2c. SAGE Source Code for Enumeration of Class number one quadratic lattices over $\mathbb{Z}$ ($\sim$ 10,000 lines of Python code in 86 pages)

[Hang] – J. Hanke

(In progress)

```
print "Loading cn1_nonmaximal.sage (Class Number One Routines)"




#########################################################################
#############
## Find a lower bound for the p-mass of any p-power index sublattice of the give
n lattice  ##
#########################################################################
#############
def find_index_p_power_sublattice_minimal_p_mass_lower_bound(L, p, proper_sublat
tices_only=False):
    """
    Returns a lower bound for the p-mass of any p-power index sublattice
    of the quadratic lattice L.  This allows the index 1 case, where
    the lattice itself attains its minimal p-mass.


    TO DO:  WE NEED TO IMPROVE WHAT IS HAPPENING WHEN p = 2!



    INPUT:
        a p-primitive quadratic lattice (over ZZ) of rank >= 3

    OUTPUT:
        a positive rational number
    """
    ## Setup some basic information about the quadratic lattice
    n = len(L.basis())
    Q = L.quadratic_form__integral()

    ## SANITY CHECK:  Require at least 3 variables!
    if n <= 2:
        raise TypeError, "We need at least 3 variables to give a lower bound on
the p-mass for now!"


    ## List the (unscaled/unimodular) Jordan blocks by their scale power
    jordan_list = Q.jordan_blocks_by_scale_and_unimodular(p)


    ## FORCE OVERRIDE OF THE proper_sublattices_only FLAG -- GIVING A POSSIBLY W
ORSE BOUND!  -- CHANGE THIS!
    proper_sublattices_only = False


    ## Deal with the case of all sublattices:
    if not proper_sublattices_only:

        ## Find a lower bound for the "cross-product" factor
        if jordan_list[-1][1].dim() == 1:    ## Here we have 3 blocks in our min
imal p-mass representative
            p_top = p**jordan_list[-1][0]
            p_middle = p**jordan_list[-2][0]
            Q_rep = DiagonalQuadraticForm(ZZ, [1] + (n-2)*[p_middle] + [p_top])
        else:                       ## Here we have 2 blocks in our minimal p-mass
 representative
            p_top = p**jordan_list[-1][0]
            Q_rep = DiagonalQuadraticForm(ZZ, [1] + (n-1)*[p_top])
        p_cross_product_bound = p**(QQ(Q_rep.conway_cross_product_doubled_power(
p)) / 2)

        ## Find a lower bound for the "diagonal" factor
        if p != 2:
            p_diagonal_factor_bound = Q.conway_diagonal_factor(p)  ## The diagon
al factor doesn't increase when p is odd.
```



```
        else:
            p_diagonal_factor_bound = Q_rep.conway_diagonal_factor(p) / 4  ## Fo
r p=2 we use the minimal p-mass representative
            #raise NotImplementedError, "Fill in the diagonal factor bound when
p=2! ...."

        ## Find a lower bound for the "type" factor, and any possible species de
viations
        if p == 2:
            p_type_bound = QQ(2)**(-2*n)      ## Assume all factors are type II
for now.  (This can be improved!)
        else:
            p_type_bound = QQ(1)

    ## Deal with the case of proper sublattices only!
    else:
        raise NotImplementedError, "Do the case of only proper sublattices too!"


    ## Return the product of the minima
    p_mass_lower_bound = p_cross_product_bound * p_diagonal_factor_bound * p_typ
e_bound
    return p_mass_lower_bound




def lattice_is_CN1_mass_eligible_at_p(L, p, mass_upper_bound=QQ(1)/2):
    """
    We say that a lattice is "mass-eligible at p" if our bounds for the
    growth of the p-mass among all proper sublattices of prime-power index
    imply that the p-mass of some such sublattice may exceed the p-mass bound.

    This routine determines if a given lattice (possibly) has a proper
    p-power index sublattice of mass <= the given mass bound.
    """
    ## Find the lower bound for the mass (over all proper p-power index sublatti
ces!)
    Q = L.quadratic_form__integral().scale_by_factor(2)         ## Scale by 2 to
 ensure an integral Gram matrix (which doesn't change any p-mass)!
    p_power_index_mass_lower_bound = (Q.conway_mass() / Q.conway_p_mass(p)) \
        * find_index_p_power_sublattice_minimal_p_mass_lower_bound(L, p, proper_
sublattices_only=True)

    ## Return whether L is "mass eligible at p"
    return p_power_index_mass_lower_bound <= mass_upper_bound







#########################################################################
########
## Find all p-power index sublattices of a given quadratic lattice with mass <=
Bound ##
#########################################################################
########
def find_p_power_index_proper_sublattices_with_mass_bounded_above_by(L, p, mass_
upper_bound=QQ(1)/2):
    """
    Find all p-power index sublattices of the given quadratic lattice L
    with mass(L) <= mass_upper_bound.



    INPUT:
```

```
        L -- an integer-valued primitive quadratic lattice
        mass_upper_bound -- a positive number

    OUTPUT:
        a list of quadratic lattices
    """
    eligible_sublattice_list = []
    L_old_list = [L]
    L_new_eligible_list = ["Start"]

    ## Find all index p sublattices that are "mass-eligible at p", until we have
 them all
    while L_new_eligible_list != []:
        print "Starting to compute a new round of index " + str(p) + " sublattic
es."

        ## Make a list of all non-isometric index p sublattices of our list of r
elated lattices
        L_new_all_list = find_all_non_isometric_index_p_sublattices_of_list_of_r
elated_lattices(L_old_list, p)
        print "Found " + str(len(L_new_all_list)) + " index " + str(p) + " subla
ttices."

        ## Make a list of these that are mass-eligible at p
        L_new_eligible_list = [L_tmp  for L_tmp in L_new_all_list  if lattice_is
_CN1_mass_eligible_at_p(L_tmp, p)]
        print "Of these, " + str(len(L_new_eligible_list)) + " of them are mass-
eligible at p=" + str(p) + "."
        eligible_sublattice_list += L_old_list
        L_old_list = L_new_eligible_list


    ## Check our list to keep only the lattices with masses that are actually be
low the desired bound!
    print "Computing the masses of the " + str(len(eligible_sublattice_list)) +"
 mass-eligible lattices to see if they meet the bound."
    final_sublattice_list = [L  for L in eligible_sublattice_list  \
                             if L.quadratic_form__integral().conway_mass() <= ma
ss_upper_bound]
    print "Of these, " + str(len(final_sublattice_list)) + " of them satisfy the
 mass bound."

    ## Return the final list
    return final_sublattice_list




######################
## Find a maximal lattice is each quadratic space, and find all sublattices up t
o isometry for each scale bound.
######################
def find_all_non_isometric_index_p_sublattices_of_list_of_related_lattices__DEPR
ECATED(lattice_list, p):
    """
    List all primitive integer-valued quadratic sublattices (up to isometry) of
    the given list of quadratic lattices (with a common quadratic space),
    assuming that the list of lattices passed are all non-isometric
    sublattices of the same index in a single (primitive integer-valued)
    quadratic lattice.  We call such lattices "related".

    INPUT:
        a list of quadratic lattices, as described above

    OUTPUT:
        a list of related non-isometric quadratic lattices

    EXAMPLES:
```



```
        sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1, 17, 1/17]))
        sage: L = QS.maximal_quadratic_lattice()
        sage: L_index_2_list = find_all_non_isometric_index_p_sublattices_of_lis
t_of_related_lattices([L], 2)
        sage: len(L_index_2_list)
        3

    """

    ## Find a maximal (integer-valued) lattice in the quadratic space
    #L_max = QS.maximal_quadratic_lattice()
    #old_lattice_list = [L_max]

    ## Setup the lists to hold the new related lattices
    new_lattice_list_L = []
    new_lattice_list_Q = []

    ## For each lattice, make a list of sublattices of index p and find non-isom
etric representatives
    #print "lattice_list = " + str(lattice_list)
    for L_old in lattice_list:
        #print "Starting the main loop"
        tmp_list = find_prime_index_sublattices_up_to_isometry(L_old, p)
        #print "tmp_list = " + str(tmp_list)

        ## Add the new representatives to the new_lattice_list
        #tmp_new_list_L = []
        #tmp_new_list_Q = []

        ## Look for new lattices in our list of sublattices
        #print "Starting L_tmp loop"
        for L_tmp in tmp_list:
            #print "Inside L_tmp loop"
            Q_tmp = L_tmp.quadratic_form__integral()

            ## Determine if this sublattice is isometric to one we have already
found
            is_new_flag = True
            #print "Starting Q_known loop"
            #print "new_lattice_list_Q = " + str(new_lattice_list_Q)
            for Q_known in new_lattice_list_Q:
                #print "Inside Q_known loop"
                if Q_tmp.is_globally_equivalent_to(Q_known):
                    is_new_flag = False

            #print "Escaped the loop!"
            ## If it is new, add it to the list of non-isometric sublattices
            if is_new_flag:
                #print "Adding this lattice."
                new_lattice_list_L.append(L_tmp)
                new_lattice_list_Q.append(Q_tmp)
#                tmp_new_list_L += L_tmp
#                tmp_new_list_Q += Q_tmp

        ## Add this new batch of index p sublattices to the others
#        new_lattice_list_L += tmp_new_list_L
#        new_lattice_list_Q += tmp_new_list_Q

    ## Return the list of non-isometric index p sublattices
    #print "About to return values."
    return new_lattice_list_L



def find_non_isometric_sublist_of_quadratic_lattice_list(lattice_list):
    """
    Find the non-isometric lattices of a given quadratic lattices.

    Note: This output can be refined to return the indices
```

```
    of the first representatives of each isomorphism class.
    """
    ## Translate them into quadratic forms
    QF_list = [QuadraticForm(ZZ, L.Hessian_matrix())  for L in lattice_list]

    ## Break them up by their basic invariants and theta series
    det_list = [Q.det()  for Q in QF_list]
    level_list = [Q.level()  for Q in QF_list]
    theta_list = [Q.theta_series(20)  for Q in QF_list]  ## Use the first 20 the
ta series coefficients as a test for non-isometry

    ## Keep track of the lattice
    big_len = len(lattice_list)
    remaining_indices = range(big_len)
    noniso_indices = []
    for i in range(big_len):

        ## Check that this form has not yet been found.
        if i in remaining_indices:

            ## Save this new index and remove it from the eligible index list
            noniso_indices.append(i)
            remaining_indices.remove(i)

            ## Remove all quadratic forms isometric to this one
            for j in range(i+1, big_len):
                if j in remaining_indices:

                    ## Check if these forms are isometric -- if so, remove the l
atter one.
                    if (det_list[i] == det_list[j]) and (level_list[i] == level_
list[j]) \
                            and (theta_list[i] == theta_list[j]) and QF_list[i].
is_globally_equivalent__souvigner(QF_list[j]):
                        remaining_indices.remove(j)

    ## Make the list of non-isometric quadratic latices
    noniso_list = [lattice_list[k]  for k in noniso_indices]

    ## Return the list of non-isometric forms
    return noniso_list




def find_all_non_isometric_index_p_sublattices_of_list_of_related_lattices(latti
ce_list, p):
    """
    List all primitive integer-valued quadratic sublattices (up to isometry) of
    the given list of quadratic lattices (with a common quadratic space),
    assuming that the list of lattices passed are all non-isometric
    sublattices of the same index in a single (primitive integer-valued)
    quadratic lattice.  We call such lattices "related".

    INPUT:
        a list of quadratic lattices, as described above

    OUTPUT:
        a list of related non-isometric quadratic lattices

    EXAMPLES:
        sage: QS = QuadraticSpace(QQ, DiagonalQuadraticForm(QQ, [1, 17, 1/17]))
        sage: L = QS.maximal_quadratic_lattice()
        sage: L_index_2_list = find_all_non_isometric_index_p_sublattices_of_lis
t_of_related_lattices([L], 2)
        sage: len(L_index_2_list)
        3
```



```
    """
    sublattice_list = []

    ## Run over all index p sublattices of each lattice.
    for L in lattice_list:
        sublattice_list += find_fixed_index_sublattices_up_to_big_automorphisms(
L, p)


    ## Find the index p sublattices up to an automorphism of the big lattice
    return find_non_isometric_sublist_of_quadratic_lattice_list(sublattice_list)









    ## Return the list of sublattices that are "mass-eligible at p"

    ################
    ## DO THIS!!! ##
    ################




################################################################################
################################
## Find all sublattices of index p up to isometry under an isometry from the lar
ger lattice automorphism group ##
################################################################################
################################
def find_fixed_index_sublattices_up_to_big_automorphisms(L, sublattice_index):
    """
    Find all sublattices of index p up to isometry under an
    isometry from the larger lattice automorphism group.

    """
    ## Set the number of variables of the quadratic lattice
    n = len(L.basis())
    #print "n = " + str(n )


    ## Boilerplate GAP Code
    GAP_STR = """
    SHV:=ShortestVectorDutourVersion(GramMat);
    TheOption:="";
    GRP:=ArithmeticAutomorphismMatrixFamily_HackSouvignier_V2(TheOption, [GramMa
t], SHV);
    TheMod:=SublatticeIndex;

    Ans_list := GetSublattices(GramMat, GRP, TheMod);"""
    GAP_STR = GAP_STR.replace("SublatticeIndex", str(sublattice_index))


    ## Make a GAP even Gram matrix from a Quadratic Form Q
    gap_gram_list = [list(r)  for r in L.Hessian_matrix().rows()]
    #print "gap_gram_list = " + str(gap_gram_list)


    ## This is necessary to use the Polyhedral GAP code
    gap.eval('RequirePackage("polyhedral")');
```

```
    ## Find the sublattices in GAP (via polyhedral)
    gap.eval("GramMat:=" + str(gap_gram_list) + ";")


    ## Run the GAP code
    gap.eval(GAP_STR)

    ## Find the number of sublattices returned by GAP
    ns = ZZ(gap.eval("Length(Ans_list);"))


    ## PRINT THE ANSWER
    #print gap.eval("Ans_list;")


    ## Get the GAP Gram Matrices for these subforms (but they may be isometric s
till...)
    Subform_list = []
    Sublattice_list = []
    for i in range(ns):
        sage_sublattice_basis_raw = eval(gap.eval("Ans_list[" + str(i+1) + "].Ba
sis;"))
        sage_basis_as_rows = Matrix(ZZ, n, n, sage_sublattice_basis_raw)
        #print "sage_basis_as_rows = \n" + str(sage_basis_as_rows) + "\n"
        #L_new = L.apply_linear_transformation_on_right(sage_basis_as_rows)
         ## WHY DOESN'T THIS WORK???
        L_new = L.apply_linear_transformation_on_left(sage_basis_as_rows.transpo
se())
        #print "L_new = " + str(L_new)
        Sublattice_list.append(L_new)

        ## Testing that the gram matrices are recovered correctly from the subla
ttice bases
        ## ---------------------------------------------------------------------
-----------
        sage_gram_coeffs = eval(gap.eval("Ans_list[" + str(i+1) + "].GramMat;"))
        print "sage_gram_coeffs = " + str(sage_gram_coeffs)
        sage_QF = QuadraticForm(ZZ, Matrix(ZZ, n, n, sage_gram_coeffs))
        ##print sage_QF
        ##print sage_QF.theta_series(10)
        ##print "\n\n"
        #Subform_list.append(sage_QF)

        print "L_new.Hessian_matrix() = \n" + str(L_new.Hessian_matrix()) + "\n"
        print "sage_QF.Hessian_matrix() = \n" + str(sage_QF.Hessian_matrix()) +
"\n"

        print L_new.Hessian_matrix() == sage_QF.Hessian_matrix()
        #print "We have a correct Gram matrix?  " + str(L_new.quadratic_form__in
tegral().is_globally_equivalent_to(sage_QF))

        #print "\n -------------------- \n"


    return Sublattice_list

    ## Find the sublattices that are not isomorphic (test theta series, then use
 Souvigner)
```



```
##################################################
## Find all sublattices of index p up to isometry ##
##################################################
def find_prime_index_sublattices_up_to_isometry(L, p):
    """
    List all index p sublattices  of L up to isometry, where L is assumed to be
integer-valued!
    """
    ## Find a basis for the given lattice -- NOTE: THIS PRESENTLY ASSUMES THAT T
HE LATTCE IS FREE!
    B = L.basis()
    n = len(B)

    ## Make lists to store the quadratic lattices, and their associated integral
 quadratic forms
    L_list = []
    Q_list = []

    ## Run through all index p sublattices
    for v in normalized_finite_projective_space_generator(n, p):
        #print "\n\n ------------------------------------"
        #print "Using the vector v = " + str(v) + " in F_" + str(p)

        ## Find the matrix for the complementary hyperplane (orthogonal w.r.t. t
he standard dot product on (F_p)^n)
        M = Matrix(GF(p), v)
        M_perp = M.transpose().kernel().matrix()  ## Find the matrix of row vect
ors orthogonal to v over F_p

        ## Adjust this matrix to give rows generating our index p sublattice in
the standard basis
        M1 = M_perp.lift()
        M2 = Matrix(ZZ, n, n, p)
        M_std_sublattice = M1.stack(M2)   ## The rows of this generate the stand
ard sublattice of index p
        #print " The rows of this matrix gives the sublattice of index p in the
standard lattice:\n" + str(M_std_sublattice)

        ## Make the generators of our index p sublattice
        B_matrix = Matrix(QQ, B)             ## Basis of L as rows
        #print "The matrix of rows of this give a basis for L:\n" + str(B_matrix
)
        hyperplane_genenerators_row_matrix = M_std_sublattice * B_matrix
        #print " The rows of this matrix gives the sublattice of index p in our
lattice:\n" + str(hyperplane_genenerators_row_matrix)

        ## Find the quadratic sublattice of index p
        L_sub = QuadraticLattice(L.quadratic_space(), hyperplane_genenerators_ro
w_matrix.rows())
        Q_sub = L_sub.quadratic_form__integral()


        ## Determine if this sublattice is isometric to one we have already foun
d
        is_new_flag = True
        for Q_old in Q_list:
            if Q_sub.is_globally_equivalent_to(Q_old):
                is_new_flag = False
                #print "Found an isomorphism!"

        ## If it is new, add it to the list of non-isometric sublattices
        if is_new_flag:
            L_list.append(L_sub)
            Q_list.append(Q_sub)
            #print "Found a new sublattice (up to isometry)!"

    ## Return the list of non-isometric index p sublattices!
    return L_list
```

```
#############################################################################
############################
## Find the minimal p-mass adjustment factor for a non-unimodular lattice [in an
 odd-dimensional space!]  ##
#############################################################################
############################
def minimal_p_mass_adjustment_for_nonunimodular_lattice_of_odd_dimension(p, n):
    """
    Returns the minimal p-mass adjustment factor for a non-unimodular lattice at
 p.

    TO DO:  Fix this for p = 2!  THIS IS A REALLY BAD BOUND!
    """
    ## SANITY CHECK: Check that n is odd and n >= 3
    if not ((n >= 3) and (n % 2 == 1)):
        raise TypeError,  "The dimension n must be an odd integer >= 3."

    ## Return the minimal adjustment factor (over all non-deg. p-adic quadratic
spaces of odd dim n)
    n1 = ZZ(n-1) / 2
    if p == 2:
        return 1             ## FIX THIS!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!  REALLY!!!!!
        return 2 * prod([(1 - QQ(1)/(2**i))  for i in range(1, n1 + 1)]) * (QQ(1
)/8)**n
    else:
        return QQ(p**(n1) - 1)/2




#############################################################################
############################
## Find all quadratic spaces that may support a class number one (possibly non-m
aximal) lattice ##
#############################################################################
############################
def find_cn1_eligible_quadratic_spaces(n):
    """
    Assumes that the dimension n is odd for now!
    """
    #prime_scale_max_list = []

    ## State the relevant mass bound
    Mass_upper_bound = QQ(1)/2

    ## Make the generic product of this dimension (using the auxiliary quantity
n1)
    n1 = (n - 1) / 2
    generic_zeta_prod = 2**(1-n1) * prod([abs(zeta__exact(1-2*r))  for r in rang
e(1, n1+1)])
    #print "generic_zeta_prod = " + str(generic_zeta_prod)


    ## Find all primes that can support a non-generic (p | level) factor at p:
    ## ---------------------------------------------------------------------
    tmp_mass_product_min = generic_zeta_prod
```



```
    ## Be sure to include the p=2 adjustment cumulatively first, since it's goin
g to be < 1.
    eligible_small_prime_list = []               ## This is the list we really c
are about!
    eligible_small_p_mass_adjustment_list = []
    prime_small_flag = True
    p = 2
    while prime_small_flag:
        p_mass_adjustment = minimal_p_mass_adjustment_for_nonunimodular_lattice_
of_odd_dimension(p, n)
        if p_mass_adjustment <= 1:
            tmp_mass_product_min *= p_mass_adjustment
            eligible_small_prime_list.append(p)
            eligible_small_p_mass_adjustment_list.append(p_mass_adjustment)
            p = next_prime(p)
        else:
            prime_small_flag = False

    print " eligible_small_prime_list = " + str(eligible_small_prime_list)
    print " eligible_small_p_mass_adjustment_list = " + str(eligible_small_p_mas
s_adjustment_list)

    ## Deal with all adjustment factors >= 1, not cumulatively!
    eligible_big_prime_list = []                ## This is the list we really car
e about!
    eligible_big_p_mass_adjustment_list = []
    prime_too_big_flag = False
    while not prime_too_big_flag:
        if (tmp_mass_product_min * p_mass_adjustment) > Mass_upper_bound:
## This uses the fact that for p>2 the adjustment factor is an increasing fncn o
f p.
            prime_too_big_flag = True
        else:
            eligible_big_prime_list.append(p)
            eligible_big_p_mass_adjustment_list.append(p_mass_adjustment)
            p = next_prime(p)
            p_mass_adjustment = minimal_p_mass_adjustment_for_nonunimodular_latt
ice_of_odd_dimension(p, n)

    print " eligible_big_prime_list = " + str(eligible_big_prime_list)
    print " eligible_big_p_mass_adjustment_list = " + str(eligible_big_p_mass_ad
justment_list)


    ## Find the maximum number of big prime factors that we can take
    total_mass_prod_min = tmp_mass_product_min
    for i in range(len(eligible_big_prime_list)):
        if total_mass_prod_min > Mass_upper_bound:
            break
        else:
            total_mass_prod_min *= eligible_big_p_mass_adjustment_list[i]
    max_number_of_big_primes = i-1

    print "The maximal number of big primes we can use is " + str(max_number_of_
big_primes)


    ## Find the big (squarefree) conductors (where we have a non-generic local i
nvariant) for an eligible quadratic space
    import itertools
    big_squarefree_list = []
    for num_big_primes in range(1, max_number_of_big_primes + 1):
        for v in itertools.combinations(range(len(eligible_big_prime_list)), num
_big_primes):
            if tmp_mass_product_min * prod([eligible_big_p_mass_adjustment_list[
i]  for i in v]) <= Mass_upper_bound:
                t_big = prod([eligible_big_prime_list[i]  for i in v])
                big_squarefree_list.append(t_big)
```

```
    print " big_squarefree_list = " + str(big_squarefree_list)
    print " len(big_squarefree_list) = " + str(len(big_squarefree_list))


    ## Make all eligible quadratic conductors, and all global spaces that have t
his conductor
    eligible_quadratic_spaces = []
    small_squarefree_list = divisors(prod([p  for p in eligible_small_prime_list
]))
    for t_small in small_squarefree_list:
        for t_big in big_squarefree_list:
            t = t_small * t_big
            print "Making quadratic spaces of conductor t = " + str(t)

            QS_list = find_quadratic_spaces_supported_on_squareclass(n, SquareCl
ass(QQ, t), exact_conductor=True)
            eligible_quadratic_spaces += QS_list
            print "  Found quadratic spaces of conductor t = " + str(t)
            print


    print "We have found "  + str(len(eligible_quadratic_spaces)) + " rational q
uadratic spaces to consider for a CN1 lattice."


    ## Return the list of eligible quadratic spaces
    return eligible_quadratic_spaces




def find_class_number_one_lattices_for_quadratic_space_list(eligible_quadratic_s
paces):

    ## Find all class number one lattices in each eligible quadratic space
    big_cn1_list = []
    ind = 0
    for QS in eligible_quadratic_spaces:
        print "\n\n\n==================================\nChecking the i = " + st
r(ind) + " space."

        ## Find a maximal lattice in the space
        t = cputime()
        L_max = QS.maximal_quadratic_lattice()
        print "Found a maximal lattice."
        print " This took " + str(cputime(t)) + " seconds."

        ## Find the set S of primes we have to consider p-power sublattices for
        #S = []

        ## Iteratively find all prime-power index sublattices with mass <= 1/2,
for all primes!
        done_flag = False
        p = 1
        L_eligible_list = [L_max]
        L_new_list = []
        while not done_flag:
            p = next_prime(p)
            print "Looking for eligible proper sublattices of " + str(p) + "-pow
er index."
            t = cputime()
            L_new_list = [find_p_power_index_proper_sublattices_with_mass_bounde
d_above_by(L, p, QQ(1)/2)  for L in flatten(L_eligible_list)]
            print "Found " + str(len(L_new_list)) + " new elible proper sublatti
ces when p = " + str(p) + "."
            print " This took " + str(cputime(t)) + " seconds."
```



```
            ## Append the new list, and move on to the next prime
            L_eligible_list += L_new_list

            ## Check if we have no new sublattices (which for p >2 signals all m
asses are too big already!)
            if L_new_list == []:
                done_flag = True

        ## Check each of these lattices to see if it has class number one!
        print "We have a total of " + str(len(L_eligible_list)) + " eligible lat
tices to check."
        CN1_list_for_QS = [L  for L in L_eligible_list  if L.quadratic_form__int
egral().has_class_number_one()]
        print "Finishing the quadratic space, and found " + str(len(CN1_list_for
_QS)) + " distinct class number one lattices!"

        ## Append this list to the big list
        big_cn1_list.append(CN1_list_for_QS)

    ## Return the list of class number one lattices
    return big_cn1_list









################################################################################
##############################
## Find the eligible quadratic spaces with determinant squareclass supported on
(at most) the squareclass t ##
################################################################################
##############################
def find_quadratic_spaces_supported_on_squareclass(n, t, exact_conductor=False):
    """
    Returns a list of the (non-isometric) n-dimensional quadratic spaces support
ed
    on the squareclass T(F^*)^2.

    If exact_conductor==True then we require that at p | t either p | det or c_p
 = -1.

    NOTE: Presently only F = QQ is supported, an the space will be positive defi
nite!
    """

    ## List all eligible squareclasses based on their local invariants:
    ## -----------------------------------------------------------------
    ## For each non-generic place, add all eligible spaces
    QS_possible_list_by_place =[ [QuadraticSpace(DiagonalQuadraticForm(RR, n*[1]
))] ]
    nongeneric_prime_list = prime_divisors(t.normalized_representative() * 2)
    for p in nongeneric_prime_list:
        ## Deal with the exact or inexact conductor condition
        if not exact_conductor:
            QS_possible_list_by_place.append([local_quadratic_space_by_invariant
s(n,SquareClass(Qp(p),d),c)  \
                for d in local_squareclass_representatives_list(p)  for c in [1,
 -1]])
        else:
            QS_possible_list_by_place.append([local_quadratic_space_by_invariant
s(n,SquareClass(Qp(p),d),-1)  \
                for d in local_squareclass_representatives_list(p)])
            QS_possible_list_by_place.append([local_quadratic_space_by_invariant
s(n,SquareClass(Qp(p),d),1)  \
                for d in local_squareclass_representatives_list(p)  if d % p ==
```

```
0])

    ## Combine these in all possible ways to make a global space (after checking
 existence by invariants!)
    global_space_list = []
    num_of_places = len(QS_possible_list_by_place)
    index_range_vec = [len(QS_possible_list_by_place[i])  for i in range(num_of_
places)]
    for index_vec in mrange(index_range_vec):
        eligible_space_list = [QS_possible_list_by_place[i][index_vec[i]]  for i
 in range(num_of_places)]
        try:
            tmp_space = rational_quadratic_space_from_local_space_list(eligible_
space_list)
            global_space_list.append(tmp_space)
        except (ValueError):
            pass



    ## SANITY CHECK: Check that we have the correct number of global quadratic s
paces!       -- THINK ABOUT THIS!
    #if len(global_space_list) != 2**(num_of_places - 1) / 2:
    #    raise RuntimeError, "We have " + str(len(global_space_list)) +\
    #        "spaces, but " + str(num_of_places - 1) + "non-generic primes."


    ## Return the list of global spaces
    return global_space_list

















    #
    #
    ### Find all primes that can support a non-generic (p | level) factor
    #too_many_primes_flag = False
    #p = 1
    #tmp_mass_product = generic_zeta_prod
    #eligible_prime_list = []               ## This is the list we really care a
bout!
    #eligible_p_mass_adjustment_list = []
    #    while not too_many_primes_flag:
    #    p = next_prime(p)
    #    p_mass_adjustment = minimal_p_mass_adjustment_for_nonunimodular_lattice
_of_odd_dimension(p, n)
    #    tmp_mass_product *= p_mass_adjustment
    #    if tmp_mass_product > Mass_upper_bound:       ## This uses the fact tha
t for p>2 the adjustment factor is an increasing fncn of p.
    #        too_many_primes_flag = True
    #    else:
    #        eligible_prime_list.append(p)
    #        eligible_p_mass_adjustment_list.append(p_mass_adjustment)
    #
    #
    ### Find the possible squarefree t where the quadratic space could be non-ge
neric (i.e. c_p = -1 or p | D_p)
```



```
    #for p in
    #
    #
    #
    #
    #
    #
    #
    #
    #
    ### List all primes where a non-generic (non-unimodular) mass factor can exi
st in a cn1 mass-eligible quadratic space
    #eligible_primes = []
    #eligible_prime_p_mass_minima = []
    #p = 2
    #done_flag = False
    #while not done_flag:
    #    p = next_prime(p)
    #
    #    ## Find the target p-mass upper bound
    #    p_max_mass =
    #    p_mass_upper_bound = Mass_upper_bound / (generic_zeta_prod * nongeneric
_min_p_mass / generic_p_mass)
    #
    #
    #    ## Make a generic lattice to use for constructing the bound!  ----  THI
S NEEDS TO BE FIXED!
    #    Id_matrix = Matrix(QQ, n, n, 1)
    #    L = QuadraticLattice(QuadraticSpace(QQ, Id_matrix), Id_matrix.rows())
    #    nongeneric_p_mass_min = find_index_p_power_sublattice_minimal_p_mass_lo
wer_bound(L, p)
    #
    #    ## Store these p-masses until they became too large to be eligible
    #    if
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    ### Find the max_scale at p=2
    #s_2 = floor((ZZ(2) / ZZ(n-1)) * log(2**n / generic_zeta_prod, 2))
    #prime_scale_max_list.append(2**s_2)
    #prime_list.append(2)
    ##print "log(2**n / generic_zeta_prod, 2) = " + str(log(2**n / generic_zeta_
prod, 2))
    ##print "s_2 = " + str(s_2)
    #
    ### For each odd prime, determine the largest power of p that can occur as a
 Jordan scale
    #done_flag = False
    #p = 2
    #while not done_flag:
    #    p = next_prime(p)
    #    s_p = floor((ZZ(2) / ZZ(n-1)) * log(1/generic_zeta_prod, p))
    #    #print "p = " + str(p)
    #    #print "log(1/generic_zeta_prod, p) = " + str(log(1/generic_zeta_prod,
p))
    #    #print "s_p = " + str(s_p)
    #    if s_p <= 0:
    #        done_flag = True
    #    else:
    #        prime_scale_max_list.append(p**s_p)
    #        prime_list.append(p)
    #
    #
```

```
    ### Return the list of non-generic primes!
    #return prime_list
    #




#mrange_inc_strict(list_of_arguments):
#    """
#    Returns a strictly increasing list indices, which iterate
#    over a the elements of the given cartesian product of ranges.
#    """







## ============================================================================
============================
## ==================================== DEPRECATED CODE BELOW ===============
============================
## ============================================================================
============================



######################
## Find a maximal lattice is each quadratic space, and find all sublattices up t
o isometry for each scale bound.
######################
#def find_all_non_isometric_index_p_sublattices_of_list_of_mass_eligible_related
_lattices(lattice_list, p, p_mass_bound):
#    """
#    List all primitive integer-valued quadratic sublattices (up to isometry) of
#    the given list of quadratic lattices (with a common quadratic space),
#    assuming that the list of lattices passed are all non-isometric
#    sublattices of the same index in a single (primitive integer-valued)
#    quadratic lattice.  We call such lattices "related".
#
#    We say that a lattice is "mass-eligible at p" if our bounds for the
#    growth of the p-mass among all sublattices of prime-power index imply
#    that the p-mass of some such sublattice may exceed the p-mass bound.
#
#    INPUT:
#        a list of quadratic lattices, as described above
#
#    OUTPUT:
#        a list of related non-isometric quadratic lattices
#
#    EXAMPLES:
#
#    """
#    pass


#########################################################################
## Find all quadratic lattices of dimension n with class number one ##
#########################################################################
def DEPRECATED__find_class_number_one_lattices_of_dimension(n):
    """
    Finds all of the positive definite class number one
    lattices of dimension n over ZZ.

    ASSUMPTION: We're assuming that n is odd for now!
```



```
    """

    ## Find all eligible quadratic spaces (for such a lattice)
    nongeneric_primes = find_nongeneric_primes_for_cn1_spaces_of_odd_dimension(n
)
    t = prod(nongeneric_primes)
    eligible_spaces = find_quadratic_spaces_supported_on_squareclass(n, t)
    print "We have " + str(len(eligible_spaces)) + " eligible quadratic spaces t
o check for a class number one lattice."


    ## Find all class number one lattices in each eligible quadratic space
    big_cn1_list = []
    ind = 0
    for QS in eligible_spaces:
        print "\n\n\n==================================\nChecking the i = " + st
r(ind) + " space."

        ## Find a maximal lattice in the space
        L_max = QS.maximal_quadratic_lattice()
        print "Found a maximal lattice."

        ## Find the set S of primes we have to consider p-power sublattices for
        #S = []

        ## Iteratively find all prime-power index sublattices with mass <= 1/2,
for all primes!
        done_flag = False
        p = 1
        L_eligible_list = [L_max]
        L_new_list = []
        while not done_flag:
            print "Looking for eligible proper sublattices of " + str(p) + "-pow
er index."
            p = next_prime(p)
            L_new_list = [find_p_power_index_proper_sublattices_with_mass_bounde
d_above_by(L, p, QQ(1)/2)  for L in L_eligible_list]
            print "Found " + str(len(L_new_list)) + " elible proper sublattices
when p = " + str(p) + "."

            ## Append the new list, and move on to the next prime
            L_eligible_list += L_new_list

            ## Check if we have no new sublattices (which for p >2 signals all m
asses are too big already!)
            if L_new_list == []:
                done_flag = True

        ## Check each of these lattices to see if it has class number one!
        print "We have a total of " + str(len(L_eligible_list)) + " eligible lat
tices to check."
        CN1_list_for_QS = [L  for L in L_eligible_list  if L.quadratic_form__int
egral().has_class_number_one()]
        print "Finishing the quadratic space, and found " + str(len(CN1_list_for
_QS)) + " distinct class number one lattices!"

        ## Append this list to the big list
        big_cn1_list.append(CN1_list_for_QS)

    ## Return the list of class number one lattices
    return big_cn1_list




#########################################################################
######
```

```
## Find the primes where we have a non-generic mass factor (i.e. p = 2 or p | de
t)! ##
################################################################################
######
def find_nongeneric_primes_for_cn1_spaces_of_odd_dimension(n):
    """
    Return a list of primes on which the determinant squareclass of
    a quadratic space of dimension n supporting an integer-valued
    lattice of class number one is supported.

    ALGORITHM:
        1) Compute the generic mass factor, and then
            bound the Jordan scale a mass-eligible
            lattice at every prime using the p-mass
            bound developed in the routine:

                find_index_p_power_sublattice_minimal_p_mass_lower_bound(L, p)

        2) Check all products to determine the
            maximum number of primes that can
            appear in an eligible quadratic space.

        3)


    NOTE: We assume that n is odd for now.
    """
    prime_scale_max_list = []
    prime_list = []               ## This is the list we really care about!

    ## Make the generic product of this dimension (using the auxiliary quantity
n1)
    n1 = (n - 1) / 2
    generic_zeta_prod = 2**(1-n1) * prod([abs(zeta__exact(1-2*r))  for r in rang
e(1, n1+1)])
    #print "generic_zeta_prod = " + str(generic_zeta_prod)

    ## Find the max_scale at p=2
    s_2 = floor((ZZ(2) / ZZ(n-1)) * log(2**n / generic_zeta_prod, 2))
    prime_scale_max_list.append(2**s_2)
    prime_list.append(2)
    #print "log(2**n / generic_zeta_prod, 2) = " + str(log(2**n / generic_zeta_p
rod, 2))
    #print "s_2 = " + str(s_2)

    ## For each odd prime, determine the largest power of p that can occur as a
Jordan scale
    done_flag = False
    p = 2
    while not done_flag:
        p = next_prime(p)
        s_p = floor((ZZ(2) / ZZ(n-1)) * log(1/generic_zeta_prod, p))
        #print "p = " + str(p)
        #print "log(1/generic_zeta_prod, p) = " + str(log(1/generic_zeta_prod, p
))
        #print "s_p = " + str(s_p)
        if s_p <= 0:
            done_flag = True
        else:
            prime_scale_max_list.append(p**s_p)
            prime_list.append(p)


    ## Return the list of non-generic primes!
    return prime_list

    ## Return the list of prime scale maxima
    #return prime_scale_max_list
```

```
print "Loading cn1.sage (Class Number One Routines)"

from itertools import product


#################################################################
##                                                             ##
##  Routines to enumerate the class number 1 ternary forms.    ##
##                                                             ##
#################################################################

## Steps:
##
##     1) Find eligible spaces from the mass formula
##
##     2) Construct a rational quadratic space from each
##
##     3) Find a maximal lattice in this space.
##
##     4) Compute its class number.


## ======================================= STEP 1 ============================
========================


#############################################################################
###################
## Returns the lambda_v adjustment factors in the GHY Mass formula for orthogona
l groups over QQ ##
#############################################################################
###################
def GHY_orthog_lambda(p, dim, delta, w):
    """
    Returns the lambda_v adjustment factors in the GHY Mass formula
    for odd dim'l orthogonal groups over the rationals.

    INPUT:
        p -- a prime number > 0
        dim -- an integer > 0
        delta -- an integer
        w -- 1 or -1

    OUTPUT:
        a rational number

    EXAMPLES:


    """
    ## Sanity Checks
    assert dim >= 2
    assert dim in IntegerRing()
    assert sgn == 1 or sgn == -1

    ## Odd dim'l case
    if is_odd(dim):
        n = (dim - 1) // 2
        if w == -1  and  delta % p != 0:
            return QQ(p**(2*n) - 1) / (2 * (p -1))   ## Type 1
        else:
            return QQ(p**n + w) / 2                ## Type 2

    ## Even dim'l case
    if is_even(dim):
        n = dim // 2
        if sgn == -1  and  is_padic_square(delta, p):
            return QQ(p**n - 1) * (p**(n-1) - 1) / (2 * (p + 1))   ## Type 1
```



```
        elif sgn == -1  and  not is_padic_square(delta, p)  and  fundamental_dis
criminant(delta) % p != 0:
            return QQ(p**n + 1) * (p**(n-1) + 1) / (2 * (p + 1))   ## Type 2
        else:
            return QQ(1)/2                                         ## Type 3




def GHY_odd_orthog_lambda(p, dim, symbol):
    """
    Returns the lambda_v adjustment factors in the GHY Mass formula
    for odd dim'l orthogonal groups over the rationals.

    INPUT:
        p -- a prime number > 0
        dim -- an integer > 0
        symbol -- either 'I', 'II+', or 'II-'

    OUTPUT:
        a rational number

    EXAMPLES:


    """
    ## Sanity Checks
    if (dim <= 2) or not is_odd(dim):
        raise TypeError, "The dimension must be an odd number >= 3."


    ## Odd dim'l case
    n = (dim - 1) / 2
    if symbol == 'I':
        return QQ(p**(2*n) - 1) / (2 * (p + 1))   ## Type 1
    elif (symbol == 'II+'):
        return QQ(p**n + 1) / 2                   ## Type 2+
    elif (symbol == 'II-'):
        return QQ(p**n - 1) / 2                   ## Type 2-
    else:
        raise TypeError, "You must enter a symbol type of either 'I', 'II+', or
'II-'."



def GHY_even_orthog_lambda(p, dim, symbol):
    """
    Returns the lambda_v adjustment factors in the GHY Mass formula
    for even dim'l orthogonal groups over the rationals.

    INPUT:
        p -- a prime number > 0
        dim -- an integer > 0
        symbol -- either 'I', 'II', or 'III'

    OUTPUT:
        a rational number

    EXAMPLES:


    """
    ## Sanity Checks
    if (dim <= 2) or not is_even(dim):
        raise TypeError, "The dimension must be an even number >= 4."
```

```
    ## Even dim'l case
    n = dim / 2
    if symbol == 'I':
        return QQ(p**(n-1) - 1) * QQ(p**n - 1) / QQ(2 * (p + 1))   ## Type 1
    elif (symbol == 'II'):
        return QQ(p**(n-1) + 1) * QQ(p**n + 1) / QQ(2 * (p + 1))   ## Type 2
    elif (symbol == 'III'):
        return QQ(1) / 2                  ## Type 3
    else:
        raise TypeError, "You must enter a symbol type of either 'I', 'II', or '
III'."



#########################################################################
## Make a list of eligible (integral and rational) adjustment factors ##
#########################################################################
def eligible_odd_adjustment_factor_lists(dim, generic_factor, Mass_max=1):
    """
    Computes two lists (rational/integral) of eligible adjustment
    factors given the "generic factor" denominator.  These factors
    could (individually) occur in the GHY mass formula for a quadratic
    space supporting a class number one maximal lattice.

    INPUT:
        dim = dimension of the form -- an integer > 0
        generic_factor -- a rational number > 0
        Mass_max -- a rational number > 0  (optional)

    OUTPUT:
        a list of lists of the form [p, ['I', 3], ..., ['II+', 5] ]
        where p is a prime number and the sublists have two elements,
        the first of which is one of the strings 'I', 'II+', II-', and
        the second of which is a rational number.

    EXAMPLES:


    """
    ## Find the bound for eligible primes by size
    n = (dim - 1) / 2
    prime_bound = (QQ(2*Mass_max)/generic_factor + 1)**(QQ(1) / n)  ## This is b
ased on the smallest adjustment: (p^n - 1) / 2


    ## Compute a list of possible eligible adjustment types
    Big_adj_list = []    ## List of adjustment types [..., [p, ['I', 3], ['II+',
 5]], ...]
    p = 2
    while p <= max(2, prime_bound):      ## Makes sure that we always check p =
2!
        p_adj_list = [ p ]

        ## Check all possible adjustment types for p
        for adj_type_symbol in ['I', 'II+', 'II-']:
            lambda_num = GHY_odd_orthog_lambda(p, dim, adj_type_symbol)
            if (not lambda_num in ZZ) or (generic_factor * lambda_num <= Mass_ma
x):
                p_adj_list.append([adj_type_symbol, lambda_num])

        ## Append all eligible adjustment factors for p, and increment p
        if len(p_adj_list) != 1:
            Big_adj_list.append(p_adj_list)
        p = next_prime(p)

    ## Return the list of eligible adjustments
    return Big_adj_list
```



```
def eligible_even_adjustment_factor_lists__for_given_twist(dim, generic_factor,
twist_d, Mass_max=1):
    """
    Computes two lists (rational/integral) of eligible adjustment
    factors given the "generic factor" denominator.  These factors
    could (individually) occur in the GHY mass formula for a quadratic
    space supporting a class number one maximal lattice.

    Note: Since the dimension of the space is even, there is at
    most one mass adjustment type per prime!

    INPUT:
        dim = dimension of the form -- an integer > 0
        generic_factor -- a rational number > 0
        twist_d -- a fundamental discriminant for an imaginary quadratic field.
        Mass_max -- a rational number > 0  (optional)

    OUTPUT:
        a list of lists of the form [p, ['I', 3], ..., ['II+', 5] ]
        where p is a prime number and the sublists have two elements,
        the first of which is one of the strings 'I', 'II+', II-', and
        the second of which is a rational number.

    EXAMPLES:


    """
    ## Find the bound for eligible primes by size
    n = dim / 2
    prime_bound = (QQ(2*Mass_max)/generic_factor + 1)**(QQ(1) / n)  ## This is b
ased on the smallest (non-III) adjustment: (p^n - 1) / 2
    #print "Using n = " + str(n) + " and prime_bound = " + str(prime_bound)

    ## Compute a list of possible eligible adjustment types  -- Note that in the
 even dim'l case there is only one possibility per prime!
    Big_adj_list = []    ## List of adjustment types [..., [p, ['I', 3], ['II+',
 5]], ...]
    p = 2
    while p <= max(2, prime_bound):
        p_adj_list = [ p ]            ## Makes sure that we always check p = 2!

        ## Check which adjustment types are possible based on the splitting beha
vior of the local qudratic extension Q_p(\sqrt(twist_d))
        if SquareClass(Qp(p), twist_d).is_unit_squares():

            ## These have type I (or they are not adjustemnts!)
            adj_type_symbol = 'I'    ## This automatically has w=-1

        else:
            ## These must be adjustment factors!
            if  twist_d % p == 0:    ## Check if Q_p(\sqrt(d)).is_ramified(), si
nce we assume that d is a fundamental discriminant.
                adj_type_symbol = 'III'   ## This w=-1 or w=-1
            else:
                adj_type_symbol = 'II'    ## This automatically has w=-1

        ## Append all eligible adjustment factors for p, and increment p
        lambda_num = GHY_even_orthog_lambda(p, dim, adj_type_symbol)
        if (not lambda_num in ZZ) or (generic_factor * lambda_num <= Mass_max):

            p_adj_list.append([adj_type_symbol, lambda_num])

        ## Append all eligible adjustment factors for p, and increment p
        if len(p_adj_list) != 1:
            Big_adj_list.append(p_adj_list)
```

```
        p = next_prime(p)

    ## Return the list of eligible adjustments
    return Big_adj_list




#########################################################################
## Make a list of all possible odd dim'l class number 1 spaces over Q ##
#########################################################################
def eligible_odd_rational_GHY_spaces(dim, Mass_max=1):
    """
    Uses the Mass formula to find the odd dimensional eligible
    quadratic spaces (over Q) of dimension d for containing a
    lattice of class number 1.

    INPUT:
        dim -- an odd integer >= 3

    OUTPUT:
        A list of the form [mass, [p, adj_type, adj_factor], ... ]
    """
    ## Make the generic product of this dimension
    n = (dim - 1) / 2
    generic_zeta_prod = 2**(1-n) * prod([abs(zeta__exact(1-2*r))  for r in range
(1, n+1)])


    ## Compute a list of possible eligible adjustment types
    Big_adj_list = eligible_odd_adjustment_factor_lists(dim, generic_zeta_prod,
Mass_max)

    print "n = ", n
    print
    print " generic_zeta_prod = ", generic_zeta_prod
    print " Big_adj_list = ", Big_adj_list



    ## Compute the largest possible number of simultaneous adjustment factors (b
y size)
    minimal_adjustment_list = [min([entry[i][1]  for i in range(1, len(entry))])
  for entry in Big_adj_list]
    minimal_adjustment_list.sort()
    print "sorted minimal adjustment list = ", minimal_adjustment_list

    #tmp_min_mass = generic_zeta_prod
    #max_ind = -1
    #while (tmp_min_mass <= 1) and (max_ind < len(minimal_adjustment_list)):
    #    print "tmp_min_mass = ", tmp_min_mass
    #    max_ind += 1
    #    tmp_min_mass = tmp_min_mass * minimal_adjustment_list[max_ind]
    #
    #print "We can use at most " + str(max_ind) + " adjustment factors before th
e mass > 1."

    tmp_min_mass = generic_zeta_prod
    max_num_of_adjustments = 0
    while (tmp_min_mass <= 1) and (max_num_of_adjustments < len(minimal_adjustme
nt_list)):
        print "tmp_min_mass = ", tmp_min_mass
        tmp_min_mass = tmp_min_mass * minimal_adjustment_list[max_num_of_adjustm
ents]
        max_num_of_adjustments += 1
        print "  Finishing with max_num_of_adjustments = " + str(max_num_of_adju
stments)
```



```
    print "We can use at most " + str(max_num_of_adjustments) + " adjustment fac
tors before the mass > 1."



    ## Make a list of all eligible adjustment products:
    eligible_product_list = []
    #for prime_indices in powerset(range(len(Big_adj_list))):    ## Run through
all possible combinations of adjustment primes
    for r in range(max_num_of_adjustments):

        for prime_indices in increasing_indices_mrange_generator(r * [len(Big_ad
j_list)], '<'):  ## Run through all possible combinations of r adjustment primes


            for adjustment_indices_vec in mrange([len(Big_adj_list[i]) - 1 for i
 in prime_indices]):  ## Then run through all adjustment types (for each p)
                ## Make the associated mass
                tmp_mass = generic_zeta_prod
                tmp_mass *= prod([Big_adj_list[prime_indices[i]][adjustment_indi
ces_vec[i] + 1][1] \
                                        for i in range(len(prime_indices))])

                ## Test if it's eligible (by size, and for mass_max==1 by numera
tor being 1)
                if ((abs(numerator(tmp_mass)) == 1) or (Mass_max != 1)) \
                        and abs(tmp_mass) <= Mass_max:

                    ## Make the final entry for this eligible quadratic space
                    current_entry = [abs(tmp_mass)]
                    for i in range(len(prime_indices)):
                        current_entry.append([Big_adj_list[prime_indices[i]][0]]
 + Big_adj_list[prime_indices[i]][adjustment_indices_vec[i] + 1])

                    ## Append it to the eligible list
                    eligible_product_list.append(current_entry)



                #print " adjustment_indices_vec = ", adjustment_indices_vec

            #print " prime_indices = ", prime_indices

    #                ## DIAGNOSTICS
    #                if len(prime_indices) <= 2:
    #                    print " prime_indices = ", prime_indices
    #                    print " adjustment_indices_vec = ", adjustment_indices_
vec
    #                    print " current_entry = ", current_entry
    #                    print " tmp_mass = ", tmp_mass


    ## Return the eligible lists
    print " Done! =) "
    return eligible_product_list









#########################################################################
#########################################################################
#########################################################################

# The adjustment factors are larger than the twist information, so they rule out
```

```
 more spaces.  Sort by them first!
#   Unfortunately the even dim'l case gives a problem since we can get lots of 1
/2 factors, which puts us back into our twist sorting!




#########################################################################
## Make a list of all possible even dim'l class number 1 spaces over Q ##
#########################################################################
def eligible_even_rational_GHY_spaces(dim, Mass_max=1):
    """
    Uses the Mass formula to find the even dim'l eligible
    quadratic spaces (over Q) of dimension d for containing
    a lattice of class number 1.

    INPUT:
        dim -- an even integer >= 4

    OUTPUT:
        A list of the form [generic_mass, twist_d, [p, adj_type, adj_factor], ..
. ]
    """
    ## Make the generic zeta product of this dimension
    n = dim / 2
    generic_zeta_prod = 2**(1-n) * prod([abs(zeta__exact(1-2*r))  for r in range
(1, n)])
    print "Our dim = " + str(dim) + " gives n = " + str(n) + " has generic_zeta_
prod = " + str(RR(generic_zeta_prod))



    ## STEP 1 new:  Compute all eligible quadratic twists:
    ## ===================================================

    ## Compute the minimum adjustment < 1 to this generic zeta product
    if n == 2:
        minimal_adjustment = QQ(1)/2  ## This is 1/2 to allow p=2 to be Type I.
    else:
        minimal_adjustment = 1
    print "The minimal_adjustment factor from the non-generic places is " + str(
minimal_adjustment)

    ## Find all primes that can divide an eligible twist conductor.
    K = QQ(1) / (2 * generic_zeta_prod * minimal_adjustment)
    F_max = ceil(((K * ((2*pi)**n) * RR(zeta(n))) / (gamma__exact(n) * zeta__exa
ct(2*n)))**(QQ(2)/(2*n - 1)))
    print "F_max = " + str(F_max)
    #eligible_conductor_primes = prime_range(p_max + 1)
    twist_sign = (-1)**(n)  ## The sign of the twist is given by the parity of d
im/2
    #print "Our quadratic twists must have twist_sign = " + str(twist_sign) + "
and uses only primes p < p_max = " + str(F_max)
    print "Our quadratic twists must have twist_sign = " + str(twist_sign) + " a
nd uses only primes p < F_max = " + str(F_max)


    ## Find all eligible twists by squarefree representative
    F = lambda p : ((p**(deepcopy(n) - QQ(1)/2)) / 2)
    twist_t_repns = [t  for t in squarefree_where_increasing_multiplicative_func
tion_satisfies_bound(F, '<=', F_max)]
    print "twist_t_repns = " + str(twist_t_repns)
    eligible_twist_list = [fundamental_discriminant(twist_sign * t)  for t in tw
ist_t_repns]


    ### STEP 1:  Compute all eligible quadratic twists:
    ### ================================================
```



```
    #
    ### Compute the minimum adjustment < 1 to this generic zeta product
    #if n == 2:
    #    minimal_adjustment = QQ(1)/2  ## This is 1/2 to allow p=2 to be Type I.
    #else:
    #    minimal_adjustment = 1
    #print "The minimal_adjustment factor from the non-generic places is " + str
(minimal_adjustment)
    #
    #
    ### Find all primes that can divide an eligible twist conductor.
    #K = QQ(1) / (2 * generic_zeta_prod * minimal_adjustment)
    #p_max = ceil(((K * ((2*pi)**n) * RR(zeta(n))) / (gamma__exact(n) * zeta__ex
act(2*n)))**(QQ(2)/(2*n - 1)))
    #print "p_max = " + str(p_max)
    #eligible_conductor_primes = prime_range(p_max + 1)
    #twist_sign = (-1)**(n)  ## The sign of the twist is given by the parity of
dim/2
    #print "Our quadratic twists must have twist_sign = " + str(twist_sign) + "
and uses only primes p < p_max = " + str(p_max)
    #
    ### Make all possible twist discriminants, by incrementally using more prime
 factors:  (We can do this more efficiently by increasing the )
    ### ------------------------------------------------------------------------
---------
    #eligible_twist_list = []
    #t = 0
    #twists_done_flag = False
    #
    ### Loop over the number of prime factors in the fund disc of the quad chara
cter
    #while not twists_done_flag:
    #    eligible_twists_with_t_primes = []
    #    t += 1
    #    print "Trying to construct twists with " + str(t) + " primes in its con
ductor."
    #
    #    ## Make twists whose conductor has t prime factors, and increment the p
rimes one at a time.
    #    for cond_prime_indices in increasing_indices_mrange_generator(t * [len(
eligible_conductor_primes)], '<'):  ## Run through all possible combinations of
t conductor primes
    #        twist_d = fundamental_discriminant(twist_sign * prod([eligible_cond
uctor_primes[cond_prime_indices[i]] for i in range(len(cond_prime_indices))]))
    #        #print " Using the primes: " + str([eligible_conductor_primes[cond_
prime_indices[i]] for i in range(len(cond_prime_indices))])
    #        #print "Checking twist_d = " + str(twist_d)
    #
    #        ## Determine if the twist is eligible
    #        min_generic_prod = minimal_adjustment * generic_zeta_prod * abs(qua
dratic_L_function__exact(1-n, twist_d))  ## THIS NEEDS TO BE FIXED!!!
    #        if min_generic_prod <= 1/2:
    #            eligible_twists_with_t_primes.append(twist_d)
    #            print "Appended twist_d = " + str(twist_d)
    #
    #    ## Check if we got any new eligible twists  (which works because the tw
ist testing condition is an increasing function in the number of primes!)
    #    if eligible_twists_with_t_primes != []:
    #        eligible_twist_list += eligible_twists_with_t_primes
    #    else:
    #        twists_done_flag = True
    #    print "Found " + str(len(eligible_twists_with_t_primes)) + " eligible t
wists with t=" + str(t) + " primes in its conductor."



    print " We have " + str(len(eligible_twist_list)) + " (generic) quadratic tw
ists to consider."
```

```
    ## STEP 2:  Compute the eligible adjustment types:
    ## ================================================
    eligible_product_list = []

    ## Find eligible adjustment types for each eligible twist
    for twist_d in eligible_twist_list:
        generic_prod = generic_zeta_prod * abs(quadratic_L_function__exact(1-n,
twist_d))


        ## Compute the (unique) eligible adjustment type
        Big_adj_list = eligible_even_adjustment_factor_lists__for_given_twist(di
m, generic_prod, twist_d, Mass_max)
        print
        print " Checking twist_d = ", twist_d
        print " generic_prod = ", generic_prod
        print " Big_adj_list = ", Big_adj_list


        ## Compute the largest possible number of simultaneous adjustment factor
s (by size)
        minimal_adjustment_list = [min([entry[i][1]  for i in range(1, len(entry
))])  for entry in Big_adj_list]
        minimal_adjustment_list.sort()
        print "sorted minimal adjustment list = ", minimal_adjustment_list

        tmp_min_mass = generic_prod
        max_num_of_adjustments = 0
        while (tmp_min_mass <= 1) and (max_num_of_adjustments < len(minimal_adju
stment_list)):
            print "tmp_min_mass = ", tmp_min_mass
            tmp_min_mass = tmp_min_mass * minimal_adjustment_list[max_num_of_adj
ustments]
            max_num_of_adjustments += 1
            print "  Finishing with max_num_of_adjustments = " + str(max_num_of_
adjustments)

        print "We can use at most " + str(max_num_of_adjustments) + " adjustment
 factors before the mass > 1."



        ## Separate out the Type III adjustments from the Type I and II adjustme
nts
        Type_I_and_II_adjustments = []
        Type_III_adjustments = []
        for adj_type in Big_adj_list:
            if adj_type[1][0]== 'III':
                Type_III_adjustments.append(adj_type)
            else:
                Type_I_and_II_adjustments.append(adj_type)
        print "Type_I_and_II_adjustments = " + str(Type_I_and_II_adjustments)
        print "Type_III_adjustments = " + str(Type_III_adjustments)




        ## Loop over possible combinations of Type I and II eligible adjustment
types to find the eligible combinations
        for r in range(max_num_of_adjustments - len(Type_III_adjustments)):

            for Type_I_and_II_prime_indices in increasing_indices_mrange_generat
or(r * [len(Type_I_and_II_adjustments)], '<'):

                #print "Using Type_I_and_II_prime_indices = " + str(Type_I_and_I
I_prime_indices)
```



```
                ## Make the associated mass
                tmp_mass = generic_prod
                tmp_mass *= prod([adj_entry[1][1]  for adj_entry in Type_III_adj
ustments])    ## We could do his outside the loop, but it's not much savings.
                tmp_mass *= prod([Type_I_and_II_adjustments[j][1][1]  for j in T
ype_I_and_II_prime_indices])

                #print "Made the associated mass of " + str(tmp_mass)


                ## Test if it's eligible (by size, and for mass_max==1 by numera
tor being 1)
                if ((abs(numerator(tmp_mass)) == 1) or (Mass_max != 1)) \
                    and abs(tmp_mass) <= Mass_max:

                    print "Found an eligible entry."

                    ## Make the final entry for this eligible quadratic space
                    current_entry = [abs(tmp_mass), twist_d]
                    for j in Type_I_and_II_prime_indices:
                        adj_entry = Type_I_and_II_adjustments[j]
                        current_entry.append([adj_entry[0]] + adj_entry[1])
                    for adj_entry in Type_III_adjustments:
                        current_entry.append([adj_entry[0]] + adj_entry[1])

                    ## Append it to the eligible list
                    print "  current_entry = " + str(current_entry)
                    print "Adding this eligible entry to the list."
                    eligible_product_list.append(current_entry)



        #                ## DIAGNOSTICS
        #                if len(prime_indices) <= 2:
        #                    print " prime_indices = ", prime_indices
        #                    print " adjustment_indices_vec = ", adjustment_indi
ces_vec
        #                    print " current_entry = ", current_entry
        #                    print " tmp_mass = ", tmp_mass


    ## Return the eligible lists
    print " Done! =) "
    return eligible_product_list







## ======================================= STEP 2 ============================
========================


################################################################################
##############################
## Find a list of all rational quadratic spaces which satisfy a given GHY local
type (from the mass formula) ##
################################################################################
##############################
def rational_spaces_from_odd_dim_GHY_mass_type(n, local_mass_type_list_unsorted)
:
    """
    Find a list of all rational quadratic spaces which satisfy a given
    GHY local type (from the mass formula).

    NOTE: We should produce at most one (positive definite) rational
    space from each mass type, since if the Hasse invariant condition
```

```
    is satisfied then the discriminant is forced up to sign by the
    mass type, and the sign is then forced by the positive definite
    condition.

    INPUT:
        n -- positive integer >= 3
        local_mass_type_list_unsorted -- a list of pairs [p, type]
            which have the exceptional local mass type "type" at the
            prime p.  This actually accepts lists of lists [p, type,
            ...] and only looks at the first two entries.


    OUTPUT:
        a list of rational quadratic spaces

    """
    ## Sort the local mass type list
    local_mass_type_list = deepcopy(local_mass_type_list_unsorted)
    local_mass_type_list.sort()


    #print "mass_type_list = ", local_mass_type_list


    ## Odd-dimensional cases
    if is_odd(n):

        ## Step 1: Check the mass type for compatibility with the product formul
a
        #w_product = 1
        #for p_mass_type in local_mass_type_list:
        #    if (p_mass_type[1] == "I") or (p_mass_type[1] == "II-"):
        #        w_product *= -1

        ## If we're not compatible, then return the empty list.
        #if w_product != 1:
        #    return []

        #print "Finished Step 1 -- the product formula checks out."


        ## Step 2: Find the determinant squareclass (which determines a unique q
uadratic space / QQ).
        d_product = 1
        for p_mass_type in local_mass_type_list:
            if (p_mass_type[1] == "II+") or (p_mass_type[1] == "II-"):
                d_product *= p_mass_type[0]


        #print "Finished Step 2 -- we can find a rational squareclass for the de
terminant -- it's ", d_product



        ## Step 3: Construct the quadratic space associated to this mass-type:
        ## ----------------------------------------------------------------------

        ## Initialize the list with a positive definite space at Infinity (i.e.
at RR)!
        local_space_list_by_place = [ [QuadraticSpace(DiagonalQuadraticForm(RR,
n*[1]))] ]


        ## If p = 2 has no mass-type (i.e. it's a generic mass type), be sure to
 add it to the list as a generic mass type!
        local_mass_type_list_primes = [LMT[0]  for LMT in local_mass_type_list]

        if not 2 in local_mass_type_list_primes:
            local_mass_type_list_with_2 = [[2, 'Generic']] + local_mass_type_lis
```



```
t
        else:
            local_mass_type_list_with_2 = local_mass_type_list

        ## Construct a list of all possible local spaces for each prime
        for pair in local_mass_type_list_with_2:
            #print " pair = ", pair
            #print
            p = pair[0]
            T = pair[1]


            ## Sanity Check: p is a prime
            if not is_prime(p):
                raise TypeError, "The local mass-type " + str(pair) + " was used
, where the first entry is not a prime."


            ## Find the possible GHY invariants for each mass type
            if T == 'Generic':    ## Special type to allow p=2 to be included in
 the enumeration even if it isn't exceptional in the GHY sense.
                w = 1
                delta_list = [s  for s in local_squareclass_representatives_list
(p)  if s % p != 0]
            elif T == 'I':
                w = -1
                delta_list = [s  for s in local_squareclass_representatives_list
(p)  if s % p != 0]
            elif T == 'II+':
                w = 1
                delta_list = [s  for s in local_squareclass_representatives_list
(p)  if s % p == 0]
            elif T == 'II-':
                w = -1
                delta_list = [s  for s in local_squareclass_representatives_list
(p)  if s % p == 0]
            else:
                raise TypeError, "Oops!, The GHY mass type " + str(T) + " is not
 recognized!"


            ## Generate the associated local quadratic spaces (at a given place)
            tmp_list_at_place = []
            for delta in delta_list:
                n1, d, c = local_quadratic_space_GHY_to_Standard_invariants(n, S
quareClass(Qp(p), delta), w)
                tmp_list_at_place.append(local_quadratic_space_by_invariants(n1,
 d, c))
                #print "Constructed at p = " + str(p) + " a local quadratic spac
e with invariants (n1, d, c) = " + str((n1, d, c))

            ## Check that we have at least one possible local space for each mas
s type!
            if tmp_list_at_place == []:
                raise RuntimeError, "The local mass-type " + str(pair) + " for d
im " + str(n) + " genereated no local quadratic spaces!"

            ## Append the list of local spaces at p to the big list (for all pla
ces)
            local_space_list_by_place.append(tmp_list_at_place)

            #print " The local spaces at p = ", p, " are ", tmp_list_at_place


        ## Step 4: Run through all combinations of local spaces and generate a r
ational space if possible
        rational_space_list = []
        for local_quadratic_list in product(*local_space_list_by_place):
```

```
            ## Check that the product of the Hasse invariants c_p is 1  (Note: t
he local quadratic list includes p=2, so it has all p where c_p is -1.)
            c_prod = prod([local_space.hasse_invariant()  for local_space in loc
al_quadratic_list])
            if c_prod == 1:

                ## Generate the rational quadratic space (if possible) and add i
t to the list
                tmp_space = None
                try:
                    #print "local_quadratic_list = ", local_quadratic_list
                    tmp_space = rational_quadratic_space_from_local_space_list(l
ocal_quadratic_list)
                    rational_space_list.append(tmp_space)
                except (ValueError):  ## ValueError comes from find_locally_repr
esented_number() or rational_quadratic_space_from_local_space_list() when imposs
ible invariants are passed.
                    pass

                ## SANTIY CHECK: Check that the mass type of the space we produc
ed matches what we asked for
                if (tmp_space != None)  and (local_mass_type_list != GHY_mass_ty
pe_list_from_odd_dim_definite_quadratic_space(tmp_space)):
                    raise RuntimeError, "Somehow we produced a rational space wi
th GHY mass-type " +  str(GHY_mass_type_list_from_odd_dim_definite_quadratic_spa
ce(tmp_space)) + ", but we wanted one with mass-type " + str(local_mass_type_lis
t) + "."

                #print "tmp_space = ", tmp_space


        ## Step 5: Return the list of eligible rational spaces
        return rational_space_list









def rational_spaces_from_even_dim_GHY_mass_type(n, delta, local_mass_type_list_u
nsorted):
    """
    Find a list of all rational quadratic spaces which satisfy a given
    GHY local type (from the mass formula).

    NOTE: We should produce at most one (positive definite) rational
    space from each mass type, since if the Hasse invariant condition
    is satisfied then the discriminant is forced up to sign by the
    mass type, and the sign is then forced by the positive definite
    condition.

    INPUT:
        n -- positive integer >= 3
        delta -- an integer representing a quadratic extension
        local_mass_type_list_unsorted -- a list of pairs [p, type]
            which have the exceptional local mass type "type" at the
            prime p.  This actually accepts lists of lists [p, type,
            ...] and only looks at the first two entries.


    OUTPUT:
        a list of rational quadratic spaces

    """
    ## Sort the local mass type list
    local_mass_type_list = deepcopy(local_mass_type_list_unsorted)
```



```
    local_mass_type_list.sort()


    #print "mass_type_list = ", local_mass_type_list


    ## Even dimensional cases
    if is_even(n):

        ## Step 1: Check the mass type for compatibility with the product formul
a
        #w_product = 1
        #for p_mass_type in local_mass_type_list:
        #    if (p_mass_type[1] == "I") or (p_mass_type[1] == "II-"):
        #        w_product *= -1

        ## If we're not compatible, then return the empty list.
        #if w_product != 1:
        #    return []

        #print "Finished Step 1 -- the product formula checks out."


        ## Step 2: Find the determinant squareclass (which determines a unique q
uadratic space / QQ).
        #d_product = delta


        #print "Finished Step 2 -- we can find a rational squareclass for the de
terminant -- it's ", d_product



        ## Step 3: Construct the quadratic space associated to this mass-type:
        ## ----------------------------------------------------------------------

        ## Initialize the list with a positive definite space at Infinity (i.e.
at RR)!
        local_space_list_by_place = [ [QuadraticSpace(DiagonalQuadraticForm(RR,
n*[1]))] ]


        ## If p = 2 has no mass-type (i.e. it's a generic mass type), be sure to
 add it to the list as a generic mass type!
        local_mass_type_list_primes = [LMT[0]  for LMT in local_mass_type_list]

        if not 2 in local_mass_type_list_primes:
            local_mass_type_list_with_2 = [[2, 'Generic']] + local_mass_type_lis
t
        else:
            local_mass_type_list_with_2 = local_mass_type_list

        ## Construct a list of all possible local spaces for each prime
        for pair in local_mass_type_list_with_2:
            #print " pair = ", pair
            #print
            p = pair[0]
            T = pair[1]


            ## Sanity Check: p is a prime
            if not is_prime(p):
                raise TypeError, "The local mass-type " + str(pair) + " was used
, where the first entry is not a prime."


            ## Find the possible GHY invariants for each mass type
            if T == 'Generic':    ## Special type to allow p=2 to be included in
 the enumeration even if it isn't exceptional in the GHY sense.
```

```
                w_list = [1]
            elif T == 'I':
                w_list = [-1]
            elif T == 'II':
                w_list = [-1]
            elif T == 'III':
                w_list = [1, -1]
            else:
                raise TypeError, "Oops!, The GHY mass type " + str(T) + " is not
 recognized!"


            ## Generate the associated local quadratic spaces (at a given place)
            tmp_list_at_place = []
            for w in w_list:
                n1, d, c = local_quadratic_space_GHY_to_Standard_invariants(n, S
quareClass(Qp(p), delta), w)
                #print "Using the standard local invariants (n1,d,c) = " + str((
n1, d, c))
                tmp_list_at_place.append(local_quadratic_space_by_invariants(n1,
 d, c))
                #print "Constructed at p = " + str(p) + " a local quadratic spac
e with invariants (n1, d, c) = " + str((n1, d, c))

            ## Check that we have at least one possible local space for each mas
s type!
            if tmp_list_at_place == []:
                raise RuntimeError, "The local mass-type " + str(pair) + " for d
im " + str(n) + " genereated no local quadratic spaces!"

            ## Append the list of local spaces at p to the big list (for all pla
ces)
            local_space_list_by_place.append(tmp_list_at_place)

            #print " The local spaces at p = ", p, " are ", tmp_list_at_place


        ## Step 4: Run through all combinations of local spaces and generate a r
ational space if possible
        rational_space_list = []
        for local_quadratic_list in product(*local_space_list_by_place):

            ## Check that the product of the Hasse invariants c_p is 1  (Note: t
he local quadratic list includes p=2, so it has all p where c_p is -1.)
            c_prod = prod([local_space.hasse_invariant()  for local_space in loc
al_quadratic_list])
            if c_prod == 1:

                ## Generate the rational quadratic space (if possible) and add i
t to the list
                tmp_space = None
                try:
                    #print "local_quadratic_list = ", local_quadratic_list
                    tmp_space = rational_quadratic_space_from_local_space_list(l
ocal_quadratic_list)
                    rational_space_list.append(tmp_space)
                except (ValueError):  ## ValueError comes from find_locally_repr
esented_number() or rational_quadratic_space_from_local_space_list() when imposs
ible invariants are passed.
                    pass

                ## SANTIY CHECK: Check that the mass type of the space we produc
ed matches what we asked for
                if (tmp_space != None)  and (local_mass_type_list != GHY_mass_ty
pe_list_from_even_dim_definite_quadratic_space(tmp_space)):
                    print "Failed with tmp_space = " + str(tmp_space)
                    raise RuntimeError, "Somehow we produced a rational space wi
th GHY mass-type " +  str(GHY_mass_type_list_from_even_dim_definite_quadratic_sp
ace(tmp_space)) + ", but we wanted one with mass-type " + str(local_mass_type_li
```



```
st) + "."

                #print "tmp_space = ", tmp_space


        ## Step 5: Return the list of eligible rational spaces
        return rational_space_list



    ## ====================================================================
====================




def GHY_mass_type_list_from_odd_dim_definite_quadratic_space(QS):
    """
    Compute the GHY mass-type list for a maximal lattice on the
    quadratic space QS.

    INPUT:
        QS -- a quadratic space defined over QQ

    OUTPUT:
        a list of lists [local mass, prime, local mass-type]

    EXAMPLES:


    """
    ## Determine if QS is a local or global space... (assume it to be global for
 now)



    ## Determine the finite list of places to consider (making sure to append 2
if it's not there)
    eligible_bad_primes = list(Set([2] + QS.local_characteristic_primes_of_QQ())
)
    eligible_bad_primes.sort()


    ## Run through each of them looking for the GHY mass type
    n = QS.dim()
    mass_type_list = []
    for p in eligible_bad_primes:
        QS_p = QS.localize_at_place(p)
        n1, delta, w = local_quadratic_space_Standard_to_GHY_invariants(n, QS_p.
determinant(), QS_p.hasse_invariant())

        if delta.valuation() == 0:
            if (w == 1):
                pass          ## Do nothing, since this is the generic case!
                #print "Generic GHY Mass-type for p = ", p
            else:
                mass_type_list.append([p, 'I'])
        else:
            if (w == 1):
                mass_type_list.append([p, 'II+'])
            else:
                mass_type_list.append([p, 'II-'])

    ## Return the GHY mass-type list
    return mass_type_list
```

```
def GHY_mass_type_list_from_even_dim_definite_quadratic_space(QS):
    """
    Compute the GHY mass-type list for a maximal lattice on the
    quadratic space QS.

    INPUT:
        QS -- a quadratic space defined over QQ

    OUTPUT:
        a list of lists [local mass, prime, local mass-type]

    EXAMPLES:


    """
    ## Determine if QS is a local or global space... (assume it to be global for
 now)



    ## Determine the finite list of places to consider (making sure to append 2
if it's not there)
    eligible_bad_primes = list(Set([2] + QS.local_characteristic_primes_of_QQ())
)
    eligible_bad_primes.sort()


    ## Run through each of them looking for the GHY mass type
    n = QS.dim()
    mass_type_list = []
    for p in eligible_bad_primes:
        QS_p = QS.localize_at_place(p)
        n1, delta, w = local_quadratic_space_Standard_to_GHY_invariants(n, QS_p.
determinant(), QS_p.hasse_invariant())

        if delta.is_unit_squares():
            if (w == 1):
                pass          ## Do nothing, since this is the generic case!
                #print "Generic GHY Mass-type for p = ", p
            else:
                mass_type_list.append([p, 'I'])
        else:
            #print "delta = " + str(delta)
            #print "delta valuation() = " + str(delta.valuation())
            ## Test if the local quadratic extension is unramified
            if ((p > 2) and (delta.valuation() == 0)) or ((p == 2) and (delta ==
 SquareClass(Qp(2), 5))):
                if (w == -1):
                    mass_type_list.append([p, 'II'])
            else:
                mass_type_list.append([p, 'III'])

    ## Return the GHY mass-type list
    return mass_type_list




##############################################
## Find all class number one maximal lattices ##
##############################################
def maximal_CN1_quadratic_lattices_of_dim(n):
    """
    List the integer-valued quadratic forms of
    dimension n arising from a maximal lattice,
    for any n > 2.
```



```
    INPUT:
        n -- an integer

    OUTPUT:
        a list of quadratic forms

    EXAMPLES:

    """
    ## SANITY CHECK: n >= 3
    if not n >= 3:
        raise RuntimeError, "This only computes quadratic forms in 3 or more var
iables!"


    ## Find all eligible rational spaces:
    ## ----------------------------------
    if is_odd(n):
        ## Find the GHY mass types of eligible quadratic spaces
        E = eligible_odd_rational_GHY_spaces(n)

        ## Extract the adjustment factors
        E_types = []
        for i in range(len(E)):
            E_types.append([LT[:2]  for LT in E[i][1:]])

        ## Find the GHY mass types of eligible quadratic spaces
        time R = [rational_spaces_from_odd_dim_GHY_mass_type(n, E_types[i]) for
i in range(len(E_types))]

    else:
        ## Find the GHY mass types of eligible quadratic spaces
        E = eligible_even_rational_GHY_spaces(n)

        ## Extract the adjustment factors
        E_types = []
        E_deltas = []
        for i in range(len(E)):
            E_deltas.append(E[i][1])
            E_types.append([LT[:2]  for LT in E[i][2:]])

        ## Find the GHY mass types of eligible quadratic spaces
        time R = [rational_spaces_from_even_dim_GHY_mass_type(n, E_deltas[i], E_
types[i]) for i in range(len(E_types))]



    ## Find the indices that produce a quadratic space
    R_nz_indices = [i  for i in range(len(R))  if R[i] != []]
    #len(R_nz_indices)


    ## Check that all of the spaces we produce are non-isomorphic
    F = flatten(R)
    I = find_distinct_quadratic_space_indices_in_list(F)
    #print "len(F) =", len(F)
    #print "len(I) =", len(I)
    #len(F) == len(I)

    ## Write these spaces to a list:
    #QF_file = open("/Users/jonhanke/Documents/SAGE/my_new_" + str(n) + "ary_CN1
_QF_space_forms.sage", 'w')
    #QF_file.write("new_eligible_4ary_QS_space_list = [ \n")
    #for i in range(len(I)):
    #    QS = F[i]
    #    QF_file.write("QuadraticSpace(QQ, Matrix(QQ,n,n,")
    #    QF_file.write(str(QS.gram_matrix().list()))
    #    QF_file.write("))")
```

```
    #    if (i != len(I) - 1):
    #        QF_file.write(", \n")
    #QF_file.write("] \n")
    #QF_file.close()

    ## Make a list of the eligible rational quadratic spaces
    #load("/Users/jonhanke/Documents/SAGE/my_new_4ary_CN1_QF_space_forms.sage")
    #Q4 = new_eligible_4ary_QS_space_list
    #len(Q4)


    ## Make the list of distinct eligible quadratic spaces
    Q_list = [F[i]  for i in I]

    ## Find the list of maximal lattices
    time L_list = [QS.maximal_quadratic_lattice()  for QS in Q_list]

    ## Check which of these spaces has a class number one maximal lattice
    CN1_maximal_space_indices = []
    CN1_maximal_lattices = []

    ## CN1 maximal lattice finding code
    i=0
    for L in L_list:
        print "-------------"
        print "i = " + str(i)
        print L.Hessian_matrix()
        if L.quadratic_form__integral().has_class_number_one():
            print "Has class number one! =)"
            CN1_maximal_space_indices.append(i)
            CN1_maximal_lattices.append(L)
        i += 1

    ## How many CN1 rank 4 maximal lattices do we get?
    print "We have " + str(len(CN1_maximal_lattices)) + " maximal lattices in "
+ str(n) + " variables with class number one."

    #My_CN1_det_list = [L.Hessian_matrix().det()  for L in CN1_maximal_lattices]
    #My_CN1_det_list.sort()
    #My_CN1_det_list

    ## Return the list of maximal lattices
    return CN1_maximal_lattices



def store_maximal_lattice_lists_for_dimensions(dim_list):
    """
    Compute and store the maximal lattice lists for
    the dimensions listed in dim_list.

    INPUT:
        dim_list -- a list of integers >= 3.

    OUTPUT:
        none
    """
    DIR_String = "/Users/jonhanke/Documents/SAGE/"
    for d in dim_list:
        L_list = maximal_CN1_quadratic_lattices_of_dim(d)
        save(L_list, DIR_String + "L" + str(d) + "_maximal_list.sobj")




## ======================================== STEP 3 ============================
========================
```



```


## ======================================== STEP 4 ============================
========================



## ========================================  MISC  ============================
========================



def squarefree_where_increasing_multiplicative_function_satisfies_bound(F, rule,
 bound):
    """
    Iterate over all squarefree numbers t where F(t) <= bound
    (if rule = '<=', or strictly < when rule='<').  Here we
    assume that the function F(t) has the properties:

        1) F(t) > 0,

        2) F(t) is multiplicative (but not necessarily strictly mult.),

        3) If p < q are primes, then F(p) < F(q).



    INPUT:
        F -- a function of positive squarefree integers, or just on primes.

        rule -- one of "<" or "<=".

        bound -- a number > 0.

    OUTPUT:
        This is an iterator over positive squarefree numbers
        satisfying the bound above.

    EXAMPLES:
        sage: g = lambda x: x
        sage: T = [t  for t in squarefree_where_increasing_multiplicative_functi
on_satisfies_bound(g, '<', 20)]; T
        [1, 2, 3, 5, 7, 11, 13, 17, 19, 6, 10, 14, 15]

        sage: T = [t  for t in squarefree_where_increasing_multiplicative_functi
on_satisfies_bound(g, '<', 30)]; T
        [1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 6, 10, 14, 22, 26, 15, 21]

        sage: T = [t  for t in squarefree_where_increasing_multiplicative_functi
on_satisfies_bound(g, '<=', 30)]; T
        [1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 6, 10, 14, 22, 26, 15, 21, 30]

    """

    ## Setup the inequality bound test
    if rule == "<":
        Bound_test = lambda x : (RR(x) < RR(bound))
    elif  rule == "<=":
        Bound_test = lambda x : (RR(x) <= RR(bound))
    else:
        raise TypeError, "You must use the rule '<' or '<='."



    ## Start with no primes
    r = 0

    ## Some initial settings
    p_list = []
```

```
    last_inc_ptr = -1

    ## Make the cumulative product lists: t_list and F_list (whose k-th entry is
 the product of the F(p) for the first k primes in p_list)
    t_cumulative_list = [1]
    F_cumulative_list = [1]

    ## Make the t and the F(t) value we're checking
    t = t_cumulative_list[-1]
    F_t = F_cumulative_list[-1]


    ## Main Loop:
    ## ==========
    done_flag = False
    while not done_flag:

        #print "\n\n"
        #print "Starting the main loop."

        done_flag = True       ## Reset this if we find one eligible entry


        #print "1 ---------------------------"
        #print "r = " + str(r)
        #print "last_inc_ptr = " + str(last_inc_ptr)
        #print "p_list = " + str(p_list)
        #print "t_cumulative_list = " + str(t_cumulative_list)
        #print "F_cumulative_list = " + str(F_cumulative_list)
        #print "t = " + str(t)
        #print "F_t = " + str(F_t)
        #print "bound = " + str(bound)
        #print "done_flag = " + str(done_flag)
        #print "---------------------------"




        ## If we're too big, carry until we're small or we can't carry anymore
        while (last_inc_ptr >= 0):

            #print "Starting the main increment subloop!"
            #
            #
            #print "2 ---------------------------"
            #print "r = " + str(r)
            #print "last_inc_ptr = " + str(last_inc_ptr)
            #print "p_list = " + str(p_list)
            #print "t_cumulative_list = " + str(t_cumulative_list)
            #print "F_cumulative_list = " + str(F_cumulative_list)
            #print "t = " + str(t)
            #print "F_t = " + str(F_t)
            #print "bound = " + str(bound)
            #print "done_flag = " + str(done_flag)
            #print "---------------------------"




            ## Return the eligible value
            #print "Bound_test(F_t) = " + str(Bound_test(F_t))
            if Bound_test(F_t):
                #print "  -- Our Bound is satisfied (1) ! =)"
                yield t
                done_flag = False   ## We found an entry with r primes, so defin
itely try r+1 primes also.

            #print "Finished the yield phase"
```



```
            ## Keep carrying until we have an eligible squarefree number, or we
can't carry anymore.
            if last_inc_ptr >= 0:

                ## Decide whether to increment or carry
                if F_t <= bound:
                    # Reset index and Increment!
                    last_inc_ptr = r - 1
                else:
                    # Carry
                    last_inc_ptr += -1


                ## Increment if we are able to
                if last_inc_ptr >= 0:

                    ## Perform the increment
                    p_list[last_inc_ptr] = next_prime(p_list[last_inc_ptr])

                    ## Generate the t_list and F_list values for this newly incr
emented index
                    if last_inc_ptr == 0:
                        t_cumulative_list[last_inc_ptr] = p_list[last_inc_ptr]
                        F_cumulative_list[last_inc_ptr] = F(p_list[last_inc_ptr]
)
                    else:
                        t_cumulative_list[last_inc_ptr] = t_cumulative_list[last
_inc_ptr-1] * p_list[last_inc_ptr]
                        F_cumulative_list[last_inc_ptr] = F_cumulative_list[last
_inc_ptr-1] * F(p_list[last_inc_ptr])

                    ## Generate the next set of primes (and associated lists) af
ter a possible carry
                    for i in range(last_inc_ptr + 1, r):
                        p_list[i] = next_prime(p_list[i-1])
                        t_cumulative_list[i] = t_cumulative_list[i-1] * p_list[i
]
                        F_cumulative_list[i] = F_cumulative_list[i-1] * F(p_list
[i])

                    ## Make the new values
                    t = t_cumulative_list[-1]
                    F_t = F_cumulative_list[-1]




        #print "Finished the carry phase"
        #
        #
        #print "3 ---------------------------"
        #print "r = " + str(r)
        #print "last_inc_ptr = " + str(last_inc_ptr)
        #print "p_list = " + str(p_list)
        #print "t_cumulative_list = " + str(t_cumulative_list)
        #print "F_cumulative_list = " + str(F_cumulative_list)
        #print "t = " + str(t)
        #print "F_t = " + str(F_t)
        #print "bound = " + str(bound)
        #print "done_flag = " + str(done_flag)
        #print "---------------------------"
        #
```

```
        ## Check if we are below the most significant carry index
        if last_inc_ptr < 0:

            #print "Beginning the loop to increase the number of primes."


            ## Return the eligible value  (This is the last possible return -- w
hen we are about to use more primes!)
            if Bound_test(F_t):
                #print "  -- Our Bound is satisfied (2) ! =)"
                yield t
                done_flag = False   ## We found an entry with r primes, so defin
itely try r+1 primes also.

            #print "Finished the final yield phase"







            #print ""
            #print "Incrementing the number of primes."


            ## Increment the number of primes to consider
            r += 1

            ## Some initial settings for products with r primes
            p_list = primes_first_n(r)
            last_inc_ptr = r-1

            #print "p_list = " + str(p_list)
            #print "last_inc_ptr = " + str(last_inc_ptr)

            ## Make the cumulative t_list and F_list (whose k-th entry is the pr
oduct of the F(p) for the first k primes in p_list)
            t_cumulative_list = []
            F_cumulative_list = []
            for p in p_list:
                if len(t_cumulative_list) == 0:
                    t_cumulative_list.append(p)
                    F_cumulative_list.append(F(p))
                else:
                    t_cumulative_list.append(t_cumulative_list[-1] * p)
                    F_cumulative_list.append(F_cumulative_list[-1] * F(p))

            ## Make the new values
            t = t_cumulative_list[-1]
            F_t = F_cumulative_list[-1]

        #print "Finished the increment # of primes phase"









def increasing_indices_mrange_generator(L, rule='le'):
    """
    Create an increasing list of elements which vary in the ranges
    specified by the components of L.

    INPUT:
        a list of integers
```



```
    OUTPUT:
        a list of integrs >= 0

    EXAMPLES:
        sage: for v in increasing_indices_mrange_generator([3,3], 'none'):
        .....:     print v
        .....:
        [0, 0]
        [0, 1]
        [0, 2]
        [1, 0]
        [1, 1]
        [1, 2]
        [2, 0]
        [2, 1]
        [2, 2]

        sage: for v in increasing_indices_mrange_generator([3,3], '<='):
        .....:     print v
        .....:
        [0, 0]
        [0, 1]
        [0, 2]
        [1, 1]
        [1, 2]
        [2, 2]

        sage: for v in increasing_indices_mrange_generator([3,3], '<'):
        print v
        .....:
        [0, 1]
        [0, 2]
        [1, 2]

    """
    ## Initialize the indexing list (according to the specified rule)
    n = len(L)
    if (rule == 'none') or (rule == 'leq') or (rule == '<='):
        v = n * [0]
    elif (rule == 'le') or (rule == '<'):
        v = range(n)
    else:
        raise TypeError, "The specified rule is not recognized."

    inc_index = n - 1
    yield v


    ## Increase the last index where we can
    while inc_index != -1:

        ## Check if we can increment the current index
        if v[inc_index] < L[inc_index] - 1:

            v[inc_index] += 1    ## Increment the current index


            for i in range(inc_index+1, n):    ## Clear future indices according
 to our specified rule
                if rule == 'none':
                    v[i] = 0
                elif (rule == 'leq') or (rule == '<='):
                    v[i] = v[inc_index]
                    if v[i] >= L[i]:      ## Check to see that this is still a v
alid entry
                        return
                elif (rule == 'le') or (rule == '<'):
                    v[i] = v[inc_index] + 1
```

```
                    if v[i] >= L[i]:      ## Check to see that this is still a v
alid entry
                        return
                else:
                    raise TypeError, "The specified rule is not recognized."


            inc_index = n - 1    ## Reset the index, and return the result
            yield v

        else:
            inc_index = inc_index - 1
```

```
print "Loading Jagy's Regular Ternary Quadratic Forms from Jagy_regular_ternary_
list.sage"


Jagy_regular_ternary_QF_list = [
QuadraticForm(ZZ, 3, ['1', '1', '1', '1', '1', '1']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '1', '0', '1']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '1', '0', '1']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '1', '1', '2']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '1', '0', '2']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '1', '1', '2']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '1', '0', '2']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '1', '0', '2']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '1', '1', '3']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '1', '0', '3']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '1', '1', '3']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '2', '2', '2']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '1', '0', '3']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '1', '0', '3']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '1', '0', '4']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '2', '1', '2']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '2', '2', '2']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '2', '1', '2']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '1', '1', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '1', '0', '4']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '2', '1', '2']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '1', '0', '4']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '2', '0', '2']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '2', '2', '3']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '1', '1', '5']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '1', '0', '6']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '2', '2', '3']), \
QuadraticForm(ZZ, 3, ['2', '2', '2', '2', '1', '2']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '1', '0', '5']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '1', '1', '7']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '2', '1', '3']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '2', '2', '3']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '2', '0', '3']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '2', '1', '3']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '2', '0', '3']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '1', '0', '6']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '2', '0', '3']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '2', '2', '4']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '3', '3', '3']), \
QuadraticForm(ZZ, 3, ['2', '1', '1', '2', '-1', '2']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '1', '0', '7']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '1', '0', '9']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '2', '1', '4']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '3', '3', '3']), \
QuadraticForm(ZZ, 3, ['2', '1', '1', '2', '1', '2']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '3', '2', '3']), \
QuadraticForm(ZZ, 3, ['2', '2', '2', '2', '2', '3']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '1', '0', '10']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '3', '1', '3']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '1', '0', '8']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '2', '0', '4']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '3', '1', '3']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '3', '2', '3']), \
QuadraticForm(ZZ, 3, ['2', '0', '2', '2', '2', '3']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '2', '1', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '1', '0', '9']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '1', '0', '12']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '2', '2', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '3', '0', '3']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '3', '3', '4']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '2', '0', '3']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '2', '0', '5']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '3', '2', '4']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '1', '1', '11']), \
```



```
QuadraticForm(ZZ, 3, ['1', '0', '0', '2', '2', '6']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '3', '0', '4']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '2', '2', '7']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '3', '0', '4']), \
QuadraticForm(ZZ, 3, ['2', '1', '0', '2', '0', '3']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '3', '3', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '1', '0', '12']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '2', '0', '6']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '3', '0', '4']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '3', '3', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '4', '4', '4']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '2', '0', '3']), \
QuadraticForm(ZZ, 3, ['2', '2', '2', '3', '2', '3']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '2', '0', '7']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '2', '2', '7']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '4', '3', '4']), \
QuadraticForm(ZZ, 3, ['2', '2', '2', '3', '1', '3']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '3', '1', '5']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '1', '0', '18']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '4', '2', '4']), \
QuadraticForm(ZZ, 3, ['2', '2', '2', '2', '1', '5']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '3', '3', '3']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '3', '1', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '3', '2', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '4', '4', '5']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '2', '0', '5']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '3', '0', '3']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '3', '3', '6']), \
QuadraticForm(ZZ, 3, ['2', '1', '2', '2', '2', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '1', '0', '16']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '2', '0', '8']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '4', '0', '4']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '4', '4', '5']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '3', '2', '3']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '3', '-2', '3']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '2', '0', '9']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '3', '0', '6']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '3', '2', '7']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '3', '3', '7']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '4', '2', '5']), \
QuadraticForm(ZZ, 3, ['2', '1', '1', '2', '1', '5']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '3', '0', '3']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '4', '0', '5']), \
QuadraticForm(ZZ, 3, ['2', '1', '0', '2', '0', '5']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '5', '4', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '2', '0', '10']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '3', '1', '7']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '3', '2', '7']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '3', '0', '4']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '3', '2', '3']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '3', '0', '7']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '4', '3', '6']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '1', '0', '21']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '3', '2', '8']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '1', '0', '30']), \
QuadraticForm(ZZ, 3, ['2', '0', '2', '3', '3', '5']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '3', '2', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '1', '0', '24']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '4', '0', '6']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '4', '4', '7']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '5', '2', '5']), \
QuadraticForm(ZZ, 3, ['2', '0', '2', '2', '2', '7']), \
QuadraticForm(ZZ, 3, ['3', '2', '0', '3', '0', '3']), \
QuadraticForm(ZZ, 3, ['3', '1', '1', '3', '-1', '3']), \
QuadraticForm(ZZ, 3, ['2', '0', '1', '3', '3', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '2', '2', '13']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '5', '0', '5']), \
QuadraticForm(ZZ, 3, ['2', '1', '1', '2', '-1', '7']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '3', '0', '5']), \
```

```
QuadraticForm(ZZ, 3, ['3', '1', '1', '3', '1', '3']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '1', '0', '36']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '3', '0', '9']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '3', '3', '10']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '4', '0', '7']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '5', '5', '7']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '6', '6', '6']), \
QuadraticForm(ZZ, 3, ['2', '1', '2', '2', '2', '8']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '2', '0', '9']), \
QuadraticForm(ZZ, 3, ['2', '0', '2', '3', '0', '5']), \
QuadraticForm(ZZ, 3, ['3', '3', '3', '3', '3', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '4', '4', '8']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '3', '2', '5']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '3', '2', '6']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '3', '0', '10']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '3', '3', '11']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '3', '0', '11']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '4', '3', '9']), \
QuadraticForm(ZZ, 3, ['2', '2', '1', '3', '3', '7']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '5', '2', '7']), \
QuadraticForm(ZZ, 3, ['3', '0', '3', '3', '3', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '2', '0', '16']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '4', '0', '8']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '6', '4', '6']), \
QuadraticForm(ZZ, 3, ['2', '2', '2', '3', '2', '7']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '4', '4', '5']), \
QuadraticForm(ZZ, 3, ['3', '2', '0', '3', '0', '4']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '3', '-2', '5']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '5', '1', '7']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '3', '3', '12']), \
QuadraticForm(ZZ, 3, ['2', '1', '0', '2', '0', '9']), \
QuadraticForm(ZZ, 3, ['2', '2', '1', '5', '5', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '2', '2', '18']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '3', '0', '12']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '3', '3', '13']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '6', '0', '6']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '3', '0', '6']), \
QuadraticForm(ZZ, 3, ['2', '2', '2', '5', '4', '5']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '3', '0', '4']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '4', '4', '4']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '2', '0', '21']), \
QuadraticForm(ZZ, 3, ['3', '1', '2', '3', '-2', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '5', '5', '9']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '5', '5', '5']), \
QuadraticForm(ZZ, 3, ['2', '0', '2', '3', '0', '7']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '5', '0', '8']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '3', '-2', '6']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '6', '0', '7']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '7', '5', '7']), \
QuadraticForm(ZZ, 3, ['2', '2', '2', '5', '1', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '4', '2', '11']), \
QuadraticForm(ZZ, 3, ['2', '1', '1', '5', '-3', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '6', '6', '9']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '7', '3', '7']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '2', '0', '15']), \
QuadraticForm(ZZ, 3, ['3', '3', '3', '5', '5', '5']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '4', '2', '13']), \
QuadraticForm(ZZ, 3, ['2', '0', '1', '3', '0', '8']), \
QuadraticForm(ZZ, 3, ['2', '1', '1', '5', '1', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '4', '0', '12']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '4', '4', '13']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '8', '8', '8']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '3', '0', '8']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '3', '2', '10']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '5', '2', '5']), \
QuadraticForm(ZZ, 3, ['3', '0', '2', '3', '2', '6']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '3', '-2', '7']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '4', '0', '4']), \
QuadraticForm(ZZ, 3, ['4', '0', '4', '4', '4', '5']), \
```



```
QuadraticForm(ZZ, 3, ['1', '0', '1', '7', '7', '9']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '5', '-4', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '5', '0', '10']), \
QuadraticForm(ZZ, 3, ['3', '1', '3', '3', '3', '7']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '3', '2', '7']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '3', '0', '18']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '3', '3', '19']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '6', '0', '9']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '3', '0', '9']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '5', '0', '6']), \
QuadraticForm(ZZ, 3, ['2', '1', '0', '5', '3', '6']), \
QuadraticForm(ZZ, 3, ['3', '3', '3', '5', '2', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '5', '4', '12']), \
QuadraticForm(ZZ, 3, ['3', '0', '2', '4', '4', '6']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '4', '0', '15']), \
QuadraticForm(ZZ, 3, ['2', '1', '0', '2', '0', '15']), \
QuadraticForm(ZZ, 3, ['2', '0', '1', '5', '5', '7']), \
QuadraticForm(ZZ, 3, ['2', '0', '2', '3', '3', '11']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '4', '4', '16']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '5', '2', '13']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '5', '0', '6']), \
QuadraticForm(ZZ, 3, ['2', '0', '2', '6', '6', '7']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '3', '0', '22']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '7', '0', '9']), \
QuadraticForm(ZZ, 3, ['2', '0', '1', '3', '3', '11']), \
QuadraticForm(ZZ, 3, ['2', '2', '1', '5', '5', '8']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '9', '8', '9']), \
QuadraticForm(ZZ, 3, ['2', '2', '2', '3', '1', '13']), \
QuadraticForm(ZZ, 3, ['2', '0', '2', '5', '2', '7']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '3', '0', '7']), \
QuadraticForm(ZZ, 3, ['3', '3', '0', '5', '1', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '2', '0', '32']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '4', '0', '16']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '5', '2', '13']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '8', '0', '8']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '3', '2', '11']), \
QuadraticForm(ZZ, 3, ['3', '2', '0', '3', '0', '8']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '5', '2', '5']), \
QuadraticForm(ZZ, 3, ['4', '4', '4', '5', '2', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '6', '6', '13']), \
QuadraticForm(ZZ, 3, ['3', '3', '3', '3', '3', '11']), \
QuadraticForm(ZZ, 3, ['5', '5', '5', '5', '4', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '9', '6', '9']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '3', '0', '12']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '3', '0', '8']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '4', '4', '7']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '5', '2', '5']), \
QuadraticForm(ZZ, 3, ['5', '4', '4', '5', '-2', '5']), \
QuadraticForm(ZZ, 3, ['3', '3', '2', '5', '1', '6']), \
QuadraticForm(ZZ, 3, ['5', '4', '3', '5', '-3', '5']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '6', '3', '13']), \
QuadraticForm(ZZ, 3, ['2', '1', '1', '5', '-2', '8']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '10', '10', '10']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '3', '0', '15']), \
QuadraticForm(ZZ, 3, ['3', '0', '3', '5', '5', '7']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '8', '8', '12']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '3', '-2', '11']), \
QuadraticForm(ZZ, 3, ['3', '0', '2', '4', '0', '7']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '6', '-4', '6']), \
QuadraticForm(ZZ, 3, ['4', '0', '4', '4', '4', '7']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '7', '5', '13']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '9', '0', '9']), \
QuadraticForm(ZZ, 3, ['2', '0', '2', '3', '0', '14']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '5', '0', '9']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '3', '2', '11']), \
QuadraticForm(ZZ, 3, ['3', '1', '0', '3', '0', '10']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '3', '0', '30']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '3', '3', '31']), \
QuadraticForm(ZZ, 3, ['2', '1', '1', '7', '3', '7']), \
```

```
QuadraticForm(ZZ, 3, ['2', '1', '1', '7', '-1', '7']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '9', '9', '13']), \
QuadraticForm(ZZ, 3, ['2', '2', '2', '5', '1', '11']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '4', '0', '24']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '6', '0', '16']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '10', '4', '10']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '7', '4', '8']), \
QuadraticForm(ZZ, 3, ['4', '0', '4', '4', '0', '7']), \
QuadraticForm(ZZ, 3, ['4', '4', '0', '5', '0', '6']), \
QuadraticForm(ZZ, 3, ['4', '4', '4', '5', '2', '7']), \
QuadraticForm(ZZ, 3, ['3', '1', '2', '3', '-2', '12']), \
QuadraticForm(ZZ, 3, ['3', '2', '0', '5', '0', '7']), \
QuadraticForm(ZZ, 3, ['3', '3', '0', '5', '2', '8']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '3', '0', '20']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '5', '0', '10']), \
QuadraticForm(ZZ, 3, ['3', '1', '2', '3', '2', '12']), \
QuadraticForm(ZZ, 3, ['3', '0', '2', '5', '0', '7']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '7', '-6', '7']), \
QuadraticForm(ZZ, 3, ['2', '1', '2', '5', '2', '11']), \
QuadraticForm(ZZ, 3, ['2', '1', '1', '8', '7', '8']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '3', '0', '36']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '3', '3', '37']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '4', '4', '28']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '6', '0', '18']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '9', '0', '12']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '12', '12', '12']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '3', '0', '18']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '5', '0', '12']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '6', '0', '9']), \
QuadraticForm(ZZ, 3, ['2', '0', '2', '6', '6', '11']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '4', '4', '10']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '5', '4', '8']), \
QuadraticForm(ZZ, 3, ['3', '3', '0', '5', '3', '9']), \
QuadraticForm(ZZ, 3, ['4', '0', '4', '6', '6', '7']), \
QuadraticForm(ZZ, 3, ['5', '2', '2', '5', '-2', '5']), \
QuadraticForm(ZZ, 3, ['2', '1', '1', '8', '-5', '8']), \
QuadraticForm(ZZ, 3, ['3', '3', '0', '6', '0', '7']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '6', '2', '7']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '7', '6', '7']), \
QuadraticForm(ZZ, 3, ['5', '2', '2', '5', '2', '5']), \
QuadraticForm(ZZ, 3, ['3', '3', '3', '7', '4', '7']), \
QuadraticForm(ZZ, 3, ['5', '5', '0', '5', '0', '6']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '6', '6', '21']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '12', '12', '13']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '4', '4', '11']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '3', '0', '44']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '6', '0', '11']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '7', '0', '18']), \
QuadraticForm(ZZ, 3, ['3', '1', '0', '3', '0', '14']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '5', '0', '25']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '9', '5', '15']), \
QuadraticForm(ZZ, 3, ['2', '0', '2', '5', '0', '13']), \
QuadraticForm(ZZ, 3, ['3', '3', '2', '7', '1', '7']), \
QuadraticForm(ZZ, 3, ['3', '3', '3', '5', '4', '11']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '8', '0', '16']), \
QuadraticForm(ZZ, 3, ['3', '2', '0', '3', '0', '16']), \
QuadraticForm(ZZ, 3, ['3', '0', '2', '4', '0', '11']), \
QuadraticForm(ZZ, 3, ['3', '2', '0', '5', '4', '10']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '7', '-2', '7']), \
QuadraticForm(ZZ, 3, ['4', '4', '0', '5', '0', '8']), \
QuadraticForm(ZZ, 3, ['4', '4', '4', '7', '6', '7']), \
QuadraticForm(ZZ, 3, ['4', '4', '0', '7', '6', '7']), \
QuadraticForm(ZZ, 3, ['2', '0', '2', '3', '0', '23']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '7', '4', '7']), \
QuadraticForm(ZZ, 3, ['5', '5', '4', '5', '2', '8']), \
QuadraticForm(ZZ, 3, ['6', '6', '6', '6', '6', '7']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '4', '0', '36']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '6', '0', '24']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '12', '0', '12']), \
```



```
QuadraticForm(ZZ, 3, ['2', '2', '2', '5', '4', '17']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '3', '-2', '19']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '4', '0', '12']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '7', '2', '7']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '8', '8', '8']), \
QuadraticForm(ZZ, 3, ['4', '0', '4', '4', '4', '11']), \
QuadraticForm(ZZ, 3, ['4', '0', '4', '6', '0', '7']), \
QuadraticForm(ZZ, 3, ['4', '4', '4', '7', '2', '7']), \
QuadraticForm(ZZ, 3, ['5', '4', '4', '5', '4', '8']), \
QuadraticForm(ZZ, 3, ['5', '2', '4', '5', '-4', '8']), \
QuadraticForm(ZZ, 3, ['3', '1', '1', '3', '-1', '17']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '7', '0', '7']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '5', '0', '15']), \
QuadraticForm(ZZ, 3, ['5', '5', '5', '7', '6', '7']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '6', '6', '11']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '7', '5', '25']), \
QuadraticForm(ZZ, 3, ['5', '1', '4', '5', '4', '8']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '4', '0', '45']), \
QuadraticForm(ZZ, 3, ['3', '3', '0', '7', '5', '10']), \
QuadraticForm(ZZ, 3, ['5', '0', '0', '6', '3', '6']), \
QuadraticForm(ZZ, 3, ['5', '3', '2', '5', '-2', '8']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '9', '7', '21']), \
QuadraticForm(ZZ, 3, ['3', '2', '3', '5', '1', '13']), \
QuadraticForm(ZZ, 3, ['5', '0', '0', '6', '2', '6']), \
QuadraticForm(ZZ, 3, ['5', '4', '3', '5', '3', '9']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '8', '8', '24']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '6', '0', '15']), \
QuadraticForm(ZZ, 3, ['2', '0', '2', '6', '6', '17']), \
QuadraticForm(ZZ, 3, ['3', '3', '3', '5', '3', '15']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '8', '4', '8']), \
QuadraticForm(ZZ, 3, ['4', '0', '4', '7', '4', '8']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '10', '0', '19']), \
QuadraticForm(ZZ, 3, ['3', '1', '2', '3', '2', '22']), \
QuadraticForm(ZZ, 3, ['3', '3', '0', '7', '0', '10']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '9', '0', '21']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '13', '3', '15']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '5', '0', '21']), \
QuadraticForm(ZZ, 3, ['5', '1', '2', '5', '2', '8']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '8', '0', '24']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '16', '16', '16']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '5', '4', '20']), \
QuadraticForm(ZZ, 3, ['3', '2', '0', '3', '0', '24']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '7', '6', '11']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '8', '0', '8']), \
QuadraticForm(ZZ, 3, ['4', '4', '4', '5', '2', '13']), \
QuadraticForm(ZZ, 3, ['4', '4', '0', '7', '0', '8']), \
QuadraticForm(ZZ, 3, ['5', '2', '0', '5', '0', '8']), \
QuadraticForm(ZZ, 3, ['5', '2', '2', '7', '6', '7']), \
QuadraticForm(ZZ, 3, ['7', '6', '6', '7', '-2', '7']), \
QuadraticForm(ZZ, 3, ['3', '2', '0', '5', '0', '14']), \
QuadraticForm(ZZ, 3, ['4', '0', '4', '7', '0', '8']), \
QuadraticForm(ZZ, 3, ['5', '4', '2', '5', '-2', '10']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '5', '0', '40']), \
QuadraticForm(ZZ, 3, ['4', '4', '4', '6', '2', '11']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '7', '5', '31']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '9', '0', '24']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '12', '12', '21']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '5', '0', '24']), \
QuadraticForm(ZZ, 3, ['2', '2', '2', '11', '4', '11']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '4', '4', '19']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '8', '8', '11']), \
QuadraticForm(ZZ, 3, ['4', '4', '4', '7', '2', '10']), \
QuadraticForm(ZZ, 3, ['5', '2', '0', '5', '0', '9']), \
QuadraticForm(ZZ, 3, ['5', '0', '4', '6', '0', '8']), \
QuadraticForm(ZZ, 3, ['5', '4', '2', '8', '8', '8']), \
QuadraticForm(ZZ, 3, ['2', '2', '2', '11', '1', '11']), \
QuadraticForm(ZZ, 3, ['5', '3', '2', '5', '2', '10']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '6', '4', '14']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '15', '15', '19']), \
```

```
QuadraticForm(ZZ, 3, ['3', '0', '0', '10', '10', '10']), \
QuadraticForm(ZZ, 3, ['5', '0', '0', '6', '6', '9']), \
QuadraticForm(ZZ, 3, ['5', '5', '0', '7', '2', '8']), \
QuadraticForm(ZZ, 3, ['2', '2', '2', '7', '2', '19']), \
QuadraticForm(ZZ, 3, ['5', '2', '4', '5', '-4', '12']), \
QuadraticForm(ZZ, 3, ['5', '0', '0', '8', '8', '8']), \
QuadraticForm(ZZ, 3, ['6', '6', '0', '7', '4', '8']), \
QuadraticForm(ZZ, 3, ['7', '6', '4', '7', '-4', '8']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '7', '0', '36']), \
QuadraticForm(ZZ, 3, ['2', '0', '2', '9', '0', '14']), \
QuadraticForm(ZZ, 3, ['5', '4', '3', '8', '-6', '9']), \
QuadraticForm(ZZ, 3, ['6', '2', '0', '6', '0', '7']), \
QuadraticForm(ZZ, 3, ['3', '2', '1', '7', '-3', '13']), \
QuadraticForm(ZZ, 3, ['5', '4', '4', '8', '4', '8']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '13', '13', '23']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '8', '0', '32']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '16', '0', '16']), \
QuadraticForm(ZZ, 3, ['3', '2', '0', '3', '0', '32']), \
QuadraticForm(ZZ, 3, ['3', '0', '2', '8', '0', '11']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '11', '-10', '11']), \
QuadraticForm(ZZ, 3, ['4', '4', '0', '5', '0', '16']), \
QuadraticForm(ZZ, 3, ['5', '2', '4', '5', '4', '12']), \
QuadraticForm(ZZ, 3, ['5', '5', '4', '7', '2', '10']), \
QuadraticForm(ZZ, 3, ['3', '0', '3', '9', '3', '11']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '9', '6', '11']), \
QuadraticForm(ZZ, 3, ['6', '6', '3', '7', '7', '10']), \
QuadraticForm(ZZ, 3, ['1', '0', '1', '10', '5', '29']), \
QuadraticForm(ZZ, 3, ['2', '1', '1', '7', '-6', '22']), \
QuadraticForm(ZZ, 3, ['5', '0', '5', '6', '3', '11']), \
QuadraticForm(ZZ, 3, ['5', '3', '3', '9', '9', '9']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '3', '0', '48']), \
QuadraticForm(ZZ, 3, ['2', '0', '0', '12', '12', '15']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '8', '0', '12']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '8', '8', '14']), \
QuadraticForm(ZZ, 3, ['5', '2', '0', '5', '0', '12']), \
QuadraticForm(ZZ, 3, ['5', '4', '2', '5', '2', '14']), \
QuadraticForm(ZZ, 3, ['5', '2', '4', '5', '-4', '14']), \
QuadraticForm(ZZ, 3, ['5', '3', '1', '5', '1', '13']), \
QuadraticForm(ZZ, 3, ['5', '2', '3', '8', '6', '9']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '10', '0', '30']), \
QuadraticForm(ZZ, 3, ['4', '0', '4', '10', '10', '11']), \
QuadraticForm(ZZ, 3, ['5', '0', '0', '8', '4', '8']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '8', '0', '40']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '11', '6', '11']), \
QuadraticForm(ZZ, 3, ['4', '0', '4', '8', '8', '13']), \
QuadraticForm(ZZ, 3, ['7', '2', '2', '7', '-2', '7']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '4', '4', '28']), \
QuadraticForm(ZZ, 3, ['4', '4', '0', '7', '6', '15']), \
QuadraticForm(ZZ, 3, ['2', '1', '0', '8', '0', '21']), \
QuadraticForm(ZZ, 3, ['5', '4', '5', '5', '2', '17']), \
QuadraticForm(ZZ, 3, ['6', '0', '3', '7', '7', '10']), \
QuadraticForm(ZZ, 3, ['4', '0', '4', '4', '4', '23']), \
QuadraticForm(ZZ, 3, ['1', '1', '1', '19', '8', '19']), \
QuadraticForm(ZZ, 3, ['5', '0', '5', '9', '9', '11']), \
QuadraticForm(ZZ, 3, ['7', '1', '1', '7', '-1', '7']), \
QuadraticForm(ZZ, 3, ['2', '2', '2', '11', '8', '18']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '5', '-4', '26']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '7', '6', '18']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '12', '12', '33']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '4', '4', '31']), \
QuadraticForm(ZZ, 3, ['7', '3', '2', '7', '2', '8']), \
QuadraticForm(ZZ, 3, ['5', '0', '4', '8', '8', '12']), \
QuadraticForm(ZZ, 3, ['2', '0', '2', '5', '0', '38']), \
QuadraticForm(ZZ, 3, ['5', '0', '0', '6', '6', '14']), \
QuadraticForm(ZZ, 3, ['6', '2', '2', '6', '2', '11']), \
QuadraticForm(ZZ, 3, ['7', '1', '2', '7', '-2', '8']), \
QuadraticForm(ZZ, 3, ['5', '4', '1', '8', '4', '11']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '16', '0', '24']), \
QuadraticForm(ZZ, 3, ['4', '4', '0', '5', '0', '24']), \
```



```
QuadraticForm(ZZ, 3, ['4', '0', '0', '7', '6', '15']), \
QuadraticForm(ZZ, 3, ['4', '4', '0', '7', '0', '16']), \
QuadraticForm(ZZ, 3, ['4', '4', '4', '11', '6', '11']), \
QuadraticForm(ZZ, 3, ['5', '2', '2', '7', '-6', '13']), \
QuadraticForm(ZZ, 3, ['7', '4', '4', '8', '0', '8']), \
QuadraticForm(ZZ, 3, ['3', '0', '2', '7', '0', '19']), \
QuadraticForm(ZZ, 3, ['5', '2', '2', '10', '-8', '10']), \
QuadraticForm(ZZ, 3, ['5', '2', '2', '5', '-2', '17']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '3', '-2', '51']), \
QuadraticForm(ZZ, 3, ['3', '2', '0', '7', '0', '20']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '7', '-6', '22']), \
QuadraticForm(ZZ, 3, ['4', '4', '4', '11', '2', '11']), \
QuadraticForm(ZZ, 3, ['5', '0', '0', '8', '8', '12']), \
QuadraticForm(ZZ, 3, ['7', '6', '4', '7', '-4', '12']), \
QuadraticForm(ZZ, 3, ['2', '2', '2', '14', '10', '17']), \
QuadraticForm(ZZ, 3, ['5', '2', '1', '8', '-4', '11']), \
QuadraticForm(ZZ, 3, ['8', '6', '6', '9', '9', '9']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '12', '0', '36']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '24', '24', '24']), \
QuadraticForm(ZZ, 3, ['2', '2', '2', '11', '10', '23']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '4', '0', '36']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '8', '8', '20']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '10', '8', '16']), \
QuadraticForm(ZZ, 3, ['4', '4', '4', '7', '2', '19']), \
QuadraticForm(ZZ, 3, ['4', '0', '4', '12', '12', '13']), \
QuadraticForm(ZZ, 3, ['5', '2', '4', '5', '-4', '20']), \
QuadraticForm(ZZ, 3, ['5', '4', '0', '8', '0', '12']), \
QuadraticForm(ZZ, 3, ['5', '4', '2', '8', '8', '14']), \
QuadraticForm(ZZ, 3, ['6', '6', '6', '7', '6', '15']), \
QuadraticForm(ZZ, 3, ['6', '0', '6', '8', '4', '11']), \
QuadraticForm(ZZ, 3, ['7', '2', '4', '7', '4', '10']), \
QuadraticForm(ZZ, 3, ['7', '4', '2', '7', '-2', '10']), \
QuadraticForm(ZZ, 3, ['8', '8', '0', '8', '0', '9']), \
QuadraticForm(ZZ, 3, ['8', '8', '8', '8', '4', '11']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '21', '0', '21']), \
QuadraticForm(ZZ, 3, ['5', '2', '2', '10', '6', '10']), \
QuadraticForm(ZZ, 3, ['8', '6', '6', '9', '-3', '9']), \
QuadraticForm(ZZ, 3, ['3', '0', '2', '8', '0', '19']), \
QuadraticForm(ZZ, 3, ['4', '4', '4', '5', '2', '29']), \
QuadraticForm(ZZ, 3, ['5', '0', '4', '8', '0', '12']), \
QuadraticForm(ZZ, 3, ['7', '6', '4', '7', '4', '12']), \
QuadraticForm(ZZ, 3, ['5', '0', '0', '6', '0', '15']), \
QuadraticForm(ZZ, 3, ['5', '5', '5', '11', '7', '11']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '7', '-6', '27']), \
QuadraticForm(ZZ, 3, ['4', '4', '0', '11', '10', '15']), \
QuadraticForm(ZZ, 3, ['7', '4', '7', '7', '2', '13']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '7', '0', '39']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '8', '0', '64']), \
QuadraticForm(ZZ, 3, ['3', '2', '0', '11', '0', '16']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '11', '-10', '19']), \
QuadraticForm(ZZ, 3, ['4', '4', '4', '7', '6', '23']), \
QuadraticForm(ZZ, 3, ['5', '0', '2', '8', '0', '13']), \
QuadraticForm(ZZ, 3, ['5', '4', '4', '12', '-8', '12']), \
QuadraticForm(ZZ, 3, ['7', '2', '4', '7', '-4', '12']), \
QuadraticForm(ZZ, 3, ['7', '6', '6', '7', '-2', '15']), \
QuadraticForm(ZZ, 3, ['5', '2', '1', '10', '10', '13']), \
QuadraticForm(ZZ, 3, ['5', '4', '2', '10', '10', '14']), \
QuadraticForm(ZZ, 3, ['5', '4', '2', '8', '8', '17']), \
QuadraticForm(ZZ, 3, ['5', '3', '3', '9', '9', '15']), \
QuadraticForm(ZZ, 3, ['6', '0', '0', '7', '2', '13']), \
QuadraticForm(ZZ, 3, ['6', '6', '0', '7', '6', '18']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '8', '8', '72']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '19', '0', '30']), \
QuadraticForm(ZZ, 3, ['3', '3', '0', '7', '0', '30']), \
QuadraticForm(ZZ, 3, ['9', '3', '9', '9', '9', '11']), \
QuadraticForm(ZZ, 3, ['7', '4', '2', '7', '-2', '13']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '24', '0', '24']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '8', '0', '24']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '16', '16', '16']), \
```

```
QuadraticForm(ZZ, 3, ['4', '4', '0', '7', '0', '24']), \
QuadraticForm(ZZ, 3, ['4', '4', '4', '13', '2', '13']), \
QuadraticForm(ZZ, 3, ['5', '2', '0', '5', '0', '24']), \
QuadraticForm(ZZ, 3, ['5', '4', '2', '8', '-4', '17']), \
QuadraticForm(ZZ, 3, ['6', '0', '0', '7', '6', '15']), \
QuadraticForm(ZZ, 3, ['7', '2', '6', '7', '-6', '15']), \
QuadraticForm(ZZ, 3, ['7', '4', '4', '10', '-4', '10']), \
QuadraticForm(ZZ, 3, ['8', '0', '0', '9', '6', '9']), \
QuadraticForm(ZZ, 3, ['5', '4', '4', '12', '-4', '12']), \
QuadraticForm(ZZ, 3, ['5', '0', '0', '8', '8', '17']), \
QuadraticForm(ZZ, 3, ['7', '6', '0', '7', '0', '15']), \
QuadraticForm(ZZ, 3, ['9', '9', '6', '9', '3', '11']), \
QuadraticForm(ZZ, 3, ['1', '1', '0', '9', '0', '70']), \
QuadraticForm(ZZ, 3, ['5', '2', '4', '5', '-4', '28']), \
QuadraticForm(ZZ, 3, ['5', '3', '5', '9', '6', '17']), \
QuadraticForm(ZZ, 3, ['5', '4', '3', '5', '-3', '33']), \
QuadraticForm(ZZ, 3, ['6', '0', '6', '7', '7', '19']), \
QuadraticForm(ZZ, 3, ['2', '0', '2', '15', '0', '23']), \
QuadraticForm(ZZ, 3, ['3', '3', '3', '7', '4', '37']), \
QuadraticForm(ZZ, 3, ['7', '4', '0', '7', '0', '15']), \
QuadraticForm(ZZ, 3, ['9', '6', '6', '11', '-8', '11']), \
QuadraticForm(ZZ, 3, ['7', '4', '2', '8', '8', '15']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '17', '12', '24']), \
QuadraticForm(ZZ, 3, ['4', '4', '4', '7', '2', '31']), \
QuadraticForm(ZZ, 3, ['6', '0', '6', '8', '4', '17']), \
QuadraticForm(ZZ, 3, ['8', '8', '0', '8', '0', '15']), \
QuadraticForm(ZZ, 3, ['8', '8', '8', '8', '4', '17']), \
QuadraticForm(ZZ, 3, ['4', '0', '4', '15', '12', '16']), \
QuadraticForm(ZZ, 3, ['8', '8', '4', '11', '2', '11']), \
QuadraticForm(ZZ, 3, ['3', '3', '3', '17', '8', '17']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '16', '0', '48']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '11', '-10', '27']), \
QuadraticForm(ZZ, 3, ['5', '0', '4', '8', '0', '20']), \
QuadraticForm(ZZ, 3, ['5', '2', '2', '13', '-6', '13']), \
QuadraticForm(ZZ, 3, ['7', '2', '0', '7', '0', '16']), \
QuadraticForm(ZZ, 3, ['7', '6', '4', '7', '4', '20']), \
QuadraticForm(ZZ, 3, ['7', '4', '4', '12', '-8', '12']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '19', '-18', '19']), \
QuadraticForm(ZZ, 3, ['5', '2', '2', '10', '6', '17']), \
QuadraticForm(ZZ, 3, ['5', '4', '2', '12', '12', '17']), \
QuadraticForm(ZZ, 3, ['6', '2', '4', '11', '-6', '14']), \
QuadraticForm(ZZ, 3, ['7', '4', '1', '10', '5', '13']), \
QuadraticForm(ZZ, 3, ['7', '4', '4', '12', '8', '12']), \
QuadraticForm(ZZ, 3, ['2', '0', '1', '15', '15', '32']), \
QuadraticForm(ZZ, 3, ['5', '0', '2', '6', '0', '29']), \
QuadraticForm(ZZ, 3, ['5', '4', '0', '8', '0', '24']), \
QuadraticForm(ZZ, 3, ['5', '4', '4', '14', '4', '14']), \
QuadraticForm(ZZ, 3, ['6', '0', '0', '9', '6', '17']), \
QuadraticForm(ZZ, 3, ['7', '2', '6', '10', '-6', '15']), \
QuadraticForm(ZZ, 3, ['8', '4', '4', '11', '-2', '11']), \
QuadraticForm(ZZ, 3, ['9', '6', '6', '11', '-2', '11']), \
QuadraticForm(ZZ, 3, ['8', '4', '4', '11', '1', '11']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '10', '0', '30']), \
QuadraticForm(ZZ, 3, ['4', '0', '4', '15', '0', '16']), \
QuadraticForm(ZZ, 3, ['7', '6', '2', '7', '-2', '23']), \
QuadraticForm(ZZ, 3, ['10', '0', '10', '12', '12', '13']), \
QuadraticForm(ZZ, 3, ['4', '4', '4', '5', '2', '61']), \
QuadraticForm(ZZ, 3, ['5', '0', '0', '8', '0', '24']), \
QuadraticForm(ZZ, 3, ['7', '6', '0', '7', '0', '24']), \
QuadraticForm(ZZ, 3, ['7', '4', '2', '8', '4', '19']), \
QuadraticForm(ZZ, 3, ['8', '0', '0', '11', '2', '11']), \
QuadraticForm(ZZ, 3, ['5', '4', '2', '8', '8', '29']), \
QuadraticForm(ZZ, 3, ['5', '0', '4', '8', '8', '28']), \
QuadraticForm(ZZ, 3, ['7', '4', '4', '8', '0', '20']), \
QuadraticForm(ZZ, 3, ['12', '0', '12', '12', '12', '13']), \
QuadraticForm(ZZ, 3, ['5', '5', '0', '11', '3', '21']), \
QuadraticForm(ZZ, 3, ['3', '2', '0', '11', '0', '32']), \
QuadraticForm(ZZ, 3, ['5', '4', '4', '12', '8', '20']), \
QuadraticForm(ZZ, 3, ['5', '2', '4', '13', '-12', '20']), \
```



```
QuadraticForm(ZZ, 3, ['5', '2', '0', '10', '0', '21']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '11', '10', '35']), \
QuadraticForm(ZZ, 3, ['9', '0', '6', '12', '12', '14']), \
QuadraticForm(ZZ, 3, ['7', '1', '1', '13', '-7', '13']), \
QuadraticForm(ZZ, 3, ['2', '0', '2', '15', '0', '38']), \
QuadraticForm(ZZ, 3, ['5', '5', '0', '11', '6', '24']), \
QuadraticForm(ZZ, 3, ['6', '6', '0', '14', '0', '15']), \
QuadraticForm(ZZ, 3, ['7', '2', '7', '8', '6', '23']), \
QuadraticForm(ZZ, 3, ['7', '2', '2', '13', '-4', '13']), \
QuadraticForm(ZZ, 3, ['5', '3', '3', '9', '0', '27']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '8', '0', '48']), \
QuadraticForm(ZZ, 3, ['5', '2', '0', '5', '0', '48']), \
QuadraticForm(ZZ, 3, ['5', '0', '4', '12', '0', '20']), \
QuadraticForm(ZZ, 3, ['5', '4', '4', '14', '-8', '20']), \
QuadraticForm(ZZ, 3, ['5', '2', '2', '17', '-14', '17']), \
QuadraticForm(ZZ, 3, ['8', '0', '0', '12', '12', '15']), \
QuadraticForm(ZZ, 3, ['8', '0', '8', '12', '12', '17']), \
QuadraticForm(ZZ, 3, ['8', '4', '0', '11', '6', '15']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '40', '40', '40']), \
QuadraticForm(ZZ, 3, ['4', '4', '4', '11', '2', '31']), \
QuadraticForm(ZZ, 3, ['8', '8', '8', '12', '4', '17']), \
QuadraticForm(ZZ, 3, ['10', '10', '0', '11', '8', '16']), \
QuadraticForm(ZZ, 3, ['11', '8', '6', '11', '-6', '14']), \
QuadraticForm(ZZ, 3, ['7', '2', '6', '13', '-12', '18']), \
QuadraticForm(ZZ, 3, ['2', '2', '0', '18', '0', '35']), \
QuadraticForm(ZZ, 3, ['4', '4', '4', '13', '10', '29']), \
QuadraticForm(ZZ, 3, ['7', '4', '0', '12', '0', '16']), \
QuadraticForm(ZZ, 3, ['7', '6', '6', '15', '-2', '15']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '8', '8', '56']), \
QuadraticForm(ZZ, 3, ['4', '4', '4', '19', '2', '19']), \
QuadraticForm(ZZ, 3, ['5', '4', '0', '8', '0', '36']), \
QuadraticForm(ZZ, 3, ['8', '8', '8', '11', '4', '20']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '7', '0', '63']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '14', '14', '35']), \
QuadraticForm(ZZ, 3, ['5', '5', '2', '17', '1', '17']), \
QuadraticForm(ZZ, 3, ['7', '7', '7', '13', '8', '19']), \
QuadraticForm(ZZ, 3, ['12', '8', '12', '12', '12', '15']), \
QuadraticForm(ZZ, 3, ['7', '1', '2', '7', '-2', '28']), \
QuadraticForm(ZZ, 3, ['7', '2', '0', '13', '0', '15']), \
QuadraticForm(ZZ, 3, ['8', '8', '4', '13', '2', '17']), \
QuadraticForm(ZZ, 3, ['8', '4', '8', '13', '2', '17']), \
QuadraticForm(ZZ, 3, ['6', '0', '6', '13', '0', '21']), \
QuadraticForm(ZZ, 3, ['4', '4', '0', '11', '8', '40']), \
QuadraticForm(ZZ, 3, ['5', '2', '0', '13', '0', '24']), \
QuadraticForm(ZZ, 3, ['5', '2', '4', '13', '-12', '28']), \
QuadraticForm(ZZ, 3, ['7', '6', '0', '15', '0', '16']), \
QuadraticForm(ZZ, 3, ['7', '6', '4', '15', '-12', '20']), \
QuadraticForm(ZZ, 3, ['11', '6', '8', '11', '8', '16']), \
QuadraticForm(ZZ, 3, ['11', '2', '8', '11', '-8', '16']), \
QuadraticForm(ZZ, 3, ['11', '8', '3', '11', '3', '15']), \
QuadraticForm(ZZ, 3, ['3', '2', '2', '27', '-26', '27']), \
QuadraticForm(ZZ, 3, ['5', '0', '0', '8', '0', '40']), \
QuadraticForm(ZZ, 3, ['7', '4', '2', '12', '12', '23']), \
QuadraticForm(ZZ, 3, ['8', '8', '8', '17', '14', '17']), \
QuadraticForm(ZZ, 3, ['8', '8', '4', '11', '2', '23']), \
QuadraticForm(ZZ, 3, ['3', '3', '0', '17', '5', '35']), \
QuadraticForm(ZZ, 3, ['5', '5', '5', '17', '7', '23']), \
QuadraticForm(ZZ, 3, ['7', '7', '4', '13', '2', '22']), \
QuadraticForm(ZZ, 3, ['9', '6', '9', '11', '3', '21']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '16', '16', '112']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '24', '0', '72']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '48', '48', '48']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '8', '0', '72']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '16', '16', '40']), \
QuadraticForm(ZZ, 3, ['4', '4', '0', '7', '0', '72']), \
QuadraticForm(ZZ, 3, ['4', '4', '4', '13', '2', '37']), \
QuadraticForm(ZZ, 3, ['4', '4', '0', '19', '0', '24']), \
QuadraticForm(ZZ, 3, ['5', '2', '0', '5', '0', '72']), \
QuadraticForm(ZZ, 3, ['5', '4', '4', '20', '-8', '20']), \
```

```
QuadraticForm(ZZ, 3, ['7', '2', '6', '7', '-6', '39']), \
QuadraticForm(ZZ, 3, ['7', '4', '4', '10', '8', '28']), \
QuadraticForm(ZZ, 3, ['7', '2', '6', '13', '6', '21']), \
QuadraticForm(ZZ, 3, ['7', '6', '2', '15', '-6', '19']), \
QuadraticForm(ZZ, 3, ['8', '0', '0', '9', '0', '24']), \
QuadraticForm(ZZ, 3, ['8', '4', '4', '11', '10', '23']), \
QuadraticForm(ZZ, 3, ['8', '8', '0', '11', '0', '24']), \
QuadraticForm(ZZ, 3, ['8', '0', '8', '12', '12', '23']), \
QuadraticForm(ZZ, 3, ['8', '0', '0', '15', '6', '15']), \
QuadraticForm(ZZ, 3, ['9', '6', '6', '17', '-14', '17']), \
QuadraticForm(ZZ, 3, ['12', '12', '0', '13', '8', '16']), \
QuadraticForm(ZZ, 3, ['8', '4', '8', '11', '2', '23']), \
QuadraticForm(ZZ, 3, ['7', '4', '2', '12', '-4', '23']), \
QuadraticForm(ZZ, 3, ['5', '0', '0', '21', '18', '21']), \
QuadraticForm(ZZ, 3, ['11', '2', '0', '11', '0', '15']), \
QuadraticForm(ZZ, 3, ['4', '4', '4', '7', '2', '79']), \
QuadraticForm(ZZ, 3, ['8', '0', '8', '11', '0', '24']), \
QuadraticForm(ZZ, 3, ['5', '5', '4', '17', '2', '26']), \
QuadraticForm(ZZ, 3, ['4', '0', '4', '20', '20', '31']), \
QuadraticForm(ZZ, 3, ['7', '6', '4', '7', '-4', '52']), \
QuadraticForm(ZZ, 3, ['7', '7', '6', '13', '3', '27']), \
QuadraticForm(ZZ, 3, ['10', '10', '10', '13', '8', '22']), \
QuadraticForm(ZZ, 3, ['7', '4', '4', '12', '-8', '28']), \
QuadraticForm(ZZ, 3, ['7', '6', '2', '15', '10', '23']), \
QuadraticForm(ZZ, 3, ['5', '1', '3', '13', '-6', '33']), \
QuadraticForm(ZZ, 3, ['7', '6', '2', '15', '-6', '23']), \
QuadraticForm(ZZ, 3, ['11', '6', '2', '11', '2', '19']), \
QuadraticForm(ZZ, 3, ['5', '2', '4', '5', '-4', '92']), \
QuadraticForm(ZZ, 3, ['7', '6', '2', '18', '6', '19']), \
QuadraticForm(ZZ, 3, ['7', '2', '2', '19', '14', '19']), \
QuadraticForm(ZZ, 3, ['12', '12', '12', '13', '6', '21']), \
QuadraticForm(ZZ, 3, ['13', '2', '2', '13', '2', '13']), \
QuadraticForm(ZZ, 3, ['9', '6', '3', '14', '14', '23']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '16', '0', '48']), \
QuadraticForm(ZZ, 3, ['4', '4', '0', '13', '0', '48']), \
QuadraticForm(ZZ, 3, ['5', '4', '2', '20', '20', '29']), \
QuadraticForm(ZZ, 3, ['7', '6', '0', '15', '0', '24']), \
QuadraticForm(ZZ, 3, ['7', '6', '4', '15', '-12', '28']), \
QuadraticForm(ZZ, 3, ['9', '6', '6', '17', '2', '17']), \
QuadraticForm(ZZ, 3, ['11', '8', '8', '16', '0', '16']), \
QuadraticForm(ZZ, 3, ['13', '2', '8', '13', '8', '16']), \
QuadraticForm(ZZ, 3, ['12', '12', '12', '17', '6', '17']), \
QuadraticForm(ZZ, 3, ['9', '6', '3', '11', '-4', '29']), \
QuadraticForm(ZZ, 3, ['5', '2', '3', '17', '9', '33']), \
QuadraticForm(ZZ, 3, ['9', '6', '0', '11', '0', '30']), \
QuadraticForm(ZZ, 3, ['11', '8', '2', '11', '-2', '26']), \
QuadraticForm(ZZ, 3, ['12', '12', '0', '13', '10', '25']), \
QuadraticForm(ZZ, 3, ['5', '0', '0', '24', '8', '24']), \
QuadraticForm(ZZ, 3, ['7', '0', '6', '16', '16', '31']), \
QuadraticForm(ZZ, 3, ['8', '0', '0', '15', '0', '24']), \
QuadraticForm(ZZ, 3, ['8', '8', '0', '17', '0', '24']), \
QuadraticForm(ZZ, 3, ['8', '4', '0', '17', '12', '24']), \
QuadraticForm(ZZ, 3, ['8', '0', '0', '21', '18', '21']), \
QuadraticForm(ZZ, 3, ['11', '8', '2', '16', '8', '19']), \
QuadraticForm(ZZ, 3, ['12', '12', '12', '15', '6', '23']), \
QuadraticForm(ZZ, 3, ['4', '4', '4', '31', '26', '31']), \
QuadraticForm(ZZ, 3, ['8', '8', '8', '20', '4', '23']), \
QuadraticForm(ZZ, 3, ['7', '2', '1', '13', '13', '37']), \
QuadraticForm(ZZ, 3, ['7', '4', '2', '20', '-4', '23']), \
QuadraticForm(ZZ, 3, ['7', '2', '2', '23', '-18', '23']), \
QuadraticForm(ZZ, 3, ['7', '0', '0', '15', '6', '30']), \
QuadraticForm(ZZ, 3, ['3', '2', '0', '19', '0', '56']), \
QuadraticForm(ZZ, 3, ['5', '4', '0', '12', '0', '56']), \
QuadraticForm(ZZ, 3, ['11', '6', '8', '11', '-8', '32']), \
QuadraticForm(ZZ, 3, ['12', '12', '8', '17', '4', '20']), \
QuadraticForm(ZZ, 3, ['7', '1', '6', '13', '-9', '39']), \
QuadraticForm(ZZ, 3, ['6', '6', '6', '19', '8', '34']), \
QuadraticForm(ZZ, 3, ['7', '2', '6', '13', '-12', '42']), \
QuadraticForm(ZZ, 3, ['10', '10', '0', '19', '6', '21']), \
```



```
QuadraticForm(ZZ, 3, ['11', '4', '2', '14', '14', '26']), \
QuadraticForm(ZZ, 3, ['5', '0', '2', '24', '0', '29']), \
QuadraticForm(ZZ, 3, ['8', '4', '8', '11', '8', '44']), \
QuadraticForm(ZZ, 3, ['9', '6', '0', '17', '0', '24']), \
QuadraticForm(ZZ, 3, ['11', '2', '8', '11', '-8', '32']), \
QuadraticForm(ZZ, 3, ['11', '6', '2', '15', '-6', '23']), \
QuadraticForm(ZZ, 3, ['15', '6', '12', '15', '12', '20']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '40', '40', '40']), \
QuadraticForm(ZZ, 3, ['10', '10', '10', '13', '8', '37']), \
QuadraticForm(ZZ, 3, ['11', '2', '10', '11', '-10', '35']), \
QuadraticForm(ZZ, 3, ['12', '12', '12', '13', '6', '33']), \
QuadraticForm(ZZ, 3, ['13', '4', '2', '13', '-2', '22']), \
QuadraticForm(ZZ, 3, ['7', '2', '6', '23', '-22', '31']), \
QuadraticForm(ZZ, 3, ['11', '2', '2', '19', '6', '19']), \
QuadraticForm(ZZ, 3, ['8', '0', '8', '9', '0', '56']), \
QuadraticForm(ZZ, 3, ['7', '0', '0', '24', '8', '24']), \
QuadraticForm(ZZ, 3, ['10', '4', '10', '13', '2', '34']), \
QuadraticForm(ZZ, 3, ['5', '2', '2', '29', '10', '29']), \
QuadraticForm(ZZ, 3, ['7', '2', '6', '19', '-18', '39']), \
QuadraticForm(ZZ, 3, ['11', '2', '6', '11', '6', '39']), \
QuadraticForm(ZZ, 3, ['7', '6', '0', '18', '0', '39']), \
QuadraticForm(ZZ, 3, ['5', '4', '0', '20', '0', '48']), \
QuadraticForm(ZZ, 3, ['5', '4', '2', '20', '20', '53']), \
QuadraticForm(ZZ, 3, ['8', '0', '0', '15', '6', '39']), \
QuadraticForm(ZZ, 3, ['8', '8', '8', '17', '10', '41']), \
QuadraticForm(ZZ, 3, ['12', '12', '0', '17', '16', '32']), \
QuadraticForm(ZZ, 3, ['15', '12', '12', '20', '8', '20']), \
QuadraticForm(ZZ, 3, ['17', '14', '4', '17', '-4', '20']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '40', '0', '120']), \
QuadraticForm(ZZ, 3, ['4', '4', '0', '11', '0', '120']), \
QuadraticForm(ZZ, 3, ['4', '4', '0', '31', '0', '40']), \
QuadraticForm(ZZ, 3, ['11', '8', '2', '16', '8', '31']), \
QuadraticForm(ZZ, 3, ['17', '14', '10', '17', '-10', '25']), \
QuadraticForm(ZZ, 3, ['13', '8', '10', '16', '8', '29']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '16', '16', '112']), \
QuadraticForm(ZZ, 3, ['7', '6', '2', '15', '-6', '55']), \
QuadraticForm(ZZ, 3, ['16', '16', '0', '19', '18', '27']), \
QuadraticForm(ZZ, 3, ['5', '2', '4', '17', '-16', '68']), \
QuadraticForm(ZZ, 3, ['13', '2', '6', '13', '-6', '33']), \
QuadraticForm(ZZ, 3, ['16', '0', '8', '16', '8', '23']), \
QuadraticForm(ZZ, 3, ['7', '0', '4', '15', '0', '52']), \
QuadraticForm(ZZ, 3, ['13', '2', '2', '22', '-16', '22']), \
QuadraticForm(ZZ, 3, ['5', '4', '2', '12', '12', '101']), \
QuadraticForm(ZZ, 3, ['8', '0', '8', '21', '14', '37']), \
QuadraticForm(ZZ, 3, ['12', '12', '0', '21', '12', '28']), \
QuadraticForm(ZZ, 3, ['5', '0', '0', '8', '8', '152']), \
QuadraticForm(ZZ, 3, ['5', '0', '0', '24', '24', '56']), \
QuadraticForm(ZZ, 3, ['21', '18', '18', '21', '2', '21']), \
QuadraticForm(ZZ, 3, ['9', '6', '6', '11', '-8', '71']), \
QuadraticForm(ZZ, 3, ['11', '8', '10', '16', '-8', '43']), \
QuadraticForm(ZZ, 3, ['11', '2', '8', '19', '-8', '32']), \
QuadraticForm(ZZ, 3, ['11', '3', '6', '15', '-3', '39']), \
QuadraticForm(ZZ, 3, ['5', '4', '4', '20', '-8', '68']), \
QuadraticForm(ZZ, 3, ['17', '4', '4', '20', '-8', '20']), \
QuadraticForm(ZZ, 3, ['3', '2', '0', '27', '0', '80']), \
QuadraticForm(ZZ, 3, ['11', '8', '4', '16', '16', '44']), \
QuadraticForm(ZZ, 3, ['17', '14', '16', '17', '16', '32']), \
QuadraticForm(ZZ, 3, ['8', '8', '0', '23', '6', '39']), \
QuadraticForm(ZZ, 3, ['1', '0', '0', '48', '0', '144']), \
QuadraticForm(ZZ, 3, ['5', '4', '2', '20', '20', '77']), \
QuadraticForm(ZZ, 3, ['7', '6', '4', '15', '-12', '76']), \
QuadraticForm(ZZ, 3, ['7', '4', '6', '28', '-12', '39']), \
QuadraticForm(ZZ, 3, ['9', '6', '0', '17', '0', '48']), \
QuadraticForm(ZZ, 3, ['11', '6', '8', '27', '-24', '32']), \
QuadraticForm(ZZ, 3, ['12', '12', '0', '23', '16', '32']), \
QuadraticForm(ZZ, 3, ['13', '8', '2', '16', '8', '37']), \
QuadraticForm(ZZ, 3, ['13', '6', '4', '21', '12', '28']), \
QuadraticForm(ZZ, 3, ['15', '6', '6', '23', '14', '23']), \
QuadraticForm(ZZ, 3, ['16', '8', '16', '19', '4', '28']), \
```

```
QuadraticForm(ZZ, 3, ['4', '4', '4', '43', '2', '43']), \
QuadraticForm(ZZ, 3, ['19', '18', '6', '19', '-6', '27']), \
QuadraticForm(ZZ, 3, ['7', '4', '2', '12', '12', '103']), \
QuadraticForm(ZZ, 3, ['11', '8', '2', '16', '8', '51']), \
QuadraticForm(ZZ, 3, ['15', '0', '10', '16', '16', '39']), \
QuadraticForm(ZZ, 3, ['7', '6', '4', '27', '-24', '52']), \
QuadraticForm(ZZ, 3, ['8', '8', '8', '28', '4', '41']), \
QuadraticForm(ZZ, 3, ['11', '10', '6', '19', '-14', '51']), \
QuadraticForm(ZZ, 3, ['5', '2', '4', '29', '-28', '68']), \
QuadraticForm(ZZ, 3, ['7', '0', '4', '24', '0', '52']), \
QuadraticForm(ZZ, 3, ['13', '0', '4', '24', '0', '28']), \
QuadraticForm(ZZ, 3, ['19', '14', '4', '19', '4', '28']), \
QuadraticForm(ZZ, 3, ['13', '4', '4', '28', '-16', '28']), \
QuadraticForm(ZZ, 3, ['13', '8', '4', '16', '16', '52']), \
QuadraticForm(ZZ, 3, ['13', '6', '10', '21', '6', '37']), \
QuadraticForm(ZZ, 3, ['17', '4', '2', '20', '20', '33']), \
QuadraticForm(ZZ, 3, ['8', '8', '8', '17', '4', '92']), \
QuadraticForm(ZZ, 3, ['9', '6', '6', '41', '-38', '41']), \
QuadraticForm(ZZ, 3, ['11', '2', '6', '26', '6', '39']), \
QuadraticForm(ZZ, 3, ['11', '10', '6', '35', '-30', '39']), \
QuadraticForm(ZZ, 3, ['25', '10', '20', '25', '-20', '28']), \
QuadraticForm(ZZ, 3, ['11', '2', '10', '19', '-18', '59']), \
QuadraticForm(ZZ, 3, ['17', '16', '8', '32', '32', '32']), \
QuadraticForm(ZZ, 3, ['21', '18', '18', '29', '26', '29']), \
QuadraticForm(ZZ, 3, ['15', '0', '6', '16', '16', '55']), \
QuadraticForm(ZZ, 3, ['15', '6', '12', '23', '-20', '44']), \
QuadraticForm(ZZ, 3, ['17', '4', '10', '20', '-12', '41']), \
QuadraticForm(ZZ, 3, ['13', '6', '8', '21', '12', '52']), \
QuadraticForm(ZZ, 3, ['11', '8', '4', '32', '-16', '44']), \
QuadraticForm(ZZ, 3, ['11', '10', '4', '35', '28', '44']), \
QuadraticForm(ZZ, 3, ['15', '12', '0', '20', '16', '56']), \
QuadraticForm(ZZ, 3, ['15', '6', '12', '23', '-4', '44']), \
QuadraticForm(ZZ, 3, ['3', '0', '0', '40', '0', '120']), \
QuadraticForm(ZZ, 3, ['7', '2', '4', '23', '12', '92']), \
QuadraticForm(ZZ, 3, ['12', '12', '0', '13', '0', '120']), \
QuadraticForm(ZZ, 3, ['12', '12', '0', '33', '0', '40']), \
QuadraticForm(ZZ, 3, ['13', '6', '2', '33', '-18', '37']), \
QuadraticForm(ZZ, 3, ['16', '16', '16', '19', '8', '64']), \
QuadraticForm(ZZ, 3, ['27', '12', '12', '28', '-24', '28']), \
QuadraticForm(ZZ, 3, ['9', '6', '6', '17', '-14', '113']), \
QuadraticForm(ZZ, 3, ['13', '2', '2', '37', '-22', '37']), \
QuadraticForm(ZZ, 3, ['20', '4', '4', '31', '-22', '31']), \
QuadraticForm(ZZ, 3, ['7', '4', '4', '52', '-16', '52']), \
QuadraticForm(ZZ, 3, ['8', '0', '8', '15', '0', '152']), \
QuadraticForm(ZZ, 3, ['15', '0', '0', '24', '24', '56']), \
QuadraticForm(ZZ, 3, ['17', '4', '6', '20', '12', '57']), \
QuadraticForm(ZZ, 3, ['17', '16', '10', '32', '16', '41']), \
QuadraticForm(ZZ, 3, ['11', '8', '10', '32', '8', '59']), \
QuadraticForm(ZZ, 3, ['11', '8', '4', '16', '16', '124']), \
QuadraticForm(ZZ, 3, ['16', '8', '16', '31', '4', '44']), \
QuadraticForm(ZZ, 3, ['13', '4', '2', '28', '28', '61']), \
QuadraticForm(ZZ, 3, ['8', '0', '8', '35', '0', '72']), \
QuadraticForm(ZZ, 3, ['16', '8', '16', '19', '4', '76']), \
QuadraticForm(ZZ, 3, ['12', '12', '12', '17', '6', '129']), \
QuadraticForm(ZZ, 3, ['12', '0', '12', '28', '28', '73']), \
QuadraticForm(ZZ, 3, ['17', '8', '4', '32', '32', '52']), \
QuadraticForm(ZZ, 3, ['13', '8', '4', '32', '32', '68']), \
QuadraticForm(ZZ, 3, ['19', '14', '12', '19', '-12', '84']), \
QuadraticForm(ZZ, 3, ['17', '16', '6', '32', '-16', '57']), \
QuadraticForm(ZZ, 3, ['23', '8', '4', '32', '32', '44']), \
QuadraticForm(ZZ, 3, ['23', '16', '22', '32', '16', '47']), \
QuadraticForm(ZZ, 3, ['11', '8', '4', '32', '32', '92']), \
QuadraticForm(ZZ, 3, ['16', '8', '8', '31', '22', '71']), \
QuadraticForm(ZZ, 3, ['27', '0', '18', '40', '40', '43']), \
QuadraticForm(ZZ, 3, ['19', '4', '10', '28', '-4', '67']), \
QuadraticForm(ZZ, 3, ['17', '10', '12', '41', '-36', '68']), \
QuadraticForm(ZZ, 3, ['9', '6', '0', '41', '0', '120']), \
QuadraticForm(ZZ, 3, ['11', '10', '0', '35', '0', '120']), \
QuadraticForm(ZZ, 3, ['25', '0', '20', '48', '48', '52']), \
```



```
QuadraticForm(ZZ, 3, ['36', '36', '24', '39', '12', '44']), \
QuadraticForm(ZZ, 3, ['16', '8', '8', '31', '-10', '103']), \
QuadraticForm(ZZ, 3, ['28', '28', '0', '37', '12', '60']), \
QuadraticForm(ZZ, 3, ['13', '4', '2', '28', '28', '157']), \
QuadraticForm(ZZ, 3, ['21', '6', '18', '29', '-26', '101']), \
QuadraticForm(ZZ, 3, ['21', '0', '12', '40', '40', '76']), \
QuadraticForm(ZZ, 3, ['24', '0', '24', '45', '30', '61']), \
QuadraticForm(ZZ, 3, ['13', '2', '12', '37', '-36', '132']), \
QuadraticForm(ZZ, 3, ['33', '24', '12', '48', '48', '52']), \
QuadraticForm(ZZ, 3, ['19', '6', '16', '27', '-24', '136']), \
QuadraticForm(ZZ, 3, ['19', '4', '2', '28', '28', '139']), \
QuadraticForm(ZZ, 3, ['39', '12', '12', '44', '8', '44']), \
QuadraticForm(ZZ, 3, ['28', '12', '12', '57', '-42', '57']), \
QuadraticForm(ZZ, 3, ['17', '4', '2', '20', '20', '257']), \
QuadraticForm(ZZ, 3, ['33', '12', '12', '52', '-8', '52']), \
QuadraticForm(ZZ, 3, ['9', '6', '6', '41', '-38', '281']), \
QuadraticForm(ZZ, 3, ['16', '8', '16', '43', '4', '172']), \
QuadraticForm(ZZ, 3, ['27', '12', '6', '28', '28', '187']), \
QuadraticForm(ZZ, 3, ['39', '12', '6', '44', '-36', '111']), \
QuadraticForm(ZZ, 3, ['52', '36', '24', '57', '-12', '84']), \
QuadraticForm(ZZ, 3, ['48', '24', '0', '73', '56', '112'])\
]


Jagy_regular_ternary_disc_list = [
2, 3, 4, 5, 6, 6, 7, 8, 8, 9, 10, 10, 11, 12, 12, 12, 12, 13, 14, 15, 15, 16, 16
, 17, 18, 18, 18, 18, 20, 20, 20, 20, 21, 21, 22, 24, 24, 24, 24, 25, 27, 27, 27
, 27, 27, 28, 28, 30, 30, 32, 32, 32, 32, 32, 33, 36, 36, 36, 36, 36, 36, 40, 40
, 42, 44, 44, 45, 45, 45, 46, 48, 48, 48, 48, 48, 48, 48, 49, 50, 50, 50, 52, 54
, 54, 54, 54, 56, 56, 60, 60, 60, 63, 63, 64, 64, 64, 64, 64, 64, 64, 70, 72, 72, 72
, 72, 72, 72, 75, 75, 78, 80, 80, 80, 80, 80, 81, 81, 84, 84, 90, 90, 92, 96, 96
, 96, 96, 96, 96, 98, 99, 100, 100, 100, 100, 100, 108, 108, 108, 108, 108, 108,
 108, 108, 108, 108, 112, 112, 112, 120, 120, 121, 125, 125, 126, 126, 128, 128,
 128, 128, 128, 128, 128, 132, 135, 135, 135, 140, 144, 144, 144, 144, 144, 144,
 144, 147, 147, 150, 150, 156, 160, 160, 162, 162, 162, 168, 169, 180, 180, 180,
 180, 189, 189, 189, 192, 192, 192, 192, 192, 192, 192, 192, 192, 192, 196, 196,
 200, 200, 208, 216, 216, 216, 216, 216, 216, 224, 224, 225, 225, 225, 234,
 240, 240, 240, 240, 242, 243, 243, 243, 250, 250, 252, 252, 252, 256, 256, 256,
 256, 256, 256, 256, 256, 270, 270, 270, 288, 288, 288, 288, 288, 288, 289, 294,
 297, 297, 300, 300, 300, 320, 320, 320, 320, 320, 324, 324, 324, 324, 336, 350,
 360, 360, 363, 375, 378, 378, 384, 384, 384, 384, 384, 384, 384, 392, 392, 396,
 400, 400, 400, 400, 400, 405, 405, 432, 432, 432, 432, 432, 432, 432, 432, 432,
 432, 432, 432, 432, 432, 432, 441, 441, 448, 448, 448, 450, 450, 468, 480, 480,
 484, 484, 486, 490, 500, 500, 500, 500, 504, 512, 512, 512, 512, 512, 512, 512,
 528, 540, 540, 540, 540, 576, 576, 576, 576, 576, 576, 576, 576, 576, 576, 576,
 576, 576, 588, 588, 600, 600, 640, 648, 648, 675, 675, 675, 676, 686, 686, 700,
 702, 704, 720, 720, 720, 720, 720, 750, 750, 750, 756, 756, 756, 756, 768, 768,
 768, 768, 768, 768, 768, 768, 768, 768, 784, 784, 784, 800, 800, 810, 864,
 864, 864, 864, 864, 864, 864, 864, 864, 864, 882, 882, 896, 900, 900, 900, 900,
 960, 960, 960, 960, 960, 972, 972, 972, 980, 1000, 1008, 1014, 1024, 1024, 1024
, 1024, 1024, 1024, 1024, 1058, 1080, 1080, 1089, 1125, 1125, 1125, 1134, 1152,
1152, 1152, 1152, 1152, 1152, 1152, 1176, 1188, 1200, 1200, 1200, 1280, 1280, 12
80, 1280, 1296, 1296, 1323, 1323, 1323, 1344, 1350, 1350, 1350, 1372, 1372, 1404
, 1440, 1440, 1452, 1472, 1500, 1500, 1500, 1500, 1512, 1536, 1536, 1536, 1536,
1536, 1536, 1536, 1568, 1568, 1584, 1600, 1600, 1600, 1600, 1600, 1600, 1620, 16
20, 1620, 1728, 1728, 1728, 1728, 1728, 1728, 1728, 1728, 1728, 1728, 1728, 1728
, 1728, 1728, 1728, 1728, 1728, 1764, 1764, 1764, 1792, 1792, 1792, 1792, 1800,
1800, 2000, 2000, 2025, 2028, 2048, 2048, 2048, 2048, 2048, 2048, 2048, 2048, 20
58, 2116, 2160, 2160, 2160, 2160, 2240, 2250, 2250, 2250, 2268, 2304, 2304, 2304
, 2304, 2304, 2304, 2304, 2304, 2304, 2352, 2400, 2400, 2430, 2450,
2496, 2592, 2646, 2646, 2700, 2700, 2700, 2700, 2704, 2880, 2880, 2880, 2880, 28
80, 3024, 3024, 3042, 3072, 3072, 3072, 3072, 3072, 3072, 3072, 3136, 3136, 3136
, 3200, 3267, 3328, 3375, 3456, 3456, 3456, 3456, 3456, 3456, 3456, 3528, 3600,
3600, 3600, 3600, 3840, 3840, 3840, 3840, 3840, 3888, 4032, 4032, 4032, 4050, 40
96, 4096, 4096, 4116, 4320, 4320, 4356, 4500, 4500, 4500, 4500, 4500, 4536, 4608
, 4608, 4608, 4608, 4608, 4608, 4608, 4752, 4800, 4800, 4800, 4800, 4800, 4860,
4900, 5120, 5120, 5120, 5184, 5184, 5184, 5184, 5292, 5292, 5292, 5292, 5376, 54
00, 5400, 5808, 6000, 6084, 6144, 6144, 6144, 6144, 6144, 6144, 6144, 6174, 6400
, 6400, 6400, 6400, 6480, 6750, 6750, 6750, 6750, 6912, 6912, 6912, 6912, 6912,
```

```
6912, 6912, 6912, 6912, 6912, 6912, 6912, 6912, 6912, 6912, 6912, 6912, 6912, 69
12, 6912, 6912, 7056, 7168, 7200, 7200, 7488, 7744, 7938, 8000, 8000, 8100, 8100
, 8192, 8192, 8232, 8448, 8448, 8640, 8640, 8640, 8640, 8640, 9126, 9216, 9216,
9216, 9216, 9216, 9216, 9216, 9408, 10125, 10584, 10800, 10800, 10800, 112
00, 11520, 11520, 11520, 11520, 11520, 11520, 11520, 12096, 12096, 12150, 12288,
 12288, 12348, 12544, 12544, 12544, 12544, 13068, 13500, 13500, 13500, 13500, 13
824, 13824, 13824, 13824, 13824, 13824, 14400, 14400, 14400, 14400, 14400, 15360
, 15360, 15552, 15680, 15876, 16128, 17424, 18000, 18252, 18432, 18432, 18432, 1
8432, 18432, 18432, 18432, 19200, 19200, 19200, 19200, 19200, 20480, 20736, 2073
6, 20736, 21168, 21168, 21504, 21600, 21600, 21952, 21952, 22464, 24000, 24000,
24000, 24300, 24576, 24576, 24696, 25344, 25344, 25600, 25600, 25600, 25920, 276
48, 27648, 27648, 27648, 27648, 27648, 27648, 27648, 27648, 27648, 27648, 28224,
 28224, 32000, 32000, 32000, 32400, 32448, 33856, 34560, 34560, 34560, 34560, 36
288, 36864, 36864, 37632, 43200, 43200, 43200, 43200, 43200, 43264, 46080, 46080
, 48384, 48384, 50176, 52272, 55296, 55296, 55296, 55296, 57600, 57600, 57600, 5
7600, 57600, 57600, 57600, 62208, 64512, 65856, 72000, 72000, 72000, 73728, 7372
8, 76032, 76800, 76800, 77760, 78400, 82944, 84672, 84672, 92928, 96000, 97344,
102400, 103680, 110592, 112896, 128000, 129600, 138240, 150528, 172800, 172800,
172800, 172800, 193536, 197568, 216000, 216000, 216000, 216000, 230400, 230400,
254016, 278784, 288000, 292032, 338688, 338688, 388800, 451584, 518400, 691200,
836352, 1354752]


Jagy_regular_ternary_non_cn1_indices = [12, 20, 23, 32, 38, 40, 41, 42, 43, 51,
65, 67, 73, 78, 86, 91, 93, 101, 102, 106, 110, 114, 129, 134, 136, 137, 138, 14
8, 149, 162, 163, 167, 173, 190, 208, 213, 217, 221, 225, 226, 233, 234, 235, 25
0, 252, 269, 281, 286, 289, 291, 292, 293, 296, 303, 307, 316, 337, 353, 355, 35
7, 359, 368, 378, 407, 417, 424, 425, 426, 432, 435, 436, 455, 457, 486, 493, 49
5, 517, 520, 525, 536, 545, 561, 577, 590, 604, 610, 613, 623, 625, 642, 646, 65
3, 675, 676, 677, 681, 683, 689, 709, 710, 718, 727, 734, 738, 746, 782, 788, 78
9, 805, 806, 808, 812, 813, 816, 826, 833, 855, 856, 889]
Jagy_regular_ternary_cn1_indices = [i  for i in range(913)  if not i in Jagy_reg
ular_ternary_non_cn1_indices]
Jagy_CN1_ternary_QF_list = [Jagy_regular_ternary_QF_list[i]  for i in Jagy_regul
ar_ternary_cn1_indices]
Jagy_nonCN1_ternary_QF_list = [Jagy_regular_ternary_QF_list[i]  for i in Jagy_re
gular_ternary_non_cn1_indices]
```

```
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())
```

```
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   1 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
```

```
## 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 *especially* 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.anisotropic_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))
```

```
def make_ternary_list_file_from_Jagy_raw_data(input_filename="regular_ternary_li
st_from_Jagy__2011_02_17.txt"):
    """
    Makes a Python list of regular ternary coefficients
    from Will Jagy's coeficient list.

    INPUT:
        input_filename -- the name of the Jagy datafile

    OUTPUT:
        none

    EXAMPLES:

    """
    raw_datafile = open(input_filename, 'r')
    regular_ternary_coeff_list = []
    regular_ternary_disc_list = []

    ## Read the lines of the raw datafile
    for line in raw_datafile:

        ## Parse the information from this line
        split_line = line.split()
        disc = split_line[0]
        coeffs = [split_line[1], split_line[6], split_line[5], split_line[2], sp
lit_line[4], split_line[3]]  ## Now in upper-triangular order! =)

        ## Append the data to our lists
        regular_ternary_coeff_list.append(coeffs)
        regular_ternary_disc_list.append(disc)

    raw_datafile.close()


    ## Write our lists to a python datafile
    outfilename = "Jagy_regular_ternary_list.sage"
    outfile = open(outfilename, 'w')
    outfile.write("""print "Loading Jagy's Regular Ternary Quadratic Forms from
""" + outfilename + '"\n\n\n')


    ## Write the coefficients list
    outfile.write("Jagy_regular_ternary_QF_list = [\n")
    ct = 0
    for c_list in regular_ternary_coeff_list:
        #print "ct = " + str(ct)
        outfile.write("QuadraticForm(ZZ, 3, " + str(c_list) + ")")
        if (ct != len(regular_ternary_coeff_list) - 1):
            outfile.write(", ")
        outfile.write("\\\n")
        ct += 1
    outfile.write("]\n\n\n")

    ## Write the disc list
    outfile.write("Jagy_regular_ternary_disc_list = [\n")
    ct = 0
    for D in regular_ternary_disc_list:
        outfile.write(str(D))
        if ct != len(regular_ternary_disc_list) - 1:
            outfile.write(",")
#       outfile.write("\\\n")
        ct += 1
    outfile.write("]\n\n\n")


    ## Write the lists of CN1 and non-CN1 ternaries
```



```
    CN1_string = """Jagy_regular_ternary_non_cn1_indices = [12, 20, 23, 32, 38,
40, 41, 42, 43, 51, 65, 67, 73, 78, 86, 91, 93, 101, 102, 106, 110, 114, 129, 13
4, 136, 137, 138, 148, 149, 162, 163, 167, 173, 190, 208, 213, 217, 221, 225, 22
6, 233, 234, 235, 250, 252, 269, 281, 286, 289, 291, 292, 293, 296, 303, 307, 31
6, 337, 353, 355, 357, 359, 368, 378, 407, 417, 424, 425, 426, 432, 435, 436, 45
5, 457, 486, 493, 495, 517, 520, 525, 536, 545, 561, 577, 590, 604, 610, 613, 62
3, 625, 642, 646, 653, 675, 676, 677, 681, 683, 689, 709, 710, 718, 727, 734, 73
8, 746, 782, 788, 789, 805, 806, 808, 812, 813, 816, 826, 833, 855, 856, 889]
Jagy_regular_ternary_cn1_indices = [i  for i in range(913)  if not i in Jagy_reg
ular_ternary_non_cn1_indices]
Jagy_CN1_ternary_QF_list = [Jagy_regular_ternary_QF_list[i]  for i in Jagy_regul
ar_ternary_cn1_indices]
Jagy_nonCN1_ternary_QF_list = [Jagy_regular_ternary_QF_list[i]  for i in Jagy_re
gular_ternary_non_cn1_indices]"""
    outfile.write(CN1_string)

    outfile.close()




## This tests that the discriminants we have are correct!
## -----------------------------------------------------
#[i  for i in range(913)  if Jagy_regular_ternary_QF_list[i].det() != 2 * Jagy_r
egular_ternary_disc_list[i]]
```

```
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..."
```

```
print "Loading watson_all_spsf_forms.sage"

##  This has the table of Watson's strongly primitive square-free discriminint f
orms
##  from his "One-class genera of positive quadratic formn in at least five vari
ables"
##  1975 Acta Arithmetica paper.



Watson_1975_raw_spsf_table1 = [ \
    ["dim", "index", "subform index", "QF border list", "disc"], \
    [1, 1, None, [1], 1], \
    [2, 2, 1, [1,1], -3], \
    [3, 3, 2, [1,1,1], -2], \
    [3, 4, 2, [0,0,1], -3], \
    [4, 5, 3, [0,1,1,1], 4], \
    [4, 6, 3, [1,1,1,1], 5], \
    [4, 7, 3, [0,0,0,1], 8], \
    [4, 8, 3, [0,1,1,2], 12], \
    [4, 9, 4, [0,0,1,1], 9], \
    [4, 10, 4, [0,0,0,1], 12], \
    [5, 11, 5, [1,1,1,1,1], 2], \
    [5, 12, 5, [0,0,0,0,1], 4], \
    [5, 13, 5, [1,1,1,1,2], 6], \
    [5, 14, 6, [1,1,1,1,1], 3], \
    [5, 15, 6, [0,0,0,0,1], 5], \
    [5, 16, 6, [0,0,1,1,2], 7], \
    [5, 17, 7, [0,0,0,1,1], 6], \
    [5, 18, 7, [0,1,1,1,2], 10], \
    [5, 19, 7, [0,0,1,1,2], 11], \
    [5, 20, 7, [0,1,1,0,2], 12], \
    [5, 21, 8, [1,1,1,-1,2], 15], \
    [5, 22, 9, [0,0,0,0,1], 9], \
    [5, 23, 10, [1,1,1,1,2], 14], \
    [5, 24, 10, [0,0,1,1,2], 18], \
    [6, 25, 11, [0,1,1,1,1,1], -3], \
    [6, 26, 11, [1,1,1,1,1,1], -4], \
    [6, 27, 11, [0,0,0,0,0,1], -8], \
    [6, 28, 11, [0,1,1,1,1,2], -11], \
    [6, 29, 11, [1,1,1,1,1,2], -12], \
    [6, 30, 12, [0,0,0,0,1,1], -12], \
    [6, 31, 12, [0,0,0,0,0,1], -16], \
    [6, 32, 14, [1,1,1,1,1,1], -7], \
    [6, 33, 14, [0,0,1,1,1,2], -15], \
    [6, 34, 15, [0,0,0,0,1,1], -15], \
    [6, 35, 15, [0,0,1,1,1,2], -23], \
    [6, 36, 17, [0,1,1,1,1,2], -28], \
    [6, 37, 22, [0,0,0,0,1,1], -27], \
    [6, 38, 24, [0,0,1,-1,0,2], -108], \
    [7, 39, 25, [0,1,1,1,1,1,1], -1], \
    [7, 40, 25, [0,0,0,0,0,0,1], -3], \
    [7, 41, 26, [1,1,1,1,1,1,1], -2], \
    [7, 42, 26, [0,0,0,0,0,0,1], -4], \
    [7, 43, 26, [1,0,0,0,0,0,2], -5], \
    [7, 44, 27, [0,0,0,0,0,1,1], -6], \
    [7, 45, 30, [0,0,0,0,1,1,1], -8], \
    [8, 46, 39, [0,0,0,0,0,0,1,1], 1], \
    [8, 47, 39, [0,0,0,0,0,0,1,2], 5], \
    [8, 48, 40, [0,0,0,0,0,0,1,1], 9], \
    [8, 49, 41, [1,1,1,1,1,1,1,1], 4], \
    [8, 50, 45, [0,0,0,0,0,1,1,1], 16], \
    [9, 51, 46, [0,0,0,0,0,0,0,0,1], 1], \
    [10, 52, 51, [0,0,0,0,0,0,0,1,1], -3] \
]


def make_watson_spsf_forms_of_dim(n):
    """
```



```
    Give the list of Watson's strongly primitive square-free
    quadratic forms in n variables.
    """
    Table = Watson_1975_raw_spsf_table1
    form_list = []

    ## Loop over all quadratic forms of that dimension
    for entry in Table:

        ## Make the upper-triangular matrix of coefficients
        if entry[0] == n:
            upper_triangular_matrix = Matrix(ZZ, n, n, 0)
            tmp_dim = n
            tmp_entry = entry
            while tmp_dim != 0:

#                print 'a'

                ## Write the present coefficients
                tmp_coeff_list = tmp_entry[3]
                for j in range(len(tmp_coeff_list)):
                    upper_triangular_matrix[j, tmp_dim - 1] = tmp_coeff_list[j]

#                print upper_triangular_matrix
#                print 'b'

                ## Move to the next (inner) entry
                tmp_dim += -1
                if tmp_dim != 0:
                    tmp_entry = Table[tmp_entry[2]]

#                print 'c'


            ## Make the associated quadratic form
            tmp_QF = QuadraticForm(ZZ, n)
            for i in range(n):
                for j in range(i, n):
                    tmp_QF[i,j] = upper_triangular_matrix[i,j]

#            print 'd'

            ## Add the quadratic form to our list
            form_list.append(tmp_QF)

    ## Return the list of assembled quadratic forms
    return form_list




## Test that all forms are accounted for, and have class number one!
#L = [make_watson_spsf_forms_of_dim(i)  for i in range(1,11)]
#LL = flatten(L)
#len(LL) == 52
#for Q in LL:
#    if Q.has_class_number_one():
#        print "ok"
#    else:
#        print "not ok"



## Compare these forms to the maximal lattices
```

```
#############################################################################
######
## Tables and forms from Watson's papers:
##      - 1963 "One-class genera of positive quadratic forms"
## Created on 02-11-2011 by Jonathan Hanke =)
#############################################################################
######

print "Loading watson_special_forms.sage"



D1 = DiagonalQuadraticForm(QQ, [1])
D8 = DiagonalQuadraticForm(QQ, [1] * 8)

M1 = Matrix(QQ, 8, 8, 1)
M1[7,7] = 0
for i in range(7):
    M1[i,7] = QQ(3)/8

M2 = Matrix(QQ, 1, 8, [1, 1, 1, 1, 1, 1, 1, QQ(21)/8])

M3 = Matrix(QQ, 1, 8, [0, 0, 0, 0, 0, 0, 0, QQ(1)/4])

## The three class number one forms listed in Watson's 1963 "One-class genera of
 positive quadratic forms" paper
Watson_E8 = QuadraticForm(ZZ, 8, ((D8(M1).scale_by_factor(QQ(1)/2)).sum_by_coeff
icients_with(D1(M2).scale_by_factor(QQ(1)/2)).sum_by_coefficients_with(D1(M3))).
coefficients())
Watson_F9 = Watson_E8 + QuadraticForm(ZZ, 1, [1])
Watson_F10 = Watson_E8 + QuadraticForm(ZZ, 2, [1,1,1])

#sage: Watson_E8.has_class_number_one()
#True
#sage: Watson_F9.has_class_number_one()
#True
#sage: Watson_F10.has_class_number_one()
#True




## Some perfect forms form Watson's 1963 "The class number of a positive quadrat
ic form" paper
def Watson_Phi(n):
    """
    Make the Watson perfect forms from (6.1) of p558 of his
    1963 "The class number of a positive quadratic form" paper.

    """
    coeff_list = [1] * ZZ(n*(n+1)/2)
    #for i in range(n):
    #    tmp_coeff_row = [1]*(n - i)
    #    coeff_list += tmp_coeff_row

    return QuadraticForm(ZZ, n, coeff_list)


def Watson_Phi_subst(n, u, c):
    """
    Make the Watson perfect forms from (6.2) on p558 of his
    1963 "The class number of a positive quadratic form" paper.

    """
    ## Check that u and c are compatible
    if not c>0:
        raise TypeError, "We must have c > 0."
```



```
    if (((n-1) * u**2 + c) % (2*n) != 0):
        raise TypeError, "The u and c given are not compatible!"


    ## Construct the ambient perfect form and its transformation matrix
    Phi = QuadraticForm(QQ, n-1, Watson_Phi(n-1).coefficients())
    M = Matrix(QQ, n-1, n)
    for i in range(n-1):
        M[i,i] = 1
        M[i,n-1] = QQ(u) / n

    ## Construct and return the desired form
    new_QF__QQ = Phi(M)
    new_QF__QQ[n-1,n-1] += QQ(c)/(2*n)
    return new_QF__QQ
    #return QuadraticForm(ZZ, n, new_QF__QQ.coefficients())



## Perfect forms form Lemma 7 on p559.
Watson_E6 = Watson_Phi_subst(6, 3, 3)
Watson_E7 = Watson_Phi_subst(7, 3, 2)
Watson_E8_again = Watson_Phi_subst(8, 3, 1)
Watson_E16 = Watson_Phi_subst(16, 7, 1)
Watson_E17 = Watson_Phi_subst(17, 6, 2)
Watson_E24_1 = Watson_Phi_subst(24, 5, 1)
Watson_E24_2 = Watson_Phi_subst(24, 11, 1)


#############################################################################
#####
## THERE IS A PROBLEM WITH THIS ONE...  MAYBE I'M MISUNDERSTANDING THE DEFINITIO
N? ##
#############################################################################
#####
## An additional 9-variable class number one form in his 1978 "One-class genera
of positive quadratic forms in nine and ten variables" paper, (2.11) on p 58
M4 = Matrix(QQ, 7, 9)
M4[6,8] = QQ(1)/2
for i in range(7):
    M4[i,i] = 1
M5 = Matrix(QQ, 1, 9, [0, 0, 0, 0, 0, 0, 0, 1, QQ(1)/2])
M6 = Matrix(QQ, 1, 9, [0, 0, 0, 0, 0, 0, 0, 0, 1])
#Watson_G9 = QuadraticForm(ZZ, 9, (Watson_E7(M4).sum_by_coefficients_with(D1(M5)
).sum_by_coefficients_with(D1(M6))).coefficients())
Watson_G9_rational= QuadraticForm(QQ, 9, (Watson_E7(M4).sum_by_coefficients_with
(D1(M5)).sum_by_coefficients_with(D1(M6))).coefficients())



################################
## Watson 7 variable CN1 forms ##
################################

## Note: Watson defines these as genera based on their local quadratic forms at
all primes.
## Only Table 2 entry 2(a) is self-reciprocal (sum of 7 squares), and we need to
 also take
## the reciprocal of all other forms in the three tables.  This will give 87 for
ms in total! =)

## Table 1
Q = QuadraticForm(ZZ, 1, [1])
P = QuadraticForm(ZZ, 2, [0,1,0])
B = QuadraticForm(ZZ, 2, [1,1,1])
Watson_7_variable_CN1_raw_table_1 = [ \
    ["(1)", 1, None], \
```

```
    ["(1)1", 9, P + P + P + Q.scale_by_factor(-9)], \
    ["(1)2", 4, P + P + P + Q.scale_by_factor(-4)], \
    ["(1)3", 4, P + P + B + Q.scale_by_factor(12)], \
    ["(1)4", 16, P + P + P.scale_by_factor(4) + Q.scale_by_factor(-1)], \
    ["(1)5", 64, P + P + P.scale_by_factor(4) + Q.scale_by_factor(-4)], \
    ["(1)6", 64, P + B + P.scale_by_factor(4) + Q.scale_by_factor(12)], \
    ["(1)7", 2**10, P + P.scale_by_factor(4) + P.scale_by_factor(4) + Q.scale_by
_factor(-4)], \
    ["(1)8", 2**10, B + P.scale_by_factor(4) + P.scale_by_factor(4) + Q.scale_by
_factor(12)], \
    ["(1)9", 16, P + P + Q + Q.scale_by_factor(-1) + Q.scale_by_factor(-4)], \
    ["(1)10", 256, P + P.scale_by_factor(4) + Q + Q.scale_by_factor(-1) + Q.scal
e_by_factor(-4)], \
    ["(1)11", 2**12, P.scale_by_factor(4) + P.scale_by_factor(4) + Q + Q.scale_b
y_factor(-1) + Q.scale_by_factor(-4)], \
    ["(2)", 2, P + P + Q.scale_by_factor(-2)], \
    ["(2)1", 8, P + P + Q.scale_by_factor(-8)], \
    ["(2)2", 256, P.scale_by_factor(2) + P.scale_by_factor(2) + Q + Q.scale_by_f
actor(-1) + Q.scale_by_factor(-4)], \
    ["(3)", 3, P + P + P + Q.scale_by_factor(-3)], \
    ["(3)1", 12, P + P + P + Q.scale_by_factor(-12)], \
    ["(4)", 4, P + P + P.scale_by_factor(2) + Q.scale_by_factor(-1)], \
    ["(4)1", 16, P + P + P.scale_by_factor(2) + Q.scale_by_factor(-4)], \
    ["(4)2", 16, P + B + P.scale_by_factor(2) + Q.scale_by_factor(12)], \
    ["(4)3", 128, P + P.scale_by_factor(2) + P.scale_by_factor(2) + Q.scale_by_f
actor(-8)], \
    ["(4)4", 64, P + P.scale_by_factor(2) + P.scale_by_factor(4) + Q.scale_by_fa
ctor(-1)], \
    ["(4)5", 256, P + P.scale_by_factor(2) + P.scale_by_factor(4) + Q.scale_by_f
actor(-4)], \
    ["(4)6", 256, B + P.scale_by_factor(2) + P.scale_by_factor(4) + Q.scale_by_f
actor(12)], \
    ["(4)7", 128, P.scale_by_factor(2) + P.scale_by_factor(2) + Q + Q.scale_by_f
actor(-1) + Q.scale_by_factor(-2)], \
    ["(4)8", 512, P.scale_by_factor(2) + P.scale_by_factor(2) + Q + Q.scale_by_f
actor(-1) + Q.scale_by_factor(-8)], \
    ["(4)9", 2**12, P.scale_by_factor(4) + P.scale_by_factor(4) + Q + Q.scale_by
_factor(-1) + Q.scale_by_factor(4)], \
    ["(4)10", 64, P + P.scale_by_factor(2) + Q + Q.scale_by_factor(-1) + Q.scale
_by_factor(-4)], \
    ["(4)11", 2**10, P.scale_by_factor(2) + P.scale_by_factor(4) + Q + Q.scale_b
y_factor(-1) + Q.scale_by_factor(-4)], \
    ["(5)", 5, P + P + P + Q.scale_by_factor(-5)], \
    ["(6)", 6, P + P + P + Q.scale_by_factor(-6)], \
    ["(7)", 8, P + P + P.scale_by_factor(2) + Q.scale_by_factor(-2)], \
    ["(7)1", 32, P + P + P.scale_by_factor(2) + Q.scale_by_factor(-8)], \
    ["(7)2", 64, P + P.scale_by_factor(2) + P.scale_by_factor(2) + Q.scale_by_fa
ctor(-4)], \
    ["(7)3", 64, B + P.scale_by_factor(2) + P.scale_by_factor(2) + Q.scale_by_fa
ctor(12)] \
]

## Table 2
Watson_7_variable_CN1_raw_table_2 = [ \
    ["(2)a", 64, P.scale_by_factor(2) + P.scale_by_factor(2) + P.scale_by_factor
(2) + Q.scale_by_factor(-1)], \
    ["(4)a", 32, P + P.scale_by_factor(2) + P.scale_by_factor(2) + Q.scale_by_fa
ctor(-2)], \
    ["(6)a", 192, P.scale_by_factor(2) + P.scale_by_factor(2) + P.scale_by_facto
r(2) + Q.scale_by_factor(-3)], \
    ["(7)a", 16, P + P.scale_by_factor(2) + P.scale_by_factor(2) + Q.scale_by_fa
ctor(-1)] \
]

## Table 3
Watson_7_variable_CN1_raw_table_3 = [ \
    ["(1)1b", 9 * 4**6, P.scale_by_factor(4) + P.scale_by_factor(4) + P.scale_by
_factor(4) + Q.scale_by_factor(-9)], \
    ["(3)b", 3**6, P.scale_by_factor(3) + P.scale_by_factor(3) + P.scale_by_fact
```



```
or(3) + Q.scale_by_factor(-1)], \
    ["(3)1b", 4 * 3**6, P.scale_by_factor(3) + P.scale_by_factor(3) + P.scale_by
_factor(3) + Q.scale_by_factor(-4)], \
    ["(5)b", 5 * 4**6, P.scale_by_factor(4) + P.scale_by_factor(4) + P.scale_by_
factor(4) + Q.scale_by_factor(-5)], \
    ["(6)b", 2 * 3**6, P.scale_by_factor(3) + P.scale_by_factor(3) + P.scale_by_
factor(3) + Q.scale_by_factor(-2)] \
]


def make_form_from_watson_7var_genus_information(neg_disc, local_quadratic_form_
at_disc):
    """
    Find a representative for a given Watson 7 variable genus of class number on
e.
    """
    H = HyperbolicPlane_quadratic_form(ZZ)
    local_generic_QF = H + H + H + QuadraticForm(ZZ, 1, [-neg_disc])

    ## Need to lift local quadratic forms to a global form... DO THIS!




################################
## Watson 8 variable CN1 forms ##
################################

## Note that here we must use Watson's "Integral Quadratic Forms" book, p52, Thr
m 30 to find the local behavior of Q at primes p where $p \nmid d$.

Watson_8_variable_CN1_raw_table_1_info = [ \
    ["(1)", 1, None], \
    ["(1)1", 9, P + P + P + Q + Q.scale_by_factor(-9)], \
    ["(1)2", 16, P + P + P + P.scale_by_factor(4)], \
    ["(1)3", 16, P + P + B + B.scale_by_factor(4)], \
    ["(1)4", 256, P + P + P.scale_by_factor(4) + P.scale_by_factor(4)], \
    ["(1)5", 256, P + B + P.scale_by_factor(4) + B.scale_by_factor(4)], \
    ["(1)6", 4, P + P + P + Q + Q.scale_by_factor(-1)], \
    ["(1)7", 64, P + P + P.scale_by_factor(4) + Q + Q.scale_by_factor(-1)], \
    ["(1)8", 4**8, P.scale_by_factor(4) + P.scale_by_factor(4) + P.scale_by_fact
or(4) + Q + Q.scale_by_factor(-4)], \
    ["(2)", 4, P + P + P + P.scale_by_factor(2)], \
    ["(2)1", 64, P + P + P.scale_by_factor(2) + P.scale_by_factor(4)], \
    ["(2)2", 64, P + B + P.scale_by_factor(2) + B.scale_by_factor(4)], \
    ["(2)3", 16, P + P + P.scale_by_factor(2) + Q + Q.scale_by_factor(-1)], \
    ["(2)4", 256, P + P.scale_by_factor(2) + P.scale_by_factor(4) + Q + Q.scale_
by_factor(-1)], \
    ["(2)5", 256, P.scale_by_factor(2) + P.scale_by_factor(2) + P.scale_by_facto
r(2) + Q + Q.scale_by_factor(-1)], \
    ["(3)", 5, P + P + P + Q + Q.scale_by_factor(-5)], \
    ["(4)", 9, P + P + B + B.scale_by_factor(-1)], \
    ["(4)1", 3**8, P.scale_by_factor(3) + P.scale_by_factor(3) + P.scale_by_fact
or(3) + Q + Q.scale_by_factor(-9)], \
    ["(5)", 16, P + P + P.scale_by_factor(2) + P.scale_by_factor(2)], \
    ["(5)1", 256, B + P.scale_by_factor(2) + P.scale_by_factor(2) + B.scale_by_f
actor(4)], \
    ["(5)2", 256, P + P.scale_by_factor(2) + P.scale_by_factor(2) + P.scale_by_f
actor(4)], \
    ["(5)3", 64, P + P.scale_by_factor(2) + P.scale_by_factor(2) + Q + Q.scale_b
y_factor(-1)] \
]



################################
## Watson 9 variable CN1 forms ##
```

```
################################
```

```
################################################################################
######
## Tables from Watson's "One-class genera of positive ternaries" papers
## Created on 9-29-2006 by Jonathan Hanke =)
## Revised/Updated on 5-20-2009
################################################################################
######


print "Loading watson_tables.sage"




## a11, a12, a13(should be a22),  a13, a23, a33  <--- Typo here!!!
Watson_1972_Table1_forms = [ \
[1, 1, 1,  0, 1, 1], \
[1, 1, 1,  0, 0, 1], \
[1, 1, 1,  0, 1, 2], \
[1, 1, 1,  0, 0, 2], \
[1, 1, 1,  0, 1, 5], \
[1, 1, 1,  0, 0, 10], \
#
[1, 0, 1,  1, 1, 2], \
[1, 0, 1,  0, 1, 2], \
[1, 0, 1,  1, 1, 3], \
[1, 0, 1,  0, 1, 4], \
[1, 0, 1,  1, 1, 11], \
#
[1, 1, 2,  0, 2, 2], \
[1, 1, 2,  0, 1, 2], \
[1, 1, 2,  0, 4, 7], \
#
[1, 0, 2,  1, 1, 3], \
[1, 0, 2,  1, 0, 3], \
[1, 0, 2,  1, 0, 9], \
#
[1, 1, 3,  0, 5, 5], \
[1, 1, 3,  0, 3, 5], \
#
[1, 1, 5,  0, 6, 6] \
]

## Note that Watson defines this as -1/2 * det(2Q), where 2Q is the doubled symm
etric Gram matrix of Q.
Watson_1972_Table1_dets = [ \
-2, -3, -5, -6, -14, -30, \
-6, -7, -10, -15, -42, \
-10, -13, -33, \
-21, -22, -70, \
-30, -46, \
-78\
]


## We assume that these are in the same form as above: a11, a12, a22,  a13, a23,
 a33  <--- Typo here!!!
Watson_1975_Table1_forms = [ \
## 1 - 5
[1, 0, 2,  1, 2, 3], \
[1, 0, 4,  1, 2, 5], \
[1, 0, 1,  0, 0, 8], \
[2, 2, 3,  2, 2, 7], \
[1, 0, 2,  0, 0, 16], \
## 6 - 10
[2, 0, 3,  0, 2, 11], \
[3, 2, 5,  0, 4, 10], \
```



```
[3, 0, 4,  2, 0, 11], \
[3, 2, 7,  2, -2, 7], \
[3, 2, 11,  2, -10, 19], \
## 11 - 15
[5, 4, 12,  4, -8, 12], \
[5, 0, 8,  2, 0, 13], \
[5, 4, 12,  4, 8, 20], \
[2, 1, 2,  2, 2, 8], \
[1, 1, 7,  1, 5, 13], \
## 16 - 20
[3, 0, 5,  0, 4, 8], \
[1, 1, 2,  0, 1, 3], \
[1, 0, 3,  0, 2, 7], \
[1, 1, 7,  0, 3, 7], \
[1, 1, 9,  0, 5, 15], \
## 21 - 25
[1, 0, 6,  1, 0, 7], \
[2, 2, 11,  2, 10, 23], \
[6, 6, 7,  6, 6, 15], \
[7, 6, 15,  2, -6, 19], \
[1, 1, 5,  1, -1, 7], \
## 26 - 30
[1, 1, 9,  0, 7, 21], \
[1, 0, 5,  0, 4, 12], \
[3, 2, 6,  2, 2, 7], \
[6, 6, 7,  0, 4, 8], \
[7, 4, 8,  2, 4, 19], \
## 31 - 35
[3, 0, 11,  0, 10, 35], \
[7, 6, 18,  2, 6, 19], \
[3, 3, 17,  6, 8, 38], \
[1, 1, 7,  1, 5, 31], \
[5, 2, 7,  2, -6, 13], \
## 36 - 40
[5, 2, 13,  0, 0, 24], \
[5, 2, 13,  4, -12, 28], \
[7, 4, 20,  2, -4, 23], \
[1, 0, 9,  0, 0, 24], \
[2, 2, 5,  0, 0, 24], \
## 41 - 45
[7, 2, 10,  6, -6, 15], \
[6, 0, 9,  0, 6, 17], \
[5, 0, 6,  2, 0, 29], \
[7, 2, 13,  6, 6, 21], \
[7, 4, 10,  4, 8, 28], \
## 46 - 50
[11, 6, 15,  2, -6, 23], \
[8, 4, 11,  8, 8, 44], \
[11, 10, 35,  4, 28, 44], \
[15, 12, 20,  0, 16, 56], \
[1, 1, 3,  1, -1, 3], \
## 51 - 55
[1, 1, 9,  1, -7, 9], \
[3, 2, 7,  1, -3, 13], \
[5, 5, 5,  2, -2, 8], \
[7, 1, 7,  2, -2, 8], \
[1, 0, 4,  1, 2, 11], \
## 56 - 60
[1, 0, 9,  1, 9, 13], \
[2, 2, 5,  2, 1, 11], \
[5, 2, 10,  1, 10, 13], \
[1, 0, 5,  0, 0, 8], \
[3, 2, 6,  0, 6, 11], \
## 61 - 65
[1, 1, 3,  1, 0, 5], \
[5, 2, 8,  3, 6, 9], \
[1, 1, 13,  0, 3, 15], \
[1, 0, 2,  1, 0, 3], \
[1, 0, 2,  1, 0, 9], \
```

```
## 66 - 68
[5, 5, 17,  5, -2, 23], \
[1, 1, 3,  0, 3, 5], \
[5, 4, 5,  3, 3, 9] \
]

## Note that Watson defines this as -1/2 * det(2Q), where 2Q is the doubled symm
etric Gram matrix of Q.
Watson_1975_Table1_dets = [ \
-18, -72, -32, -128, -128, \
-256, -512, -512, -512, -2048, \
-2048, -2048, -4096, -108, -324, \
-432, -20, -80, -180, -500, \
-162, -1728, -1728, -6912, -126, \
-686, -224, -448, -960, -3840, \
-4320, -8640, -6750, -810, -1536, \
-6144, -6144, -12288, -864, -864, \
-3456, -3456, -3456, -6912, -6912, \
-13824, -13824, -55296, -55296, -28, \
-250, -1000, -540, -1500, -168, \
-378, -378, -2058, -160, -640, \
-52, -1188, -756, -22, -70, \
-6750, -46, -702]


## Watson's incomplete list of class number 1 quaternaries
Watson_4var_Table_forms = [ \
## 1 - 5
[1, 1, 1,  1, 1, 1,  0, 1,  1, 1], \
[1, 1, 1,  1, 1, 1,  1, 1,  1, 1], \
[1, 1, 1,  1, 1, 1,  0, 0,  0, 1], \
[1, 1, 1,  1, 1, 1,  0, 1,  1, 2], \
[1, 1, 1,  1, 1, 1,  1, 1,  1, 2], \
## 6 - 10
[1, 1, 1,  1, 1, 1,  0, 1,  1, 3], \
[1, 1, 1,  1, 1, 1,  1, 1,  1, 3], \
[1, 1, 1,  0, 0, 1,  0, 0,  1, 1], \
[1, 1, 1,  0, 0, 1,  0, 0,  0, 1], \
[1, 1, 1,  0, 0, 1,  1, 1,  1, 2], \
## 11 - 15
## d4 check failed for Form #14 (see below)
[1, 1, 1,  0, 0, 1,  0, 0,  1, 2], \
[1, 1, 1,  0, 0, 1,  0, 0,  1, 4], \
[1, 1, 1,  1, 1, 2,  1, 1,  -1, 2], \
[1, 1, 1,  1, 1, 2,  1, 1,  2, 1], \
[1, 1, 1,  0, 0, 2,  0, 0,  2, 2], \
## 16 - 20
[1, 1, 1,  0, 0, 2,  0, 0,  1, 2], \
[1, 0, 1,  0, 0, 1,  1, 1,  1, 2], \
[1, 0, 1,  0, 0, 1,  0, 1,  1, 2], \
[1, 0, 1,  1, 1, 2,  0, 1,  -1, 2], \
[1, 0, 1,  1, 1, 2,  1, 1,  1, 2], \
## 21 - 25
## Check failed for d3 and d4 of Watson's #23 (see below)
[1, 0, 1,  0, 1, 2,  1, 0,  0, 2], \
[1, 0, 1,  0, 1, 2,  1, 1,  1, 3], \
[1, 0, 1,  1, 1, 1,  1, 1,  1, 3], \
[1, 1, 2,  0, 2, 2,  0, 1,  2, 2], \
[1, 1, 2,  0, 2, 2,  0, 1,  2, 3], \
## 26 - 27
[1, 1, 2,  0, 1, 2,  1, 1,  2, 4], \
[1, 0, 2,  1, 0, 3,  0, 2,  0, 6] \
]

Watson_4var_Table_dets = [ \
## 1 - 7
[-3, -2, 4], \
[-3, -2, 5], \
[-3, -2, 8], \
```



```
[-3, -2, 12], \
[-3, -2, 13], \
[-3, -2, 20], \
[-3, -2, 21], \
## 8 - 12
[-3, -3, 9], \
[-3, -3, 12], \
[-3, -3, 17], \
[-3, -3, 21], \
[-3, -3, 45], \
## 13 - 14
## Check failed for Watson's #14 -> [-3, -5, 28]
[-3, -5, 25], \
[-3, -5, 8], \
## 15 - 16
[-3, -6, 36], \
[-3, -6, 45], \
## 17 - 18
[-4, -4, 20], \
[-4, -4, 24], \
## 19 - 20
[-4, -6, 33], \
[-4, -6, 36], \
## 21 - 22
[-4, -7, 49], \
[-4, -7, 69], \
## 23
## Check failed for Watson's #13 -> [-4, -10, 100]
[-4, -2, 20], \
## 24 - 25
[-7, -10, 60], \
[-7, -10, 100], \
## 26
[-7, -13, 169], \
## 27
[-8, -22, 484] \
]



def parse_Watson_ternary_coefficients(coeff_list, Watson_det = None):
    """
    Takes a list of (integer) coefficients

       [a11, a12, a22,  a13, a23, a33]

    as in Watson's 1972 and 1975 ternary papers, and creates a
    quadratic lattice from them.

    If Watson's determinant is passed, then this is checked against
    the form.
    """
    ## Check that we have 6 coefficients
    if not (isinstance(coeff_list, (list, tuple)) and len(coeff_list) == 6):
        raise TypeError, "Oops! You need to pass in a 6 element list/tuple!"

    ## Check that all coefficients are integers
    for i in range(6):
        if not coeff_list[i] in ZZ:
            raise TypeError, "Oops! Every coefficient must be an integer!"

    ## Construct the associated doubled matrix
    matrix_coeffs = [ 2 * coeff_list[0], coeff_list[1], coeff_list[3], \
                      coeff_list[1], 2 * coeff_list[2], coeff_list[4], \
                      coeff_list[3], coeff_list[4], 2 * coeff_list[5] ]
    A = MatrixSpace(ZZ,3,3)(matrix_coeffs)

    ## Check it's determinant
    if Watson_det != None  and  (ZZ(-1)/2) * A.det() != Watson_det:
```

```
        #print "Oops!  The matrix \n" + str(A) + " has Watson determinant " + st
r((ZZ(-1)/2) * A.det()) + ", but should have Watson determinant " + str(Watson_d
et) + "!\n"
        raise RuntimeError, "Oops!  The matrix \n" + str(A) + " has Watson deter
minant " + str((ZZ(-1)/2) * A.det()) + ", but should have Watson determinant " +
 str(Watson_det) + "!"

    ## Return the quadratic lattice
    return QuadraticForm(A)



def parse_Watson_quaternary_coefficients(coeff_list, Watson_det_list = None):
    """
    Takes a list of (integer) coefficients

       [a11, a12, a22,  a13, a23, a33,  a14, a24, a34, a44]

    as in Watson's 1974 paper, and creates a quadratic lattice from
    them.

    If the list of Watson's determinants [d2, d3, d4] is passed, then
    these are checked against the form.
    """
    ## Check that we have 10 coefficients
    if not (isinstance(coeff_list, (list, tuple)) and len(coeff_list) == 10):
        raise TypeError, "Oops! You need to pass in a 10 element list/tuple!"

    ## Check that all coefficients are integers
    for i in range(10):
        if not coeff_list[i] in ZZ:
            raise TypeError, "Oops! Every coefficient must be an integer!"

    ## Construct the associated doubled matrix
    matrix_coeffs = [ 2 * coeff_list[0], coeff_list[1], coeff_list[3], coeff_lis
t[6], \
                      coeff_list[1], 2 * coeff_list[2], coeff_list[4], coeff_lis
t[7], \
                      coeff_list[3], coeff_list[4], 2 * coeff_list[5], coeff_lis
t[8], \
                      coeff_list[6],  coeff_list[7], coeff_list[8], 2 * coeff_li
st[9] ]
    A = MatrixSpace(ZZ,4,4)(matrix_coeffs)

    ## Check it's determinants
    if Watson_det_list != None:

        ## 2 x 2
        det2 = ZZ(-1) * A.matrix_from_rows_and_columns([0,1], [0,1]).det()
        if Watson_det_list[0] != det2:
            raise RuntimeError, "Oops!  The matrix \n" + str(A) + " should have
2x2 Watson determinant " \
                  + str(Watson_det_list[0]) + "!\n  We computed " + str(det2) +
"."


        ## 3 x 3
        det3 = (ZZ(-1)/2) * A.matrix_from_rows_and_columns([0,1,2], [0,1,2]).det
()
        if Watson_det_list[1] != det3:
            raise RuntimeError, "Oops!  The matrix \n" + str(A) + " should have
3x3 Watson determinant " \
                  + str(Watson_det_list[1]) + "!\n  We computed " + str(det3) +
"."

        ## 4 x 4
        if Watson_det_list[2] != A.det():
            raise RuntimeError, "Oops!  The matrix \n" + str(A) + " should have
Watson determinant " \
                  + str(Watson_det_list[2]) + "!\n  We computed " + str(A.det())
```



```
 + "."

    ## Return the quadratic lattice
    return QuadraticForm(A)



## Parse these as QuadraticLattice objects
W72_forms = [parse_Watson_ternary_coefficients(Watson_1972_Table1_forms[i], Wats
on_1972_Table1_dets[i]) \
             for i in range(len(Watson_1972_Table1_forms))]
W75_forms = [parse_Watson_ternary_coefficients(Watson_1975_Table1_forms[i], Wats
on_1975_Table1_dets[i]) \
             for i in range(len(Watson_1975_Table1_forms))]

W4_forms = [parse_Watson_quaternary_coefficients(Watson_4var_Table_forms[i], Wat
son_4var_Table_dets[i]) \
             for i in range(len(Watson_4var_Table_forms))]

### Check the integrity of the tables
#W72_dets = [(-1/2) * f.gram_matrix().det()  for f in W72_forms]
#W75_dets = [(-1/2) * f.gram_matrix().det()  for f in W75_forms]
#assert W72_dets == Watson_1972_Table1_dets
#assert W75_dets == Watson_1975_Table1_dets

## Make an associated list of maximal lattices
#time W75_maximal = [form.maximal_superlattice() for form in W75_forms]

## Find the associated quadratic spaces
#W72_spaces = [W72_forms[i].base_change_to(QQ)  for i in range(len(W72_forms))]
#W75_spaces = [W75_forms[i].base_change_to(QQ)  for i in range(len(W75_forms))]
#
#W4_spaces = [W4_forms[i].base_change_to(QQ)  for i in range(len(W4_forms))]



## Find the associated list of quadratic spaces
W72_spaces = [QuadraticSpace(QQ, W72_forms[i])  for i in range(len(W72_forms))]
W75_spaces = [QuadraticSpace(QQ, W75_forms[i])  for i in range(len(W75_forms))]

W4_spaces = [QuadraticSpace(QQ, W4_forms[i])  for i in range(len(W4_forms))]




## Check Watson's 1975 paper Table 2 -- has mistakes! =(
# time W75_Table2 = [[i+1, j+1] for i in range(68)  for j in range(20)  if W75_s
paces[i] == W72_spaces[j]]
### Wall time: 50.91
# assert len(W75_Table2) == 68   ## Nope. It only has 57 forms... so 11 are miss
ing...
#######
# time W75_doubled_Table2 = [[i+1, j+1] for i in range(68)  for j in range(20)
if W75_spaces[i].scale_by(2) == W72_spaces[j]]



## Check that Watson's 20 forms from 1972 have distinct quadratic spaces -- YES!
 =)
# time is_distinct(W72_spaces)
### Wall time: 7.37




def find_distinct_quadratic_space_indices_in_list(L):
    """
    Find representatives in a list of quadratic spaces up to isomorphism.
```

```
    """
    distinct_indices = []
    for i in range(len(L)):

        ## Look for repeated quadratic spaces
        distinct_flag = True
        for j in range(i):
            if L[i].is_isomorphic_to(L[j]):
                distinct_flag = False
                break

        ## Store the index if there are no isomorphic spaces before it.
        if distinct_flag:
            distinct_indices.append(i)

    ## Return the list of distinct indices
    return distinct_indices




## Make tables for Magma
def MakeWatsonForMAGMA(filename = 'watson_tables.magma'):
    """
    Writes Watson's tables of forms to a MAGMA readable file.
    """
    ## Check that filename is a short string ending in .m or .magma
    if not (isinstance(filename, str)  and len(filename) < 40  \
            and (filename.endswith(".m") or filename.endswith(".magma")) ):
        raise TypeError, "Oops!  We didn't like that filename...  Try again!"

    ## Make the 1972 Table 1 ternary forms
    magma_72_forms = "Watson_1972_Table1_forms := [ \n"
    for i in range(len(Watson_1972_Table1_forms)):
        coeff_list = Watson_1972_Table1_forms[i]
        matrix_coeffs = [ 2 * coeff_list[0], coeff_list[1], coeff_list[3], \
                      coeff_list[1], 2 * coeff_list[2], coeff_list[4], \
                      coeff_list[3], coeff_list[4], 2 * coeff_list[5] ]
        magma_72_forms += str(matrix_coeffs)
        if i != len(Watson_1972_Table1_forms) - 1:
            magma_72_forms += ', '
        magma_72_forms += '\n'
    magma_72_forms += ']; \n\n'

    ## Make the 1975 Table 1 ternary forms
    magma_75_forms = "Watson_1975_Table1_forms := [ \n"
    for i in range(len(Watson_1975_Table1_forms)):
        coeff_list = Watson_1975_Table1_forms[i]
        matrix_coeffs = [ 2 * coeff_list[0], coeff_list[1], coeff_list[3], \
                      coeff_list[1], 2 * coeff_list[2], coeff_list[4], \
                      coeff_list[3], coeff_list[4], 2 * coeff_list[5] ]
        magma_75_forms += str(matrix_coeffs)
        if i != len(Watson_1975_Table1_forms) - 1:
            magma_75_forms += ', '
        magma_75_forms += '\n'
    magma_75_forms += ']; \n\n'


    ## Make the 1974 Table of quaternary forms
    magma_4_forms = "Watson_4var_Table_forms := [ \n"
    for i in range(len(Watson_4var_Table_forms)):
        coeff_list = Watson_4var_Table_forms[i]
        matrix_coeffs = [ 2 * coeff_list[0], coeff_list[1], coeff_list[3], coeff
_list[6], \
                      coeff_list[1], 2 * coeff_list[2], coeff_list[4], coeff_lis
t[7], \
                      coeff_list[3], coeff_list[4], 2 * coeff_list[5], coeff_lis
```



```
t[8], \
                      coeff_list[6],  coeff_list[7], coeff_list[8], 2 * coeff_li
st[9] ]
        magma_4_forms += str(matrix_coeffs)
        if i != len(Watson_4var_Table_forms) - 1:
            magma_4_forms += ', '
        magma_4_forms += '\n'
    magma_4_forms += ']; \n\n'


    ## Write code to translate them into MAGMA quadratic lattices
    magma_parsing = \
                    """
// Basic Definitions
Z := Integers();
Q := Rationals();
QQ3 := RMatrixSpace(Q,3,3);
QQ4 := RMatrixSpace(Q,4,4);
ThetaRing<q>:=PowerSeriesRing(Q);

// Make Watson's two tables of forms
W72_forms := [ LatticeWithGram(QQ3 ! Matrix(3, entry)) : entry in Watson_1972_Ta
ble1_forms];
W72_form_class_numbers := [ #GenusRepresentatives(L) : L in W72_forms];

W75_forms := [ LatticeWithGram(QQ3 ! Matrix(3, entry)) : entry in Watson_1975_Ta
ble1_forms];
W75_form_class_numbers := [ #GenusRepresentatives(L) : L in W75_forms];

W4_forms := [ LatticeWithGram(QQ4 ! Matrix(4, entry)) : entry in Watson_4var_Tab
le_forms];
W4_form_class_numbers := [ #GenusRepresentatives(L) : L in W4_forms];

//AvgAuto := &+[ 1 / #AutomorphismGroup(L) : L in Gen];
"""

    ## Write them to the specified (MAGMA) file
    f = open(filename, 'w')
    f.write(magma_72_forms)
    f.write(magma_75_forms)
    f.write(magma_4_forms)
    f.write(magma_parsing)
    f.close()
```