EXAHD Teachlet
An Introduction to the Combination Technique

July 1, 2016

1 The combination technique step by step

In this demo, we will show how the combination technique works step by step. Our goal is to interpolate a simple function (2D parabola) using the combination technique.

Our 2D parabola is given by

$$f(x, y) = 16x(x - 1)y(y - 1)$$

in the unit square $(x, y) = [0, 1]^2$

First, we import several functions.

```
In [1]: %pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
In [2]: from sys import path
        path.append('./src/')

        # General useful Python modules
        import time
        import pgf_plots as pgf
        import matplotlib.animation as animation
        from IPython.display import clear_output

        # Modules specific to the combination technique, from our Git repository
        from combinationScheme import combinationSchemeArbitrary, combinationSchemeFaultTolerant
        from combiGrid import combiGridDummy2D, combiGrid
        from combineGrids import combineGrids
        from HelperFunctions import *
        import ActiveSetFactory

        # These are some useful functions to plot with latex (you can ignore the warning)
        pgf_plot = pgf.pgf_plotter(plt)
        pgf_plot.update_rc(plt)
        plt.rc('text', usetex=True)
```

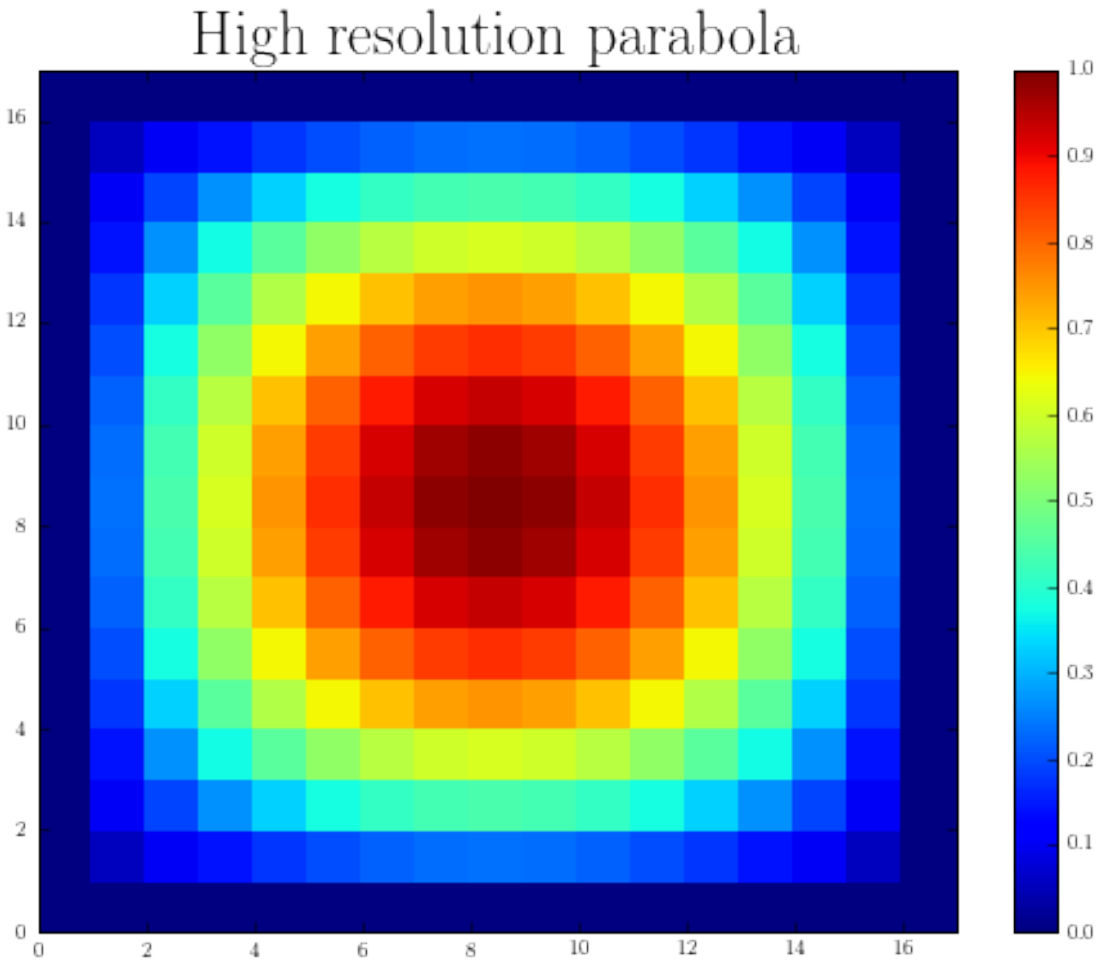
```
/home/sccs/anaconda/lib/python2.7/site-packages/matplotlib/__init__.py:855: UserWarning: text.fontsize is
  warnings.warn(self.msg_depr % (key, alt_key))
```

First we generate a parabola on a "high" resolution full grid with 17 x 17 points. You can choose a higher resolution by varying lmax

```
In [3]: # The solution will have (2^lmax + 1) discretization points
        lmax = (4,4)
```

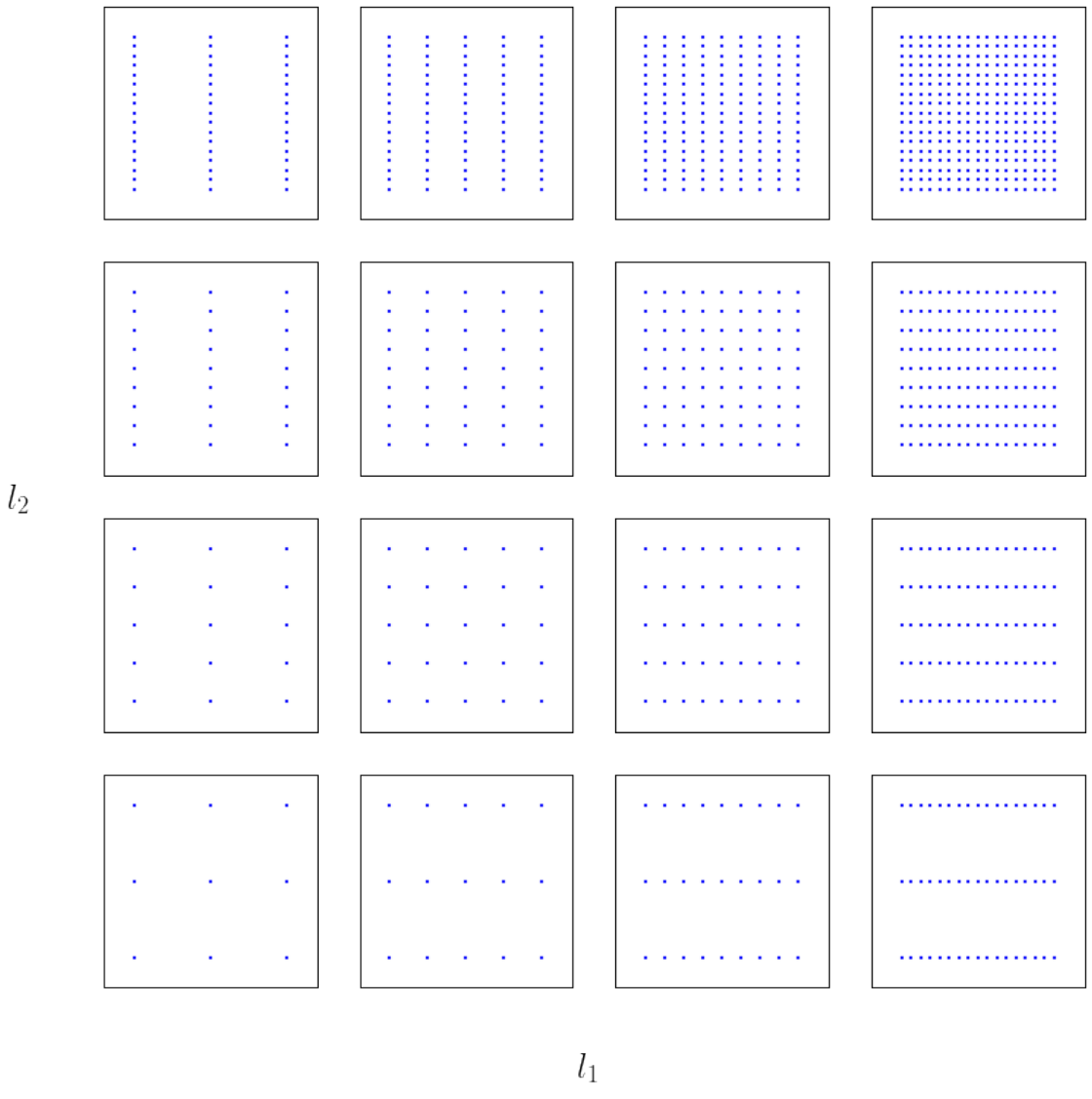
```
# We create a 2D grid.
# The "True" values indicate that the grid has boundary points in both dimensions
grid = combiGridDummy2D(lmax,(True,True))

# We fill the grid with our 2D parabola function
grid.fillData(parabola_2d)
data = grid.getData()

# Plot the solution
fig = figure(figsize=(8,6))
ax = gca()
title('High resolution parabola',fontsize=24)
im = ax.pcolormesh(data)
ax.axes.set_xlim(0, data.shape[1])
ax.axes.set_ylim(0, data.shape[0])
fig.colorbar(im)
```

Out[3]: <matplotlib.colorbar.Colorbar instance at 0x7f7131214878>



Now take a look at the following "grid of grids", which shows 16 different full grids with varying discretization resolutions in x and y directions.

In [4]:
```
# See the Appendix for a brief explanation of the data structures used
lmin = (1,1)
lmax = (4,4)
factory = ActiveSetFactory.ClassicDiagonalActiveSet(lmax, lmin, 0)
activeSet = factory.getActiveSet()
scheme = combinationSchemeArbitrary(activeSet)
combiG = combineGrids(scheme)
for s in scheme.hierSubs((1,1),lmax):
    grid = combiGridDummy2D(s,(True,True))
    grid.fillData(fun)
    combiG.addGrid(grid)
sz=(12,12)
plot_combination_technique(lmin, lmax, {}, combiG, figure(figsize=sz))
```

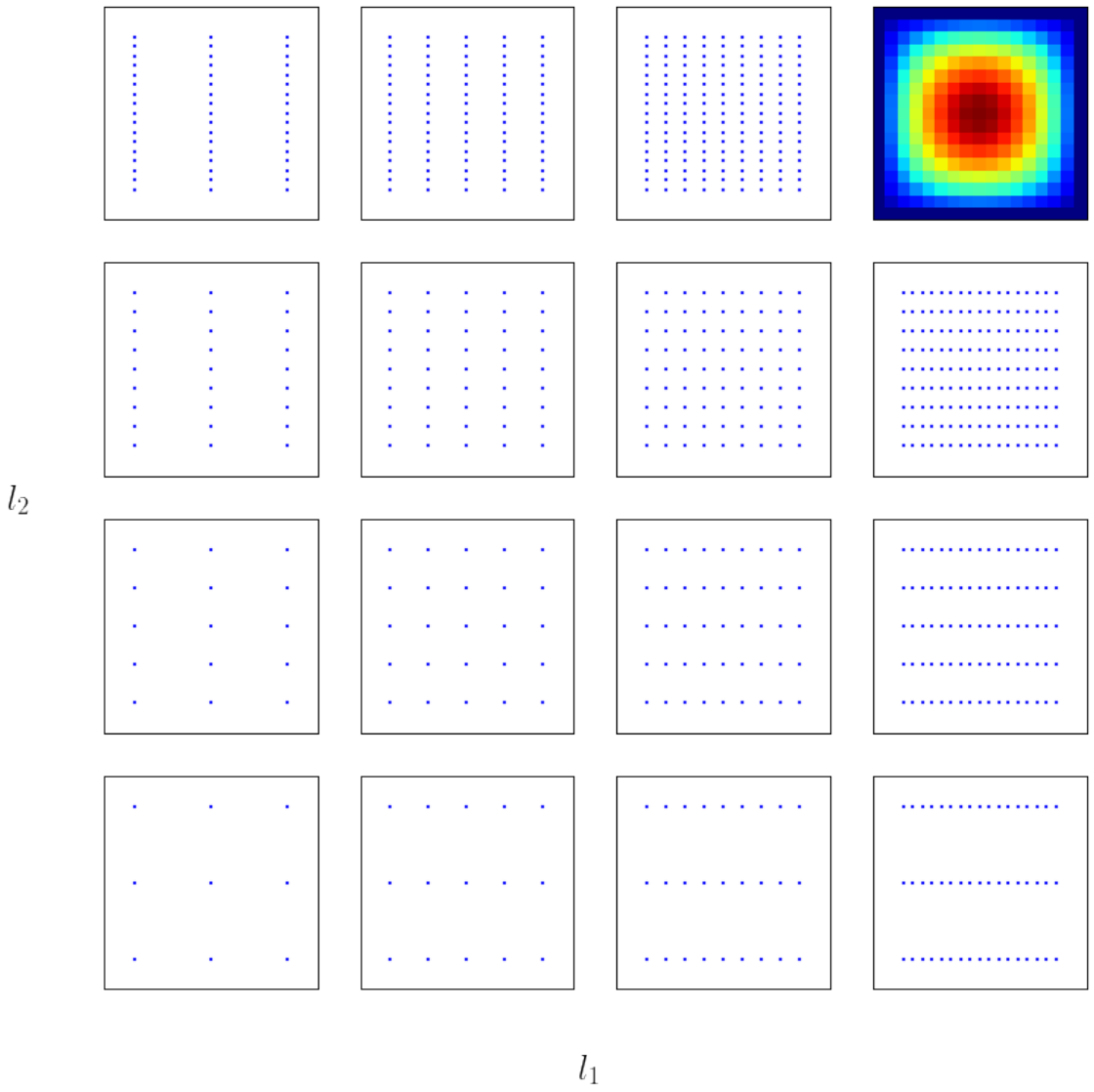


We can refer to the different grids by their position on the chart. For example, the coarsest grid (bottom left, with 3x3 grid points) is located in coordinate (1,1). This coordinate is the grid's *level* l. A grid of level

l has $2^l + 1$ grid points. So the grid on the bottom right has level (4,1) (17x3 grid points).

If we could choose among any of these 16 grids to perform our interpolation, and wanted to have the highest resolution possible, we'd choose the right top grid:

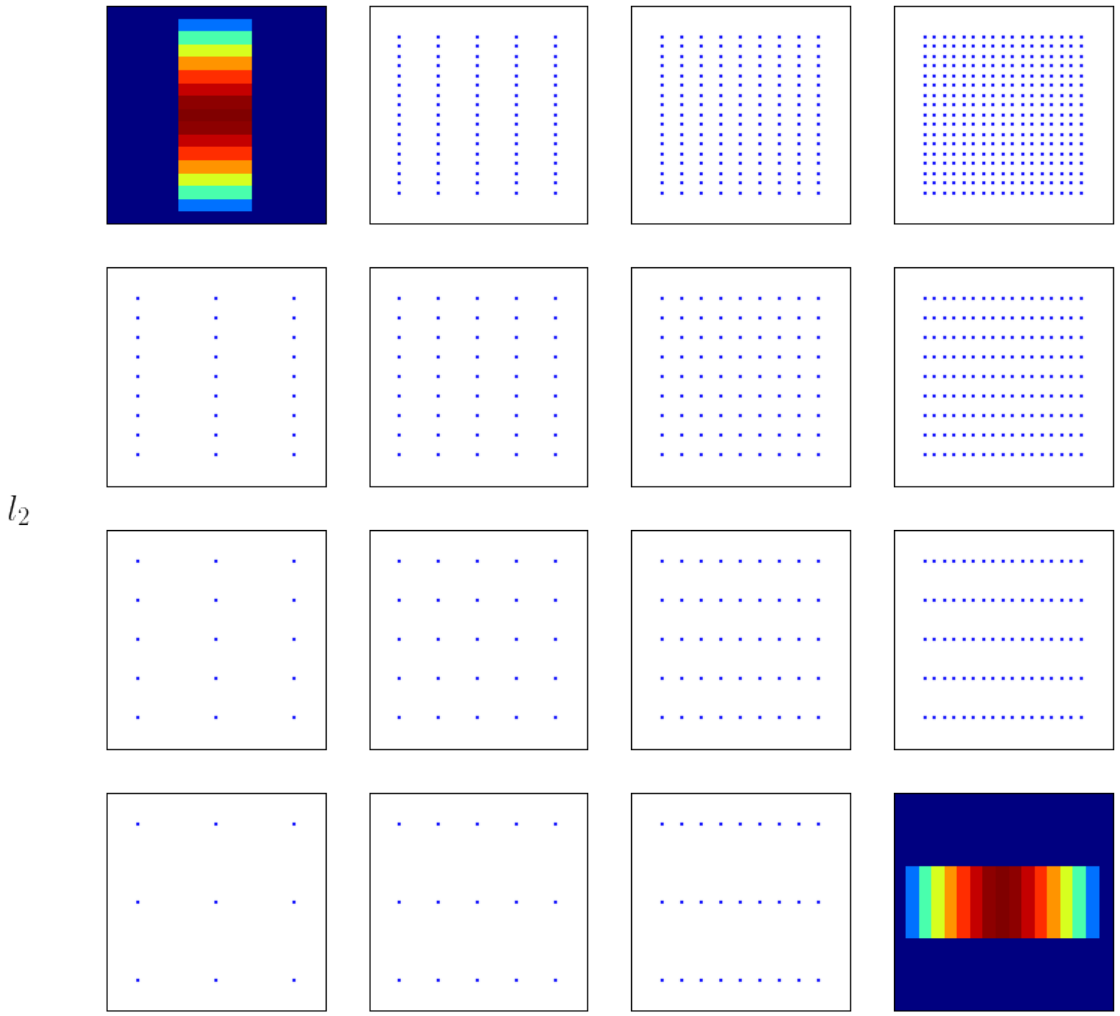
```
In [5]: combiG.gridsDict[(lmax),].fillData(parabola_2d)
        plot_combination_technique( lmin, lmax, {lmax:1}, combiG, \
                                    figure( figsize=sz ), plusMinus=False )
```



Suppose that you cannot afford interpolating your function with such a high resolution (17x17). The combination technique gives you an alternative. Take a look at the two solutions plotted below. The grid with level (4,1) has a high resolution in the x dimension and a low resolution in the y dimension (17x3 points); the grid with level (1,4) has a high resolution in the y dimension but a low resolution in the x dimension (3x17 points). We could ask ourselves: is there a way to combine these solutions to approximate the full grid solution (17x17 points)?

```
In [6]: keyX = (lmax[0],lmin[0])
        keyY = (lmin[0],lmax[0])
```

```
combiG.gridsDict[(keyX),].fillData(parabola_2d)
combiG.gridsDict[(keyY),].fillData(parabola_2d)
combiG.gridsDict[(lmax),].fillData(fun)
fig = figure(figsize=sz)
plot_combination_technique( lmin, lmax, {keyX:1, keyY:1}, combiG, fig, plusMinus=0 )
```



It turns out that we can "add" these two coarse parabolas to approximate the high resolution parabola (17x17).

Three questions arise:

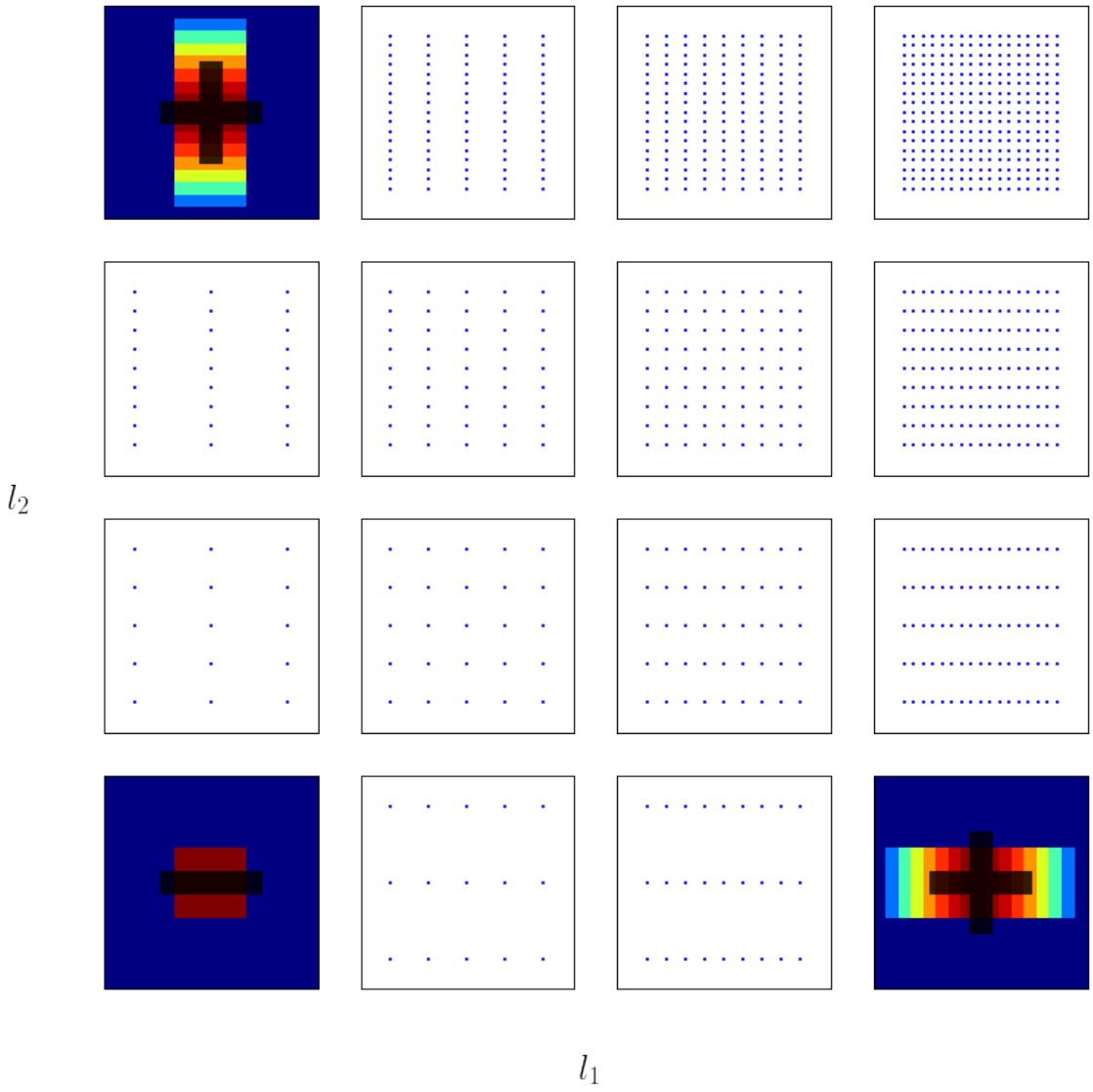
1. How does one "add" grids of different resolutions?
2. How good is this approximation compared to the full grid?
3. Why would one want to do this?

Let's start with the first question. Adding grids of different resolutions requires us to do some interpolation (or hierarchization) but we won't go into the details. For now you can imagine superposing the values of

both grids. The only problem is that the two grids have some grid points in common (9 of them, actually. Can you spot which ones?). At these grid points we will *add both solutions together* and subtract a *third solution*, namely, the one on grid (1,1). This additional grid contains exactly the 9 points that the grids (1,4) and (4,1) have in common.

```
In [7]: key_common = tuple(map(min,zip(*[keyX,keyY])))
        combiG.gridsDict[(key_common),].fillData(parabola_2d)
        dict_small = {keyX:1, keyY:1, key_common:-1}
```

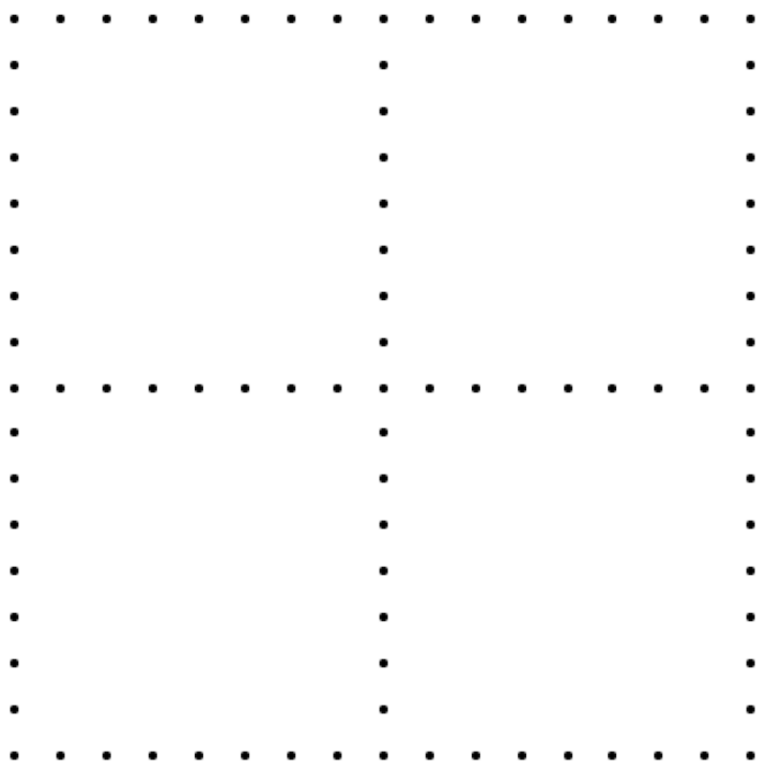
```
In [8]: fig = figure( figsize=sz )
        plot_combination_technique( lmin, lmax, dict_small, combiG, fig )
```



To summarize, we have to perform the following steps: - Solve your problem on grids (4,1), (1,4), and (1,1) - Superpose / add solutions from grids (4,1) and (1,4) - Subtract the solution from grid (1,1)

This combination of grids results in a *sparse grid*. In this case, the sparse grid looks as follows:

```
In [9]: activeSetSmall = set([keyX,keyY])
        schemeSmall = combinationSchemeArbitrary(activeSetSmall)
        combiSG = combineGrids(schemeSmall)

        for s in [keyX, keyY, key_common]:
            grid = combiGridDummy2D(s,(True,True))
            grid.fillData(fun)
            combiSG.addGrid(grid)

        sg = combiSG.plotSparseGrid(size=(8,8),ptsize=9,color='black')
```



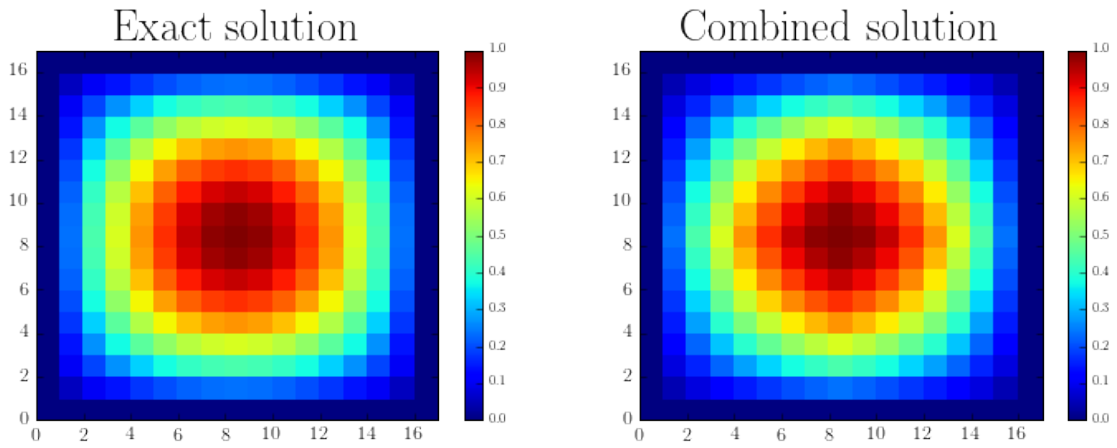
Look at the combined solution below, compared to the full grid solution. You can already start to see the answer to our second question: how good is the approximation? In this case, pretty good! (See the error calculation below, and observe the plots)

```
In [10]: combiSimple = combineGrids(schemeSmall)
         for s in [keyX,keyY,key_common]:
```

```
            grid = combiGridDummy2D(s,(True,True))
            grid.fillData(parabola_2d)
            combiSimple.addGrid(grid)
        solution_ex = parabola_2d(lmax)
        solution_combi = combiSimple.getCombination()
```

In [11]:
```
sz2=(12,4)
fig = figure( figsize=sz2 )
plot_all_solutions( fig, solution_ex, solution_combi)

err_ct = eN.L1AverageError( solution_combi, solution_ex )

print "Combination technique error:",  "%0.2f" %(err_ct*100),"%"
```

```
Combination technique error: 2.38 %
```



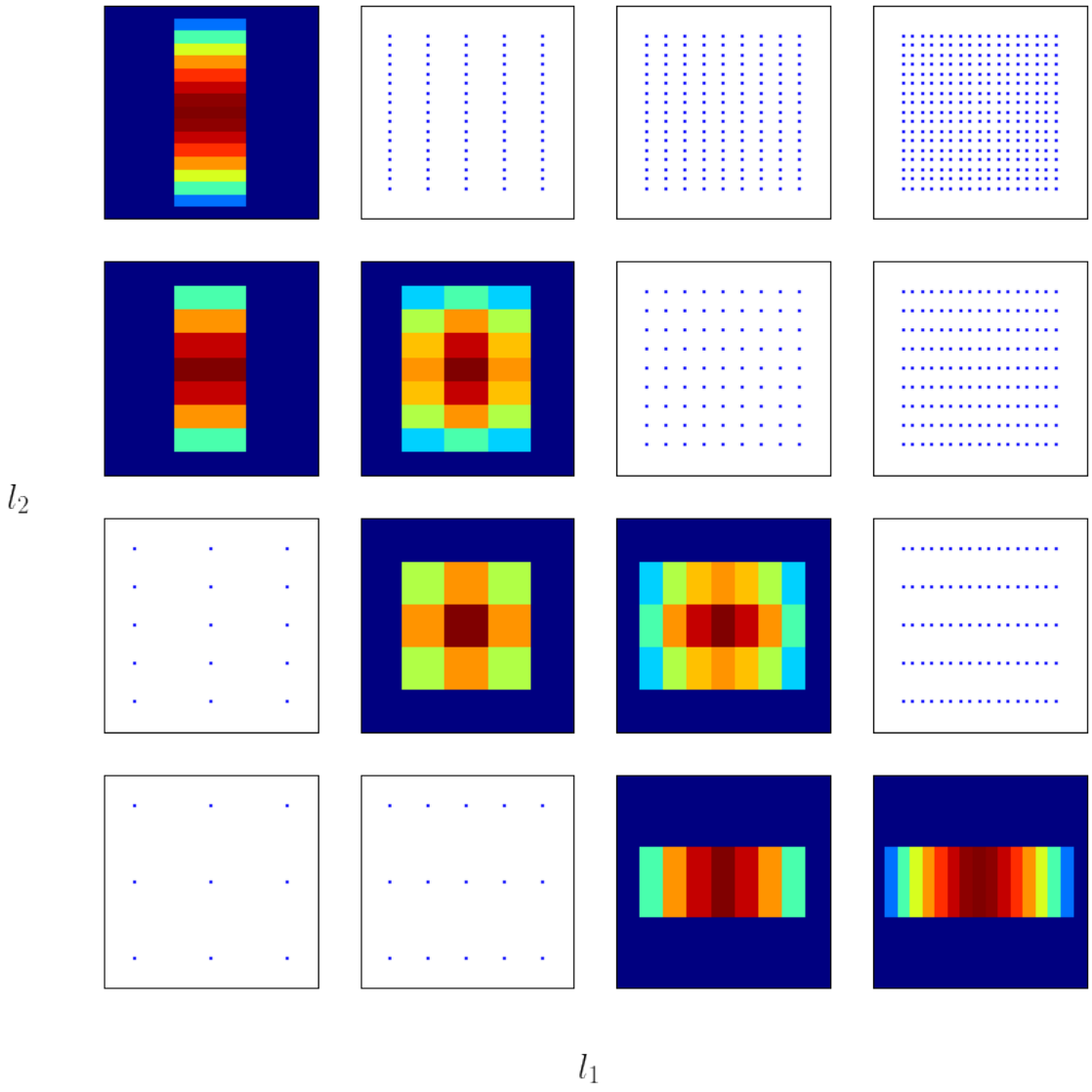
In order to plot the combined grid above, we interpolated the sparse grid to the full grid space. They look quite similar!

The answer to the third question is simple: the three solutions we combined are much cheaper to compute than the full solution. Additionally, since these three problems are independent of one another, they can be solved in parallel!

Let's take a step further and consider an even smarter combination of grids. Have a look at the seven solutions plotted below.

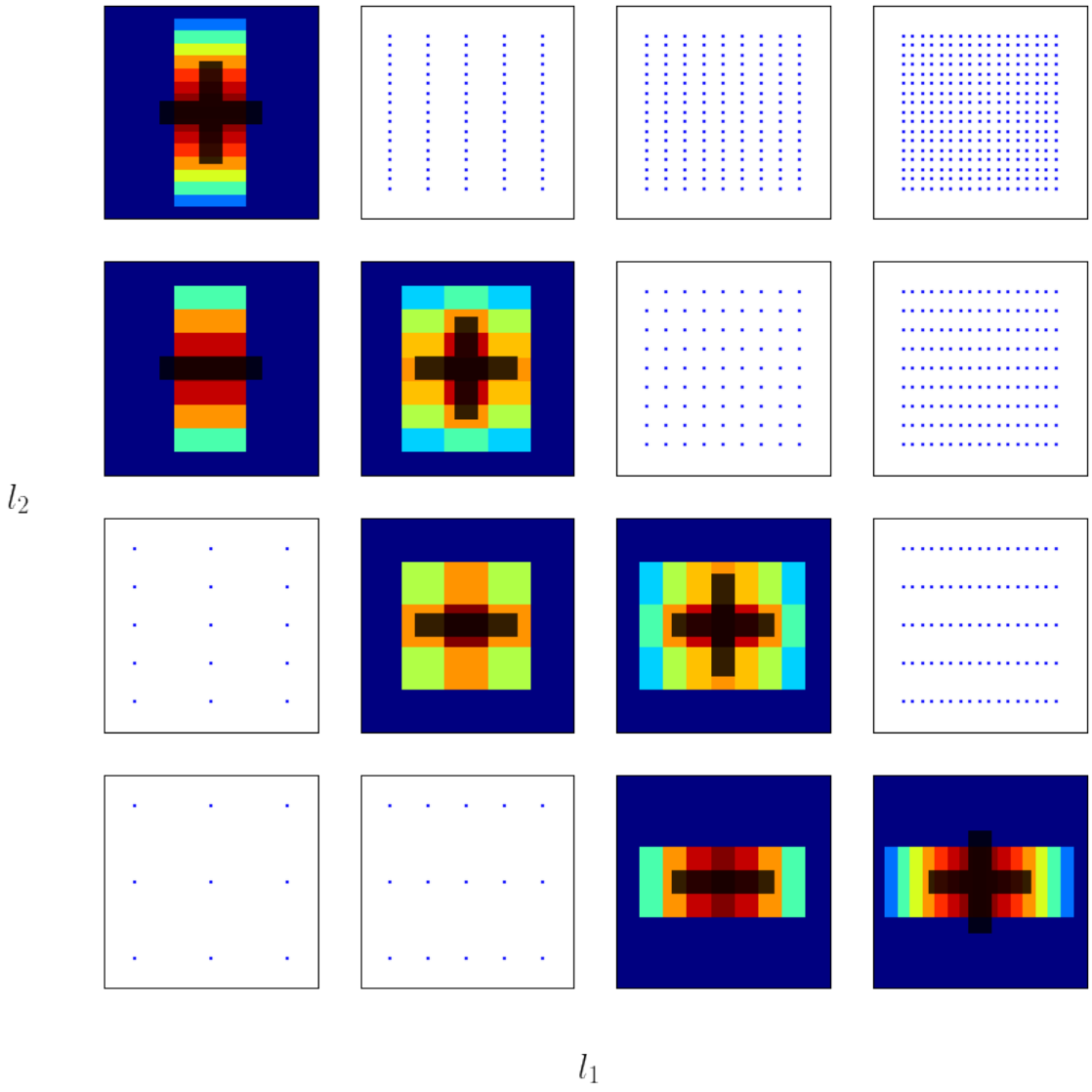
In [12]:
```
keys = scheme.dictOfScheme.keys()
combi_classical = combineGrids(scheme)

for s in scheme.hierSubs((1,1),lmax):
    grid = combiGridDummy2D(s,(True,True))
    if s in keys:
        grid.fillData(parabola_2d)
    else:
        grid.fillData(fun)
    combi_classical.addGrid(grid)
```

In [13]:
```
plot_combination_technique( lmin, lmax, scheme.dictOfScheme, \
                                    combi_classical, figure(figsize=sz), plusMinus=False )
```



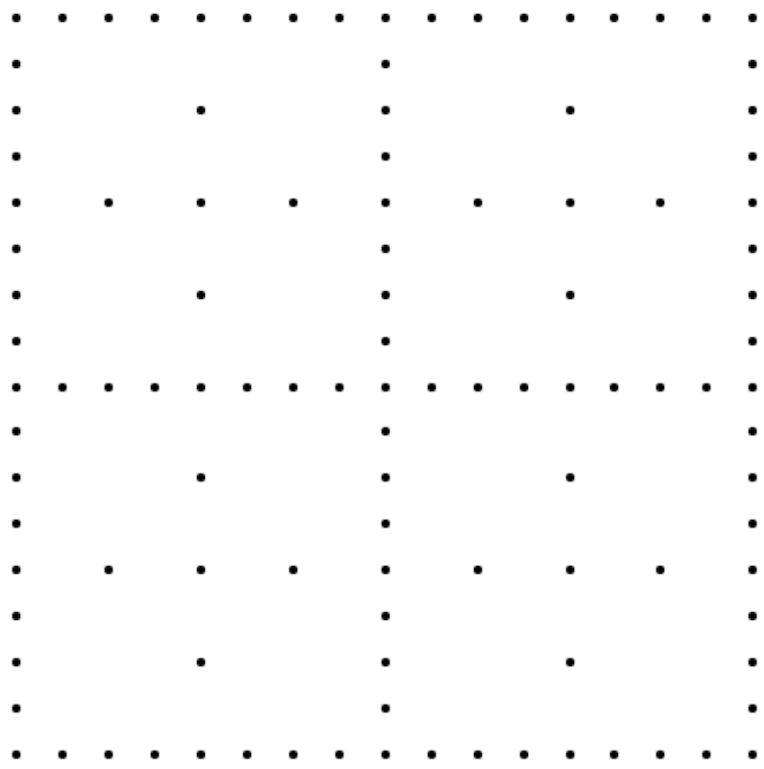
Once again, the solutions plotted above are much cheaper than the full grid with 17x17 points. We can again combine these grids to obtain an approximation of the full grid. The way to combine them is to take weights +1 for grids on the "main diagonal" and -1 for the grids on the "lower diagonal", as illustrated below.

```
In [14]: plot_combination_technique( lmin, lmax, scheme.dictOfScheme, \
                                     combi_classical, figure(figsize=sz), plusMinus=True )
```



This is called the *Classical Combination Technique*. The sparse grid that results from this combination looks as follows:

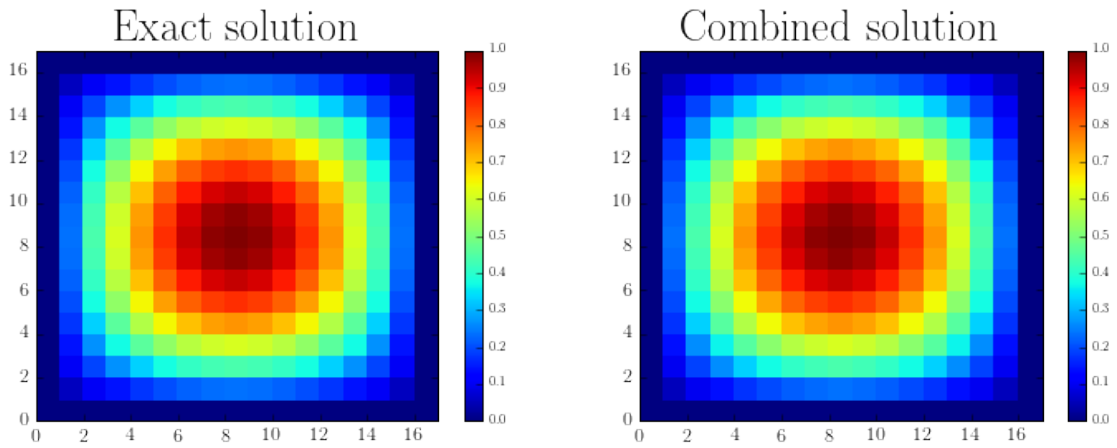
```
In [15]: keys = scheme.dictOfScheme.keys()
         combiSG = combineGrids(scheme)

         for s in keys:
             grid = combiGridDummy2D(s,(True,True))
             grid.fillData(fun)
             combiSG.addGrid(grid)
         sg = combiSG.plotSparseGrid(size=(8,8),ptsize=9,color='black')
```



Combining these seven grids above gives us an even better approximation than our original combination of three grids.

```
In [16]: solution_ex = parabola_2d(lmax)
         solution_combi = combi_classical.getCombination()

         plot_all_solutions( figure(figsize=sz2), solution_ex, solution_combi )

         err_ct = eN.L1AverageError( solution_combi,solution_ex )
         print "Combination technique error:",  "%0.2f" %(err_ct*100),"%"
```

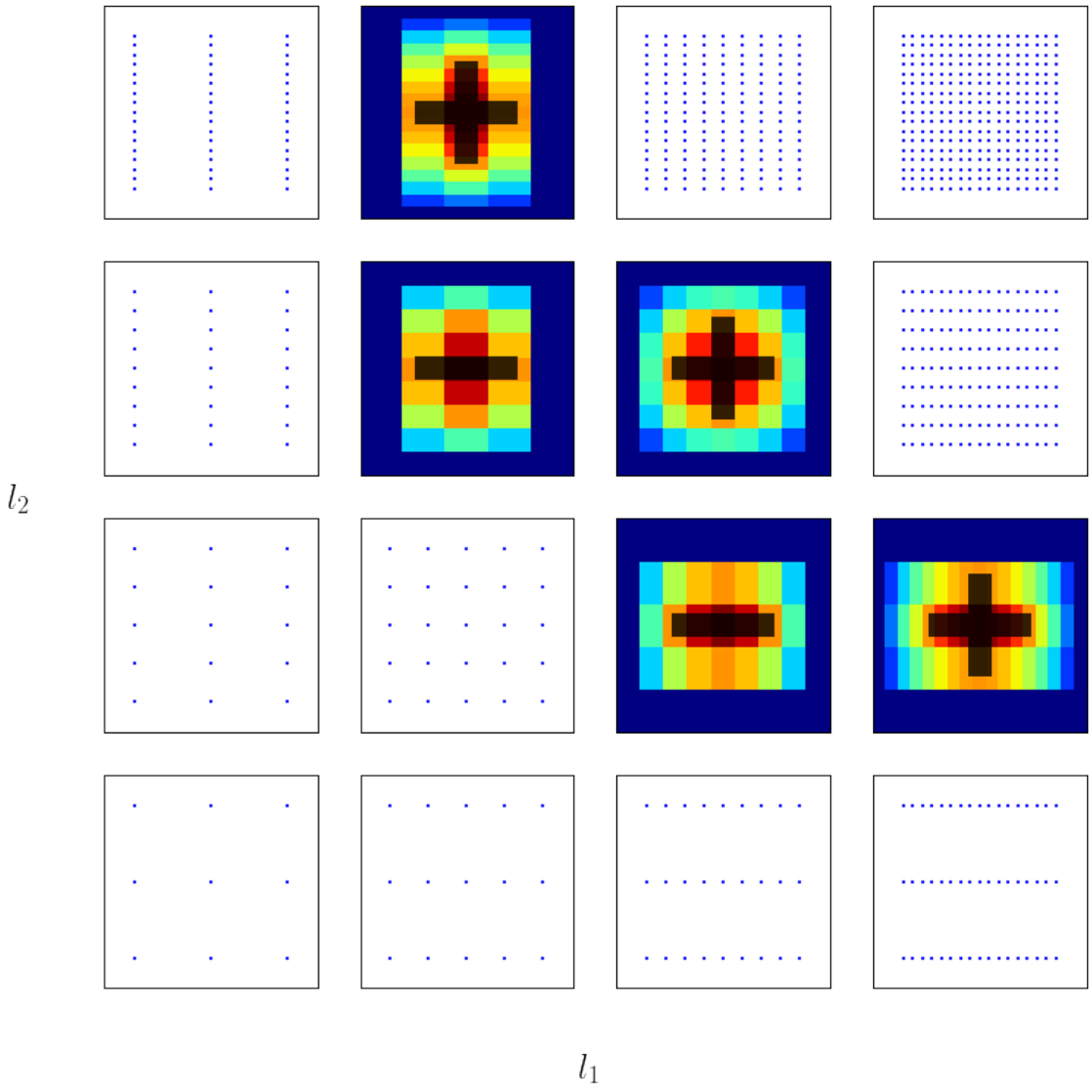
```
Combination technique error: 0.31 %
```



1.1 The truncated combination technique

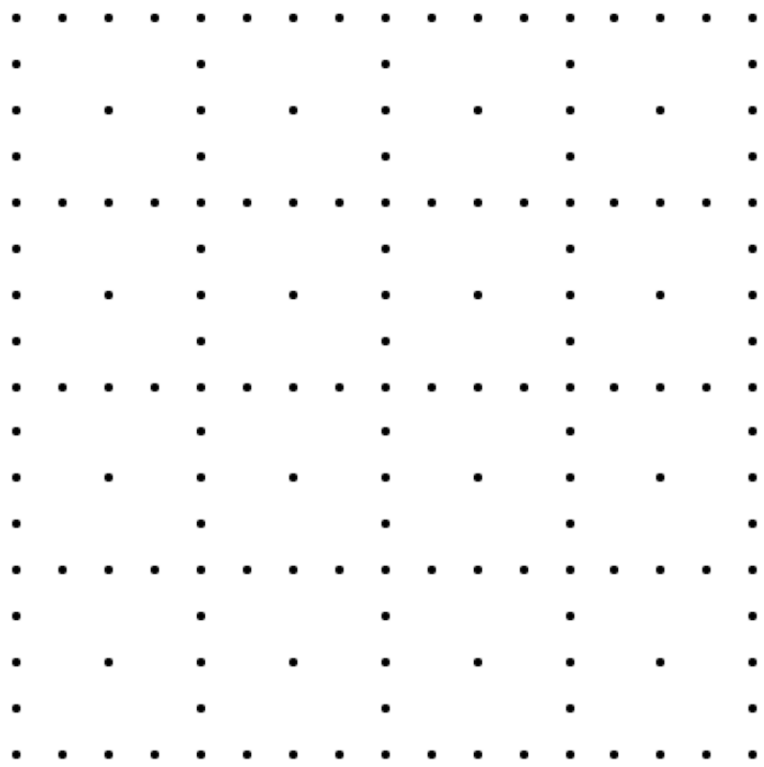
In our example above, the most anisotropic grids might introduce some unwanted error. We are talking about grids with levels (1,4), (1,3), (4,1) and (3,1). For this reason we usually truncate the combination technique, excluding these anisotropic grids. One example could look as follows:

In [17]:
```
# This is where we specify the truncation
# We choose (2,2) as minimum resolution instead of (1,1)
lmin = (2,2)
lmax = (4,4)

dim = len(lmin)

factory = ActiveSetFactory.ClassicDiagonalActiveSet(lmax, lmin, 0)
activeSet = factory.getActiveSet()
scheme = combinationSchemeArbitrary(activeSet)
keys = scheme.dictOfScheme.keys()
combi_trunc = combineGrids(scheme)

for s in scheme.hierSubs((1,1),lmax):
    grid = combiGridDummy2D(s,(True,True))
    if s in keys:
        grid.fillData(parabola_2d)
    else:
        grid.fillData(fun)
    combi_trunc.addGrid(grid)

plot_combination_technique( lmin, lmax, scheme.dictOfScheme, \
                            combi_trunc, figure(figsize=sz) )
```



The sparse grid that results from this combination looks as follows:

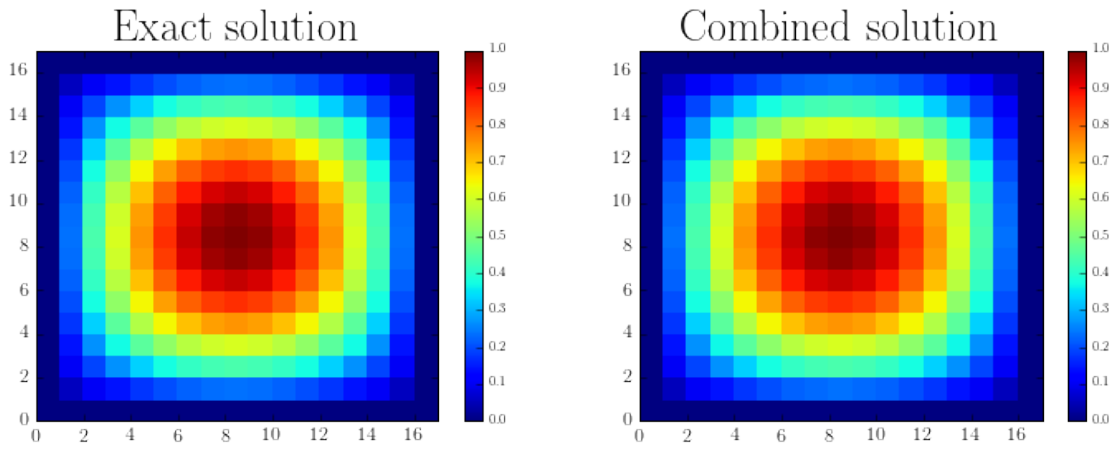
```
In [18]: keys = scheme.dictOfScheme.keys()
         combiSG = combineGrids(scheme)

         for s in keys:
             grid = combiGridDummy2D(s,(True,True))
             grid.fillData(fun)
             combiSG.addGrid(grid)
         sg = combiSG.plotSparseGrid(size=(8,8),ptsize=9,color='black')
```



This combination is slightly more expensive but yields better results.

```
In [19]: solution_ex = parabola_2d(lmax)

         solution_combi = real(combi_trunc.getCombination())

         plot_all_solutions( figure(figsize=(12,4)), solution_ex, solution_combi )
         err_ct = eN.L1AverageError( solution_combi, solution_ex )

         print "Combination technique error:",  "%0.2f" %(err_ct*100),"%"
```

```
Combination technique error: 0.05 %
```

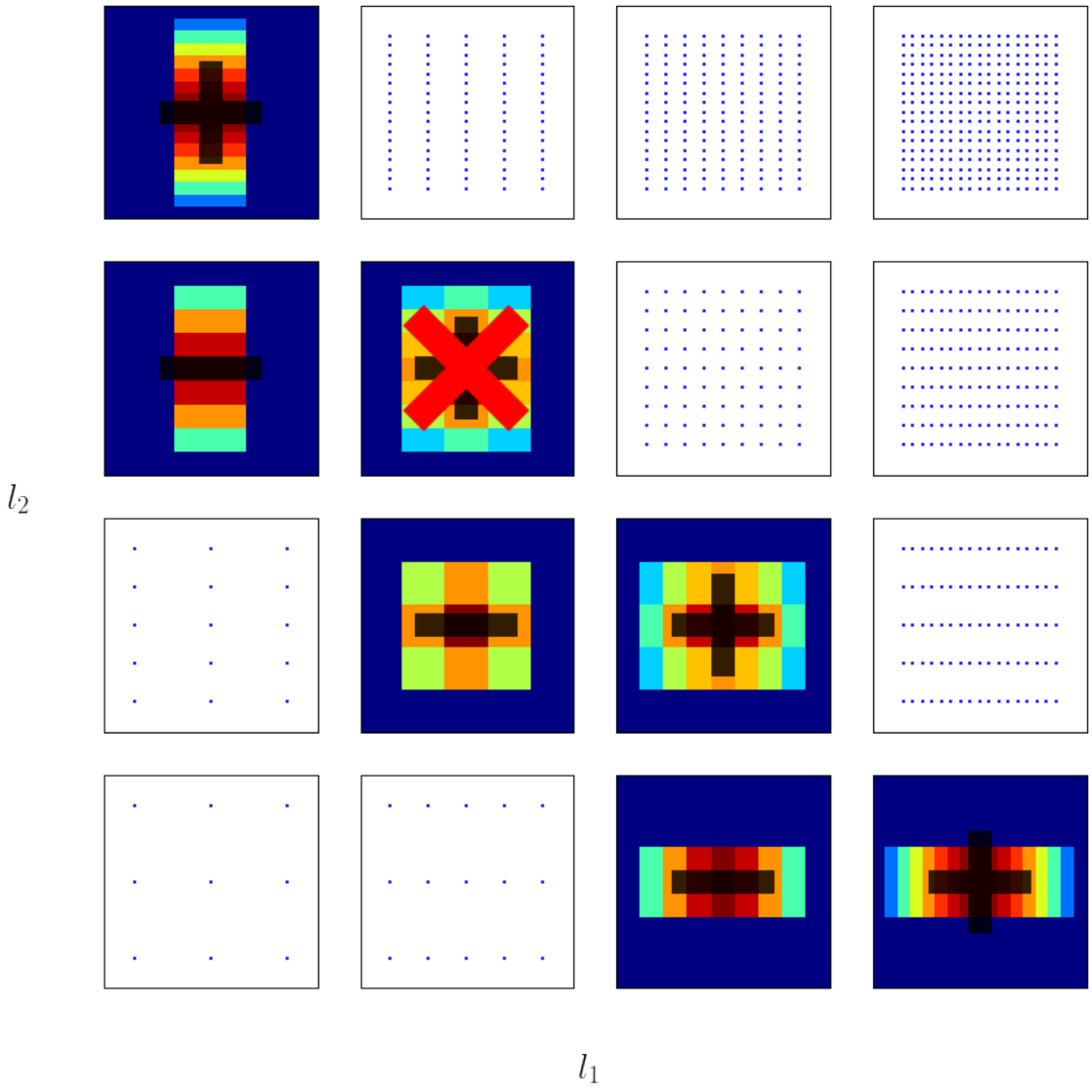


Now you can play around with the simulation parameters and see how the combination technique behaves.

2 Fault tolerance with the combination technique

Since each component solution can be solved independently of each other, the combination technique can be parallelized. However, if some processes fail, some component solutions might go lost, as shown below.

```
In [20]: lmin = (1,1)
         lmax = (4,4)

         factory = ActiveSetFactory.ClassicDiagonalActiveSet(lmax, lmin, 0)
         activeSet = factory.getActiveSet()
         scheme = combinationSchemeArbitrary(activeSet)
         combi_ft = combineGrids(scheme)

         for s in scheme.hierSubs((1,1),lmax):
             grid = combiGridDummy2D(s,(True,True))
             if s in scheme.dictOfScheme.keys():
                 grid.fillData(parabola_2d)
             else:
                 grid.fillData(fun)
             combi_ft.addGrid(grid)

         faults = [(2,3)]
         plot_combination_technique( lmin, lmax, scheme.dictOfScheme, combi_ft, \
                                     figure(figsize=sz), withFaults=True, faults=faults )
```

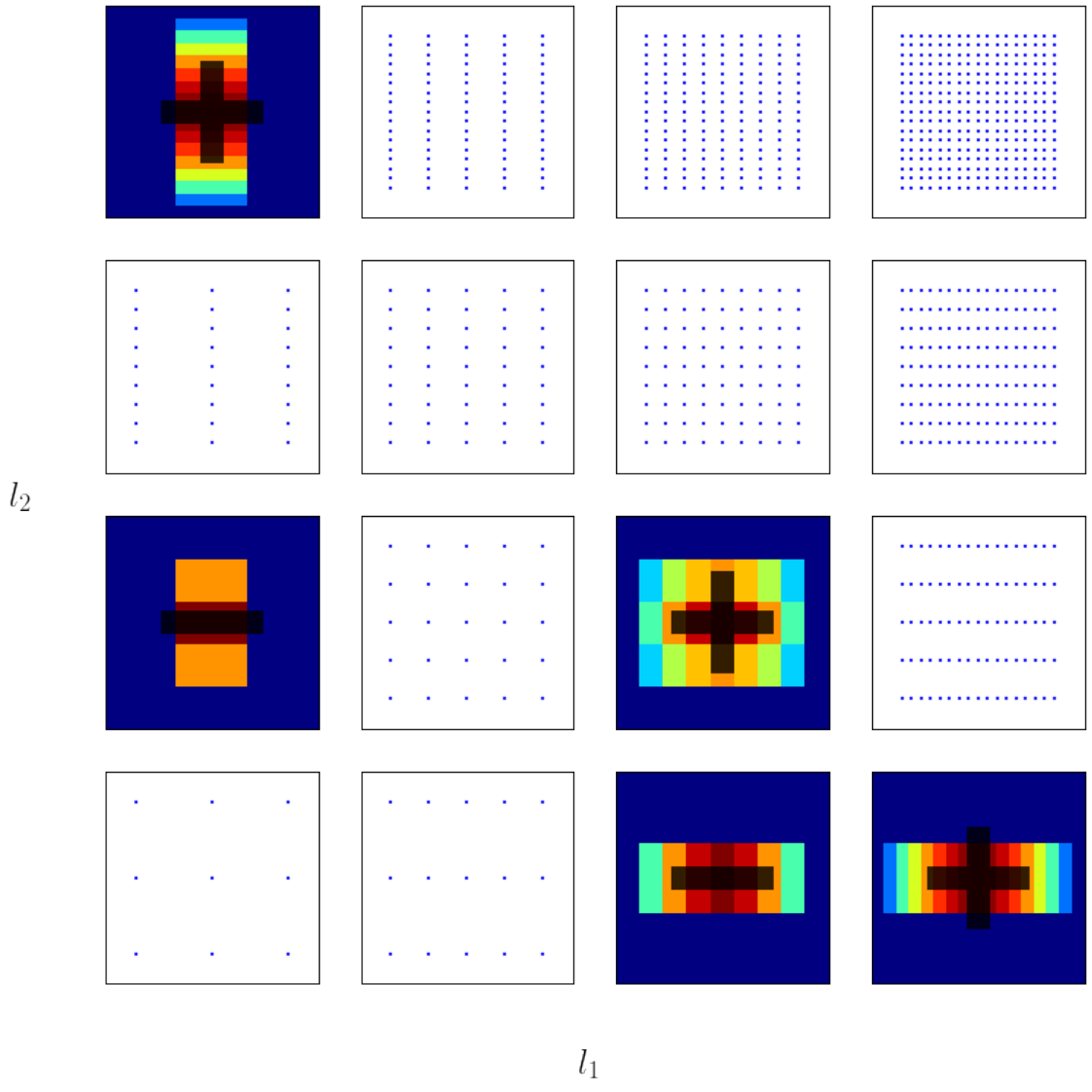


In this case, grid (2,3) has been lost due to a hardware failure. The *Fault Tolerant Combination Technique* can overcome this problem. It excludes the failed solutions by finding an alternative combination of solutions. This is done as follows:

```
In [21]: # Create fault tolerant combination technique
         schemeFT = combinationSchemeFaultTolerant( factory )
         combi_ft = combineGrids(schemeFT)

         for s in schemeFT.hierSubs((1,1),lmax):
             grid = combiGridDummy2D(s,(True,True))
             if s in schemeFT.dictOfScheme.keys():
                 grid.fillData(parabola_2d)
             else:
                 grid.fillData(fun)
             combi_ft.addGrid(grid)
```

```
# Specify which solutions will fail
faults = [(2,3)]

# Recover the scheme
schemeFT.recoverSchemeGCP(faults)

combi_ft = combineGrids(schemeFT)
for s in schemeFT.hierSubs((1,1),lmax):
    grid = combiGridDummy2D(s,(True,True))
    if s in schemeFT.dictOfScheme.keys():
        grid.fillData(parabola_2d)
    else:
        grid.fillData(fun)
    combi_ft.addGrid(grid)
```

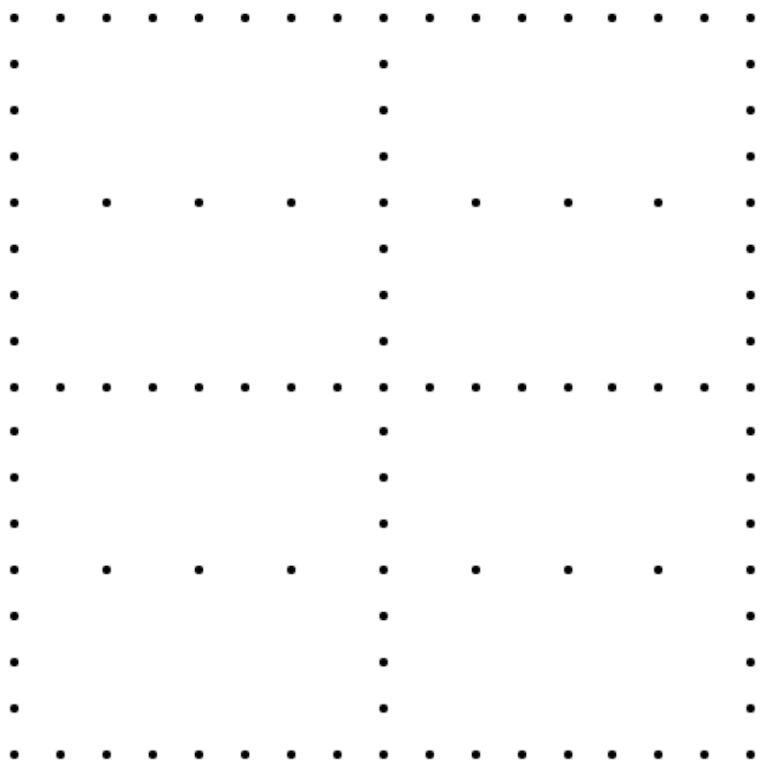
In [22]:
```
plot_combination_technique( lmin, lmax, schemeFT.dictOfScheme, \
                            combi_ft, figure(figsize=sz) )
```



This new combination is not as good as the original since it only uses 5 grids instead of 7. However, we avoid recomputing the solutions we lost, so we don't need any form of checkpoint / restarting.

The sparse grid that results from this combination looks as follows:

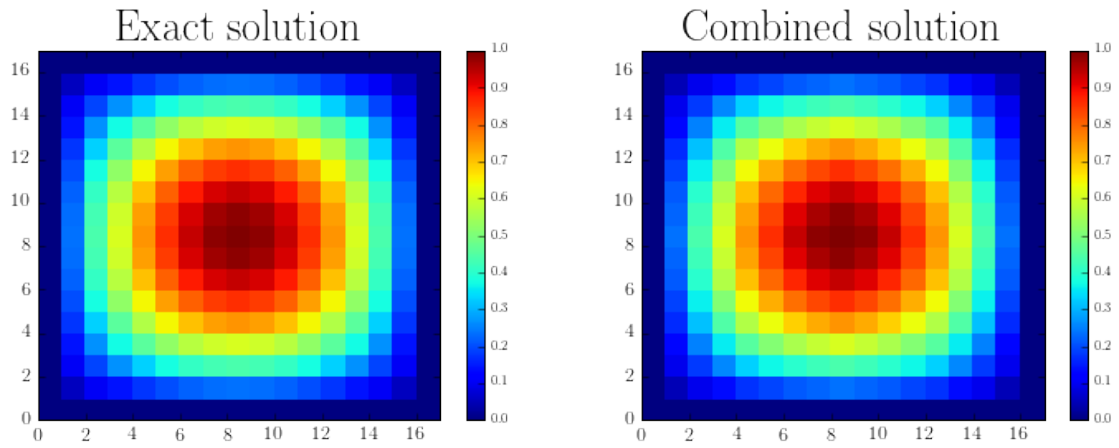
```
In [23]: keys = schemeFT.dictOfScheme.keys()
         combiSG = combineGrids(schemeFT)

         for s in keys:
             grid = combiGridDummy2D(s,(True,True))
             grid.fillData(fun)
             combiSG.addGrid(grid)
         sg = combiSG.plotSparseGrid(size=(8,8),ptsize=9,color='black')
```



You can also see that it has fewer grid points than the sparse grid we had for the combination of the 7 grids. The error of this new combination technique is slightly larger than the original combination technique

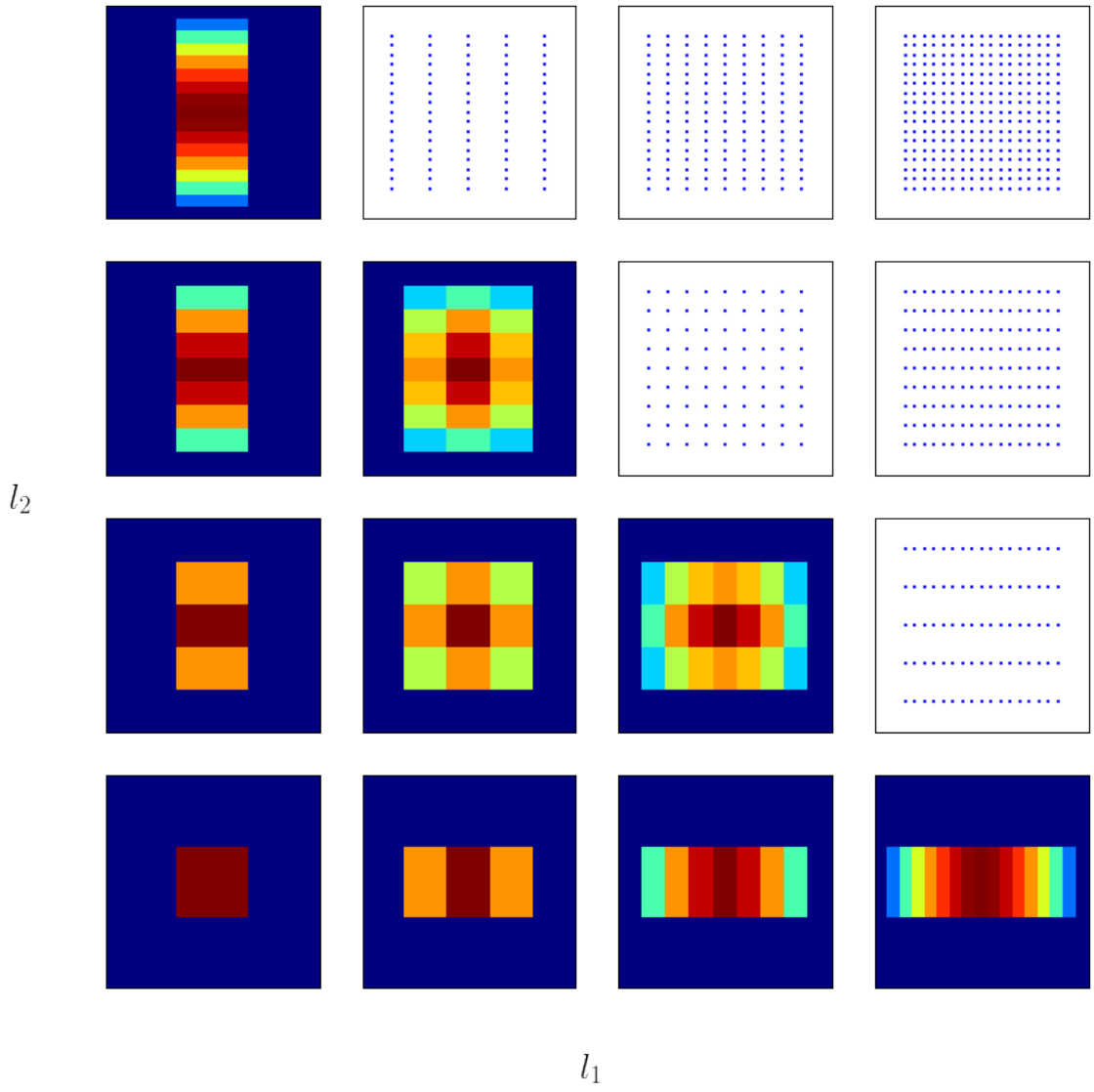
```
In [24]: solution_ex = parabola_2d(lmax)

         solution_combi = real(combi_ft.getCombination())

         plot_all_solutions( figure(figsize=(12,4)), solution_ex, solution_combi )
         err_ct = eN.L1AverageError( solution_combi, solution_ex )

         print "Combination technique error after recovering from faults:",  "%0.2f" %(err_ct*100),"%"
```

```
Combination technique error after recovering from faults: 0.65 %
```



Notice that now we have used a component solution that was not in the original scheme (solution (1,2)). In order for the fault tolerant combination technique to work we have to include these additional solutions in the original scheme, and use them in case any faults occur. This is what the original fault tolerant scheme really should look like:

```
In [25]: schemeFT = combinationSchemeFaultTolerant( factory )
         combi_ft = combineGrids(schemeFT)

         for s in schemeFT.hierSubs((1,1),lmax):
             grid = combiGridDummy2D(s,(True,True))
             if s in schemeFT.dictOfScheme.keys():
                 grid.fillData(parabola_2d)
             else:
                 grid.fillData(fun)
             combi_ft.addGrid(grid)
         plot_combination_technique( lmin, lmax, schemeFT.dictOfScheme, combi_ft, \
                                     figure(figsize=sz), plusMinus=False)
```

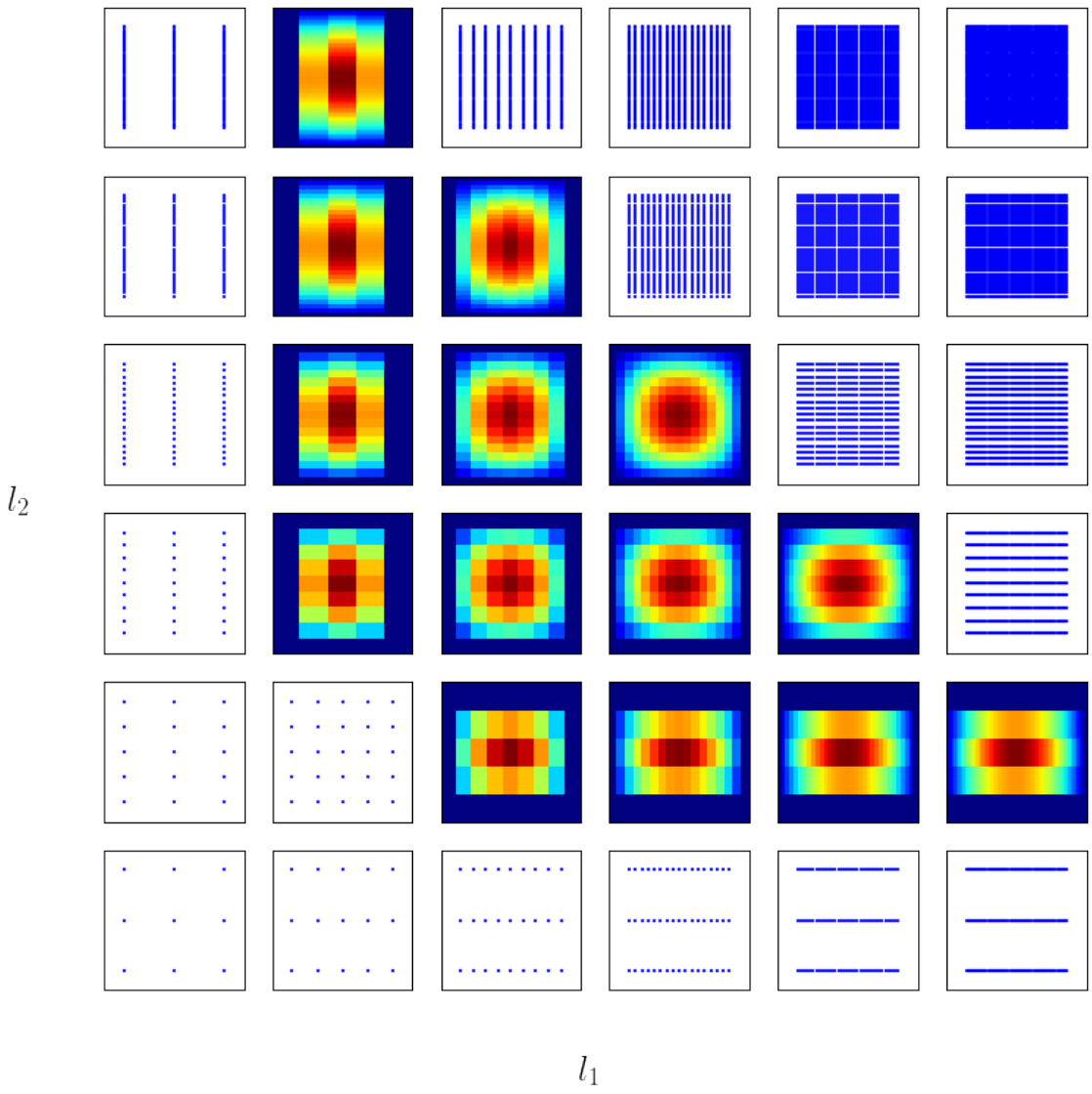


This extra effort (computing 3 additional solutions) is in fact quite small compared to the overall computational effort. In general, we will always compute two extra diagonals of grids to ensure fault tolerance.

Here's another example:

```
In [26]: lmin = (2,2)
         lmax = (6,6)
         factory = ActiveSetFactory.ClassicDiagonalActiveSet(lmax, lmin, 0)
         activeSet = factory.getActiveSet()
         schemeFT = combinationSchemeFaultTolerant( factory )
         combi_ft = combineGrids(schemeFT)

         for s in schemeFT.hierSubs((1,1),lmax):
             grid = combiGridDummy2D(s,(True,True))
             if s in schemeFT.dictOfScheme.keys():
                 grid.fillData(parabola_2d)
             else:
```

```
        grid.fillData(fun)
    combi_ft.addGrid(grid)
plot_combination_technique( lmin, lmax, schemeFT.dictOfScheme, \
                            combi_ft, figure(figsize=sz), plusMinus=False)
```



2.1 Solving time-dependent PDEs with the Combination Technique

The Combination Technique can also be used to solve different classes of PDEs, including PDEs whose solution depends on time.

As an example, consider the 2D linear advection equation:

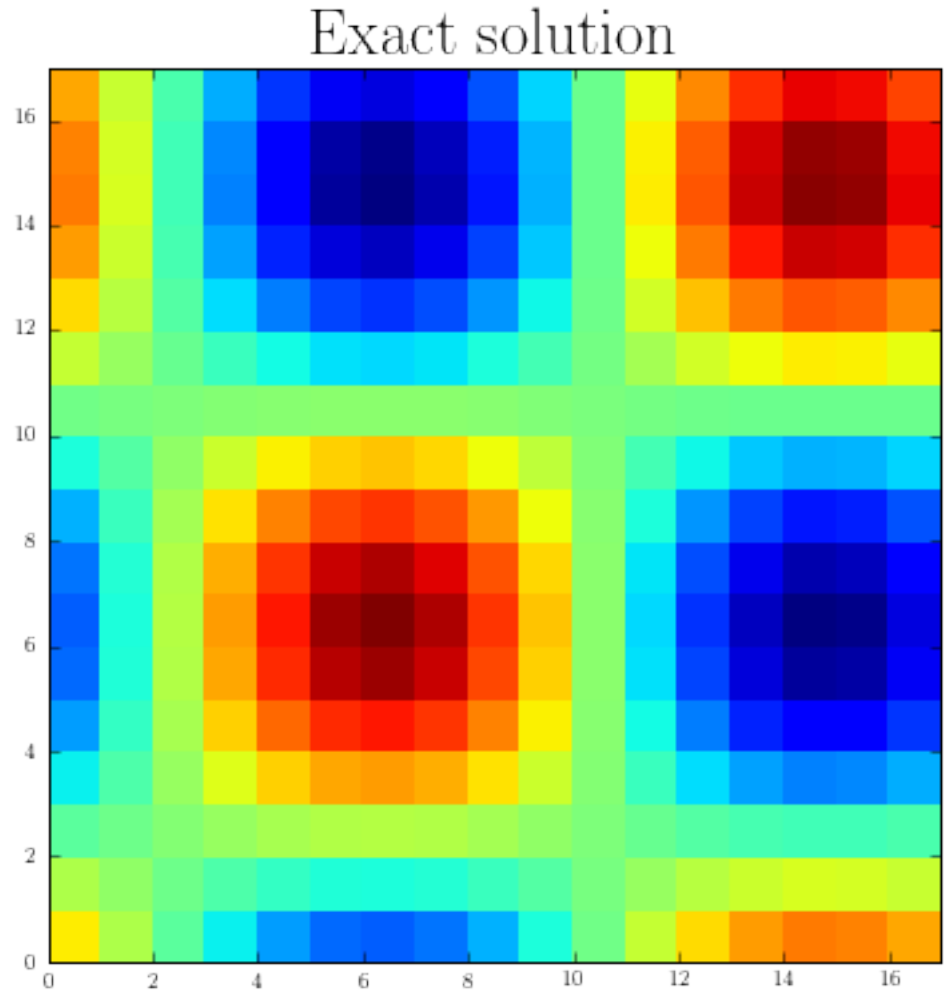
$$\frac{\partial u}{\partial t} + c_x \frac{\partial u}{\partial x} + c_y \frac{\partial u}{\partial y} = 0,$$

in the unit square $[0,1]^2$ with periodic boundary conditions and initial condition $u(x,y,t=0) = \sin(2\pi x)\sin(2\pi y)$. The analytical solution is given by $u(x,y,t) = \sin(2\pi(x-c_x t))\sin(2\pi(y-c_y t))$, where c_x

and c_y are constant advection velocities, and we want to know the solution at a time $t = 2$. We can choose for example $c_x = c_y = 0.5$.

The exact solution to our equation, given by $u(x, y, t) = \sin(2\pi(x - c_x t)) \sin(2\pi(y - c_y t))$, looks like this (you can change the resolution by varying lmax):

```
In [27]: from Advection import *
         c  = (0.5,0.5)
         dt = .01
         tf = 0.2
         Nt = int(ceil(tf/dt))
         lmax = (4,4) # The solution will have (2^lmax + 1) discretization points

         fig = figure(figsize=(6,6))
         ax = gca()
         title('Exact solution',fontsize=24)
         for i in xrange(Nt):
             time.sleep(.001)
             advectionScheme = AdvectionSimple(c,i*dt,dt)
             advectionFunction = lambda l: advectionScheme.exact_advection(l,i*dt)
             grid = combiGridDummy2D(lmax,(True,True))
             grid.fillData(advectionFunction)
             data = grid.getData()
             ax.pcolormesh(data.transpose())
             ax.axes.set_xlim(0, data.shape[0])
             ax.axes.set_ylim(0, data.shape[1])
             clear_output(wait=True)
             display(fig)

         plt.close()
```

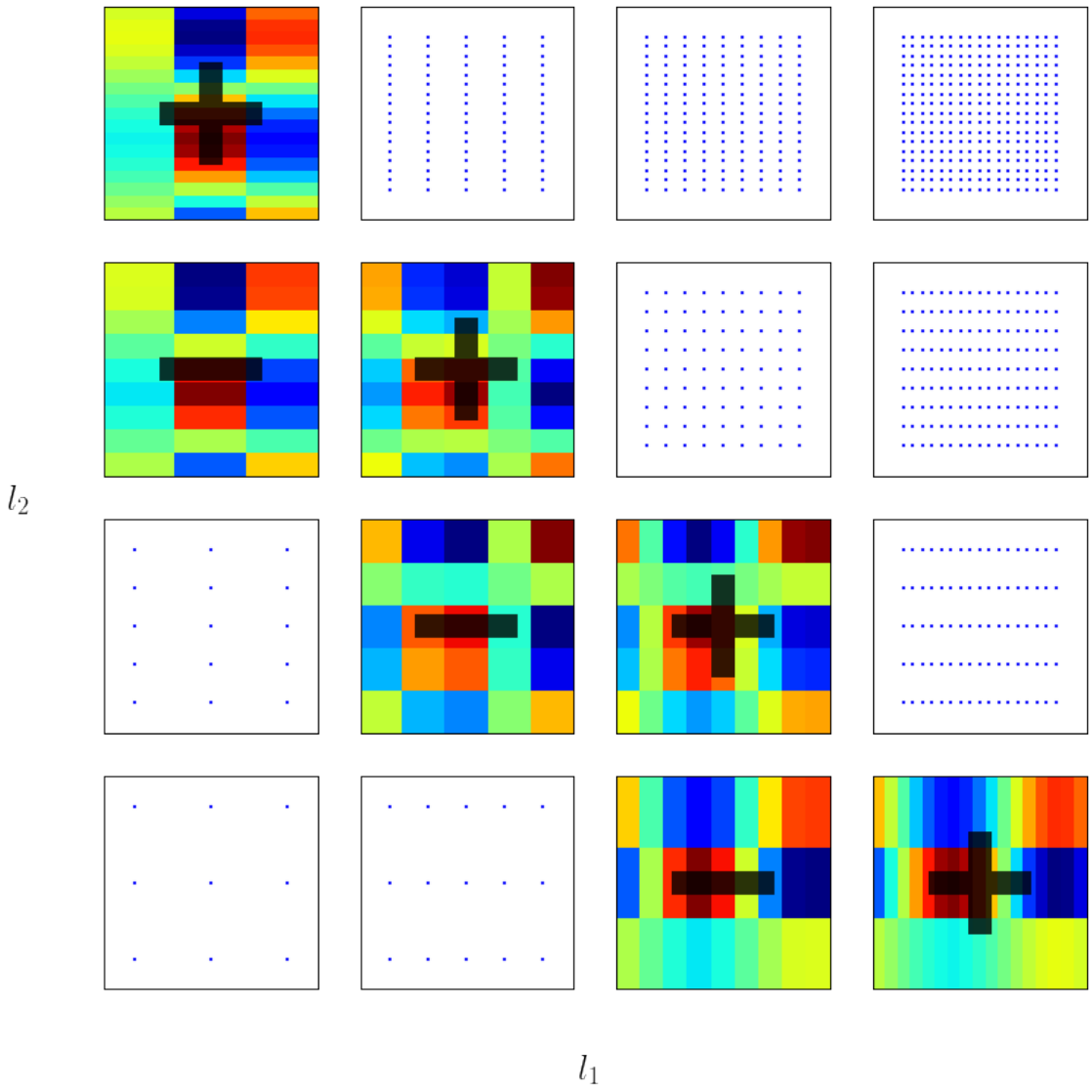


We can use the classical Combination Technique to solve this PDE. This means solving the PDE on each of the seven grids below, and combining them.

In [28]:
```
# Min and max levels
lmin = (1,1)
lmax = (4,4)

factory = ActiveSetFactory.ClassicDiagonalActiveSet(lmax, lmin, 0)
activeSet = factory.getActiveSet()
scheme = combinationSchemeArbitrary(activeSet)

keys = scheme.dictOfScheme.keys()
combiAdv = combineGrids(scheme)
advectionScheme = AdvectionSimple(c,tf,dt)
advectionFunction = lambda l: advectionScheme.solver(l)

for s in scheme.hierSubs((1,1),lmax):
    grid = combiGridDummy2D(s,(True,True))
    if s in keys:
        grid.fillData(advectionFunction)
```

```
        else:
            grid.fillData(fun)
        combiAdv.addGrid(grid)
```

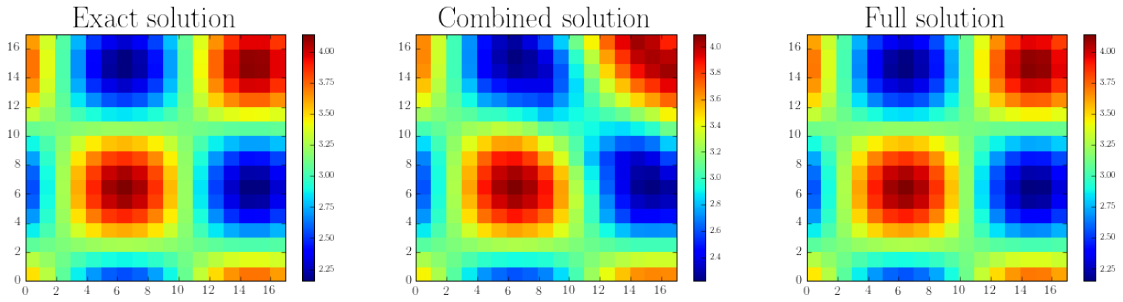
In [29]:
```
plot_combination_technique( lmin, lmax, scheme.dictOfScheme, \
                            combiAdv, figure(figsize=sz), plusMinus=True)
```



In [30]:
```
solution_ex = advectionScheme.exact_advection(lmax,Nt*dt)
solution_combi = combiAdv.getCombination()

fullgrid = combiGridDummy2D(lmax,(True,True))
fullgrid.fillData(advectionFunction)
solution_full = fullgrid.getData()

sz3=(18,4)
```

```
plot_all_solutions( figure(figsize=sz3), solution_ex, solution_combi, solution_full )

err_full = eN.L1AverageError( solution_full,solution_ex )
err_ct = eN.L1AverageError( solution_combi,solution_ex )
print "Relative errors compared to the exact solution:"
print "Full solution:", "%0.2f" %(err_full*100),"%"
print "Combination technique solution:",  "%0.2f" %(err_ct*100),"%"
```

```
Relative errors compared to the exact solution:
Full solution: 1.10 %
Combination technique solution: 6.92 %
```



2.2 References

- Griebel, M., Schneider, M., Zenger, C.: *A combination technique for the solution of sparse grid problems*. In: Iterative Methods in Lin. Alg., pp. 263–281 (1992)
- Harding, B., Hegland, M., Larson, J., Southern, J.: *Fault tolerant computation with the sparse grid combination technique*. SIAM Journal on Scientific Computing 37(3), C331–C353 (2015)

2.3 Appendix

Brief explanation of the code Throughout this notebook we have repeatedly used several functions that perform the combination technique. The first step to generate a combination technique is to specify the minimum and maximum levels of resolution:

lmin = (2,2) lmax = (5,5)

With these parameters we create an object of the class *ActiveSetFactory*, with which we can create an active set. For the classical combination technique, the active set is the set of levels on the main diagonal (those with coefficients +1)

```
In [31]: factory = ActiveSetFactory.ClassicDiagonalActiveSet(lmax, lmin, 0)
         activeSet = factory.getActiveSet()
         print activeSet
```

```
set([(4, 1), (3, 2), (2, 3), (1, 4)])
```

With this active set we create a combination technique scheme which stores all levels and coefficients of the combination technique

```
In [32]: scheme = combinationSchemeArbitrary(activeSet)
         print scheme.dictOfScheme
```

```
{(3, 2): 1, (1, 3): -1, (3, 1): -1, (1, 4): 1, (2, 3): 1, (2, 2): -1, (4, 1): 1}
```

Then we use an object of the class *combineGrids* to store the data from all the combination grids. Each combination grid is an object of the class *combiGridDummy2D*, which uses the method *fillData* to fill its data. The grids are then added to the *combi* object.

```
In [33]: combiG = combineGrids(scheme)
         keys = scheme.dictOfScheme.keys()
         combi = combineGrids(scheme)

         for s in keys:
             grid = combiGridDummy2D(s,(True,True))
             grid.fillData(parabola_2d)
             combi.addGrid(grid)
```

Finally, we call the method *getCombination* in order to perform the combination of the data. It returns a numpy array with the combination solution.

```
In [34]: solution_combi = combi.getCombination()
```