NB: The graded, first version of the report must be returned if you hand in a second time!

H3b: Multigrid

Gustav Hjelmare and Albin Karlsson

December 18, 2014

Task №	Points	Avail. points
Σ		

Introduction

Solving differential equations is an important part of physics, but it is often impractical to do by hand. In such cases, the equations are discretized and a computer is used to solve them. Using more points gives a more detailed solution, but is also more computationally costly due to the larger number of points to consider. In addition, a larger number of points makes corrections spread more slowly, further increasing the time spent. In order to tackle this trade off, and make the best use of computational resources, various multi-resolution discretized methods have been developed. In this home problem we investigate some of them.

Task 1

In task 1 we solved Poisson's equation

$$\nabla^2 \Phi(r) = -\frac{1}{\epsilon_0}\rho(r) \tag{1}$$

with Dirichlet boundary conditions in a square of dimension $L \times L$. The charge distribution ρ used was

$$\rho(r) = \lambda[\delta(r - r_+) - \delta(r - r_-)] \tag{2}$$

where $r_\pm = r_c \pm d/2$ and r_c, the location of the center of the dipole, is $(x = L/2, y = L/2)$ and d,the separation vector of the dipole, is taken to be $0.2\hat{x}$. The delta functions are discretized as $\delta(r_i - r_j) = (1/h^2)\delta_{ij}$, with h being the distance between grid points in the simulation. For simplicity, units were used such that $\lambda = \epsilon_0 = L = 1$.

The problem was solved by the *two-grid method*, using Gauss-Seidel relaxation (equation (3)).

Two-grid method:

(a) Presmooth: Apply a few relaxation iterations.

(b) Compute the residual R and restrict it to a coarser grid.

(c) On the coarser grid, solve the residual equation exactly.

(d) Interpolate the correction E obtained from the residual equation to the finer grid and update the solution with it.

(e) Postsmooth: Apply a few relaxation iterations.

In implementing the above algorithm, we used 4 Gauss-Seidel iterations each for pre- and postsmoothing.

Gauss-Seidel relaxation is described by

$$u_{i,j}^{n+1} = \frac{1}{4}(u_{i+1,j}^{n} + u_{i-1,j}^{n+1} + u_{i,j+1}^{n} + u_{i,j-1}^{n+1} - \frac{1}{h^2}\rho_{i,j}) \tag{3}$$

where the n superscript denotes the current iteration, and the i, j subscripts denote x and y coordinates of the grid point.

We used Gauss-Seidel both for pre- and postsmoothing (steps (a) and (e) above), and to solve the residual equation (equation (5)) "exactly" (step (c)). In our case, "exactly" means applying GS iterations until no point $E_{i,j}$ in E changes more than 10^{-5} from it's previous value. The same convergence criterion is used for deciding when to stop iterating the two-grid method.

$$R = \rho - \nabla^2\tilde{\Phi} \tag{4}$$

$$\nabla^2 E = R \tag{5}$$

The correction E is then applied to the current approximation $\tilde{\Phi}$ of the solution, in order to bring the approximation closer to the true solution.

$$E = \Phi - \tilde{\Phi} \tag{6}$$

The derivatives in the gradients (∇^2) have been discretized as in equation (7)

$$\frac{\partial^2 u(x,y)}{\partial x^2} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\triangle x)^2} \tag{7}$$

The solution over the square area can be visualized in a surface plot as in Figure 1. However, the results from different methods and grid sizes are very similar, and it is difficult to see these differences in such a plot. Thus, we plot the solution along the line $y = 1/2$ as in Figure 2, where it is easier to compare with the exact solution.

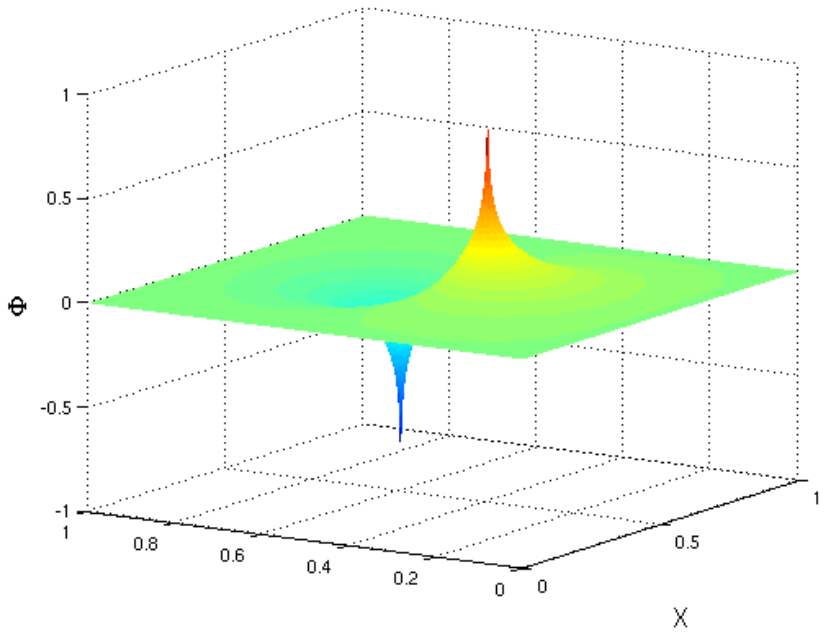

Figure 1: Throughout all tasks in this home problem the simulations resulted in figures that resemble this one, with minor differences. To be able to see these differences we will throughout the rest of the report display the result in 2D ($x \in [0, 1]$ and $y = L/2 = 1/2$).

Task 2

The two-grid method can be generalized to multiple grid sizes, which is what have been done in task 2. The implementation of the multi-grid method is described below.

Multigrid(Φ, ρ):

(a) If on the coarsest grid, solve $\nabla^2\Phi = \rho$ exactly, else:
   i. Presmoot: Apply a few relaxation iterations.
   ii. Compute the residual $R = \rho - \nabla^2\Phi$.
   iii. Restrict R to a coarser grid and put v:=0.
   iv. v:=Multigrid(v,R) γ times.
   v. Interpolate v to a finer grid.
   vi. Update $\Phi := \Phi + v$
   vii. Postsmooth: Apply a few relaxation iterations.

(b) return Φ

In this method γ determines how the problem is dealt with in different grid sizes. If $\gamma = 1$ the simulation performes the so called V-cycle, while if $\gamma = 2$ it performes a W-cycle. This behavior is illustrated in figure 8 and 9. All constants remain the same as in Task 1, as do the convergence criteria.
The multi-grid method was performed using the maximum grid sizes 81, 161, 321, 641, 1281 and the results can be seen in figure 4, 5, 6 and 7.

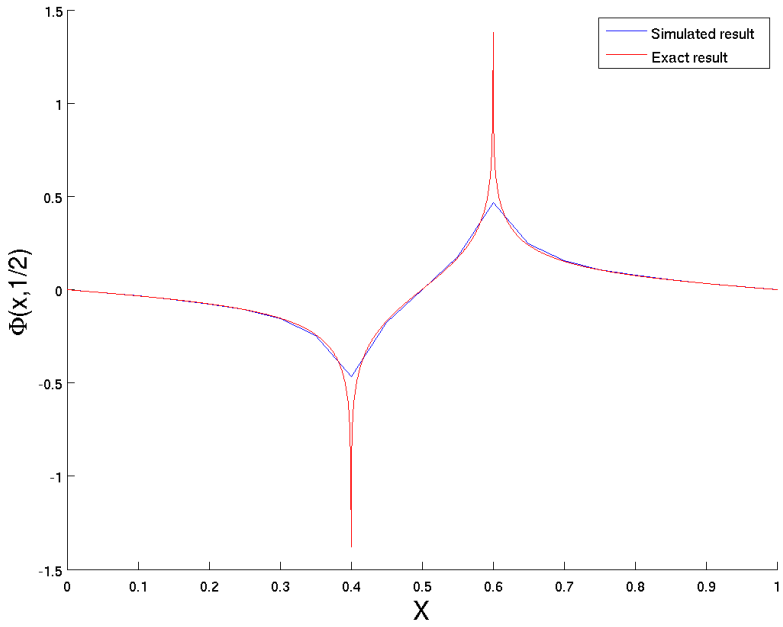

Figure 2: Simulated result using the two-grid method on 11 / 21 grid points, compared to the exact solution. The simulated solution corresponds well to the exact one, except that the sharp peaks are significantly blunter in the simulated result. Detail shown in Figure 2.

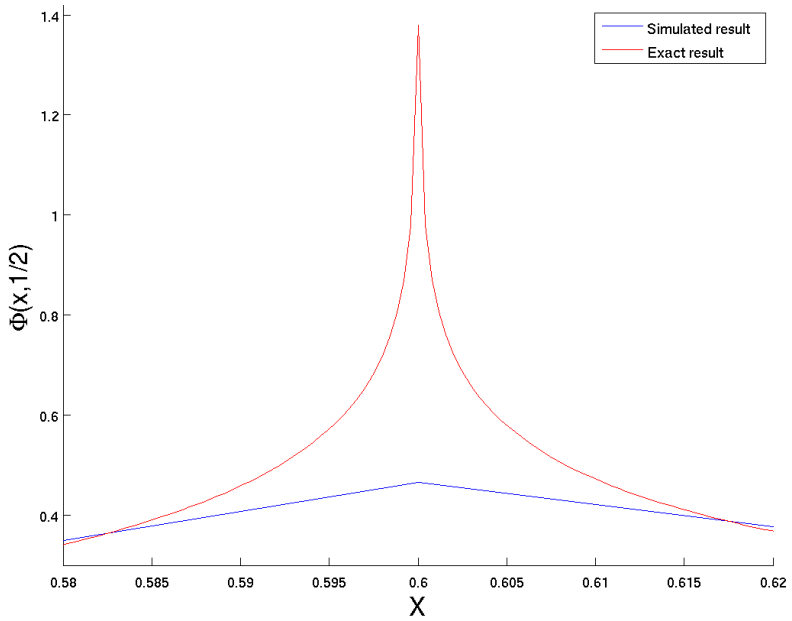

Figure 3: Using only 21 grid points for the finer of the two grids, the two-grid method does not resolve the sharp peaks very well. This figure is an enlarged part of figure 2.

According to Figures 5 and 7 we see that the errors in the solution from the simulations, in comparison with the exact solution, decrease with growing grid sizes independently of what type of cycle is used. In order to compare the results from the two different cycles, we have displayed them both in Figure 10 which shows us that there is no distinguishable difference. Allthough the two cycles result in the same

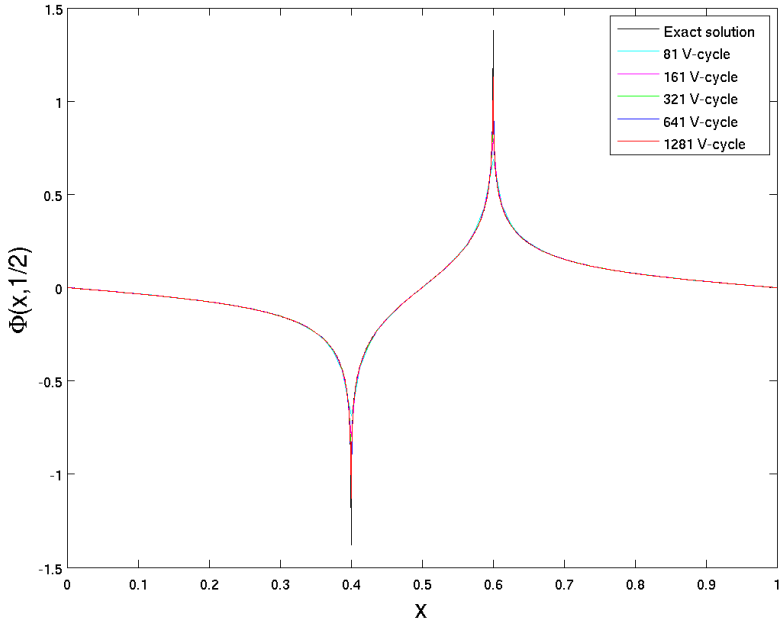

Figure 4: Multigrid V-cycle on different grid sizes. The results are virtually indistinguishable, but there are slight differences in how well the different grid sizes are able to resolve the sharp peaks, as seen in Figure 5.

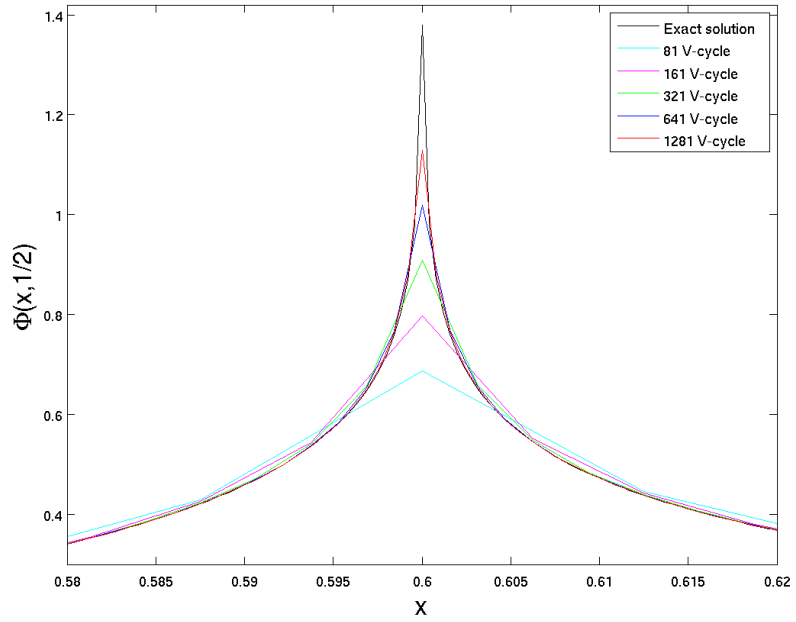

Figure 5: Multigrid V-cycle on different grid sizes, showing the difference between grid sizes in ability to resolve the sharp peaks.

solution, it is seen in figure 11 that the V-cycle requires fewer Gauss-Seidel iterations than the W-cycle. We can also see that the computational cost of the multi-grid method seems to depend on the grid size as to the power of 2, in comparison to simply applying Gauss-Seidel on one grid size (as in exercise E6) which gave a dependence on the grid size as to the power of 3.82.

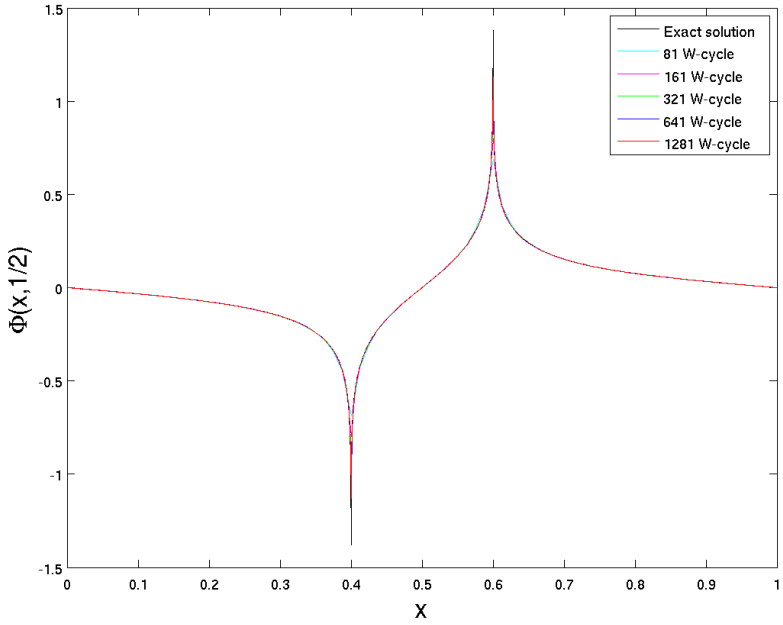

Figure 6: Multigrid W-cycle on different grid sizes. Once again, the results are virtually indistinguishable, except for in the sharpness of the peaks; detail shown in Figure 7.

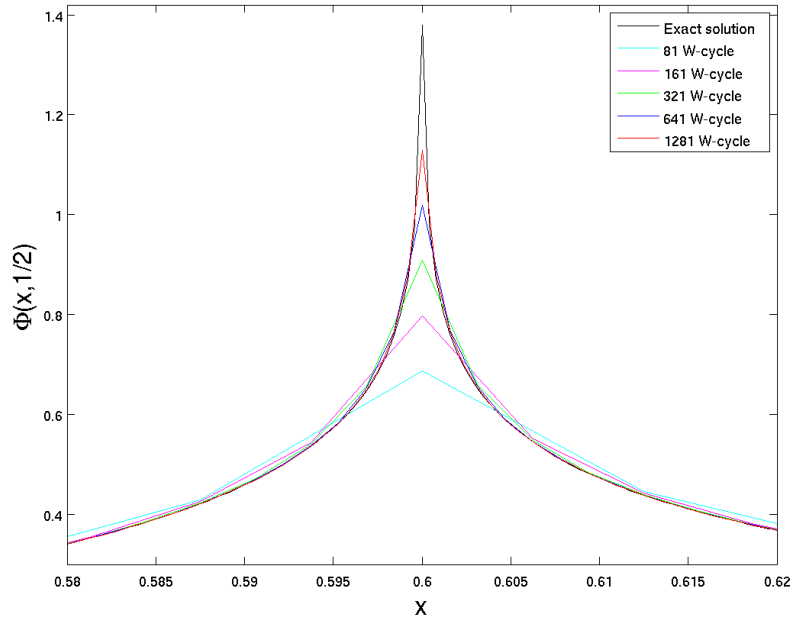

Figure 7: Multigrid W-cycle on different grid sizes, showing the difference between grid sizes in ability to resolve the sharp peaks. There are no noticeable differences between W- and V-cycles, as shown in figure 10.

Task 3

One can make small modifications to the method described in Task 2 to get the *full multigrid method*. In this method we start at the coarsest grid size, and calculate the exact solution. The solution is then interpolated to a finer grid where one calls the multigrid function as described in the previous section. The result from the multigrid

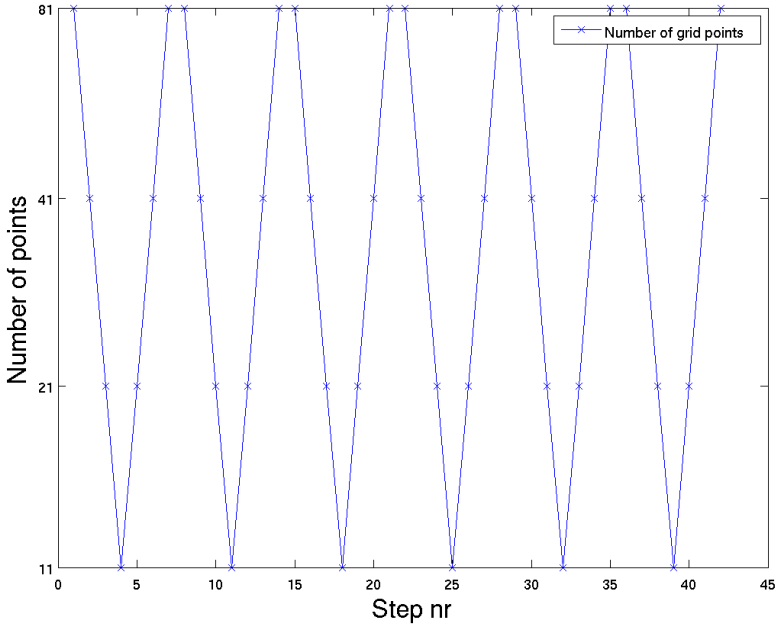

Figure 8: Multigrid V-cycle on grid size 81, showing the coarsening and interpolation of the grid from start to convergence. The doubled uppermost points are an artefact of how we save the data, and do not represent repeated calculations at the same grid size.

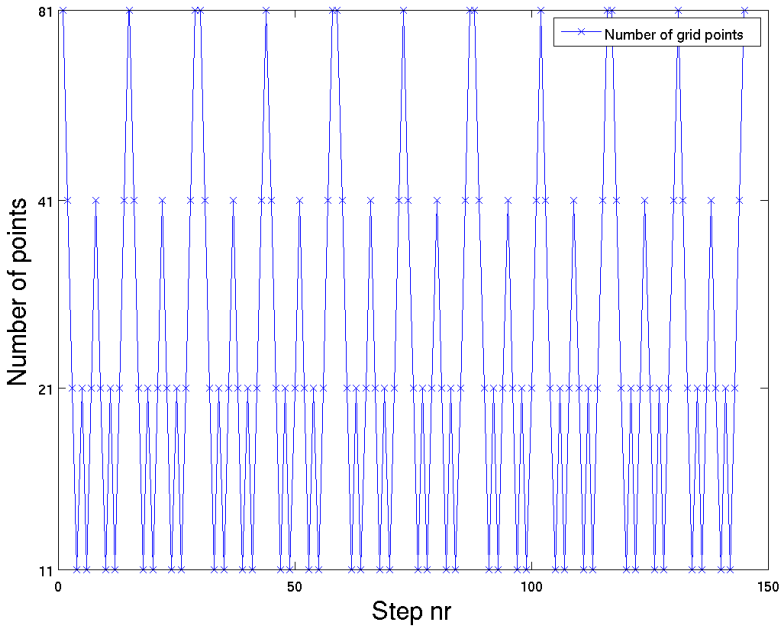

Figure 9: Multigrid W-cycle on grid size 81, showing the coarsening and interpolation of the grid from start to convergence. As in Figure 8, the doubled upper points are artefacts of how we save the data.

function is in turn interpolated to a finer grid, and so on. One keeps doing this until arriving at the finest grid. This method has been used with the same maximum grid sizes that were mentioned in Task 2. In the full multigrid method γ is equal to 1 which means that the grid sizes that are considered are a combination of V-cycles as can be seen in Figure 14. Results from the simulations are shown in Figures 12 and 13. When

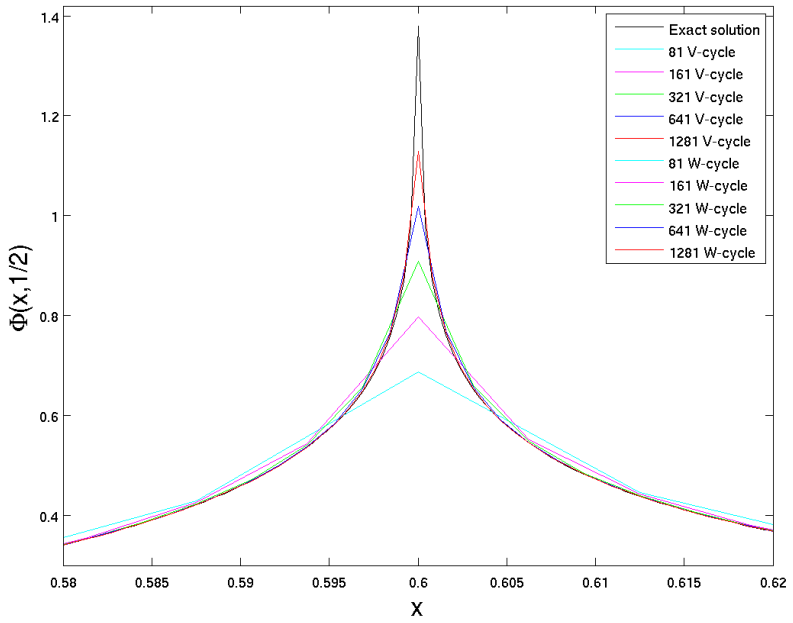

Figure 10: Comparison between V- and W-cycles, showing their identical results. For each grid size, both V- and W-cycle solutions are plotted in the same color. There is no distinguishable difference between the two cycles' solutions.

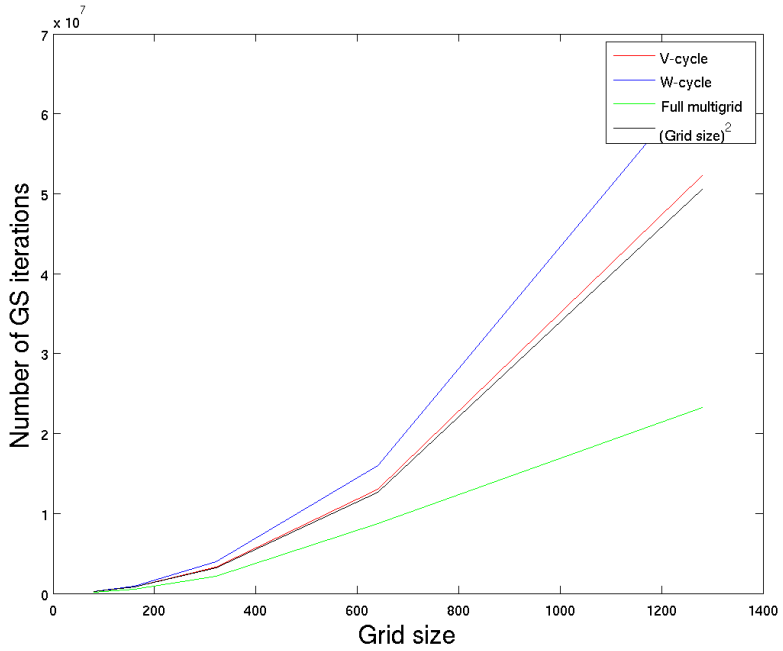

Figure 11: Number of Gauss-Seidel relaxations performed (counted as one per grid point per iteration of GS), as a function of grid size, for V- and W-cycle multigrid as well as full multigrid. The V-cycle method is superior to the W-cycle in this problem for all of the investigated grid sizes. Full multigrid beats both V- and W-cycle. The black curve shows us that the multi grid methods' computational cost is related to the grid size as to the power of 2, with the full multigrid being even cheaper at approximately grid size to the power of 1.7.

comparing the full multigrid method to the results in Task 2 we see that the full multigrid

is slightly less exact than the multigrid method with V- and W-cycles, but as seen in Figure 11 it is also significantly cheaper in terms of computational cost, scaling as grid size to the power of 1.7.

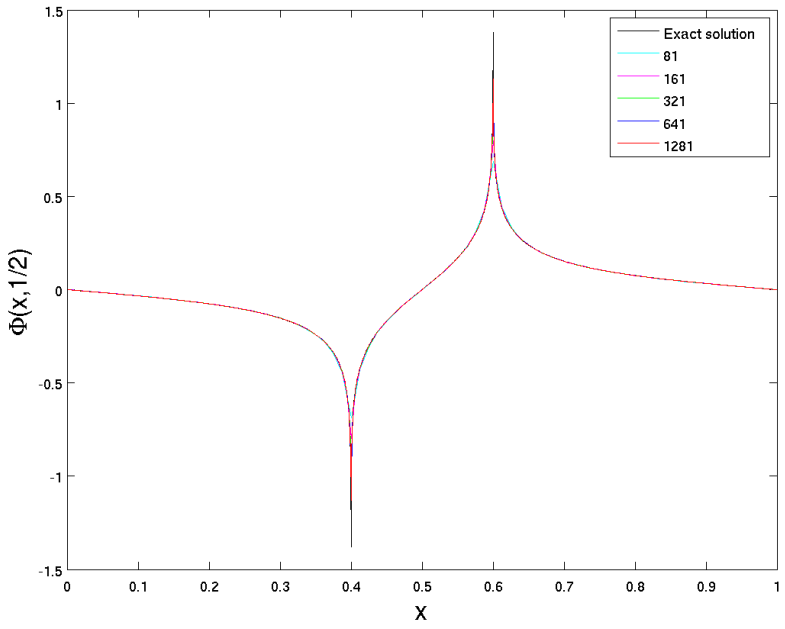

Figure 12: Full multigrid solutions for five different grid sizes, compared with the exact solution. All of the grid sizes yield results that are practically indistinguishable; see Figure 13 for a more detailed view of one of the sharp peaks, showing the differences that do arise.

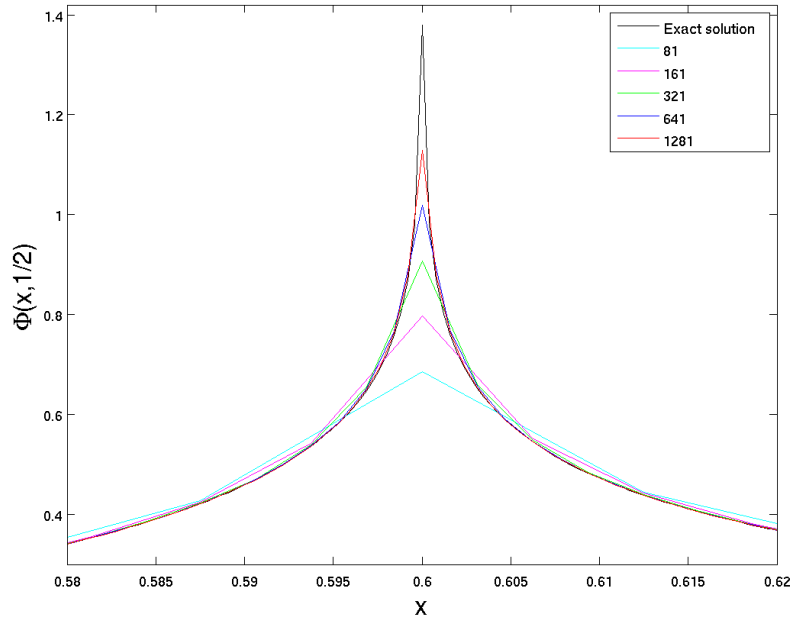

Figure 13: Restriction of Figure 12 to better show the difference in how well the different grid sizes are able to resolve the sharp peak of the exact solution.

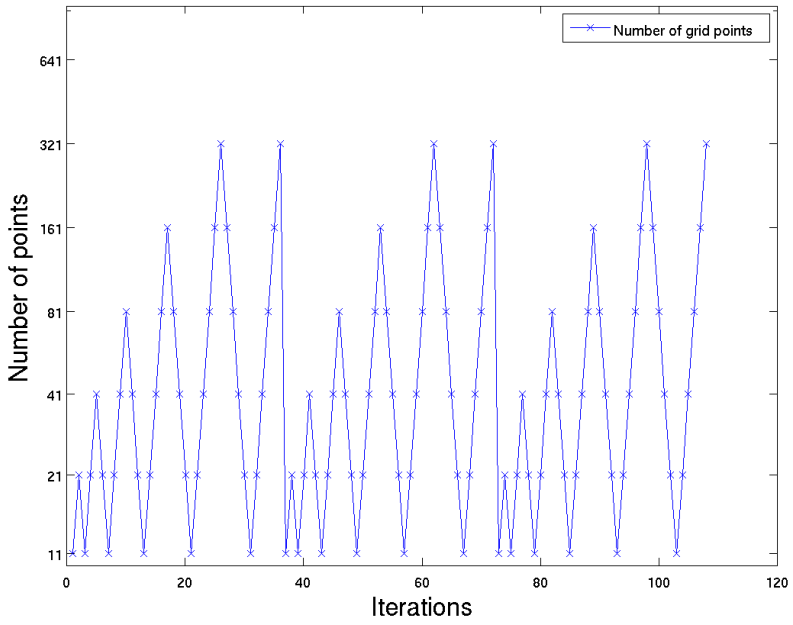

Figure 14: Number of points used in each iteration, showing the typical full multigrid behaviour. The top level in this plot uses 321 grid points.

A Source Code

Task 1

A.1 Main program: `task1/main.c`

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "mg_func.h"

int main() {

  int nCoarsePoints = 11;
  int nFinePoints = 2*(nCoarsePoints - 1) +1; // must be N = 2*(n-1)+1

  double chargeSeparation = 0.2;
  double totalLength = 1;
  double fineCellLength = totalLength / (nFinePoints - 1);
  double coarseCellLength = totalLength / (nCoarsePoints - 1);
  int fineChargeOffset = chargeSeparation / 2 * \
  (nFinePoints - 1)/totalLength;

  double fineGrid[nFinePoints][nFinePoints];
  double oldFineGrid[nFinePoints][nFinePoints];
  double fineRho[nFinePoints][nFinePoints];
  double fineResidual[nFinePoints][nFinePoints];
  double coarseResidual[nCoarsePoints][nCoarsePoints];
  double coarseRho[nCoarsePoints][nCoarsePoints];
  double coarseError[nCoarsePoints][nCoarsePoints];
  double fineError[nFinePoints][nFinePoints];

  int x, y, i, z,j;
  double diff;
  int nPresmooth = 2;   // "a few"
  int nPostsmooth = 2;
  double innerTolerance = 0.00001;
  double innerMaxDiff = 2*innerTolerance;  // high enough to start looping
  double outerTolerance = 0.00001;
  double outerMaxDiff = 2*outerTolerance;

  // Initializing
  for ( x = 0 ; x < nFinePoints ; x++ ) {
    for ( y = 0 ; y < nFinePoints ; y++ ) {
      fineGrid[x][y] = 0;
      fineRho[x][y] = 0;
```

```
      fineResidual[x][y] = 0;
      fineError[x][y] = 0;
    }
  }
  for ( x = 0 ; x < nCoarsePoints ; x++ ) {
    for ( y = 0 ; y < nCoarsePoints ; y++ ) {
      coarseResidual[x][y] = 0;
      coarseRho[x][y] = 0;
      coarseError[x][y] = 0;
    }
  }

  // Setting up the source distribution
  fineRho[nFinePoints / 2 + fineChargeOffset][nFinePoints / 2 ] = \
  -1.0 / pow(fineCellLength,2);
  fineRho[nFinePoints / 2 - fineChargeOffset][nFinePoints / 2 ] = \
  1.0 / pow(fineCellLength,2);
  // End of init part

  while( outerMaxDiff > outerTolerance) {
    // Keep a copy of the old solution to check convergence
    for( x = 0 ; x < nFinePoints ; x++ ) {
      for( y = 0 ; y < nFinePoints ; y++ ) {
        oldFineGrid[x][y] = fineGrid[x][y];
      }
    }

    for( i = 0 ; i < nPresmooth ; i++ ) {
      GaussSeidel( nFinePoints, fineCellLength, fineGrid, fineRho);
    }

    ComputeResidual(nFinePoints, fineCellLength, fineGrid, \
    fineRho, fineResidual);

    DecreaseGridDensity(nFinePoints, fineResidual, coarseResidual);

    // Solve residual eq on coarse grid "exactly" i.e. to within tolerance
    innerMaxDiff = 2*innerTolerance;
    while (innerMaxDiff > innerTolerance ) {
      innerMaxDiff = GaussSeidel( nCoarsePoints, coarseCellLength, \
      coarseError, coarseResidual);
    }

    IncreaseGridDensity(nCoarsePoints, coarseError, fineError);

    // Apply interpolated correction
    for ( x = 0 ; x < nFinePoints ; x++ ) {
      for ( y =0 ; y < nFinePoints ; y++ ) {
        fineGrid[x][y] = fineGrid[x][y] + fineError[x][y];
      }
    }

    for ( i = 0 ; i < nPostsmooth ; i++ ) {
      GaussSeidel( nFinePoints, fineCellLength, fineGrid, fineRho);
    }

    // Check size of largest change made this iteration
    outerMaxDiff = 0;
    for( x = 0 ; x < nFinePoints ; x++ ) {
      for( y = 0 ; y < nFinePoints ; y++ ) {
        diff = fabs(oldFineGrid[x][y] - fineGrid[x][y]);
        outerMaxDiff = diff > outerMaxDiff ? diff : outerMaxDiff;
      }
    }
  }

  // Save solution
  FILE *fGrid = fopen("grid.data","w");

  int nPlotPoints = nFinePoints;
  for ( x = 0 ; x < nPlotPoints ; x++ ) {
    for (  y = 0 ; y < nPlotPoints ; y++ ) {
      fprintf(fGrid,"%e\t",fineGrid[x][y]);
    }
    fprintf(fGrid,"\n");
  }

  printf("Done!\n");

  return 0;
}
```

A.2 Function header: `task1/mg_func.h`

```
#ifndef _mg_func_h
#define _mg_func_h
```

```
3
4  extern double GaussSeidel(int size, double cellLength, \
5      double grid[size][size], double rho[size][size]);
6  extern double IncreaseGridDensity(int n, double in[n][n], \
7      double out[2*n-1][2*n-1]);
8  extern double DecreaseGridDensity(int n, double in[n][n], \
9      double out[n/2+1][n/2+1]);
10 extern void ComputeResidual(int n, double c, double grid[n][n], \
11     double rho[n][n], double res[n][n]);
12
13 #endif
```

A.3 Function code: `task1/mg_func.c`

```
1  #include <stdio.h>
2  #include <math.h>
3  #include <stdlib.h>
4  #define PI 3.141592653589
5
6  double GaussSeidel(int nPoints, double cellLength, \
7  double grid[nPoints][nPoints], double rho[nPoints][nPoints]) {
8    double maxDiff = 0;
9    double oldValue, diff;
10   int x,y;
11
12   for ( x = 1 ; x < nPoints - 1 ; x++ ) {
13     for ( y = 1 ; y < nPoints - 1 ; y++ ) {
14       oldValue = grid[x][y];
15       grid[x][y] = 1.0/4.0 * (grid[x+1][y] + grid[x-1][y] \
16       + grid[x][y+1] + grid[x][y-1] - pow(cellLength,2)*rho[x][y]);
17       diff = fabs(oldValue - grid[x][y]);
18       maxDiff = diff > maxDiff ? diff : maxDiff;
19     }
20   }
21
22   return(maxDiff);
23 }
24
25 void IncreaseGridDensity(int nPoints, double inGrid[nPoints][nPoints], \
26 double outGrid[2*nPoints-1][2*nPoints-1]) {
27   int x, y;
28
29   // the exactly matching points
30   for ( x = 0 ; x < nPoints ; x++ ) {
31     for ( y = 0 ; y < nPoints ; y++ ) {
32       outGrid[2*x][2*y] = inGrid[x][y];
33     }
34   }
35   // points with four nearest neighbours
36   for ( x = 1 ; x < 2*nPoints-1 ; x += 2 ) {
37     for ( y = 1 ; y < 2*nPoints-1 ; y += 2 ) {
38       outGrid[x][y] = 1.0/4.0 * (inGrid[x/2][y/2] + inGrid[x/2+1][y/2] \
39       + inGrid[x/2][y/2+1] + inGrid[x/2+1][y/2+1]);
40     }
41   }
42   // the half way points in y
43   for ( x = 0 ; x < 2*nPoints-1 ; x += 2 ) {
44     for ( y = 1 ; y < 2*nPoints-1 ; y += 2 ) {
45       outGrid[x][y] = 1.0/2.0 * (inGrid[x/2][(y-1)/2] \
46       + inGrid[x/2][(y+1)/2]);
47     }
48   } // and in x
49   for ( x = 1 ; x < 2*nPoints-1 ; x += 2 ) {
50     for ( y = 0 ; y < 2*nPoints-1 ; y += 2 ) {
51       outGrid[x][y] = 1.0/2.0 * (inGrid[(x-1)/2][y/2] \
52       + inGrid[(x+1)/2][y/2]);
53     }
54   }
55
56   return;
57 }
58
59 void DecreaseGridDensity(int nPoints, double inGrid[nPoints][nPoints], \
60 double outGrid[nPoints/2+1][nPoints/2+1]) {
61   int x,y;
62
63   for ( x = 1 ; x < nPoints/2+1 ; x++ ) {
64     for ( y = 1 ; y < nPoints/2+1 ; y++ ) {
65       outGrid[x][y] = 1.0/4.0 * inGrid[2*x][2*y] \
66                     + 1.0/8.0 * (inGrid[2*x-1][2*y] + inGrid[2*x+1][2*y] \
67                     + inGrid[2*x][2*y-1] + inGrid[2*x][2*y+1]) \
68                     + 1.0/16.0 * (inGrid[2*x-1][2*y-1] \
69                     + inGrid[2*x+1][2*y-1] + inGrid[2*x+1][2*y+1] \
70                     + inGrid[2*x-1][2*y+1]);
71     }
72   }
```

```
73
74    return;
75  }
76
77  void ComputeResidual(int nPoints, double cellLength, \
78  double grid[nPoints][nPoints], double rho[nPoints][nPoints], \
79  double residual[nPoints][nPoints]) {
80    double oldValue, diff;
81    int x,y;
82    double tolerance = 0.00001;
83    double maxDiff = 2*tolerance;
84
85    for ( x = 1 ; x < nPoints - 1 ; x++ ) {
86      for ( y = 1 ; y < nPoints - 1 ; y++ ) {
87        residual[x][y] = rho[x][y] \
88                         - (grid[x-1][y] - 2*grid[x][y] + grid[x+1][y]) \
89                         / pow(cellLength,2) \
90                         - (grid[x][y-1] - 2*grid[x][y] + grid[x][y+1]) \
91                         / pow(cellLength,2);
92      }
93    }
94
95    return;
96  }
```

A.4 Plotting in matlab: `task1/plot_stuff.m`

```
1   %% potential plot
2   clear all
3   clc
4   clf
5   textStorlek = 14;
6   legendStorlek = 11;
7
8   data = dlmread('grid.data','\t');
9   exactData = dlmread('../phi_exact.data', '\t');
10
11  data = data(:,1:end-1);
12
13  middleIndex = (length(data)-1)/2 + 1;
14
15  xData = linspace(0,1,length(data));
16  xExact = linspace(0,1, length(exactData));
17
18  surf(xData,xData,data)
19  xlabel('Y', 'FontSize', textStorlek);
20  ylabel('X', 'FontSize', textStorlek);
21  zlabel('\Phi(x,y)', 'FontSize', textStorlek);
22  shading flat
23  %view(45,0)
24  figure
25  hold on
26  plot(xData, data(:,middleIndex))
27  plot(xExact, exactData, 'r');
28  xlabel('X', 'FontSize', textStorlek);
29  ylabel('\Phi(x,1/2)', 'FontSize', textStorlek);
30  text=legend('Simulated result', 'Exact result');
31  set(text, 'FontSize', legendStorlek);
32
33  saveas(gcf,'task1.png','png')
34
35  axis([0.58 0.62 0.3 1.42])
36
37  saveas(gcf,'task1_zoom.png','png')
```

Task 2

A.5 Main program: `task2/main.c`

```
1   #include <stdio.h>
2   #include <math.h>
3   #include <stdlib.h>
4   #include "mg_func.h"
5
6   int main() {
7     // Setting to be changed for each run
8     int nPoints = 81;
9     FILE *fGrid = fopen("grid81w.data","w");
10    int gamma = 2;
11    FILE *fLog = fopen("log81w.data","w");
```

```
12
13    fprintf(fLog, "Points\t GS its\n");
14
15    double tolerance = 0.00001;
16
17    double chargeSeparation = 0.2;
18    double totalLength = 1;
19    double cellLength = totalLength / (nPoints - 1);
20    int chargeOffset = chargeSeparation / 2 / cellLength;
21    double maxDiff = 2*tolerance;
22    double diff;
23
24    double **grid;
25    double **oldGrid;
26    double **rho;
27    Allocate2dSq(nPoints, &grid);
28    Allocate2dSq(nPoints, &oldGrid);
29    Allocate2dSq(nPoints, &rho);
30
31    int x, y, i;
32    // Set up charge distribution
33    rho[nPoints / 2 + chargeOffset][nPoints / 2 ] = -1.0 / pow(cellLength,2);
34    rho[nPoints / 2 - chargeOffset][nPoints / 2 ] = 1.0 / pow(cellLength,2);
35
36    // Iterate multigrid to convergence
37    while ( maxDiff > tolerance ) {
38      Multigrid(gamma, nPoints, totalLength, grid, rho, fLog);
39
40      // Compare before and after
41      maxDiff = 0;
42      for ( x = 0 ; x < nPoints; x++ ) {
43        for ( y = 0 ; y < nPoints; y++ ) {
44          diff = fabs( grid[x][y] - oldGrid[x][y] );
45          maxDiff = maxDiff > diff ? maxDiff : diff;
46
47          oldGrid[x][y] = grid[x][y];
48        }
49      }
50    }
51
52    fclose(fLog);
53
54    // Save solution
55    int nPlotPoints = nPoints;
56    for ( x = 0 ; x < nPlotPoints ; x++ ) {
57      for (  y = 0 ; y < nPlotPoints ; y++ ) {
58        fprintf(fGrid,"%e\t",grid[x][y]);
59
60      }
61      fprintf(fGrid,"\n");
62    }
63
64    Free2dSq(nPoints, grid);
65    Free2dSq(nPoints, oldGrid);
66    Free2dSq(nPoints, rho);
67
68    printf("Done!\n");
69
70    return 0;
71  }
```

A.6 Function header: `task2/mg_func.h`

```
1  #ifndef _mg_func_h
2  #define _mg_func_h
3
4  extern void Allocate2dSq(int, double***);
5  extern void Free2dSq(int, double**);
6  extern void Multigrid(int gamma, int size, double cellLength, \
7      double** grid, double** source, FILE *fLog);
8  extern double GaussSeidel(int size, double cellLength, double** grid, \
9      double** rho, FILE* fLog);
10 extern void IncreaseGridDensity(int n, double** in, double** out);
11 extern void DecreaseGridDensity(int n, double** in, double** out);
12 extern void ComputeResidual(int n, double c, double** grid, \
13     double** rho, double** res);
14
15 #endif
```

A.7 Function code: `task2/mg_func.c`

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "mg_func.h"
#define PI 3.141592653589
#define COARSESTPOINTS 11

void Allocate2dSq(int size, double ***array)
{
  int i;
  *array = (double**) malloc(size * sizeof(double));
  for ( i = 0 ; i < size ; i++ ) {
    (*array)[i] = (double*) calloc(size, sizeof(double));
  }

  return;
}

void Free2dSq(int size, double **array)
{
  int i;
  for ( i = 0 ; i < size ; i++) {
    free(array[i]);
    array[i] = NULL;
  }
  free(array);
  array = NULL;

  return;
}


void Multigrid(int gamma, int nPoints, double totalLength, \
double** grid, double** source, FILE *fLog)
{
  int i,j,k,x,y;
  int nPresmooth = 2, nPostsmooth = 2;
  int nCoarsePoints = nPoints/2+1;
  double cellLength = totalLength / (double) (nPoints-1);
  double** residual;
  double** coarseResidual;
  double** v;
  double** fineV;
  double tolerance = 0.00001;
  double maxDiff = 2*tolerance;

  residual = (double**) malloc(nPoints * sizeof(double*));
  fineV = (double**) malloc(nPoints * sizeof(double*));
  coarseResidual = (double**) malloc(nCoarsePoints * sizeof(double*));
  v = (double**) malloc(nCoarsePoints * sizeof(double*));

  for (i = 0 ; i < nPoints ; i++ ) {
    residual[i] = (double*) calloc(nPoints, sizeof(double));
    fineV[i] = (double*) calloc(nPoints, sizeof(double));
  }
  for ( i = 0 ; i < nCoarsePoints ; i++ ) {
    coarseResidual[i] = (double*) calloc(nCoarsePoints, sizeof(double));
    v[i] = (double*) calloc(nCoarsePoints, sizeof(double));
  }

  if ( nPoints <= COARSESTPOINTS ) {
    while (maxDiff > tolerance ) {
      maxDiff = GaussSeidel(nPoints, totalLength, grid, source, fLog);
    }
    fprintf(fLog, "%d\t0\n",nPoints);
  } else {
    for( i = 0 ; i < nPresmooth ; i++ ) {
      GaussSeidel( nPoints, totalLength, grid, source, fLog);
    }

    ComputeResidual(nPoints, totalLength, grid, source, residual);

    DecreaseGridDensity(nPoints, residual, coarseResidual);

    for ( i = 0 ; i < nCoarsePoints ; i++ ) {
      for ( j = 0 ; j < nCoarsePoints ; j++ ) {
        v[i][j] = 0;
      }
    }

    fprintf(fLog,"%d\t0\n", nPoints);
    for ( k = 0 ; k < gamma ; k++) {
      Multigrid(gamma, nCoarsePoints, totalLength, v, \
      coarseResidual, fLog);
      fprintf(fLog,"%d\t0\n", nPoints);
    }
    IncreaseGridDensity(nCoarsePoints, v, fineV);

    for ( x = 0 ; x < nPoints ; x++ ) {
      for ( y =0 ; y < nPoints ; y++ ) {
        grid[x][y] = grid[x][y] + fineV[x][y];
```

```
      }
    }

    for ( i = 0 ; i < nPostsmooth ; i++ ) {
      GaussSeidel( nPoints, totalLength, grid, source, fLog);
    }
  }

  return; // grid is "returned" as an argument...
}


double GaussSeidel(int nPoints, double totalLength, double** grid, \
double** rho, FILE* fLog) {
  double maxDiff = 0;
  double cellLength = totalLength/ (double) (nPoints-1);
  double oldValue, diff;
  int x,y, count = 0;

  for ( x = 1 ; x < nPoints - 1 ; x++ ) {
    for ( y = 1 ; y < nPoints - 1 ; y++ ) {
      oldValue = grid[x][y];
      grid[x][y] = 1.0/4.0 * (grid[x+1][y] + grid[x-1][y] + grid[x][y+1] \
      + grid[x][y-1] - pow(cellLength,2)*rho[x][y]);
      diff = fabs(oldValue - grid[x][y]);
      maxDiff = diff > maxDiff ? diff : maxDiff;
      count++;
    }
  }

  fprintf(fLog, "0\t%d\n",count);

  return(maxDiff);
}

void IncreaseGridDensity(int nPoints, double** inGrid, double** outGrid) {
  int x, y;
  // the exactly matching points
  for ( x = 0 ; x < nPoints ; x++ ) {
    for ( y = 0 ; y < nPoints ; y++ ) {
      outGrid[2*x][2*y] = inGrid[x][y];
    }
  }
  // points with four nearest neighbours
  for ( x = 1 ; x < 2*nPoints-1 ; x += 2 ) {
    for ( y = 1 ; y < 2*nPoints-1 ; y += 2 ) {
      outGrid[x][y] = 1.0/4.0 * (inGrid[x/2][y/2] + inGrid[x/2+1][y/2] \
      + inGrid[x/2][y/2+1] + inGrid[x/2+1][y/2+1]);
    }
  }
  // the half points in y
  for ( x = 0 ; x < 2*nPoints-1 ; x += 2 ) {
    for ( y = 1 ; y < 2*nPoints-1 ; y += 2 ) {
      outGrid[x][y] = 1.0/2.0 * (inGrid[x/2][(y-1)/2] \
      + inGrid[x/2][(y+1)/2]);
    }
  } // and in x
  for ( x = 1 ; x < 2*nPoints-1 ; x += 2 ) {
    for ( y = 0 ; y < 2*nPoints-1 ; y += 2 ) {
      outGrid[x][y] = 1.0/2.0 * (inGrid[(x-1)/2][y/2] \
      + inGrid[(x+1)/2][y/2]);
    }
  }

  return;
}

void DecreaseGridDensity(int nPoints, double** inGrid, double** outGrid) {
  int x,y;

  for ( x = 1 ; x < nPoints/2 ; x++ ) {
    for ( y = 1 ; y < nPoints/2 ; y++ ) {
      outGrid[x][y] = 1.0/4.0 * inGrid[2*x][2*y] \
                    + 1.0/8.0 * (inGrid[2*x-1][2*y] + inGrid[2*x+1][2*y] \
                    + inGrid[2*x][2*y-1] + inGrid[2*x][2*y+1]) \
                    + 1.0/16.0 * (inGrid[2*x-1][2*y-1] \
                    + inGrid[2*x+1][2*y-1] + inGrid[2*x+1][2*y+1] \
                    + inGrid[2*x-1][2*y+1]);
    }
  }

  return;
}

void ComputeResidual(int nPoints, double totalLength, double** grid, \
                          double** rho, double** residual) {
  double oldValue, diff;
  int x,y;
  double cellLength = totalLength / (double) (nPoints-1);
  double tolerance = 0.00001;
  double maxDiff = 2*tolerance;
```

```
183
184   for ( x = 1 ; x < nPoints - 1 ; x++ ) {
185     for ( y = 1 ; y < nPoints - 1 ; y++ ) {
186       residual[x][y] = rho[x][y] \
187                      - (grid[x-1][y] - 2*grid[x][y] + grid[x+1][y]) \
188                      / pow(cellLength,2) \
189                      - (grid[x][y-1] - 2*grid[x][y] + grid[x][y+1]) \
190                      / pow(cellLength,2);
191     }
192   }
193
194   return;
195 }
```

A.8 Plotting in matlab: `task2/plot_stuff.m`

```
1  %% potential plot - v-cycle
2
3  clear all
4  clc
5  clf
6
7  cc = [ 0 1 1 ; 1 0 1 ; 0 1 0 ; 0 0 1 ; 1 0 0];
8  textStorlek = 14;
9  legendStorlek = 11;
10
11 exactData = dlmread('../phi_exact.data','\t');
12 xExactData = linspace(0,1,length(exactData));
13 plot(xExactData,exactData,'k')
14
15 hold on
16
17 for i = 1:5
18     filename = ['grid' num2str(2^(i+2)) num2str(1) 'v.data'];
19
20     data = dlmread(filename,'\t');
21     data = data(:,1:end-1);
22     data = data(:,fix(end/2)+1);
23
24     xData = linspace(0,1,length(data));
25
26     plot(xData,data,'Color',cc(i,:))
27 end
28
29 xlabel('x','FontSize',textStorlek)
30 ylabel('\Phi(x,1/2)','FontSize',textStorlek)
31
32 h = legend('Exact solution','81 V-cycle','161 V-cycle',...
33     '321 V-cycle','641 V-cycle','1281 V-cycle')
34 set(h,'FontSize',legendStorlek);
35 hold off
36
37 saveas(gcf,'task2v.png','png')
38
39 axis([0.58 0.62 0.3 1.42])
40
41 saveas(gcf,'task2v_zoom.png','png')
42
43 %% potential plot - w-cycle
44
45 clear all
46 clc
47 clf
48
49 cc = [ 0 1 1 ; 1 0 1 ; 0 1 0 ; 0 0 1 ; 1 0 0];
50 textStorlek = 14;
51 legendStorlek = 11;
52
53 exactData = dlmread('../phi_exact.data','\t');
54 xExactData = linspace(0,1,length(exactData));
55 plot(xExactData,exactData,'k')
56
57 hold on
58
59 for i = 1:5
60     filename = ['grid' num2str(2^(i+2)) num2str(1) 'w.data'];
61
62     dataV = dlmread(filename,'\t');
63     dataV = dataV(:,1:end-1);
64     dataV = dataV(:,fix(end/2)+1);
65
66     xData = linspace(0,1,length(dataV));
67
68     plot(xData,dataV,'Color',cc(i,:))
69 end
70
```

```
xlabel('x','FontSize',textStorlek)
ylabel('\Phi(x,1/2)','FontSize',textStorlek)

h = legend('Exact solution','81 W-cycle','161 W-cycle',...
    '321 W-cycle','641 W-cycle','1281 W-cycle')
set(h,'FontSize',legendStorlek);
hold off

saveas(gcf,'task2w.png','png')

axis([0.58 0.62 0.3 1.42])

saveas(gcf,'task2w_zoom.png','png')


%% depth

clear all
clc
clf

textStorlek = 14;
legendStorlek = 11;

data = dlmread('log81w.data','\t',1,0);
savename = 'task2_w_depth.png';
data = data(:,1);
data = data( data ~= 0 );

semilogy(data,'x-');
set(gca, 'YTick', [0 11 21 41 81 161 321 641 1281]);

xlabel('Step nr','FontSize',textStorlek)
ylabel('Number of points','FontSize',textStorlek)

h = legend('Number of grid points');
set(h,'FontSize',legendStorlek);


saveas(gcf,savename,'png')

%% nIterations

clear all
clc
clf

textStorlek = 14;
legendStorlek = 11;
for i = 1:5
    filename = ['log' num2str(2^(i+2)) num2str(1) 'v.data'];

    data = dlmread(filename,'\t',1,0);
    data = data(:,2);
    data = data( data ~= 0 );
    yV(i) = sum(data);

    filename = ['log' num2str(2^(i+2)) num2str(1) 'w.data'];

    data = dlmread(filename,'\t',1,0);
    data = data(:,2);
    data = data( data ~= 0 );
    yW(i) = sum(data);

    filename = ['../task3/log' num2str(2^(i+2)) num2str(1) '.data'];

    data = dlmread(filename,'\t',1,0);
    data = data(:,2);
    yFMG(i) = sum(data);

end

xData = [81 161 321 641 1281];
yBehavior = xData.^2;
yBehavior = yBehavior * yV(1)/yBehavior(1);
yBehavior = yBehavior ;


plot(xData, yV,'r')
hold on
plot(xData, yW,'b')
plot(xData, yFMG,'g')
plot(xData, yBehavior, 'k')
hold off

xlabel('Grid size','FontSize',textStorlek)
ylabel('Number of GS iterations','FontSize',textStorlek)

h = legend('V-cycle', 'W-cycle', 'Full multigrid', '(Grid size)^2');
set(h,'FontSize',legendStorlek);

```

```
saveas(gcf,'task2_its.png','png')
```

Task 3

A.9 Main program: `task3/main.c`

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "mg_func.h"

int main() {
  // Settings to change
  int nMaxPoints = 81;
  int nMinPoints = 11;
  FILE *fLog = fopen("log81.data","w");
  FILE *fGrid = fopen("grid81.data","w");
  double tolerance = 0.00001;

  int nPoints = nMinPoints;
  int nFinerPoints = 2*(nPoints - 1) + 1;

  double chargeSeparation = 0.2;
  double totalLength = 1;
  double cellLength = totalLength / (nPoints - 1);
  int chargeOffset = chargeSeparation / 2 / cellLength;

  double maxDiff;
  double diff;

  double **grid;
  double **newGrid;
  double **oldGrid;
  double **saveGrid;
  double **rho;

  int x, y;

  fprintf(fLog, "Points\tGS its\n");

  // Initialize on the coarsest grid
  Allocate2dSq(nPoints, &grid);
  Allocate2dSq(nMaxPoints, &oldGrid);
  Allocate2dSq(nMaxPoints, &saveGrid);
  Allocate2dSq(nPoints, &rho);

  // Set up source distribution
  rho[nPoints / 2 + chargeOffset][nPoints / 2 ] = -1.0 / pow(cellLength,2);
  rho[nPoints / 2 - chargeOffset][nPoints / 2 ] = 1.0 / pow(cellLength,2);

  // Iterate until our best approximation changes less than tolerance
  maxDiff = 2*tolerance;  // Just to get the loop started
  while(maxDiff > tolerance) {
    maxDiff = 0;
    nPoints = nMinPoints;
    nFinerPoints = 2*(nPoints-1)+1;

    // Start from coarsest grid, keep going to desired size
    while( nPoints <= nMaxPoints){
      // Run multigrid
      Multigrid(nPoints, totalLength, grid, rho, fLog);

      // If finest grid, save data now, before interpolating
      if ( nPoints == nMaxPoints ) {
        for ( x = 0 ; x < nPoints ; x++ ) {
          for ( y = 0 ; y < nPoints ; y++ ) {
            saveGrid[x][y] = grid[x][y];
          }
        }
      }

      // Interpolate the result to higher gridsize
      Allocate2dSq(nFinerPoints, &newGrid);
      IncreaseGridDensity(nPoints, grid, newGrid);
      Free2dSq(nPoints, grid);
      grid = newGrid;
      newGrid = NULL;

      // Free stuff we don't need anymore
      Free2dSq(nPoints,rho);

      // Update nPoints and derived numbers
      nPoints = nFinerPoints;
      nFinerPoints = 2*(nPoints-1)+1;
      cellLength = totalLength / (nPoints - 1);
```

```
      chargeOffset = chargeSeparation / 2 / cellLength;

      // Create a new rho at current grid size
      Allocate2dSq(nPoints, &rho);
      rho[nPoints / 2 + chargeOffset][nPoints / 2 ] = \
      -1.0 / pow(cellLength,2);
      rho[nPoints / 2 - chargeOffset][nPoints / 2 ] = \
      1.0 / pow(cellLength,2);
    }

    // Since we're at the finest grid, revert last change of nPoints
    nPoints = (nPoints-1)/2 + 1;

    // Compare previous solution to new one
    for(x = 0 ; x < nPoints ; x++ ) {
      for( y = 0 ; y < nPoints ; y++) {
        diff = fabs(oldGrid[x][y] - grid[x][y]);
        oldGrid[x][y] = grid[x][y];
        maxDiff = maxDiff > diff ? maxDiff : diff;
      }
    }
  }

  // Save solution
  int nPlotPoints = nMaxPoints;
  for ( x = 0 ; x < nPlotPoints ; x++ ) {
    for (  y = 0 ; y < nPlotPoints ; y++ ) {
      fprintf(fGrid,"%e\t",saveGrid[x][y]);

    }
    fprintf(fGrid,"\n");
  }

  printf("Done\n");

  return 0;
}
```

A.10 Function header: `task3/mg_func.h`

```
#ifndef _mg_func_h
#define _mg_func_h

extern void Allocate2dSq(int, double***);
extern void Free2dSq(int, double**);
extern void Multigrid(int size, double cellLength, double** grid, \
    double** source, FILE *fLog);
extern double GaussSeidel(int size, double cellLength, double** grid, \
    double** rho, FILE *fLog);
extern void IncreaseGridDensity(int n, double** in, double** out);
extern void DecreaseGridDensity(int n, double** in, double** out);
extern void ComputeResidual(int n, double c, double** grid, \
    double** rho, double** res);

#endif
```

A.11 Function code: `task3/mg_func.c`

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "mg_func.h"
#define PI 3.141592653589
#define COARSESTPOINTS 11

void Allocate2dSq(int size, double ***array)
{
  int i;
  *array = (double**) malloc(size * sizeof(double));
  for ( i = 0 ; i < size ; i++ ) {
    (*array)[i] = (double*) calloc(size, sizeof(double));
  }

  return;
}

void Free2dSq(int size, double **array)
{
  int i;
  for ( i = 0 ; i < size ; i++) {
    free(array[i]);
```

```
    array[i] = NULL;
  }
  free(array);
  array = NULL;

  return;
}


void Multigrid(int nPoints, double totalLength, double** grid, \
double** source, FILE *fLog)
{
  int i,j,k,x,y;
  int gamma = 1;
  int nPresmooth = 2, nPostsmooth = 2;
  int nCoarsePoints = nPoints/2+1;
  double cellLength = totalLength / (double) (nPoints-1);
  double** residual;
  double** coarseResidual;
  double** v;
  double** fineV;
  double tolerance = 0.00001;
  double maxDiff = 2*tolerance;

  residual = (double**) malloc(nPoints * sizeof(double*));
  fineV = (double**) malloc(nPoints * sizeof(double*));
  coarseResidual = (double**) malloc(nCoarsePoints * sizeof(double*));
  v = (double**) malloc(nCoarsePoints * sizeof(double*));

  for (i = 0 ; i < nPoints ; i++ ) {
    residual[i] = (double*) calloc(nPoints, sizeof(double));
    fineV[i] = (double*) calloc(nPoints, sizeof(double));
  }
  for ( i = 0 ; i < nCoarsePoints ; i++ ) {
    coarseResidual[i] = (double*) calloc(nCoarsePoints, sizeof(double));
    v[i] = (double*) calloc(nCoarsePoints, sizeof(double));
  }

  if ( nPoints <= COARSESTPOINTS ) {
    while (maxDiff > tolerance ) {
      maxDiff = GaussSeidel(nPoints, totalLength, grid, source, fLog);
    }
    fprintf(fLog, "%d\t0\n",nPoints);
  } else {
    for( i = 0 ; i < nPresmooth ; i++ ) {
      GaussSeidel( nPoints, totalLength, grid, source, fLog);
    }

    ComputeResidual(nPoints, totalLength, grid, source, residual);

    DecreaseGridDensity(nPoints, residual, coarseResidual);

    for ( i = 0 ; i < nCoarsePoints ; i++ ) {
      for ( j = 0 ; j < nCoarsePoints ; j++ ) {
        v[i][j] = 0;
      }
    }

    fprintf(fLog,"%d\t0\n", nPoints);
    for ( k = 0 ; k < gamma ; k++) {
      Multigrid(nCoarsePoints, totalLength, v, coarseResidual, fLog);
      fprintf(fLog,"%d\t0\n", nPoints);
    }
    IncreaseGridDensity(nCoarsePoints, v, fineV);

    for ( x = 0 ; x < nPoints ; x++ ) {
      for ( y =0 ; y < nPoints ; y++ ) {
        grid[x][y] = grid[x][y] + fineV[x][y];
      }
    }

    for ( i = 0 ; i < nPostsmooth ; i++ ) {
      GaussSeidel( nPoints, totalLength, grid, source, fLog);
    }
  }

  return; // grid is "returned" as an argument...
}


double GaussSeidel(int nPoints, double totalLength, double** grid, \
double** rho, FILE *fLog) {
  double maxDiff = 0;
  double cellLength = totalLength/ (double) (nPoints-1);
  double oldValue, diff;
  int x,y, count = 0;

  for ( x = 1 ; x < nPoints - 1 ; x++ ) {
    for ( y = 1 ; y < nPoints - 1 ; y++ ) {
      oldValue = grid[x][y];
      grid[x][y] = 1.0/4.0 * (grid[x+1][y] + grid[x-1][y] \
```

```
      + grid[x][y+1] + grid[x][y-1] - pow(cellLength,2)*rho[x][y]);
      diff = fabs(oldValue - grid[x][y]);
      maxDiff = diff > maxDiff ? diff : maxDiff;
      count++;
    }
  }

  fprintf(fLog, "0\t%d\n", count);

  return(maxDiff);
}

void IncreaseGridDensity(int nPoints, double** inGrid, double** outGrid) {
  int x, y;
  // the exactly matching points
  for ( x = 0 ; x < nPoints ; x++ ) {
    for ( y = 0 ; y < nPoints ; y++ ) {
      outGrid[2*x][2*y] = inGrid[x][y];
    }
  }
  // points with four nearest neighbours
  for ( x = 1 ; x < 2*nPoints-1 ; x += 2 ) {
    for ( y = 1 ; y < 2*nPoints-1 ; y += 2 ) {
      outGrid[x][y] = 1.0/4.0 * (inGrid[x/2][y/2] + inGrid[x/2+1][y/2] \
      + inGrid[x/2][y/2+1] + inGrid[x/2+1][y/2+1]);
    }
  }
  // the half points in y
  for ( x = 0 ; x < 2*nPoints-1 ; x += 2 ) {
    for ( y = 1 ; y < 2*nPoints-1 ; y += 2 ) {
      outGrid[x][y] = 1.0/2.0 * (inGrid[x/2][(y-1)/2] \
      + inGrid[x/2][(y+1)/2]);
    }
  } // and in x
  for ( x = 1 ; x < 2*nPoints-1 ; x += 2 ) {
    for ( y = 0 ; y < 2*nPoints-1 ; y += 2 ) {
      outGrid[x][y] = 1.0/2.0 * (inGrid[(x-1)/2][y/2] \
      + inGrid[(x+1)/2][y/2]);
    }
  }

  return;
}

void DecreaseGridDensity(int nPoints, double** inGrid, double** outGrid) {
  int x,y;

  for ( x = 1 ; x < nPoints/2 ; x++ ) {
    for ( y = 1 ; y < nPoints/2 ; y++ ) {
      outGrid[x][y] = 1.0/4.0 * inGrid[2*x][2*y] \
                    + 1.0/8.0 * (inGrid[2*x-1][2*y] + inGrid[2*x+1][2*y] \
                    + inGrid[2*x][2*y-1] + inGrid[2*x][2*y+1]) \
                    + 1.0/16.0 * (inGrid[2*x-1][2*y-1] \
                    + inGrid[2*x+1][2*y-1] + inGrid[2*x+1][2*y+1] \
                    + inGrid[2*x-1][2*y+1]);
    }
  }

  return;
}

void ComputeResidual(int nPoints, double totalLength, double** grid, \
                         double** rho, double** residual) {
  double oldValue, diff;
  int x,y;
  double cellLength = totalLength / (double) (nPoints-1);
  double tolerance = 0.00001;
  double maxDiff = 2*tolerance;

  for ( x = 1 ; x < nPoints - 1 ; x++ ) {
    for ( y = 1 ; y < nPoints - 1 ; y++ ) {
      residual[x][y] = rho[x][y] \
                     - (grid[x-1][y] - 2*grid[x][y] + grid[x+1][y]) \
                     / pow(cellLength,2) \
                     - (grid[x][y-1] - 2*grid[x][y] + grid[x][y+1]) \
                     / pow(cellLength,2);
    }
  }

  return;
}
```

A.12 Plotting in matlab: `task3/plot_stuff.m`

```
%% potential plot

```

```
clear all
clc
clf

cc = [ 0 1 1 ; 1 0 1 ; 0 1 0 ; 0 0 1 ; 1 0 0];
textStorlek = 14;
legendStorlek = 11;

exactData = dlmread('../phi_exact.data','\t');
xExactData = linspace(0,1,length(exactData));
plot(xExactData,exactData,'k')

hold on

for i = 1:5
    filename = ['grid' num2str(2^(i+2)) num2str(1) '.data'];

    data = dlmread(filename,'\t');
    data = data(:,1:end-1);
    data = data(:,fix(end/2)+1);

    xData = linspace(0,1,length(data));

    plot(xData,data,'Color',cc(i,:))

end

xlabel('x','FontSize',textStorlek)
ylabel('\Phi(x,1/2)','FontSize',textStorlek)

h = legend('Exact solution','81','161','321','641','1281');
set(h,'FontSize',legendStorlek);
hold off

saveas(gcf,'task3.png','png')

axis([0.58 0.62 0.3 1.42])

saveas(gcf,'task3_zoom.png','png')


%% depth

clear all
clc
clf

textStorlek = 14;
legendStorlek = 11;

data = dlmread('log321.data','\t',1,0);
data = data(:,1);
data = data( data ~= 0 );

semilogy(data,'x-')
set(gca, 'YTick', [0 11 21 41 81 161 321 641 1281]);

xlabel('Iterations','FontSize',textStorlek)
ylabel('Number of points','FontSize',textStorlek)

h = legend('Number of grid points');
set(h,'FontSize',legendStorlek);

saveas(gcf,'task3_depth.png','png')

%% nIterations

for i = 1:5
    filename = ['log' num2str(2^(i+2)) num2str(1) '.data'];

    data = dlmread(filename,'\t',1,0);
    data = data(:,2);
    y(i) = sum(data);

    %xData = linspace(0,1,length(data));

    %plot(xData,data,'Color',cc(i,:))

end

plot(y)
```