```
In [1]: %load_ext sympyprinting
import time
# import the PagodaFunction class and additional helpers for plotting
from PagodaFunction import PagodaFunction
from Plotting import plotHierarchical2d, evaluateSparseGrid, printSparseGrid, plotCombi2d
from CombiGrid import createCombiGrid, evaluateCombiGridOnFullGrid, createFullGrid
```

Exercise 1: Hierarchization in Higher Dimensions

In this exercise we will implement the multi-recursive algorithm for hierarchization of a multi-dimensional regular sparse grid. The structure of the code resembles strongly the one-dimensional case, and so we again have a class (`PagodaFunction`) representing our grid points.

```
In [2]: def parabola(xvec):
    '''The parabola test function in multiple dimensions'''
    result = 1.0
    for x in xvec:
        result *= 4.0*x*(1.0-x)
    return result

def applyRefinementCriterion(sg, dim, criterion, func):
    '''
    Applies the criterion to the leaves of the sparse grid.

    Calls the "apply" member for each leaf of the grid,
    then finishes calling the "finalize" member passing the
    grid and the approximated function as arguments.
    We'll also use it to build our sparse grid.
    '''
    # iterate over leaves only
    for (lKey,iKey) in sg:
        l, i = list(lKey), list(iKey)
        isLeaf = True
        for d in xrange(dim):
            l[d] = l[d] + 1
            i[d] = 2*i[d] - 1
            leftChild =  (tuple(l), tuple(i))
            i[d] = i[d] + 2
            rightChild =  (tuple(l), tuple(i))
            l[d] = l[d] - 1
            i[d] = i[d] >> 1
            isLeaf = isLeaf and not (sg.has_key(leftChild) and sg.has_key(rightChild))
        if isLeaf:
            criterion.apply(sg[key])
    criterion.finalize(sg, func)
```

(i)

Implement the refinement criterion `MinLevelCriterion` that adds all points up to a specified level to a given grid.
Hint: In your grid traversal, try to avoid multiple visits to the same grid points.

```
In [3]: class MinLevelCriterion(object):
    '''Criterion adding all missing points up to a certain level.'''
    def __init__(self, dim, minLevel, verbose=False):
        self.dim = dim
        self.minLevel = minLevel
        self.verbose = verbose
    def apply(self, af):
        pass # work is done in finalize
    def finalize(self, sg, func):
        '''Inserts the grid points recursively'''
        def insertRecursively(minD, l, lvec, ivec, xvec):
            '''
            Local recursive helper function for grid traversal.

            Use lvec and ivec (mutable lists) to identify the current point.
            Insert it if it does not exist, descend recursively if not
            on level minLevel. minD helps to ensure to not pay any point
            multiple visits.
            '''
            key = (tuple(lvec), tuple(ivec))
            if not sg.has_key(key):
                sg[key] = PagodaFunction(self.dim, lvec, ivec, func(xvec))
            if self.verbose:
                print " " * (2 * (l - 1)), "Insert grid point", key
            # if not on highest refinement level, insert the children
            if l < self.minLevel:
                for d in xrange(minD, self.dim):
                    x = xvec[d]
                    # insert left child
                    lvec[d] += 1
                    ivec[d] = (2*ivec[d] - 1)
                    xvec[d] = x - 1.0 / (1 << lvec[d])
                    insertRecursively(d, l + 1, lvec, ivec, xvec)
                    # insert right child
                    ivec[d] += 2
                    xvec[d] = x + 1.0 / (1 << lvec[d])
                    insertRecursively(d, l + 1, lvec, ivec, xvec)
                    # reset vectors to original state
                    lvec[d] -= 1
                    ivec[d] >>= 1
                    xvec[d] = x
        # issue the top level call
        insertRecursively(0, 1, [1,]*self.dim, [1,] * self.dim, [0.5,] * self.dim)
```

(ii)

Implement the function `hierarchize` efficiently using a recursive approach.
Hint: The underlying traversal algorithm can be implemented similar to the one above.

In [4]:
```
def hierarchize(sg, dim, verbose=False):
    '''Recursive hierarchization function.'''
    def hierarchize1d(workDim, lvec, ivec, f_l, f_r):
        '''Recursive 1-D hierarchization to (lvec,ivec) in dimension workDim.'''
        key = (tuple(lvec), tuple(ivec))
        if not sg.has_key(key):
            return False
        af = sg[key]
        f_m = af.getSurplus()
        lvec[workDim] += 1
        ivec[workDim] = 2*ivec[workDim] - 1
        hierarchize1d(workDim, lvec, ivec, f_l, f_m)
        ivec[workDim] += 2
        hierarchize1d(workDim, lvec, ivec, f_m, f_r)
        lvec[workDim] -= 1
        ivec[workDim] = ivec[workDim] >> 1
        af.setSurplus(f_m - 0.5*(f_l+f_r))
        return True

    def traverseRecursively(workDim, minD, lvec, ivec, verbose = False):
        '''
        Local recursive helper function for traversal of the "main axis" w.r.t. workDim.

        Use the same approach as in MinLevelCriterion.finalize to traverse only the main
        axis of the grid, i.e. all points with coordinate x_workDim=0.5
        Hint: Don't descend into workDim, as hierarchize1d will do that.
        '''
        key = (tuple(lvec), tuple(ivec))
        # apply 1d hierarchization!
        if not hierarchize1d(workDim, lvec, ivec, 0.0, 0.0):
            return
        # being here => point exists
        if verbose:
            print " " * (2 * (sum(lvec)-dim + 1 - 1)), "dimension %d: hierarchize" % workDim, key

        for d in xrange(minD, dim):
            # skip the work dimension if necessary!
            if d != workDim:
                # descend into left child
                lvec[d] += 1
                ivec[d] = 2*ivec[d] - 1
                traverseRecursively(workDim, d, lvec, ivec, verbose)
                # descend into right child
                ivec[d] += 2
                traverseRecursively(workDim, d, lvec, ivec, verbose)
                # reset vectors to original state
                lvec[d] -= 1
                ivec[d] >>= 1
    # apply the 1-D hierarchization scheme for all dimensions, one after another
    lvec, ivec = [1,]*dim, [1,] * dim
    for d in xrange(dim):
        traverseRecursively(d, 0, lvec, ivec, verbose)
```

In [5]:
```
func = parabola
sg = {}
dim = 2
minLevel = 4
applyRefinementCriterion(sg, dim, MinLevelCriterion(dim, minLevel, verbose=False), func)
printSparseGrid(sg)
```

```
((2, 1), (3, 1)) : u_h( [0.75, 0.5] )= 0.75
((3, 1), (7, 1)) : u_h( [0.875, 0.5] )= 0.4375
((2, 3), (1, 7)) : u_h( [0.25, 0.875] )= 0.328125
((1, 2), (1, 3)) : u_h( [0.5, 0.75] )= 0.75
((3, 1), (3, 1)) : u_h( [0.375, 0.5] )= 0.9375
((2, 3), (3, 1)) : u_h( [0.75, 0.125] )= 0.328125
((4, 1), (15, 1)) : u_h( [0.9375, 0.5] )= 0.234375
((3, 2), (1, 3)) : u_h( [0.125, 0.75] )= 0.328125
((1, 4), (1, 13)) : u_h( [0.5, 0.8125] )= 0.609375
((4, 1), (5, 1)) : u_h( [0.3125, 0.5] )= 0.859375
((1, 4), (1, 11)) : u_h( [0.5, 0.6875] )= 0.859375
((3, 2), (1, 1)) : u_h( [0.125, 0.25] )= 0.328125
((1, 1), (1, 1)) : u_h( [0.5, 0.5] )= 1.0
((1, 4), (1, 7)) : u_h( [0.5, 0.4375] )= 0.984375
((3, 1), (1, 1)) : u_h( [0.125, 0.5] )= 0.4375
((2, 2), (1, 3)) : u_h( [0.25, 0.75] )= 0.5625
((3, 2), (5, 1)) : u_h( [0.625, 0.25] )= 0.703125
((2, 3), (3, 3)) : u_h( [0.75, 0.375] )= 0.703125
((4, 1), (7, 1)) : u_h( [0.4375, 0.5] )= 0.984375
((2, 1), (1, 1)) : u_h( [0.25, 0.5] )= 0.75
((1, 4), (1, 15)) : u_h( [0.5, 0.9375] )= 0.234375
((4, 1), (9, 1)) : u_h( [0.5625, 0.5] )= 0.984375
((2, 3), (3, 7)) : u_h( [0.75, 0.875] )= 0.328125
((2, 3), (1, 3)) : u_h( [0.25, 0.375] )= 0.703125
((4, 1), (1, 1)) : u_h( [0.0625, 0.5] )= 0.234375
((2, 3), (3, 5)) : u_h( [0.75, 0.625] )= 0.703125
((2, 2), (1, 1)) : u_h( [0.25, 0.25] )= 0.5625
((3, 2), (3, 1)) : u_h( [0.375, 0.25] )= 0.703125
((1, 4), (1, 5)) : u_h( [0.5, 0.3125] )= 0.859375
((3, 2), (7, 3)) : u_h( [0.875, 0.75] )= 0.328125
((1, 4), (1, 3)) : u_h( [0.5, 0.1875] )= 0.609375
((1, 3), (1, 1)) : u_h( [0.5, 0.125] )= 0.4375
((3, 1), (5, 1)) : u_h( [0.625, 0.5] )= 0.9375
((3, 2), (3, 3)) : u_h( [0.375, 0.75] )= 0.703125
((4, 1), (11, 1)) : u_h( [0.6875, 0.5] )= 0.859375
((1, 3), (1, 3)) : u_h( [0.5, 0.375] )= 0.9375
((3, 2), (7, 1)) : u_h( [0.875, 0.25] )= 0.328125
((2, 2), (3, 3)) : u_h( [0.75, 0.75] )= 0.5625
((1, 2), (1, 1)) : u_h( [0.5, 0.25] )= 0.75
((4, 1), (3, 1)) : u_h( [0.1875, 0.5] )= 0.609375
((2, 2), (3, 1)) : u_h( [0.75, 0.25] )= 0.5625
((2, 3), (1, 1)) : u_h( [0.25, 0.125] )= 0.328125
((1, 4), (1, 9)) : u_h( [0.5, 0.5625] )= 0.984375
((1, 3), (1, 7)) : u_h( [0.5, 0.875] )= 0.4375
((3, 2), (5, 3)) : u_h( [0.625, 0.75] )= 0.703125
((4, 1), (13, 1)) : u_h( [0.8125, 0.5] )= 0.609375
((1, 3), (1, 5)) : u_h( [0.5, 0.625] )= 0.9375
((2, 3), (1, 5)) : u_h( [0.25, 0.625] )= 0.703125
((1, 4), (1, 1)) : u_h( [0.5, 0.0625] )= 0.234375
```

In [6]:
```
period = time.time()
hierarchize(sg, dim)
period = time.time() - period
print "Hierarchization of sparse grid in", period, "sec"
printSparseGrid(sg)
```

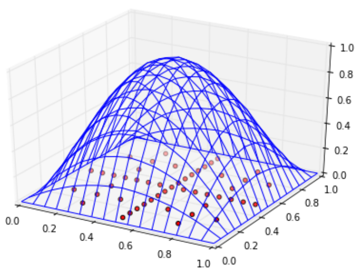
```
Hierarchization of sparse grid in 0.000777006149292 sec
((2, 1), (3, 1)) : u_h( [0.75, 0.5] )= 0.25
((3, 1), (7, 1)) : u_h( [0.875, 0.5] )= 0.0625
((2, 3), (1, 7)) : u_h( [0.25, 0.875] )= 0.015625
((1, 2), (1, 3)) : u_h( [0.5, 0.75] )= 0.25
((3, 1), (3, 1)) : u_h( [0.375, 0.5] )= 0.0625
((2, 3), (3, 1)) : u_h( [0.75, 0.125] )= 0.015625
((4, 1), (15, 1)) : u_h( [0.9375, 0.5] )= 0.015625
((3, 2), (1, 3)) : u_h( [0.125, 0.75] )= 0.015625
((1, 4), (1, 13)) : u_h( [0.5, 0.8125] )= 0.015625
((4, 1), (5, 1)) : u_h( [0.3125, 0.5] )= 0.015625
((1, 4), (1, 11)) : u_h( [0.5, 0.6875] )= 0.015625
((3, 2), (1, 1)) : u_h( [0.125, 0.25] )= 0.015625
((1, 1), (1, 1)) : u_h( [0.5, 0.5] )= 1.0
((1, 4), (1, 7)) : u_h( [0.5, 0.4375] )= 0.015625
((3, 1), (1, 1)) : u_h( [0.125, 0.5] )= 0.0625
((2, 2), (1, 3)) : u_h( [0.25, 0.75] )= 0.0625
((3, 2), (5, 1)) : u_h( [0.625, 0.25] )= 0.015625
((2, 3), (3, 3)) : u_h( [0.75, 0.375] )= 0.015625
((4, 1), (7, 1)) : u_h( [0.4375, 0.5] )= 0.015625
((2, 1), (1, 1)) : u_h( [0.25, 0.5] )= 0.25
((1, 4), (1, 15)) : u_h( [0.5, 0.9375] )= 0.015625
((4, 1), (9, 1)) : u_h( [0.5625, 0.5] )= 0.015625
((2, 3), (3, 7)) : u_h( [0.75, 0.875] )= 0.015625
((2, 3), (1, 3)) : u_h( [0.25, 0.375] )= 0.015625
((4, 1), (1, 1)) : u_h( [0.0625, 0.5] )= 0.015625
((2, 3), (3, 5)) : u_h( [0.75, 0.625] )= 0.015625
((2, 2), (1, 1)) : u_h( [0.25, 0.25] )= 0.0625
((3, 2), (3, 1)) : u_h( [0.375, 0.25] )= 0.015625
((1, 4), (1, 5)) : u_h( [0.5, 0.3125] )= 0.015625
((3, 2), (7, 3)) : u_h( [0.875, 0.75] )= 0.015625
((1, 4), (1, 3)) : u_h( [0.5, 0.1875] )= 0.015625
((1, 3), (1, 1)) : u_h( [0.5, 0.125] )= 0.0625
((3, 1), (5, 1)) : u_h( [0.625, 0.5] )= 0.0625
((3, 2), (3, 3)) : u_h( [0.375, 0.75] )= 0.015625
((4, 1), (11, 1)) : u_h( [0.6875, 0.5] )= 0.015625
((1, 3), (1, 3)) : u_h( [0.5, 0.375] )= 0.0625
((3, 2), (7, 1)) : u_h( [0.875, 0.25] )= 0.015625
((2, 2), (3, 3)) : u_h( [0.75, 0.75] )= 0.0625
((1, 2), (1, 1)) : u_h( [0.5, 0.25] )= 0.25
((4, 1), (3, 1)) : u_h( [0.1875, 0.5] )= 0.015625
((2, 2), (3, 1)) : u_h( [0.75, 0.25] )= 0.0625
((2, 3), (1, 1)) : u_h( [0.25, 0.125] )= 0.015625
((1, 4), (1, 9)) : u_h( [0.5, 0.5625] )= 0.015625
((1, 3), (1, 7)) : u_h( [0.5, 0.875] )= 0.0625
((3, 2), (5, 3)) : u_h( [0.625, 0.75] )= 0.015625
((4, 1), (13, 1)) : u_h( [0.8125, 0.5] )= 0.015625
((1, 3), (1, 5)) : u_h( [0.5, 0.625] )= 0.0625
((2, 3), (1, 5)) : u_h( [0.25, 0.625] )= 0.015625
((1, 4), (1, 1)) : u_h( [0.5, 0.0625] )= 0.015625
```

In [7]:
```
def integrateSparseGrid(sg):
    '''Computes the integral approximated by this sparse grid'''
    result = 0.0
    # simply add up the volumes of all pagodas
    for af in sg.itervalues():
        dim = len(af.getLevelVector())
        result += af.getSurplus() / (1<<(dim + af.getLevel() - 1))
    return result
```

In [8]:
```
period = time.time()
for key in sg:
    x = sg[key].computeCoordinate()
    print key, x, evaluateSparseGrid(sg, x), func(x)
period = time.time() - period
print "Evaluation of sparse grid in", period, "sec"
```

```
((2, 1), (3, 1)) [0.75, 0.5] 0.75 0.75
((3, 1), (7, 1)) [0.875, 0.5] 0.4375 0.4375
((2, 3), (1, 7)) [0.25, 0.875] 0.328125 0.328125
((1, 2), (1, 3)) [0.5, 0.75] 0.75 0.75
((3, 1), (3, 1)) [0.375, 0.5] 0.9375 0.9375
((2, 3), (3, 1)) [0.75, 0.125] 0.328125 0.328125
((4, 1), (15, 1)) [0.9375, 0.5] 0.234375 0.234375
((3, 2), (1, 3)) [0.125, 0.75] 0.328125 0.328125
((1, 4), (1, 13)) [0.5, 0.8125] 0.609375 0.609375
((4, 1), (5, 1)) [0.3125, 0.5] 0.859375 0.859375
((1, 4), (1, 11)) [0.5, 0.6875] 0.859375 0.859375
((3, 2), (1, 1)) [0.125, 0.25] 0.328125 0.328125
((1, 1), (1, 1)) [0.5, 0.5] 1.0 1.0
((1, 4), (1, 7)) [0.5, 0.4375] 0.984375 0.984375
((3, 1), (1, 1)) [0.125, 0.5] 0.4375 0.4375
((2, 2), (1, 3)) [0.25, 0.75] 0.5625 0.5625
((3, 2), (5, 1)) [0.625, 0.25] 0.703125 0.703125
((2, 3), (3, 3)) [0.75, 0.375] 0.703125 0.703125
((4, 1), (7, 1)) [0.4375, 0.5] 0.984375 0.984375
((2, 1), (1, 1)) [0.25, 0.5] 0.75 0.75
((1, 4), (1, 15)) [0.5, 0.9375] 0.234375 0.234375
((4, 1), (9, 1)) [0.5625, 0.5] 0.984375 0.984375
((2, 3), (3, 7)) [0.75, 0.875] 0.328125 0.328125
((2, 3), (1, 3)) [0.25, 0.375] 0.703125 0.703125
((4, 1), (1, 1)) [0.0625, 0.5] 0.234375 0.234375
((2, 3), (3, 5)) [0.75, 0.625] 0.703125 0.703125
((2, 2), (1, 1)) [0.25, 0.25] 0.5625 0.5625
((3, 2), (3, 1)) [0.375, 0.25] 0.703125 0.703125
((1, 4), (1, 5)) [0.5, 0.3125] 0.859375 0.859375
((3, 2), (7, 3)) [0.875, 0.75] 0.328125 0.328125
((1, 4), (1, 3)) [0.5, 0.1875] 0.609375 0.609375
((1, 3), (1, 1)) [0.5, 0.125] 0.4375 0.4375
((3, 1), (5, 1)) [0.625, 0.5] 0.9375 0.9375
((3, 2), (3, 3)) [0.375, 0.75] 0.703125 0.703125
((4, 1), (11, 1)) [0.6875, 0.5] 0.859375 0.859375
((1, 3), (1, 3)) [0.5, 0.375] 0.9375 0.9375
((3, 2), (7, 1)) [0.875, 0.25] 0.328125 0.328125
((2, 2), (3, 3)) [0.75, 0.75] 0.5625 0.5625
((1, 2), (1, 1)) [0.5, 0.25] 0.75 0.75
((4, 1), (3, 1)) [0.1875, 0.5] 0.609375 0.609375
((2, 2), (3, 1)) [0.75, 0.25] 0.5625 0.5625
((2, 3), (1, 1)) [0.25, 0.125] 0.328125 0.328125
((1, 4), (1, 9)) [0.5, 0.5625] 0.984375 0.984375
((1, 3), (1, 7)) [0.5, 0.875] 0.4375 0.4375
((3, 2), (5, 3)) [0.625, 0.75] 0.703125 0.703125
((4, 1), (13, 1)) [0.8125, 0.5] 0.609375 0.609375
((1, 3), (1, 5)) [0.5, 0.625] 0.9375 0.9375
((2, 3), (1, 5)) [0.25, 0.625] 0.703125 0.703125
((1, 4), (1, 1)) [0.5, 0.0625] 0.234375 0.234375
Evaluation of sparse grid in 0.0100841522217 sec
```

In [9]:
```
period = time.time()
plotHierarchical2d(sg, showGrid=True, minLevel=minLevel, azimuth=None, elevation=None)
period = time.time() - period
print "Plotting the sparse grid took", period, "sec"
```




```
Plotting the sparse grid took 0.341922044754 sec
```

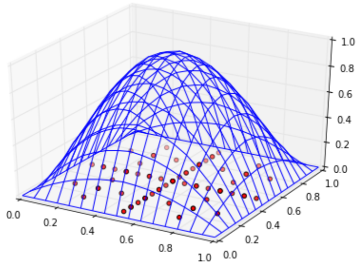
Exercise 2: The Combination Technique – A Different View on Sparse Grids

There is no exercise to be solved here, we only want to learn about the combination technique through playing with it.
Look at the plots, and try to figure out what the call to *map* does further below.
Manipulate and remove the elements in cg manually and try to always establish a consistent state of your combi grid.

In [10]:
```
cg = createCombiGrid(dim, minLevel, parabola)
print "The combi grid contains the following u_l:"
for lvec in cg:
    print "u_", lvec
    #print cg[lvec]
```

```
The combi grid contains the following u_l:
u_ (3, 2)
u_ (1, 3)
u_ (3, 1)
u_ (1, 4)
u_ (2, 3)
u_ (2, 2)
u_ (4, 1)
```

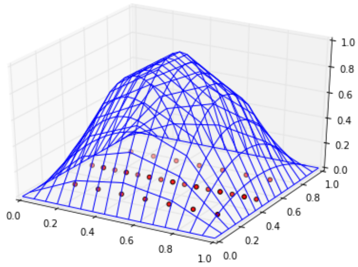
In [11]:
```
period = time.time()
plotCombi2d(cg, showGrid=True, minLevel=minLevel, azimuth=None, elevation=None)
period = time.time() - period
print "Plotting the sparse grid took", period, "sec"
```




```
Plotting the sparse grid took 0.195422172546 sec
```

In [12]:
```
# What does the following line do?
_ = map(cg.pop, filter(lambda key: key[0] <= 2, cg.iterkeys()))
```

In [13]:
```
period = time.time()
plotCombi2d(cg, showGrid=True, minLevel=minLevel, azimuth=None, elevation=None)
period = time.time() - period
print "Plotting the sparse grid took", period, "sec"
```




```
Plotting the sparse grid took 0.18644785881 sec
```

Extra: Quadrature in 2-D

This part presents some results for numerical quadrature in 2-D which are for you to interpret.

In [14]:
```
def trapezoidalRuleMultiD(fullGrid, hvec=None, implicitBoundaries=False):
    '''
    Uses nested CTR to approximate the integral of given function samples.
    '''
    def rec(d, ivec):
        offset = 0 if implicitBoundaries else 1
        if d == 0:
            # will not be called "on" implicit boundaries
            return fullGrid[tuple(ivec)]
        else:
            # still need to recurse into another dimension
            endIdx = fullGrid.shape[d-1]
            result = 0.0
            if not implicitBoundaries:
                # explicit boundaries have weight 0.5
                ivec[d-1] = 0
                result += rec(d-1, ivec)
                ivec[d-1] = endIdx-1
                result += rec(d-1, ivec)
                result *= 0.5
            for i in xrange(offset, endIdx-offset):
                # function values of inner points have weight 1
                ivec[d-1] = i
                result += rec(d-1, ivec)
            # scale result with h and return
            return hvec[d-1] * result
    dim = len(fullGrid.shape)
    if hvec is None:
        k = 1 if implicitBoundaries else -1
        hvec = [1.0 / (l+k) for l in fullGrid.shape]
    return rec(dim, [0,]*dim)
```

In [15]:
```
def simpsonRuleMultiD(fullGrid, hvec=None, implicitBoundaries=False):
    '''
    Uses CSR to approximate the integral of given function samples.
    '''
    dim = len(fullGrid.shape)
    def rec(d, ivec):
        offset = 0 if implicitBoundaries else 1
        if d == 0:
            # will not be called "on" implicit boundaries
            return fullGrid[tuple(ivec)]
        else:
            # still need to recurse into another dimension
            endIdx = fullGrid.shape[d-1]
            result = 0.0
            if not implicitBoundaries:
                # explicit boundaries have weight 0.5
                ivec[d-1] = 0
                result += rec(d-1, ivec)
                ivec[d-1] = endIdx-1
                result += rec(d-1, ivec)
                result *= 0.5
            for i in xrange(offset, endIdx-offset, 2):
                # scale with weight 2
                ivec[d-1] = i
                result += 2.0 * rec(d-1, ivec)
            for i in xrange(offset + 1, endIdx-offset-1, 2):
                # scale with weight 1
                ivec[d-1] = i
                result += rec(d-1, ivec)
            # scale result with h and return
            return 2.0 * hvec[d-1] * result / 3.0
    if hvec is None:
        k = 1 if implicitBoundaries else -1
        hvec = [1.0 / (l+k) for l in fullGrid.shape]
    return rec(dim, [0,]*dim)
```

In [16]:
```
dim = 2
level = 4

print "Composite Trapezoidal Rule:", trapezoidalRuleMultiD(createFullGrid([3,3], func=func, useBoundaries=True), hvec=None, implicitBoundaries=False)
print "Simpson Rule:", simpsonRuleMultiD(createFullGrid([1,1], func=func, useBoundaries=True), hvec=None, implicitBoundaries=False)

# sparse grid integration
sg = {}
applyRefinementCriterion(sg, dim, MinLevelCriterion(dim, level), func)
hierarchize(sg, dim)
print "Sparse grid integration:", integrateSparseGrid(sg)

# combi grid integration
cg = createCombiGrid(dim, level, func)
cg_CTR = 0.0
cg_CSR = 0.0
for cgkey in cg:
    if sum(cgkey)-dim+1 == level:
        cg_CSR += simpsonRuleMultiD(cg[cgkey],     hvec=None, implicitBoundaries=True)
        cg_CTR += trapezoidalRuleMultiD(cg[cgkey], hvec=None, implicitBoundaries=True)
    elif sum(cgkey)-dim+1 == level-1:
        cg_CSR -= simpsonRuleMultiD(cg[cgkey],     hvec=None, implicitBoundaries=True)
        cg_CTR -= trapezoidalRuleMultiD(cg[cgkey], hvec=None, implicitBoundaries=True)
print "Combi grid with CTR:", cg_CTR
print "Combi grid with CSR:", cg_CSR
```

```
Composite Trapezoidal Rule: 0.4306640625
Simpson Rule: 0.444444444444
Sparse grid integration: 0.4375
Combi grid with CTR: 0.4375
Combi grid with CSR: 0.444444444444
```

In [16]: