comparison routine for <, <=, ==, >, >= if no special comparison operator is defined. These operations are not defined at present, and it is not clear what anything but equality would mean in this context, so we raise a NotImplementedError.

INPUT:
other –– a symmetric bilinear space

OUTPUT:
boolean

EXAMPLES:
sage: SBS1 = SymmetricBilinearSpace(QQ, Matrix(QQ, 2, 2, [1,3,3,–2]))
sage: SBS2 = SymmetricBilinearSpace(QQ, Matrix(QQ, 2, 2, [5,4,4,1]))
sage: SBS1.__cmp__(SBS2)
Traceback (most recent call last):
...
NotImplementedError: Warning: The comparison operation just used is not presently defined.
sage: SBS1 > SBS2
Traceback (most recent call last):
...
NotImplementedError: Warning: The comparison operation just used is not presently defined.

```
    """
         raise NotImplementedError, "Warning: The comparison operation just used is not presently def
ined."




## ------------------------------------  Field-specific Routines  --------------
------------------------------


     def orthogonal_basis(self):
         """
```

Return an orthogonal basis for the symmetric bilinear space, or false if no such basis exists!

TO DO: MAKE THIS FIELD–INDEPENDENT!!!

INPUT:
None

OUTPUT:
A list of vectors

EXAMPLES:
sage: M1 = Matrix(QQ, 3, 3, [2, 3, 4, 3, –1/2, –1, 4, –1, 17])
sage: M1.det()
–188
sage: SBS = SymmetricBilinearSpace(QQ, M1)
sage: B = SBS.orthogonal_basis()
sage: M2 = Matrix(QQ, 3, 3, [SBS.inner_product(B[i], B[j]) for i in range(3) for j in range(3)])
sage: M2.is_diagonal()
True
sage: M2
[2 0 0]
[0 –5 0]
[0 0 94/5]
sage: M2.det()
–188

```
    """
         n = self.dim()
         B = self.gram_matrix()
         MS = MatrixSpace(self.base_field(), n, n)
         T = MS(1)
         char = self.base_field().characteristic()

         ## Clear the entries one row at a time.
         for i in range(n):
```

```
            ## Deal with rows where the diagonal entry is zero.
            if B[i,i] == 0:

                ## Look for a non-zero entry and use it to make the diagonal non
-zero (if it exists)
                for j in range(i+1, n):
                    if B[i,j] != 0:

                        ## Return False for fields of characteristic 2
                        if char == 2:
                            return False

                        ## Otherwise make a shearing elementary row/column opera
tion to create a non-zero diagonal entry
                        temp = MS(1)
                        temp[j, i] = 1

                        ## Apply the transformation
                        B = temp.transpose() * B * temp
                        T = T * temp
                        break

            ## Create a matrix which deals with off-diagonal entries (all at onc
e for each row)
            temp = MS(1)
            for j in range(i+1, n):
                if B[i,j] != 0:
                    temp[i,j] = -B[i,j] / B[i,i]    ## This should only occur wh
en B[i,i] != 0, which the above step guarantees.

            B = temp.transpose() * B * temp
            T = T * temp


        ## Return the appropriate output
        return T.columns()



    def integral_lattice(self):
        """
    Return a Z-valued lattice on this symmetric bilinear space (by scaling all of
    the lattice generators to be Z-valued).

    TO DO:
      - Deal with bilinear spaces that *do not* have an orthogonal basis!
      - Modify this to allow it to return I-valued forms for any ideal I over a number field.


    INPUT:
      None

    OUTPUT:
      a symmetric bilinear lattice over ZZ

    EXAMPLES:
      sage: SBS = SymmetricBilinearSpace(QQ, DiagonalMatrix(QQ, [2,45/11,1/12]))
      sage: SBS.integral_lattice()
      ...
      Quadratic Lattice in Quadratic space defined by the Quadratic form in 3 variables over Rational Field with coeff
icients:
      [ 2 0 0 ]
      [ * 45/11 0 ]
      [ * * 1/12 ]
      spanned by ((1, 0, 0), (0, 11, 0), (0, 0, 6)).

      sage: SBS = QuadraticSpace(QQ, QuadraticForm(QQ, 3, [2, 11/3, 5,  45/11, 1,  1/12]))
```



```
      sage: L = SBS.integral_lattice(); L
      ...
      Quadratic Lattice in Quadratic space defined by the Quadratic form in 3 variables over Rational Field with coeff
icients:
      [ 2 11/3 5 ]
      [ * 45/11 1 ]
      [ * * 1/12 ]
      spanned by ((1, 0, 0), (0, 66, 11454), (0, 0, 22908)).
      sage: Matrix(QQ, 3, 3, [L.inner_product(L.basis()[i], L.basis()[j])  for i in range(3)  for j in range(3)]) in Matrix
Space(ZZ, 3, 3)
      True
    """
        new_gen_list = []

        ## Scale all vectors in an orthogonal basis to make them integer-valued

        for v in self.orthogonal_basis():
            #print "v =", v
            d = self(v,v).denominator()
            #print "d =", d
            scale_factor = sqrt(d * squarefree_part(d))
            #print "scale_factor =", scale_factor
            new_gen_list.append(v * scale_factor)
            #print "Done with v! =) \n"

        #raise NotImplementedError, "Need to decide about and write the symmetri
c bilinear lattice class..."
        return SymmetricBilinearLattice(self, new_gen_list)    ## Note: This ope
ration does *not* preserve the given basis!

        #try:
        #except:
        #    raise NotImplementedError, "We need to write how to deal with integ
rality for non-diagonalizable quadratic spaces!"



    def find_basis_of_radical_subspace(self):
        """
    Return a basis for the radical of a this symmetric bilinear
    space.  As a second return argument, return a basis for a
    complementary subspace.
    """
        F = self.base_field()
        n = self.dim()

        radical_basis = self.orthogonal_subspace_to_vector_list(self.basis()).ba
sis()

        ## TO DO: THE FOLLOWING CODE SHOULD BE IN A VECTORSPACE ROUTINE TO FIND
A COMPLEMENTARY SET TO A SUBSPACE!

        ## Use the standard "look for pivot columns" trick to find a complementa
ry set of vectors!
        Id_matrix = Matrix(F, n, n, 1)
        radical_columns = Matrix(F, radical_basis).transpose()
        M = radical_columns.augment(Id_matrix)
        M1 = M.rref()

        ## Find the pivot columns of the identity matrix part, and these say whi
ch standard basis vectors to use as a complement!
        r = len(radical_basis)
        complementary_basis = [Id_matrix[i-r]  for i in M1.pivots()  if i >= r]

        ## Return the radical basis, and a complementary basis
        return radical_basis, complementary_basis
```

```
    def find_isotropic_vector(self, known_to_exist=False):
        """
    Returns a non-zero vector v in the symmetric blinear space with B(v,v) = 0,
    and returns False if there is no such vector.

    If the known_to_exist flag is set, then it will continue looking until it finds an isotropic vector.

    INPUT:
      None

    OUTPUT:
      a vector over self.base_field()

    EXAMPLES:
      sage: A = matrix(ZZ, 2, 2, [3,1,1,3])
      sage: QS = SymmetricBilinearSpace(GF(3), A)
      sage: v = QS.find_isotropic_vector()
      sage: QS(v) == 0
      True
      sage: v.parent()
      Vector space of dimension 2 over Finite Field of size 3

    """
        ## SANITY CHECK: Check that the space is non-degenerate
        if self.is_degenerate():
            raise NotImplementedError, "For now we require a non-degenerate bilinear space."

        d = self.dim()
        F = self.base_field()
        char_p = F.characteristic()


        ## Deal with dimension <=1 forms -- (very easy)
        if (d == 0) or (d == 1):
            return False


        ## DIAGNOSTIC
        verbose("\n self.base_field() = " + str(self.base_field()))
        verbose("\n self.gram_matrix() = " + str(self.gram_matrix()))


        ## Import QuadraticSpace here to avoid circular dependencies!
        from sage.quadratic_forms.quadratic_space import QuadraticSpace

        ## Deal with anisotropic forms -- (uses invariants of the associated qua
dratic space, if they exist!)
        if not known_to_exist and QuadraticSpace(self.base_field(), self.gram_ma
trix(), matrix_type="Gram").is_anisotropic():
            return False


        ## Deal with isotropic forms (so we have isotropic vectors)!
        PR = PolynomialRing(F, 'y')
        y = PR.gen()

        while True:                                                     ## This mus
t terminate since n >= 3
            ## Choose a random (non-degenerate) linear polynomial vector, giving
 a general line in our space
            v1 = vector([PR(F.random_element())  for i in range(d)])
            while v1 == v1.parent().zero_vector():
                v1 = vector([PR(F.random_element())  for i in range(d)])
            v2 = vector([PR(F.random_element())  for i in range(d)])
            while v2 == v1.parent().zero_vector():
                v2 = vector([PR(F.random_element())  for i in range(d)])
            v = v1 + y * v2
```



```
            ## DIAGNOSTIC
            verbose("\n \n\n Looking for an isotropic vector on the line defined by:")
            verbose("\n v1 = " + str(v1))
            verbose("\n v2 = " + str(v2))
            verbose("\n giving v = " + str(v))
            verbose("\n ---------------------------------- \n")

            ## Find its value (as a quadratic polynomial in y)  --  this could b
e sped up by not re-copying the matrix G every time, and evaluating instead!
            G = self.gram_matrix()
            G1 = matrix(PR, G)
            m1 = (v * G1 * v.transpose())[0]


            ## DIAGNOSTIC
            verbose("G = " + str(G))
            verbose("G1 = " + str(G1))
            verbose("m1 = " + str(m1))



            ## Deal with every vector being isotropic
            if m1 == 0:
                #return vector(F, v1)                                    -- This f
ails due to a coersion bug over GF(2) -- TRAC #
                return vector([F(v1[i]) for i in range(len(v1))])

            ## Otherwise find roots, and return an isotropic vector
            m1_roots = m1.roots()
            if len(m1_roots) != 0:
                a = m1_roots[0][0]  ## Take the first root over F_p
                new_v = v1 + a*v2

                ## DIAGNOSTIC
                verbose("m1_roots = " + str(m1_roots))
                verbose("new_v = " + str(new_v))

                if new_v != new_v.parent().zero_vector():
                    #return vector(F, new_v)                                --
 This fails due to a coersion bug over GF(2) -- TRAC #
                    return vector([F(new_v[i]) for i in range(len(new_v))])

#                     verbose("Ready to return the non-zero new_v = " + str(new_v
))
#                     new_v_as_vector = vector(F, new_v)
#                     verbose("new_v_as_vector = " + str(new_v_as_vector))
#                     verbose("type(new_v_as_vector) = " + str(type(new_v_as_vect
or)))
#                     return vector(F, new_v)




    def orthogonal_subspace_to_vector_list(self, v_list):
        """
    Find the subspace of the symmetric bilinear space orthogonal to the given list of vectors.

    INPUT:
      v_list -- a list of vectors coercible to the given vectorspace

    OUTPUT:
      a subspace of the underlying vectorspace

    EXAMPLES:

    """
        ## SANITY CHECK: Check that the vectors are coercible to our vectorspace
```

```
!
        try:
            v1_list = [self.vector_space()(v)  for v in v_list]
        except:
            raise TypeError, "You need to pass in a vector coercible to this symmetric bilinear space!"


        F = self.base_field()
        return (self.gram_matrix() * matrix(F, v1_list).transpose()).kernel()




    def basis(self):
        """
    Return the (standard) basis of the underlying vector space.
    """
        return self.__vector_space.basis()




    def hyperbolic_complement_for_isotropic_vector(self, v):
        """
    Returns an isotropic vector whose inner product with v is 1, if one exists.

    (Here we presently assume that the bilinear space space is non-degenerate, and has characteristic not 2!)

    """
        ## SANITY CHECK: Check that the space is non-degenerate
        if self.is_degenerate():
            raise NotImplementedError, "For now we require a non-degenerate bilinear space."


        F = self.base_field()

        ## Split off a hyperbolic plane if the characteristic is not 2
        if F.characteristic() != 2:
            for w in self.basis():
                a = self.inner_product(v,w)
                if (a != 0) and (w != v):
                    b = self.inner_product(w,w)
                    return w - v * (F(1)/F(2*a))

        ## For characteristic 2,
        else:
            raise NotImplementedError, "This is not as easy to do in characteristic 2, " + \
                "and requires we normalize (as much as) the entire space to find such a vector!"




    def characteristic_two_decomposition(self):
        """
    Decompose a bilinear space of characteristic 2 into a radical, non-deg space of isotropic vectors, and non-deg ani
sotropic space.

    Note:  The non-deg space of isotropic vectors will not be totally isotropic, except

    Returns:    radical_basis, aniso_basis, maximal_tot_iso_basis, hyperbolic_pair_list, metabolic_pair_list



    INPUT:
```



```
        None

    OUTPUT:
        five lists of vectors.


    EXAMPLES:
        sage: S2 = SymmetricBilinearSpace(GF(2), DiagonalMatrix(ZZ, [1,1,1,1]))
        sage: S2.characteristic_two_decomposition()
        ([], [(0, 0, 0, 1)], [[(1, 0, 1, 0), (0, 1, 1, 0)]], [(1, 0, 1, 0), (1, 1, 1, 1)])

    """
        F = self.base_field()

        ## Step 1. Arrange for the diagonal to be an anisotropic quadratic form:
        ## ---------------------------------------------------------------------
_
        #final_basis_list = self.basis()

        ## Find the basis vectors giving non-zero diagonal entries.
        nz_diag_basis = []
        nz_diag_elements = []
        zero_basis = []
        nz_vec_index_list = []

        B = self.basis()          ## This is the standard basis for our space
        for i in range(len(B)):
            v = B[i]
            tmp_elt = self.inner_product(v,v)
            if tmp_elt != 0:
                nz_diag_basis.append(v)
                nz_diag_elements.append(tmp_elt)
                nz_vec_index_list.append(i)
            else:
                zero_basis.append(v)

        ## Look for isotropic diagonal relations, until there are none
        v_iso = True
        while v_iso != False:

            #print "LOOKING FOR ISOTROPIC VECTORS IN Gram Matrix" + str(Diagonal
Matrix(F, nz_diag_elements))

            ## Find an isotropic vector for the associated non-zero diagonal SBS
            nz_diag_SBS = SymmetricBilinearSpace(F, DiagonalMatrix(F, nz_diag_el
ements))
            v_iso = nz_diag_SBS.find_isotropic_vector()

            #print "USING THE ISOTROPIC VECTOR v_iso = " + str(v_iso)


            ## If we have an isotropic vector, shrink our space
            if v_iso != False:

                ## Find the index in our list of a standard basis vector sharing
 a non-zero entry with v.
                for i in range(len(v_iso)):
                    if v_iso[i] != 0:
                        trim_index = i
                        break


                ## Replace the associated basis vector with v_iso in the final_b
asis_list
                #v_old = nz_diag_basis[trim_index]
                #final_basis_list[final_basis_list.index(v_old)] = v_iso

                ## Add the (expanded) isotropic vector v_iso to the zero_basis l
ist
```

```
                v_iso_big_list = len(B) * [0]
                for i in range(len(nz_vec_index_list)):
                    v_iso_big_list[nz_vec_index_list[i]] = v_iso[i]
                zero_basis.append(vector(F, v_iso_big_list))

                ## Shrink our diagonal SBS, since we have made one diagonal entr
y zero
                nz_diag_basis = nz_diag_basis[:trim_index] + nz_diag_basis[trim_
index+1:]
                nz_diag_elements = nz_diag_elements[:trim_index] + nz_diag_eleme
nts[trim_index+1:]
                nz_vec_index_list = nz_vec_index_list[:trim_index] + nz_vec_inde
x_list[trim_index+1:]


        #print "\n\n Finished Step 1:"
        #print "nz_diag_basis = " + str(nz_diag_basis)
        #print "nz_diag_elements = " + str(nz_diag_elements)
        #print "nz_vec_index_list = " + str(nz_vec_index_list)
        #print "zero_basis = " + str(zero_basis)
        #print self.gram_matrix_for_vectors(zero_basis + nz_diag_basis)


        ## Step 2.  Split off hyperbolic planes:
        ## -------------------------------------
        hyperbolic_pair_list = []
        b = 1

        ## Loop through all strictly upper triangular entries until they're all
zero
        while b != 0:

            ## Find a non-zero (off-diagonal) entry, if one exists
            z_len = len(zero_basis)
            upper_diag_list = [(i,j)  for i in range(z_len)  for j in range(i+1,
z_len)]
            b = 0  ## Needed if there are no strictly upper-triangular entries!
            for (i,j) in upper_diag_list:                    ## This is a littl
e redundant, but at least we can use a single "break" to escape!
                b = self.inner_product(zero_basis[i], zero_basis[j])
                if (b != 0):
                    B = b
                    I = i
                    J = j
                    break


            ## If one exists, then split off a hyperbolic plane
            if b != 0:

                ## Note: We now have I, J, and b
                #print "B = " + str(B)
                #print "I = " + str(I)
                #print "J = " + str(J)
                #print "--------"

                ## Normalize to get a hyperbolic plane from the (i,j) vectors, a
nd remember the plane
                zero_basis[J] = zero_basis[J] / b
                v = zero_basis[I]
                w = zero_basis[J]
                hyperbolic_pair_list.append([v,w])

                ## Split off the hyperbolic plane (top row and then bottom row,
for the hyperbolic plane rows)
                for k in range(len(zero_basis)):
                    M_ik = self.inner_product(v, zero_basis[k])
                    if (k != J) and (M_ik != 0):
                        zero_basis[k]  =  zero_basis[k] - (M_ik * w)
```



```
                for k in range(len(zero_basis)):
                    M_jk = self.inner_product(w, zero_basis[k])
                    if (k != I) and (M_jk != 0):
                        zero_basis[k]  =  zero_basis[k] - (M_jk * v)

                for k in range(len(nz_diag_basis)):
                    M_ik = self.inner_product(v, nz_diag_basis[k])
                    if (k != J) and (M_ik != 0):
                        nz_diag_basis[k]  =  nz_diag_basis[k] - (M_ik * w)

                for k in range(len(nz_diag_basis)):
                    M_jk = self.inner_product(w, nz_diag_basis[k])
                    if (k != I) and (M_jk != 0):
                        nz_diag_basis[k]  =  nz_diag_basis[k] - (M_jk * v)


                ## Remove the hyperbolic plane vectors, so we get a smaller spac
e to look in
                zero_basis = zero_basis[:I] + zero_basis[I+1:J] + zero_basis[J+1
:]


        #print "\n\n Finished Step 2 -- Split off all hyperbolic planes:"
        #print "hyperbolic_pair_list = " + str(hyperbolic_pair_list)
        #print "zero_basis = " + str(zero_basis)
        #print "nz_diag_basis = " + str(nz_diag_basis)
        #print self.gram_matrix_for_vectors(zero_basis + nz_diag_basis)


        ## Step 3:  Split off the metabolic planes
        ## ---------------------------------------
        metabolic_pair_list = []
        z_len = len(zero_basis)
        nz_len = len(nz_diag_basis)

        if (z_len > 0) and (nz_len > 0):

            ## Compute the upper-right Gram matrix block
            M = Matrix(F, z_len, nz_len)
            for i in range(len(zero_basis)):
                v = zero_basis[i]
                for j in range(len(nz_diag_basis)):
                    w = nz_diag_basis[j]
                    M[i,j] = self.inner_product(v, w)

            #print "M = " + str(M)

            ## Arrange that the upper-right block is row reduced
            M1 = M.augment(Matrix(F, z_len, z_len, 1))
            M1.subdivide(None, z_len)
            M1 = M1.rref()

            #print "M1 = " + str(M1)


            ## Find the change of variables of the zero-basis vectors to give th
is row-reduced form
            T = M1.subdivision(0,1)
            z_rows_matrix = Matrix(F, zero_basis)
            zero_basis = (T * z_rows_matrix).rows()

            ## Split off the metabolic planes
            M0 = M1.subdivision(0,0)    ## This is the RREF of the upper-right g
ram matrix
            pivot_columns = M0.pivots()
            pivot_rows = M0.pivot_rows()
            number_of_pivots = len(pivot_rows)
            for p in range(number_of_pivots):
```

```
                I = pivot_rows[p] - p          ## This '-p' is a convenient way
of maintining the RREF indexing, since both I and J are strictly increasing!
                J = pivot_columns[p] - p
                v = zero_basis[I]
                w = nz_diag_basis[J]
                metabolic_pair_list.append([v,w])

                ## Split off one metabolic plane (top row and then bottom row, f
or the hyperbolic plane rows)
                for k in range(len(nz_diag_basis)):
                    M_ik = self.inner_product(v, nz_diag_basis[k])
                    if (k != J) and (M_ik != 0):
                        nz_diag_basis[k]  =  nz_diag_basis[k] - (M_ik * w)


                ## Remove the metabolic plane basis, so we get a smaller space t
o look in
                zero_basis = zero_basis[:I] + zero_basis[I+1:]
                nz_diag_basis = nz_diag_basis[:J] + nz_diag_basis[J+1:]


        ## Step 4:  Find the relevant bases, and return the results
        ## -------------------------------------------------------

        ## The remaining zero-basis vectors form the radical of the space
        radical_basis = zero_basis

        ## The remaining nz_diag-basis vectors are an anisotropic space (but the
 full anisotropic space may have a metabolic contribution!)
        diag_aniso_basis = nz_diag_basis

        ## Find bases for a maximal isotropic and maximal anisotropic subspace!
        maximal_tot_iso_basis = radical_basis + [H[0]  for H in hyperbolic_pair_
list] + [M[0]  for M in metabolic_pair_list]
        aniso_basis = diag_aniso_basis + [M[1]  for M in metabolic_pair_list]

        ## Return the results
        return  radical_basis, aniso_basis, maximal_tot_iso_basis, hyperbolic_pa
ir_list, metabolic_pair_list









    def find_basis_of_maximal_totally_isotropic_subspace(self, return_type="list"
):
        """
    Find a basis of a maximal totally isotropic subspace of the symmetric bilinear space, as a matrix of row vectors.

    The return_type can be 'list', 'row_matrix', or 'column_matrix'.


    Note:  Presently this does not require the space to be non-degenerate -- though we may want to add this in the fut
ure!

    INPUT:
      None

    OUTPUT:
      a matrix of row vectors

    EXAMPLES:
      sage: from sage.quadratic_forms.maximal_extras import find_basis_of_maximal_totally_isotropic_subspace

      sage: MM = matrix(ZZ, 6, 6, [0, 0, 1, 2, 2, 2, 0, 0, 0, 1, 0, 1, 1, 0, 2, 2, 3, 0, 2, 1, 2, 3, 1, 2, 2, 0, 3, 1, 1, 0, 2, 1, 0,
 2, 0, 1])
```



```
      sage: QS = QuadraticSpace(GF(5), MM)
      sage: QS.find_basis_of_maximal_totally_isotropic_subspace()  ## random
      [2 0 0 3 0 2]
      [0 3 0 1 3 1]
      [0 0 2 2 2 3]

      sage: S2 = SymmetricBilinearSpace(GF(2), DiagonalMatrix(ZZ, [1,1,1,1]))
      sage: S2.find_basis_of_maximal_totally_isotropic_subspace()
      [(0, 1, 0, 1), (1, 0, 1, 0), (0, 0, 1, 1)]

    """
        F = self.base_field()

        ## Deal with characteristic 2 spaces!
        if self.base_field().characteristic() == 2:
            (R, A, M_iso, Hyp_list, Meta_list) = self.characteristic_two_decompo
sition()
            T_new = Matrix(F, M_iso)



        ## Deal with odd characteristic spaces:
        ## ------------------------------------
        else:

            ## Deal with degenerate bilinear spaces!
            if self.is_degenerate():

                ## DIAGNOSTIC
                verbose("\n\n\n\nStarting the degenerate case...")

                radical_basis, nondeg_basis = self.find_basis_of_radical_subspac
e()
                V1 = SymmetricBilinearSpace(self.base_field(), self.gram_matrix_
for_vectors(nondeg_basis))
                V1_iso_basis = V1.find_basis_of_maximal_totally_isotropic_subspa
ce()
                M1_iso_columns = Matrix(self.base_field(), V1_iso_basis).transpo
se()    ## Make a matrix of columns (w.r.t. the standard basis for V1)
                M_nondeg_columns = Matrix(self.base_field(), nondeg_basis).trans
pose()    ## Make a matrix of columns (w.r.t. the standard basis for V)

                ## Find the radical and non-degenerate vectors that give a maxim
al isotropic subspace
                T_new = Matrix(self.base_field(), radical_basis + (M_nondeg_colu
mns * M1_iso_columns).columns())


            ## Here we suppose that the space is non-degenerate...
            ## ---------------------------------------------------
            else:
                ## DIAGNOSTIC
                verbose("\n\n\n\nStarting the non-degenerate case...")


                G = self.gram_matrix()
                n = G.nrows()

                p = G.parent().base_ring().characteristic()



                ## Make the transformation matrix (of rows!!!)
                T = matrix(F, 0, n, [])

                ## Find one isotropic vector
                v = self.find_isotropic_vector()

                ## Check if we're done.
```

```
                if v == False:
                    return T


                ## Split off a hyperbolic plane, since we're non-degenerate:
                ## ----------------------------------------------------------

                ## Find a hyperbolic plane H containing the given isotropic vect
or
                w = self.hyperbolic_complement_for_isotropic_vector(v)

                ## Find the smaller subspace (orthogonal to H) and its Gram matr
ix to use in the recursion
                K1 = self.orthogonal_subspace_to_vector_list([v,w]).basis_matrix
()    ## Note: Row vectors here, in reduced row echelon form!
                G1 = K1 * G * K1.transpose()

                ## DIAGNOSIC
                verbose("\n K1 = " + str(K1))
                verbose("\n G1 = " + str(G1))

                ## Perform the recursion
                SBS1 = SymmetricBilinearSpace(self.base_field(), G1)
                T1 = SBS1.find_basis_of_maximal_totally_isotropic_subspace(retur
n_type="row_matrix")
                T_last = T1 * K1
                T_new = (T_last.transpose().augment(v.transpose())).transpose()
  ## Augment T_last by adding the row v

                ## DIAGNOSTIC
                verbose("\n T_new = " + str(T_new))
                verbose("\n Found T_new of dimension " + str(T_new.nrows()))
                verbose("\n return_type = " + str(return_type))



        ## Return the isotropic basis, in the desired format
        if return_type == "row_matrix":
            return T_new
        elif return_type == "column_matrix":
            return T_new.transpose()
        elif return_type == "list":
            return T_new.rows()
        else:
            raise TypeError, "The return_type must be either 'list', 'row_matrix', or 'column_matrix'."




    def maximal_bilinear_lattice(self):
        """
    Find a symmetric bilinear lattice equivalent to a maximal lattice
    in the given symmetric bilinear space.

    TO DO: Add support for (a)-maximal lattices too.


    INPUT:
      None

    OUTPUT:
      a symmetric bilinear lattice over ZZ

    EXAMPLES:
      sage: QS = QuadraticSpace(QQ, QuadraticForm(QQ, 3, [2, 11/3, 5,  45/11, 1,  1/12]))


      sage: SBS = SymmetricBilinearSpace(QQ, DiagonalMatrix(QQ, [1,3,9,27]))
```



```
      sage: L = SBS.maximal_bilinear_lattice(); L
      Symmetric Bilinear Lattice in Symmetric bilinear space over Rational Field of dimension 4 defined by the Gra
m matrix
      [ 1  0  0  0]
      [ 0  3  0  0]
      [ 0  0  9  0]
      [ 0  0  0 27]
      generated over the ring Integer Ring by
      [
      (0, 0, 1/3, 0),
      (1, 0, 0, 0),
      (0, 0, 0, 1/3),
      (0, 1, 0, 0)
      ]
      sage: L.gram_matrix()
      [1 0 0 0]
      [0 1 0 0]
      [0 0 3 0]
      [0 0 0 3]



      sage: SBS = SymmetricBilinearSpace(QQ, DiagonalMatrix(QQ, [1,5,25,125]))
      sage: L = SBS.maximal_bilinear_lattice(); L
      Symmetric Bilinear Lattice in Symmetric bilinear space over Rational Field of dimension 4 defined by the Gra
m matrix
      [  1   0   0   0]
      [  0   5   0   0]
      [  0   0  25   0]
      [  0   0   0 125]
      generated over the ring Integer Ring by
      [
      (0, 0, 1/5, 0),
      (1, 0, 0, 0),
      (0, -1/5, 0, -2/25),
      (0, -2/5, 0, 1/25)
      ]
      sage: L.gram_matrix()
      [1 0 0 0]
      [0 1 0 0]
      [0 0 1 0]
      [0 0 0 1]

      sage: SBS = SymmetricBilinearSpace(QQ, 2*DiagonalMatrix(QQ, [1,5,25,125]))
      sage: L = SBS.maximal_bilinear_lattice()
      sage: L.gram_matrix(rational_matrix=True)
      [1 0 0 0]
      [0 1 0 0]
      [0 0 1 0]
      [0 0 0 1]



      ## CHECK THESE BELOW!!! -- THESE ARE NOT CHECKED BY HAND! =(

      sage: SBS = SymmetricBilinearSpace(QQ, DiagonalMatrix(QQ, [1,3,9,27]))
      sage: L = SBS.maximal_bilinear_lattice()
      sage: L.gram_matrix(rational_matrix=True)
      [1 0 0 0]
      [0 1 0 0]
      [0 0 3 0]
      [0 0 0 3]
      sage:
      sage: SBS = SymmetricBilinearSpace(QQ, 2 * DiagonalMatrix(QQ, [1,3,9,27]))
      sage: L = SBS.maximal_bilinear_lattice()
      sage: L.gram_matrix(rational_matrix=True)
      [2 0 1 0]
      [0 2 0 1]
      [1 0 2 0]
      [0 1 0 2]
```

```
        sage:
        sage: SBS = SymmetricBilinearSpace(QQ, 4 * DiagonalMatrix(QQ, [1,3,9,27]))
        sage: L = SBS.maximal_bilinear_lattice()
        sage: L.gram_matrix(rational_matrix=True)
        [1 0 0 0]
        [0 1 0 0]
        [0 0 3 0]
        [0 0 0 3]

    """
        n = self.dim()

        ## Check that we're working over QQ
        if self.base_field() != QQ:
            raise NotImplementedError, "Presently only the base field of QQ is supported."

        ## Find an integral lattice in our symmetric bilinear space
        L = self.integral_lattice()

        ## DIAGNOTSIC
        verbose("\n L = " + str(L))
        verbose("\n L.gram_matrix() = \n" + str(L.gram_matrix(rational_matrix=False)))
        verbose("\n ====================== ")

        ## Enlarge the integral symmetric bilinear lattice so its discriminant m
odule L^#/L is a product of quadratic spaces
        H = L.gram_matrix()
        max_ed = H.elementary_divisors()[-1]
        while not is_squarefree(max_ed):    ## Check if the largest elementary d
ivisor is not squarefree

            ## Enlarge the lattice with the scaled dual lattice
            big_sq_factor = sqrt(max_ed * squarefree_part(max_ed))  ## This is t
he amount we scale the dual lattice by, doing all primes at once!
            L = L.sum_with(L.dual_lattice().apply_linear_transformation_on_right
(big_sq_factor))

            ## DIAGNOSTIC
            verbose("\n big_sq_factor = " + str(big_sq_factor))
            verbose("\n L = " + str(L))
            verbose("\n L.gram_matrix() \n= " + str(L.gram_matrix(rational_matrix=False
)))
            verbose("\n ====================== ")

            ## Prepare to check if we're done
            H = L.gram_matrix()
            max_ed = H.elementary_divisors()[-1]



        ## Find a maximal Z-valued Hessian bilinear lattice:
        ## -------------------------------------------------
        L_dual = L.dual_lattice()
        A = Matrix(ZZ, L_dual.gram_matrix(rational_matrix=True).inverse())     #
# This matrix describes L in the given basis of L^#
        D, U, V = A.smith_form()
        iso_eligible_primes = prime_divisors(D[-2,-2])        ## The list of pri
mes which have at most a 2-dim'l subspace in L^#/L
        dual_prime_product = D[-1,-1]          ## This is the product of all pri
mes dividing |L^#/L|
        #L1_dual = L_dual.apply_linear_transformation_on_right(U)        ## This
 is the dual lattice in the 'nice' basis, so a basis of L is multiples
        L1_dual = L_dual.apply_linear_transformation_on_right(U.inverse())
  ## This is the dual lattice in the 'nice' basis, so a basis of L is multiples


        ## DIAGNOSTIC
        verbose("\n \n\n Start looking for a maximal Z-valued Hessian bilinear lattice:")
        verbose("\n ----------------------------------------------------------------")
```



```
        verbose("\n A = " + str(A))
        verbose("\n D = " + str(D))
        verbose("\n U = " + str(U))
        verbose("\n V = " + str(V))
        verbose("\n iso_eligible_primes = " + str(iso_eligible_primes))
        verbose("\n dual_prime_product = " + str(dual_prime_product))
        verbose("\n dual primes = " + str(prime_divisors(dual_prime_product)))
        verbose("\n L_dual = " + str(L_dual))
        verbose("\n L_dual.gram_matrix() = \n" + str(L_dual.gram_matrix(rational_matrix=
True)))
        verbose("\n L1_dual = " + str(L1_dual))
        verbose("\n L1_dual.gram_matrix() = \n" + str(L1_dual.gram_matrix(rational_matri
x=True)))
        verbose("\n ====================================== ")

        ## Loop through all primes to find a maximal isotropic space for each.
        T_huge = deepcopy(D)    ## This will hold the final list of generators f
or the isotropic submodule (of L^#), together with a basis of L
        for p in iso_eligible_primes:

            ## DIAGNOSTIC
            verbose("\n \n\n\n\n Starting to compute a maximal totally isotropic subspace for p = " + str(
p))
            verbose("\n ============================================================
========= \n")

            ## Compute the dimension d_p of L^#/L at p -- (Since (1/p) * D gives
 the L1_dual basis relative to some basis of L)
            d_p = 0
            for i in range(n):
                if D[i,i] % p == 0:
                    d_p = n-i         ## Dim of L#/L
                    break

            ## Create the Gram matrix for the smaller subspace
            small_gram = L1_dual.gram_matrix(rational_matrix=True)[(n-d_p):, (n-
d_p):]

            verbose("\n small_gram = " + str(small_gram))


            ## Create the bilinear lattice in the basis
            small_SBS = SymmetricBilinearSpace(GF(p), p * small_gram)  ## This m
ult-by-p makes the gram matrix p-integral
            verbose("\n small_SBS = " + str(small_SBS))
            Tp = small_SBS.find_basis_of_maximal_totally_isotropic_subspace(retu
rn_type="row_matrix")

            ## DIAGNOSTIC
            verbose("\n \n\n After one eligible prime:")
            verbose("\n ---------------------------")
            verbose("\n p = " + str(p))
            verbose("\n d_p = " + str(d_p))
            verbose("\n small_gram = " + str(small_gram))
            verbose("\n small_SBS = " + str(small_SBS))
            verbose("\n Tp = " + str(Tp))
            verbose("\n Finished finding maximal isotropic subspace at prime " + str(p))


            ## Now add an integral lift of this subspace to the basis of our sub
module in L^#/L
            if Tp.nrows() != 0:



                ## Find the appropriate lift factor to multiply the (mod p) vect
or by to lift it to ZZ only at p (and is the zero vector at other primes!).
                dual_prime_prod_not_p = ZZ(dual_prime_product / p)
                dual_prime_lift_multiplier = (dual_prime_prod_not_p) * (GF(p)(du
```

```
al_prime_prod_not_p)**(-1)).lift()


                ## Find integer lifts of our vectors (being careful to have the
vectors to initially be primitive relative to all other primes)
                Tp_lift_list = []
                for v in Tp.rows():
                    v1 = vector(ZZ, v.lift())
                    g = GCD(list(v1))
                    v2 = v1 * (1/g)
                    Tp_lift_list.append(v2 * dual_prime_lift_multiplier)


                ## Add lifts of this subspace to our matrix of generators
                TZ_top_as_rows = Matrix(ZZ, Tp.nrows(), (n - d_p))
                TZ_bottom_as_rows = Matrix(ZZ, Tp_lift_list)
                TZ = TZ_top_as_rows.augment(TZ_bottom_as_rows).transpose()   ##
extend the isotropic basis correctly in the basis of L^#
                #dp_cols_small = U.matrix_from_columns(range(d_p))    ## These c
olumns are the basis of L^# we used to find the maximal iso subspace.
                #TZ = dp_cols_small * matrix(ZZ,Tp).transpose()  ## Add (a lift
to ZZ of) these vectors to a (column) matrix of generators.


                ## DIAGNOSTIC
                verbose("\n \n\n This prime gives the lifts:")
                verbose("\n ----------------------------")
                verbose("\n dual_prime_product = " + str(dual_prime_product))
                verbose("\n dual_prime_prod_not_p = " + str(dual_prime_prod_not_p))
                verbose("\n dual_prime_lift_multiplier = " + str(dual_prime_lift_multiplie
r))
                verbose("\n TZ_top_as_rows = " + str(TZ_top_as_rows))
                verbose("\n TZ_bottom_as_rows = " + str(TZ_bottom_as_rows))
                verbose("\n TZ = " + str(TZ))
                verbose("\n T_huge = " + str(T_huge))
                verbose("\n ")


                T_huge = T_huge.augment(TZ)          ## Add (a lift to ZZ of) th
e isotropic basis vectors to a (column) matrix of generators

                ## DIAGNOSTIC
                verbose("\n T_huge = " + str(T_huge))
                verbose("\n ")


        ## Find a basis for the maximal form, in the ??? basis
        verbose("\n\n Status -- Pre-LLL")
        verbose("\n T_huge has " + str(T_huge.nrows()) + " rows and " + str(T_huge.ncol
s()) + " columns.")
        verbose("\n " + str(type(T_huge)) + " " + str(T_huge.parent()))
        verbose("\n " + str(T_huge.rows()))

        nr = T_huge.ncols()  ## after LLL the last rows form a basis, the first
ones are 0
        T_lll = T_huge.transpose().LLL().matrix_from_rows(range(nr-n,nr)).transp
ose()

        verbose("\n Status -- Post-LLL")
        #Gram_of_max_lat = matrix(ZZ, T_lll.transpose() * B * T_lll / (max_ed* m
ax_ed))



        ## DIAGNOSTIC
        verbose("\n \n\n Generating the final maximal lattice from:")
        verbose("\n ------------------------------------------")
        verbose("\n T_lll = " + str(T_lll))
        verbose("\n L1_dual.basis_matrix_of_columns() = " + str(L1_dual.basis_matrix_of_col
```



```
umns()))
        verbose("\n ")



        ## Return a maximal symmetric bilinear lattice
        return SymmetricBilinearLattice(self, (L1_dual.basis_matrix_of_columns()
 * T_lll).columns())




    def is_degenerate(self):
        """
    Determines if the quadratic space is degenerate (i.e. it has
    some non-zero vector orthogonal to the entire space).

    INPUT:
      None

    OUTPUT:
      boolean

    EXAMPLES:
      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(ZZ, [1,1,1,1]))
      sage: QS.is_degenerate()
      False

      sage: QS = QuadraticSpace(FiniteField(9,x), DiagonalQuadraticForm(ZZ, [1,3,5,7]))
      sage: QS.is_degenerate()
      True

      sage: QS = QuadraticSpace(RR, DiagonalQuadraticForm(ZZ, []))  ## The zero-dim'l for is non-degenerate by
def'n!
      sage: QS.is_degenerate()
      False

    """
        return self.determinant_squareclass().is_zero()


    def is_nondegenerate(self):
        """
    Determines if the quadratic space is non-degenerate (i.e. it
    has no non-zero vectors orthogonal to the entire space).

    INPUT:
      None

    OUTPUT:
      boolean

    EXAMPLES:
      sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(ZZ, [1,1,1,1]))
      sage: QS.is_nondegenerate()
      True

      sage: QS = QuadraticSpace(FiniteField(9,x), DiagonalQuadraticForm(ZZ, [1,3,5,7]))
      sage: QS.is_nondegenerate()
      False

      sage: QS = QuadraticSpace(RR, DiagonalQuadraticForm(ZZ, []))  ## The zero-dim'l for is non-degenerate by
def'n!
      sage: QS.is_nondegenerate()
      True
    """
        return not self.is_degenerate()
```

```
#############################################################################
#########################################
#############################################################################
#########################################
#############################################################################
#########################################









from sage.modules.free_module import FreeModule_ambient_field



#from sage.quadratic_forms.lattice import Lattice
#from sage.quadratic_forms.symmetric_bilinear_space import SymmetricBilinearSpac
e

from sage.modules.free_module import FreeModule_submodule_with_basis_pid

#from copy import deepcopy

from sage.rings.arith import LCM
from sage.misc.functional import denominator




################################################
## Code for the Symmetric Bilinear Lattice class ##
################################################

#class SymmetricBilinearLattice(Lattice):
class SymmetricBilinearLattice():
    """
    This is a class that gives a finitely generated submodule of a
    K-vectorspace over its ring of integers O_K, where K is a number
    field.

    TO DO: Eventually add support for S-integers as well, where S is a
    set of places of K.
    """


    def __init__(self, V, basis):
        """
        Initializes with the syntax:

          Lattice(V, list_of_generators)

        Lattice(V, row_matrix_of_generators)

        where V is a symmetric bilinear space


        Note:  When subclassing this class overload the __init__() and ambient_space() methods, and change the internal
variable.
        """
            ## Check that V is a symmetric bilinear space
            if not isinstance(V, SymmetricBilinearSpace):
                raise TypeError, "The first argument must be a symmetric bilinear space!"


            ## Check that V is a defined over a number field or its completion at a
place


            ## Check that basis is a matrix or list of vectors
            if not (is_Matrix(basis) or isinstance(basis, list)):
                raise TypeError, "The second argument must be a matrix or a list of vectors."

            ## Check that the ring of integers is a PID
            #raise NotImplementedError, "Presently we only have support for modules
over principal ideal domains."


            ## Initilize the lattice
#            Lattice.__init__(self, V, basis)
            self.__ambient_bilinear_space = V     ## We completely ignore the inner
product structure on this!
            self.__lattice_module = FreeModule_submodule_with_basis_pid(V.vector_spa
ce(), basis)
            self.__base_ring = V.base_field().ring_of_integers()


        def __repr__(self):
            """
        Returns a string representing the symmetric bilinear lattice.

        EXAMPLES:
            sage: L = SymmetricBilinearLattice(QQ^3, [[1,0,0], [1,2,3]])
            sage: L.__repr__()
            Symmetric Bilinear Lattice in Vector space of dimension 3 over Rational Field generated by over the ring Integ
er Ring by
            [
            (1, 0, 0),
            (1, 2, 3)
            ]
        """
            return "Symmetric Bilinear Lattice in " + str(self.ambient_space()) + \
                   " generated by over the ring " + str(self.base_ring()) + \
                   " by \n" + str(self.generators())


        def dual_lattice(self):
            """
        Compute the dual lattice of this lattice.

        INPUT:
            None

        OUTPUT:
            a symmetric bilinear lattice in the same ambient symmetric bilinear space.

        EXAMPLES:
            sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1,1]))
            sage: L = QuadraticLattice(QS)
            sage: L_dual = L.dual_lattice(); L_dual
            Quadratic Lattice in Quadratic space defined by the Quadratic form in 2 variables over Rational Field with coeff
```

```
icients:
        [ 1 0 ]
        [ * 1 ]
        spanned by ((1/2, 0), (0, 1/2)).

        sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [2,45/11,1/12]))
        sage: L = QS.integral_lattice()
        sage: L.dual_lattice()
        Quadratic Lattice in Quadratic space defined by the Quadratic form in 3 variables over Rational Field with coeff
icients:
        [ 2 0 0 ]
        [ * 45/11 0 ]
        [ * * 1/12 ]
        spanned by ((1/4, 0, 0), (0, 1/90, 0), (0, 0, 1)).

    """
        ## Check that the lattice is free -- useful for now, but can be easily c
ircumvented by dealing with generators instead!
        if not self.is_free():
            raise NotImplementedError, "We currently don't have an implementation for non-free la
ttices..."

        ## Compute the dual basis w.r.t. the ambient bilinear form.
        B = self.ambient_space().gram_matrix()      ## Gram matrix in std basis
        A = self.basis_matrix_of_rows()          ## Matrix of basis for L, as ro
w vectors.

        ## Solve A * H * B = Id to get a basis of the dual lattice
        X = (A * B).inverse()

        return SymmetricBilinearLattice(self.ambient_space(), X.columns())



    def is_integral(self):
        """
    Determine if the symmetric bilinear lattice is integer-valued.
    """
        A_rational = self.gram_matrix(rational_matrix=True)
        n = self.rank()

        ## Check that the matrix is ZZ-valued
        try:
            A = MatrixSpace(ZZ, n, n)(A_rational)
            return True
        except:
            return False



    def is_even(self):
        """
    Determines if the given (Z-valued symmetric bilinear) lattice is even.

    """
        ## Check if the lattice is integer-valued
        if not self.is_integral():
            return False

        ## Check if the lattice is even!
        L, X, even_flag = self.even_sublattice(return_extended_info=True)
        return even_flag



    def even_sublattice(self, return_extended_info=False):
        """
    Returns the even symmetric bilinear lattice of the given
```



```
    lattice, its associated transformation (on rows), and whether
    the original lattice was even to begin with.

    INPUT:
      None

    OUTPUT:
      (an even symmetric bilinear lattice over ZZ, an integral matrix, boolean)

    EXAMPLES:
      sage: from sage.quadratic_forms.maximal_extras import even_sublattice_of_bilinear_gram_matrix

      sage: even_sublattice_of_bilinear_gram_matrix(matrix(ZZ, 2, 2, [1,3,3,1]))
      ([4 8]
      [8 8], [2 1]
      [0 1], False)

    """
        ## Check that the matrix is ZZ-valued
        if not self.is_integral():
            raise TypeError, "The symmetric bilinear lattice is not integer-valued!"


        ## Initialize some variables
        n = self.rank()
        A = self.gram_matrix()
        X = matrix(ZZ, n, n, 1)


        ## Find the first odd diagonal entry
        odd_ind = -1
        for i in range(n):
            if A[i,i] % 2 != 0:
                odd_ind = i
                break

        ## Here the lattice is already even
        if odd_ind == -1:
            if return_extended_info:
                return self, X, True
            else:
                return self

        ## Make the transformation matrix (on row vectors!)
        X[odd_ind, odd_ind] = 2
        for i in range(odd_ind + 1, n):
            if A[i,i] % 2 != 0:
                X[i, odd_ind] = 1


        #print "return_extended_info = " + str(return_extended_info)

        ## Return the even sublattice
        if return_extended_info:
            #print "Returning a tuple"
            return SymmetricBilinearLattice(self.ambient_space(), (X * self.basi
s_matrix_of_rows()).rows()), X, False
        else:
            #print "Returning a Symmetric Bilinear lattice"
            return SymmetricBilinearLattice(self.ambient_space(), (X * self.basi
s_matrix_of_rows()).rows())




    def ambient_space(self):
        """
    Returns the ambient symmetric bilinear space that this lattice sits in.
```

```
    """
        ## We should be able to use this -- but we can't due a characteristic 2
vector space bug!  (TRAC #)
        #return deepcopy(self.__ambient_bilinear_space)

        ## Here is our workaround!
        return SymmetricBilinearSpace(self.__ambient_bilinear_space.base_field()
, self.__ambient_bilinear_space.gram_matrix())



    def gram_matrix(self, rational_matrix=False):
        """
    Returns the gram matrix with respect to the generators of the
    symmetric bilinear lattice.

    If rational_matrix is False then this matrix is defined over
    the base ring of the lattice, otherwise it is defined over the
    base field of the ambient symmetric bilinear space.

    """
        ## Generate the gram matrix
        B = self.generators()
        n = len(B)
        G = Matrix(self.__ambient_bilinear_space.base_field(), n, n)
        for i in range(n):
            for j in range(n):
                G[i,j] = self.__ambient_bilinear_space(B[i], B[j])

        ## Return the appropriate matrix type!
        if rational_matrix:
            return G
        else:
            return Matrix(self.base_ring(), G)




    def sum_with(self, other):
        """
    Find the sum of this lattice with the lattice L (in the same vector space).

    Here other can be a SymmetricBilinearLattice, or a list of vectors that coerce to the SymmetricBilinearSpace!

    TO DO: Allow other to be a Lattice as well.

    """
        ## Initialize from a list of vectors
        if isinstance(other, list):
            new_gens_list = other

        ## Initialize from a SymmetricBilinearSpace
        elif isinstance(other, SymmetricBilinearLattice):
                ## Check that both lattices live on the same ambient space
            if not self.ambient_space() == other.ambient_space():
                raise TypeError, "The two lattices live on different ambient spaces!"

            new_gens_list = other.generators()

        else:
            raise TypeError, "The type of other is presently not supported!"


        ## Find a basis for the sum of the two lattices
        M = Matrix(self.__ambient_bilinear_space.base_field(), self.generators()
 + new_gens_list)
        M_denom = LCM([denominator(M[i,j])  for i in range(M.nrows())  for j in
range(M.ncols())])
```



```
        M_int = Matrix(self.base_ring(), M * M_denom)
        B = [v / M_denom  for v in M_int.row_module().basis()]


        ## Return the lattice generated by generators of both lattices
        return SymmetricBilinearLattice(self.__ambient_bilinear_space, B)



################################################################################
############
##  the methods below are copied directly from the Lattice class
################################################################################
############




    def ambient_dimension(self):
        """
    Returns the ambient vector space that this lattice sits in.
    """
        return self.ambient_space().dim()


    def base_ring(self):
        """
    Returns the ring for which this is a module.
    """
        return deepcopy(self.__base_ring)



    def rank(self):
        """
    Determines if the lattice spans the ambient vector space.
    """
        return self.__lattice_module.rank()


    def is_full_rank(self):
        """
    Determines if the lattice spans the ambient vector space.
    """
        return self.rank() == self.ambient_dimension()


    def is_free(self):
        """
    Determines if the lattice has a (free) basis.
    """
        return True



    def intersect_with(self, other):
        """
    Find the intersection of this lattice with the lattice L (in the same vector space).
    """
        ## Check that both lattices live on the same ambient space
        if not self.ambient_space() == other.ambient_space():
            raise TypeError, "The two lattices live on different ambient spaces!"

        ## Return the intersection of the two quadratic latties
        intersection_basis = self.__lattice_free_module.intersection(other.__lat
tice_free_module).basis()
        return SymmetricBilinearLattice(self.__ambient_bilinear_space, intersect
ion_basis)
```

```
    def basis(self):
        """
    Returns the list of the basis vectors for the lattice, if the generators form a basis.
    """
        return self.__lattice_module.basis()


    def basis_matrix_of_columns(self):
        """
    Returns a matrix whose columns are the ordered basis for the lattice (if a basis exists).
    """
        return self.__lattice_module.basis_matrix().transpose()


    def basis_matrix_of_rows(self):
        """
    Returns a matrix whose rows are the ordered basis for the lattice (if a basis exists).
    """
        return self.__lattice_module.basis_matrix()



    def generators(self):
        """
    Returns the list of generating vectors for the lattice.
    """
        return self.__lattice_module.basis()


    def generator_matrix_as_columns(self):
        """
    Returns a matrix whose columns are the (ordered) generators for the lattice.
    """
        return self.__lattice_module.basis_matrix().transpose()


    def generator_matrix_as_rows(self):
        """
    Returns a matrix whose rows are the (ordered) generators for the lattice.
    """
        return self.__lattice_module.basis_matrix()



    def apply_linear_transformation_on_left(self, U):
        """
    Returns the lattice generated by the generators of this
    lattice (as a matrix of row vectors), after left-multiplying
    by the matrix U.
    """
        return SymmetricBilinearLattice(self.__ambient_bilinear_space, (U * self
.generator_matrix_as_rows()).rows())


    def apply_linear_transformation_on_right(self, U):
        """
    Returns the lattice generated by the generators of this
    lattice (as a matrix of column vectors), after right-multiplying
    by the matrix U.
    """
        return SymmetricBilinearLattice(self.__ambient_bilinear_space, (self.gen
erator_matrix_as_columns() * U).columns())
```

```
from sage.quadratic_forms.square_classes import SquareClass, local_squareclass_r
adius_val

from sage.functions.generalized import sgn

from sage.rings.arith import valuation, is_prime
from sage.rings.integer_ring import ZZ, crt_basis
from sage.rings.rational_field import QQ
from sage.rings.all import is_pAdicField


from sage.rings.infinity import Infinity

from sage.misc.misc import prod



def weak_approx_for_numbers_over_QQ(approx_list):
    """
  Find a rational number x (in QQ) satisfying:

     x = a (mod p^v)

  for all given triples (p, v, a), where

     – p is a prime number or Infinity,

     – v is an integer (possibly negative),

     – a is a rational number.

  If the valuation v at Infinity is 1, then the sign of x will be
  the same as the sign of a.  Allowed valuations at Infinity are
  either 0 or 1.


  #Takes a list of triples [prime, val, num], but where each prime is
  #replaced by its NumberFieldLocalization (which allows
  #non–archimedian congruences as well).

  INPUT:
     a list of triples [p, val, num] where:
        p –– a prime number or a (possibly archimedian) NumberFieldLocalization
        val –– a positive integer
        num –– a rational number (or a NumberFieldLocalization_element?)
     if p is Infinity, then val = 0 (no condition) or 1 (same sign as num).

  OUTPUT:
     an integer (or a number field element?)

  EXAMPLES:
     sage: x = weak_approx_for_numbers_over_QQ([[2, 1, 1/2], [3, 1, 3/7]]); x
     9/2
     sage: (x – ZZ(1)/ZZ(2)) % 2 == 0
     True
     sage: (x – ZZ(3)/ZZ(7)) % 3 == 0
     True

     sage: weak_approx_for_numbers_over_QQ([[2, 1, 1/2], [Infinity, 0, 123], [3, 1, 3/7]])
     9/2

     sage: weak_approx_for_numbers_over_QQ([[Infinity, 1, 123], [2, 1, 1/2], [3, 1, 3/7]])
     9/2

     sage: weak_approx_for_numbers_over_QQ([[Infinity, 1, –123], [2, 1, 1/2], [3, 1, 3/7]])
     –3/2
```



```
    """
    ## TO DO: Check that the approximation list is internally compatible (which
is always true if the moduli are all relatively prime!)


    ## Separate out the prime and archimedean conditions
    prime_approx_list = [L  for L in approx_list  if L[0] != Infinity]
    archimedean_approx_list = [L  for L in approx_list  if L[0] == Infinity]

    ## Determine the archimedean conditions
    arch_sign_flag = False
    if len(archimedean_approx_list) > 0:
        arch_sign_flag = (archimedean_approx_list[0][1] != 0)     ## Determine i
f there is a sign condition
        if arch_sign_flag:
            arch_sign = sgn(archimedean_approx_list[0][2])        ## If so, deci
de which sign we want


    #print "archimedean_approx_list = ", archimedean_approx_list
    #print "arch_sign_flag = ", arch_sign_flag

    ## Coerce to integers where we solve a harder CRT problem, keeping track of
the denominator to divide by later
    integer_approx_list = []
    total_denom = 1
    for triple in prime_approx_list:
        p = triple[0]
        old_val = triple[1]
        old_num = triple[2]

        ## Adjust the CRT problem to make it integral (but not necessarily over
ZZ yet)
        p_denom_val = max([0, -valuation(old_num, p)])
        p_denom_adjustment = p ** p_denom_val
        total_denom *= p_denom_adjustment
        integer_approx_list += [[p, old_val + p_denom_val, old_num * p_denom_adj
ustment]]


    ## Separate out the congruence information by numbers and prime-powers
    integer_list = []
    p_pow_list = []
    for triple in integer_approx_list:
        p_power = triple[0]**triple[1]
        integer_list.append(ZZ(triple[2] % p_power))        ## TO DO: Check that
 this works ok with p-adic integers as well.
        p_pow_list.append(p_power)

    ## Solve the CRT problem over ZZ
    E = crt_basis(p_pow_list)
    crt_modulus = prod(p_pow_list)
    global_int = sum([integer_list[i] * E[i]  for i in range(len(E))]) % crt_mod
ulus   ## This is always positive (>=0) by construction.

    #print "gloabl_int = ", global_int

    ## Adjust the sign to meet our condition at Infinity
    if arch_sign_flag:

        ## If the signs don't agree, then move by the crt_modulus in the directi
on we want the sign to be.
        if arch_sign < 0:
            global_int = global_int - crt_modulus

    #print "global_int = ", global_int
```

```
    ## Divide by the denominator.
    return QQ(global_int) / total_denom




def weak_approx_for_squareclasses_over_QQ(local_squareclass_list, ensure_uniquen
ess_flag=True):
    """
    Returns a (non-zero) squareclass over QQ which localizes to the
    specified squareclasses, and is only divisible by the primes of
    the specified squareclasses with odd valuation.

    If the ensure_uniqueness_flag is set, then this routine will raise
    an error if there is more than one rational squareclass satisfying
    the given local conditions.

    INPUT:
        local_squareclass_list -- a list of squareclasses over RR and
            various p-adic fields Q_p

    OUTPUT:
        a squareclass over QQ

    EXAMPLES:
        sage: S2 = SquareClass(Qp(2), 5)
        sage: S3 = SquareClass(Qp(3), 5/9)
        sage: S5 = SquareClass(Qp(5), 5*4)
        sage: S7 = SquareClass(Qp(7), 3)
        sage: weak_approx_for_squareclasses_over_QQ([S2, S3, S5, S7])
        The squareclass represented by 5 over Rational Field

        sage: weak_approx_for_squareclasses_over_QQ([S2, S3])    ## This fails since it cannot be a unit squareclass at al
l other places!
        Traceback (most recent call last):
        ....
        RuntimeError: We have not found any global squareclasses which localize to your list of local squareclasses, and
having valuation 0 aside from these.


    """
    ## SANITY CHECK: Check that the list isn't empty
    if len(local_squareclass_list) == 0:
        raise TypeError, "Oops!  You need to pass in at least one local squareclass to approximate!"

    ## WARNING: this doesn't check compatibility if there is a repeated prime, o
r even give necessarily the right answer in that case!

    ## Determine the primes dividing the squareclass, and generate a squarefree
representative t up to integer unit multiples
    t = prod([QS.base_field().prime()  for QS in local_squareclass_list  if is_p
AdicField(QS.base_field()) and QS.valuation() == 1])     ## QUESTION: Why doesn'
t the valuation condition automatically eliminate the real and complex places?!?


    ## Run through all possible units u to check if the rational squareclass u*t
 satisfies all local conditions!
    new_s_list = []
    for u in [1, -1]:
        localizations_work_flag = True
        new_s = SquareClass(QQ, u * t)
        for s in local_squareclass_list:

            ## Check if a localization fails to agree with the given list of squ
areclasses
```



```
            if s != SquareClass(s.base_field(), new_s.representative()):
                localizations_work_flag = False
                break

        ## If all localizations agree, then keep (or return) the rational square
class
        if localizations_work_flag:

            ## If the ensure_uniqueness flag isn't set, then return this localiz
ation
            if ensure_uniqueness_flag == False:
                return new_s
            else:
                new_s_list.append(new_s)


    ## Check that we have exactly one localization, in which case we return it
    if len(new_s_list) == 1:
        return new_s_list[0]
    elif len(new_s_list) == 0:
        raise RuntimeError, "We have not found any global squareclasses which localize to your list of lo
cal squareclasses, and having valuation 0 aside from these."
    else:
        raise RuntimeError, "We have found more than one global squareclass.  Please either unset the en
sure_uniqueness_flag or add more local conditions to specify your squareclass uniquely! "




def strong_approx_for_squareclasses_by_QQ_except_at_one_prime(local_squareclass_
list, ensure_uniqueness_flag=True, return_integral_representative=False):
    """
    Returns a (non-zero) squareclass over QQ which localizes to the
    specified squareclasses, and is only divisible by the primes of
    the specified squareclasses with odd valuation and at most one
    other prime number.

    If the ensure_uniqueness_flag is set, then this routine will raise
    an error if there is more than one rational squareclass satisfying
    the given local conditions.

    INPUT:
        local_squareclass_list -- a list of squareclasses over RR and
            various p-adic fields Q_p

    OUTPUT:
        a squareclass over QQ

    EXAMPLES:
        sage: from sage.quadratic_forms.weak_approx import strong_approx_for_squareclasses_by_QQ_except_at_one_p
rime

        sage: Sinf = SquareClass(RR, 5)
        sage: S2 = SquareClass(Qp(2), 5)
        sage: S3 = SquareClass(Qp(3), 5/9)
        sage: S5 = SquareClass(Qp(5), 5*4)
        sage: S7 = SquareClass(Qp(7), 3)
        sage: a = strong_approx_for_squareclasses_by_QQ_except_at_one_prime([Sinf, S2, S3, S5, S7], return_integral_r
epresentative=True); a
        12605
        sage: strong_approx_for_squareclasses_by_QQ_except_at_one_prime([Sinf, S2, S3, S5, S7])
        The squareclass represented by 12605 over Rational Field
        sage: SquareClass(RR, a) == Sinf
        True
        sage: SquareClass(Qp(2), a) == S2
        True
        sage: SquareClass(Qp(3), a) == S3
```

```
    True
    sage: SquareClass(Qp(5), a) == S5
    True
    sage: SquareClass(Qp(7), a) == S7
    True

  """
    ## SANITY CHECK: Check that the list isn't empty
    if len(local_squareclass_list) == 0:
        raise TypeError, "Oops!  You need to pass in at least one local squareclass to approximate!"

    ## WARNING: this doesn't check compatibility if there is a repeated prime, o
r even give necessarily the right answer in that case!


    #print "local_squareclass_list = ", local_squareclass_list



    ## Determine the primes dividing the squareclass, and generate a squarefree
representative t up to integer unit multiples
    t = 1
    for QS in local_squareclass_list:
        if QS.valuation() == 1:
            if is_pAdicField(QS.base_field()):
                t *= QS.base_field().prime()
            elif isinstance(QS.base_field(), RealField):
                t *= QS.normalized_representative()   ## This should be 1 or -1,
 since we're over RR here.
            else:
                raise TypeError, "We were given a squareclass not over the p-adics or RR."


    #print "t = ", t


    ## Determine the modulus and representatives to consider the new prime facto
r in,
    new_p_modulus_list = []
    new_p_repn_list = []
    for QS in local_squareclass_list:
        if is_pAdicField(QS.base_field()):
            p = QS.base_field().prime()
            new_p_repn_list.append( (QS * t).normalized_representative() )
            new_p_modulus_list.append(p**local_squareclass_radius_val(p))

    #print "new_p_modulus_list = ", new_p_modulus_list
    #print "new_p_repn_list = ", new_p_repn_list


    ## Use CRT to find the smallest positive representative for the given local
squareclasses (without p-divisibility)
    new_p_modulus = prod(new_p_modulus_list)
    X = crt_basis(new_p_modulus_list)
    tmp_p  = sum([X[i] * new_p_repn_list[i]  for i in range(len(X))]) % new_p_mo
dulus

    #print "tmp_p = ", tmp_p
    #print "new_p_modulus = ", new_p_modulus


    ## Increment tmp_p until it is prime  (since it's already prime to all given
 squareclass places by construction)
    while not (is_prime(tmp_p)):
        tmp_p += new_p_modulus

    #print "tmp_p = ", tmp_p
```



```
    ## Return the (integral) squareclass representative or squareclass
    if return_integral_representative:
        return t * tmp_p
    else:
        return SquareClass(QQ, t * tmp_p)
```