In [20]:
```
%load_ext sympyprinting
import time
from math import sqrt
# import the wavelet function classes and additional helpers for plotting
from Wavelet import *
```

Exercise 1: Cranking The Machine

Typically the scaling function ϕ is not known explicitly, and sometimes a closed-form analytic formula does not even exist. However, for continuous ϕ we can approximate the function to arbitrarily high precision using the **Cascade Algorithm**, a fixed-point method for functions.

In this exercise we want to implement this algorithm by iterating over the expression

$$F(\gamma)(t) = \sum_k c_k \cdot \gamma(2t - k)$$

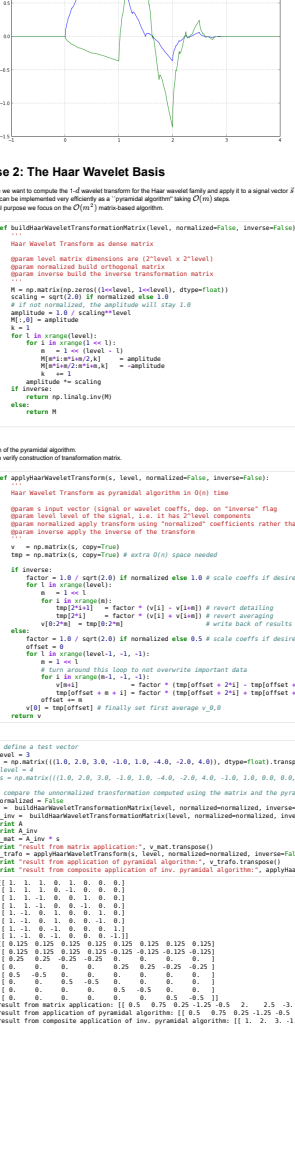
in order to find the fixed point γ of F.

Our starting point γ_0 will be the mother function of all hat functions over the interval $[-1; 1]$.

In [21]:
```
class FixedPointScalingApproximation:
    '''Encapsulates information about one approximation in fixed point iteration'''
    def __init__(self, previousApproximation, ScalingFunction):
        self.f = previousApproximation
        self.phi = ScalingFunction

    def __call__(self, t):
        resultScaling  = 0.0
        for k in xrange(ScalingFunction.nk):
            resultScaling  += ScalingFunction.c[k] * self.f(2.0*t - k)
        return resultScaling

class FixedPointWaveletApproximation:
    '''Encapsulates information about one approximation in fixed point iteration'''
    def __init__(self, previousApproximation, ScalingFunction):
        self.f = previousApproximation
        self.phi = ScalingFunction

    def __call__(self, t):
        resultWavelet  = 0.0
        nk = ScalingFunction.nk
        for k in xrange(nk):
            resultWavelet  += (-1.0)**k * ScalingFunction.c[nk-1-k] * self.f(2.0*t - k)
        return resultWavelet
```

In [22]:
```
ScalingFunction = HaarScalingFunction
#ScalingFunction = HatScalingFunction
#ScalingFunction = BattleLemarieScalingFunction
#ScalingFunction = CubicBattleLemarieScalingFunction
ScalingFunction = Daubechies4ScalingFunction
#ScalingFunction = Daubechies6ScalingFunction
#ScalingFunction = Daubechies8ScalingFunction

a, b =  -1.0, 4.0
plotLevel  = 8
iterations = 4

fig = plt.figure(1, figsize = (11, 9))
plt.suptitle('Cranking my own machine')
_ = plt.grid(True)
_ = plt.ylim(-1.5, 1.8)

# starting point in the fixed point iteration
hat = lambda t: max(0.0, 1-abs(t))

# set of sampling points used for plotting
x = np.linspace(a, b, int(b-a) << plotLevel)

# plot the hat function
ax = []
ax.append(plot(x, map(hat, x), label="hat"))

fixedPointScaling = [hat,] + [None,]* iterations
fixedPointWavelet = [hat,] + [None,]* iterations
for i in xrange(1, iterations + 1):
    fixedPointScaling[i] = FixedPointScalingApproximation(fixedPointScaling[i-1], ScalingFunction)
    fixedPointWavelet[i] = FixedPointWaveletApproximation(fixedPointScaling[i-1], ScalingFunction)
    if i % 2 == 1 or i == iterations:
        period = time.time()
        ax.append(plot(x, map(fixedPointScaling[i], x), label="scaling [%d]" % i))
        ax.append(plot(x, map(fixedPointWavelet[i], x), label="wavelet [%d]" % i))
        period = time.time() - period
        print "Done plotting iteration", i, "in", period, "sec"

_ = plt.legend()
_ = plt.show()
```

```
Done plotting iteration 1 in 0.0482399463654 sec
Done plotting iteration 3 in 0.805557157516 sec
Done plotting iteration 4 in 3.20506600519 sec
```



In [23]:
```
ScalingFunction = HaarScalingFunction
#ScalingFunction = HatScalingFunction
#ScalingFunction = BattleLemarieScalingFunction
#ScalingFunction = CubicBattleLemarieScalingFunction
ScalingFunction = Daubechies4ScalingFunction
#ScalingFunction = Daubechies6ScalingFunction
#ScalingFunction = Daubechies8ScalingFunction

a, b =  -1.0, 4.0
plotLevel  = 8
iterations = 10

period = time.time()
values = crankTheMachine(ScalingFunction, iterations, plotLevel, a=a, b=b)
period = time.time() - period
print "Done plotting iteration", iterations, "in", period, "sec"

fig = plt.figure(1, figsize = (11, 9))
plt.suptitle('Cranking the machine')
ax = []
ax.append(plot(values[0], values[1], label="scaling"))
ax.append(plot(values[0], values[2], label="wavelet"))
_ = plt.grid(True)
_ = plt.ylim(-1.5, 1.8)
_ = plt.legend()
```

```
Done plotting iteration 10 in 0.348423300415 sec
```


Exercise 2: The Haar Wavelet Basis

In this exercise we want to compute the 1-d wavelet transform for the Haar wavelet family and apply it to a signal vector $\vec{s}$ of length $m = 2^n$.
The transform can be implemented very efficiently as a ''pyramidal algorithm'' taking $\mathcal{O}(m)$ steps.
For educational purpose we focus on the $\mathcal{O}(m^2)$ matrix-based algorithm.

In [24]:
```
def buildHaarWaveletTransformationMatrix(level, normalized=False, inverse=False):
    '''
    Haar Wavelet Transform as dense matrix

    @param level matrix dimensions are (2^level x 2^level)
    @param normalized build orthogonal matrix
    @param inverse build the inverse transformation matrix
    '''
    M = np.matrix(np.zeros((1<<level, 1<<level), dtype=float))
    scaling = sqrt(2.0) if normalized else 1.0
    # if not normalized, the amplitude will stay 1.0
    amplitude = 1.0 / scaling**level
    M[:,0] = amplitude
    k = 1
    for l in xrange(level):
        for i in xrange(1 << l):
            m   = 1 << (level - l)
            M[m*i:m*i+m/2,k]    = amplitude
            M[m*i+m/2:m*i+m,k]  = -amplitude
            k  += 1
        amplitude *= scaling
    if inverse:
        return np.linalg.inv(M)
    else:
        return M
```

Implementation of the pyramidal algorithm.
Can be used to verify construction of transformation matrix.

In [25]:
```
def applyHaarWaveletTransform(s, level, normalized=False, inverse=False):
    '''
    Haar Wavelet Transform as pyramidal algorithm in O(n) time

    @param s input vector (signal or wavelet coeffs, dep. on "inverse" flag
    @param level level of the signal, i.e. it has 2^level components
    @param normalized apply transform using "normalized" coefficients rather than "nice" ones
    @param inverse apply the inverse of the transform
    '''

    v   = np.matrix(s, copy=True)
    tmp = np.matrix(s, copy=True) # extra O(n) space needed

    if inverse:
        factor = 1.0 / sqrt(2.0) if normalized else 1.0 # scale coeffs if desired
        for l in xrange(level):
            m   = 1 << l
            for i in xrange(m):
                tmp[2*i+1]   = factor * (v[i] - v[i+m]) # revert detailing
                tmp[2*i]     = factor * (v[i] + v[i+m]) # revert averaging
            v[0:2*m] = tmp[0:2*m]                      # write back of results
    else:
        factor = 1.0 / sqrt(2.0) if normalized else 0.5 # scale coeffs if desired
        offset = 0
        for l in xrange(level-1, -1, -1):
            m = 1 << l
            # turn around this loop to not overwrite important data
            for i in xrange(m-1, -1, -1):
                v[m+i]      = factor * (tmp[offset + 2*i] - tmp[offset + 2*i+1]) # detailing
                tmp[offset + m + i] = factor * (tmp[offset + 2*i] + tmp[offset + 2*i+1]) #averaging
            offset += m
        v[0] = tmp[offset] # finally set first average v_0_0
    return v
```

In [28]:
```
# define a test vector
level = 3
s = np.matrix(((1.0, 2.0, 3.0, -1.0, 1.0, -4.0, -2.0, 4.0)), dtype=float).transpose()
#level = 4
#s = np.matrix(((1.0, 2.0, 3.0, -1.0, 1.0, -4.0, -2.0, 4.0, -1.0, 1.0, 0.0, 0.0, 5.0, -5.0, 0.0, 0.0)), dtype=float).transpose()

# compare the unnormalized transformation computed using the matrix and the pyramidal algorithm
normalized = False
A = buildHaarWaveletTransformationMatrix(level, normalized=normalized, inverse=False)
A_inv =  buildHaarWaveletTransformationMatrix(level, normalized=normalized, inverse=True)
print A
print A_inv
v_mat = A_inv * s
print "result from matrix application:", v_mat.transpose()
v_trafo = applyHaarWaveletTransform(s, level, normalized=normalized, inverse=False)
print "result from application of pyramidal algorithm:", v_trafo.transpose()
print "result from composite application of inv. pyramidal algorithm:", applyHaarWaveletTransform(v_trafo, level, normalized=normalized, inverse=True).transpose()
```

```
[[ 1.  1.  1.  0.  1.  0.  0.  0.]
 [ 1.  1.  1.  0. -1.  0.  0.  0.]
 [ 1.  1. -1.  0.  0.  1.  0.  0.]
 [ 1.  1. -1.  0.  0. -1.  0.  0.]
 [ 1. -1.  0.  1.  0.  0.  1.  0.]
 [ 1. -1.  0.  1.  0.  0. -1.  0.]
 [ 1. -1.  0. -1.  0.  0.  0.  1.]
 [ 1. -1.  0. -1.  0.  0.  0. -1.]]
[[ 0.125  0.125  0.125  0.125  0.125  0.125  0.125  0.125]
 [ 0.125  0.125  0.125  0.125 -0.125 -0.125 -0.125 -0.125]
 [ 0.25   0.25  -0.25  -0.25   0.     0.     0.     0.   ]
 [ 0.     0.     0.     0.     0.25   0.25  -0.25  -0.25 ]
 [ 0.5   -0.5    0.     0.     0.     0.     0.     0.   ]
 [ 0.     0.     0.5   -0.5    0.     0.     0.     0.   ]
 [ 0.     0.     0.     0.     0.5   -0.5    0.     0.   ]
 [ 0.     0.     0.     0.     0.     0.     0.5   -0.5  ]]
result from matrix application: [[ 0.5   0.75  0.25 -1.25 -0.5   2.    2.5  -3.  ]]
result from application of pyramidal algorithm: [[ 0.5   0.75  0.25 -1.25 -0.5   2.    2.5  -3.  ]]
result from composite application of inv. pyramidal algorithm: [[ 1.  2.  3. -1.  1. -4. -2.  4.]]
```

In [29]:

```python
# define a test vector
level = 3
s = np.matrix(((1.0, 2.0, 3.0, -1.0, 1.0, -4.0, -2.0, 4.0)), dtype=float).transpose()
#level = 4
#s = np.matrix(((1.0, 2.0, 3.0, -1.0, 1.0, -4.0, -2.0, 4.0, -1.0, 1.0, 0.0, 0.0, 5.0, -5.0, 0.0, 0.0)), dtype=float).transpose()

# compare the normalized transformation computed using the matrix and the pyramidal algorithm
normalized = True
A =  buildHaarWaveletTransformationMatrix(level, normalized=normalized, inverse=False)
A_inv =  buildHaarWaveletTransformationMatrix(level, normalized=normalized, inverse=True)
print A
print A_inv
v_mat = A_inv * s
print "result from (normalized) matrix application:", v_mat.transpose()
v_trafo = applyHaarWaveletTransform(s, level, normalized=normalized, inverse=False)
print "result from application of pyramidal algorithm:", v_trafo.transpose()
print "result from composite application of inv. pyramidal algorithm:", applyHaarWaveletTransform(v_trafo, level, normalized=normalized, inverse=True).transpose()
```

```
[[ 0.35355339  0.35355339  0.5         0.          0.70710678  0.          0.
   0.        ]
 [ 0.35355339  0.35355339  0.5         0.         -0.70710678  0.          0.
   0.        ]
 [ 0.35355339  0.35355339 -0.5         0.          0.          0.70710678
   0.          0.        ]
 [ 0.35355339  0.35355339 -0.5         0.          0.         -0.70710678
   0.          0.        ]
 [ 0.35355339 -0.35355339  0.          0.5         0.          0.
   0.70710678  0.        ]
 [ 0.35355339 -0.35355339  0.          0.5         0.          0.
  -0.70710678  0.        ]
 [ 0.35355339 -0.35355339  0.         -0.5         0.          0.          0.
   0.70710678]
 [ 0.35355339 -0.35355339  0.         -0.5         0.          0.          0.
  -0.70710678]]
[[ 0.35355339  0.35355339  0.35355339  0.35355339  0.35355339  0.35355339
   0.35355339  0.35355339]
 [ 0.35355339  0.35355339  0.35355339  0.35355339 -0.35355339 -0.35355339
  -0.35355339 -0.35355339]
 [ 0.5         0.5        -0.5        -0.5         0.          0.          0.
   0.        ]
 [ 0.          0.          0.          0.          0.5         0.5        -0.5
  -0.5       ]
 [ 0.70710678 -0.70710678  0.          0.          0.          0.          0.
   0.        ]
 [ 0.          0.          0.70710678 -0.70710678  0.          0.          0.
   0.        ]
 [ 0.          0.          0.          0.          0.70710678 -0.70710678
   0.          0.        ]
 [ 0.          0.          0.          0.          0.          0.
   0.70710678 -0.70710678]]
result from (normalized) matrix application: [[ 1.41421356  2.12132034  0.5        -2.5        -0.70710678  2.82842712
   3.53553391 -4.24264069]]
result from application of pyramidal algorithm: [[ 1.41421356  2.12132034  0.5        -2.5        -0.70710678  2.82842712
   3.53553391 -4.24264069]]
result from composite application of inv. pyramidal algorithm: [[ 1.  2.  3. -1.  1. -4. -2.  4.]]
```

In [27]: