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]:

```
(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]:

```
(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 hierarchize(workDim, lvec, ivec, f_l, f_r):
        '''Recursive 1-D hierarchization to (lvec, ivec) in dimension workDim.''
        key = (tuple(lvec), tuple(ivec))
        return False
    af = sg[key]
    f_m = af.getSurplus()
    lvec[workDim] -= 1
    ivec[workDim] = 2*ivec[workDim] - 1
    hierarchize(workDim, lvec, ivec, f_l, f_m)
    lvec[workDim] += 1
    ivec[workDim] = ivec[workDim] >> 0.5
    for d in xrange(dim):
        # apply the 1-D hierarchization scheme for all dimensions, one after another
    return True

def traverseRecursively(workDim, minD, lvec, ivec, verbose = False):
    '''Local recursive helper function for traversal of the "main axis" w.r.t. workDim.''
    if not sg.has_key(key):
        return False
    if verbose:
        print 'dimension %d: hierarchize' % workDim, key
    # being here => point exists
    if verbose:
        for d in xrange(minD, dim):
            # skip the work dimension if necessary!
        # descend into left child
    lvec[d] += 1
    ivec[d] = 2*ivec[d] - 1
    traverseRecursively(workDim, d, lvec, ivec, verbose)
    # descend into right child
    lvec[d] += 1
    ivec[d] = ivec[d] >> 0.5
    # apply the 1-D hierarchization scheme for all dimensions, one after another
    lvec[d] = 1
    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)
In [6]:
period = time.time()
hierarchize(sg, dim)
period = time.time()
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
((1, 2), (1, 3)) : u_h( [0.5, 0.75] )= 0.015625
((2, 3), (3, 1)) : u_h( [0.375, 0.5] )= 0.015625
((1, 5), (15, 1)) : u_h( [0.0375, 0.5] )= 0.015625
((1, 2), (1, 3)) : u_h( [0.125, 0.5] )= 1.0
((2, 3), (3, 1)) : u_h( [0.25, 0.75] )= 0.0625
((2, 1), (1, 1)) : u_h( [0.25, 0.5] )= 0.25
((1, 2), (1, 1)) : u_h( [0.5, 0.25] )= 0.25
((1, 2), (1, 1)) : u_h( [0.5, 0.0625] )= 0.015625
((2, 2), (3, 3)) : u_h( [0.75, 0.375] )= 0.015625
((3, 1), (5, 1)) : u_h( [0.625, 0.5] )= 0.0625
((3, 1), (3, 1)) : u_h( [0.375, 0.5] )= 0.0625
((2, 3), (1, 3)) : u_h( [0.25, 0.625] )= 0.015625
((1, 3), (1, 3)) : u_h( [0.5, 0.375] )= 0.0625
((2, 3), (3, 3)) : u_h( [0.75, 0.875] )= 0.015625
((2, 3), (1, 3)) : u_h( [0.25, 0.375] )= 0.015625
((1, 4), (1, 11)) : u_h( [0.5, 0.6875] )= 0.015625
((3, 2), (1, 3)) : u_h( [0.125, 0.75] )= 0.015625
((3, 2), (1, 3)) : u_h( [0.125, 0.25] )= 0.015625
((4, 1), (13, 1)) : u_h( [0.8125, 0.5] )= 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
((1, 3), (1, 7)) : u_h( [0.5, 0.125] )= 0.015625
((2, 3), (1, 1)) : u_h( [0.25, 0.125] )= 0.015625
((1, 3), (1, 1)) : u_h( [0.5, 0.0625] )= 0.015625
((2, 2), (3, 1)) : u_h( [0.75, 0.25] )= 0.0625
((2, 1), (1, 1)) : u_h( [0.25, 0.5] )= 0.25
((2, 2), (3, 1)) : u_h( [0.75, 0.25] )= 0.0625
((2, 2), (3, 1)) : u_h( [0.75, 0.75] )= 0.0625

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
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.

```python
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")
```

Evaluation of sparse grid in 0.0100841522217 sec

```python
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

```python
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)
```
Extra: Quadrature in 2-D

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

```python
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 range(offset, endIdx - offset):
                # function values of inner points have weight 1
                ivec[d - 1] = i
                result += rec(d - 1, ivec)
            return hvec[d - 1] * result
    dim = len(fullGrid.shape)
    if hvec is None:
        k = 1 if implicitBoundaries else -1
        hvec = [1.0 / (1+k) for i in fullGrid.shape]
    return rec(dim, [0]*dim)
```
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
            result = 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 range(offset, endIdx - offset, 2):
                # scale with weight 2
                ivec[d-1] = i
                result += 2.0 * rec(d-1, ivec)
            for i in range(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 / (i*k) for i in fullGrid.shape]
        return rec(dim, [0,]*dim)

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)
    else:
        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