FMNN15: Geometric Multigrid 2D Wave Equation

Ludwig Jacobsson, Jonatan Ferm

March 2014

1 Introduction

The goal of this project is to solve the wave equation in 2D, using a geometric multigrid method. For the sake of curiosity and interest our choice of programming language is C. The language C is also known for its speed due to its low level characteristics, this makes it suitable for such heavy computations as multigrid-algorithms.

2 Theory

Bellow is the wave equation solved in this project:

$u_{tt} = c^2 \Delta u$

This equation is rewritten as two coupled first order equations and for simplicity $c = 1$.

$u_t = v$

$v_t = u_{xx} + u_{yy}$

The laplace operator is discretized assuming ($\Delta x^2 = \Delta y^2$) using the standard five-point formula.

$$\Delta u = u_{xx} + u_{yy} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{u^n_{i+1,j} + u^n_{i-1,j} - 4u^n_{i,j} + u^n_{i,j-1} + u^n_{i,j+1}}{\Delta x^2}$$

To overcome the CFL conditions from explicit timestepping methods the implicit midpoint rule is used. This leads to the following equation:

$$\left(I - \frac{\Delta t^2}{4} T\right) \mathbf{v}^{n+1} = \Delta t T \mathbf{u^n} + \left(I + \frac{\Delta t^2}{4} T\right) \mathbf{v^n}$$

T is the 5 point finite difference matrix.

The equation will be solved for $\mathbf{v^{n+1}}$ and $\mathbf{u^{n+1}}$ will then be computed as follows:

$\mathbf{u^{n+1}} = \mathbf{u^n} + \Delta t(\mathbf{v^n} + v^{n+1})/2$

3 Method

Numerically solving differential equations to a high accuracy on large grids requires many computations. The achievable accuracy depends on the number of internal grid points on the grid.
If we denote the internal grid points by N, the number of equations to be solved in 1D is $N * N$. 2D leads to $N^2 * N^2$ and 3D $N^3 * N^3$. A typical problem is in the range of $N > 2^7$, as we can see this leads to an incredible amount of computations. In order to successfully computing this we need a way to reduce the number of computations and still arrive at an answer. One approach would be to reduce the number of internal grid points until we arrive at a grid size where we can solve the problem using a standard linear equations solver and then interpolate the solution to a finer grid until we arrive at the original grid. This is roughly the outline of the multigrid-method with the exception that the calculations are done on the residuals of the problem. The multigrid-method implemented in the project is called the Full Multigrid V-cycle method (FMGV).

3.1 FMGV

A FMGV scheme starts with an initial guess $\mathbf{v}$, the cycle is then recursively continued as follows:

```
FMGV(v, Tdx, rf){
    if (N<32)
        ec = solve(rf, Tdx)     // solve: Tdx*ec = rf
    else
        rf = Tdx*v - f;         // Compute residual
        v = v - omega*rf/Ddx;   // Pre-smoothing Jacobi
        rf = Tdx*v - f;         // Compute residual
        rf = lowpass(rf);       // Optional LP filter
        rc = FMGrestrict(rf);   // Restrict to coarse
        ec = FMGV(0,T2dx,rc);   // Solve error equation,
                                // recursive call
        ef = FMGprolong(ec);    // Prolong to finer grid
        v = v - ef;             // Correct: remove error
        rf = Tdx*v - f;         // Compute residual
        v = v - omega*rf/Ddx;   // Post-smoothing Jacobi
}
```

In each recursive call the residual is computed, the solution smoothed, the residual lowpass filtered and the grid size reduced. When the roughest grid is reached the error equation is solved with a standard solver, in our case an iterative conjugate gradient method.

4 Algorithm

For our C algorithm the memory allocation was one of the major issues to consider. Since the algorithm is recursive and memory allocation calls are expensive we needed a well planed layout. We ended up using a region-based memory management where we allocated all memory we would need for the recursive calls in the initiation of the grid. In the recursive call structure all we do is move the pointers to the new correct location each time we recur the algorithm. Using this we managed to keep the allocated memory to a minimum of $3 * N^2$ where N is the size of the finest grid, see *figure 1*.

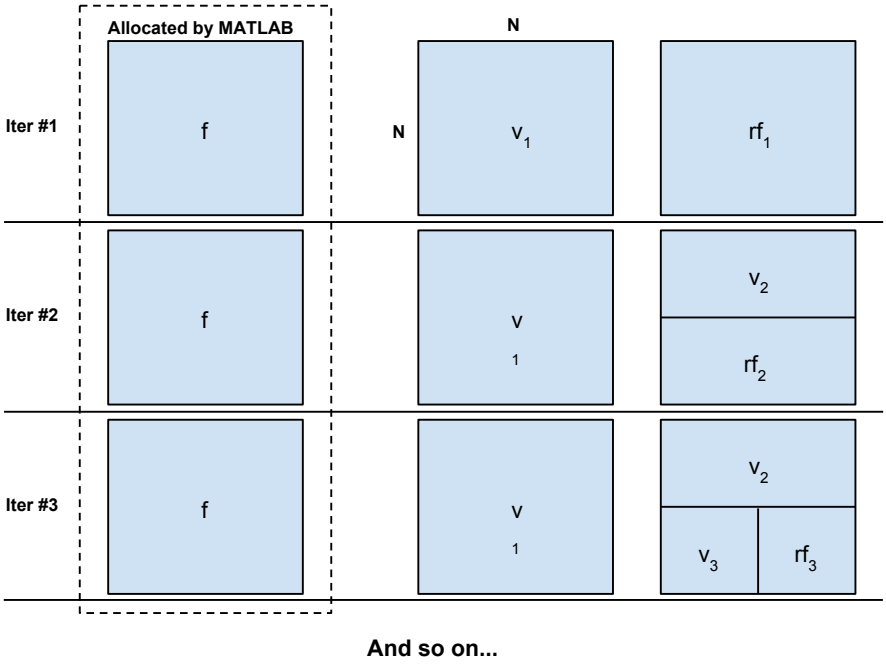

Figure 1: Memory layout for recursion 1-3.

4.1 Linear solver

To solve the linear equation for the smallest grid we needed to implement a solver. Since the T matrix is symmetric and positive definite the iterative conjugate gradient method proved to be a good choice. Since the solver is iterative and the problem to be solved only needed to be an approximation we could stop when we considered the solution to be good enough.

5 Results

The results for a gaussian peak is plotted in *figure 2*. For timestepping our algorithm runs smoothely until initial grid size is $n = 2^8 - 1$. For a single calculation we solved for a grid of size $n = 2^12 - 1$ in about 8 seconds, but the actual upper limit is highly dependent on the computer architecture.

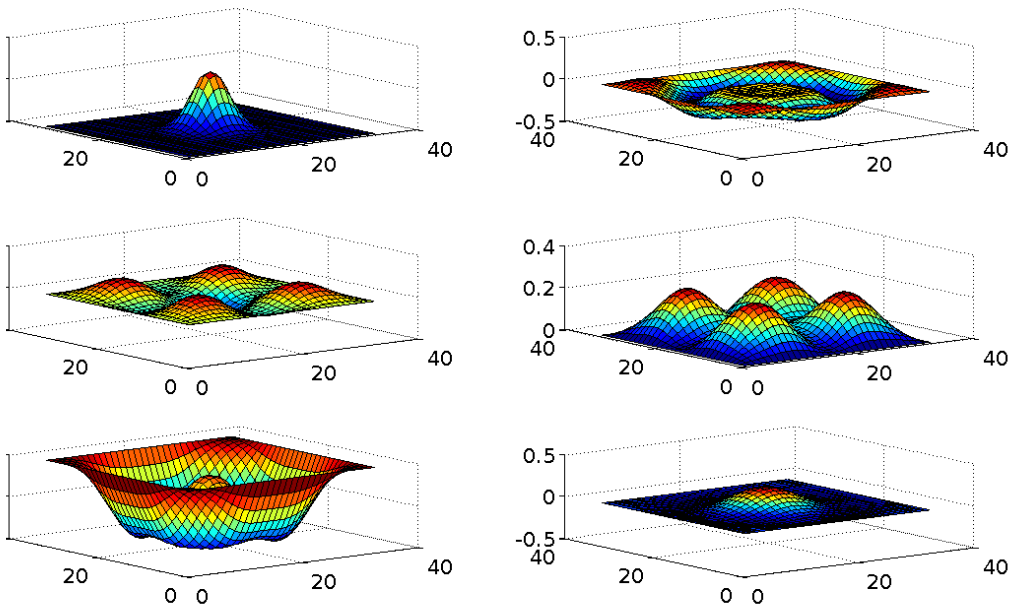

Figure 2: Result for t = 0, 0.5, 1.0, 1.5, 2.0, 2.5 s

5.1 Performace

When profiling the program it is no surprise that the convolution takes up half of the execution time. The function is called often and has a massive task each time.

```
 Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 48.14      3.88     3.88      102     0.04     0.04  conv2
 15.38      5.12     1.24        1     1.24     4.85  FMGV
 11.04      6.01     0.89                             main
  7.20      6.59     0.58        7     0.08     0.11  prolong_init
  5.40      7.03     0.43        7     0.06     0.13  prolong_grid_2d
  3.60      7.32     0.29        7     0.04     0.04  vecsub2
  3.54      7.60     0.28 16752790     0.00     0.00  lthmap
  2.11      7.77     0.17 50370121     0.00     0.00  eget
  1.36      7.88     0.11        6     0.02     0.06  postreq
  1.24      7.98     0.10        6     0.02     0.05  prereq
  0.74      8.04     0.06        7     0.01     0.06  lp_rest2
  0.12      8.05     0.01    64856     0.00     0.00  bcmap
  0.12      8.06     0.01        2     0.01     1.53  conjgrad
  0.00      8.06     0.00        1     0.00     3.31  FMGVh
  0.00      8.06     0.00        1     0.00     2.26  guess_g
  0.00      8.06     0.00        1     0.00     0.00  initiate_grid
```

Figure 3: Profiling for the execution of the algorithm doing a computation on an initial grid of size $N = 2^12 - 1$. See *appendix* for elucidation.

6 Discussion

The multigrid-method is very successful in solving these huge problems fast and accurately. The speed of the calculation is important when performing time-stepping.
A comparison with a Fourier based convolution algorithm would be interesting. Although since our convolution kernel is relatively small $3x3$ there would most likely not be any improvement. Current number of calculations are $O((3*3)*n^2)$, whereas the transformed problem would be $O(n^2)$ with an additional $O(nlog(n))$ for the Fast Fourier transform itself.

7 Appendix

Appendix 1. Program instructions

1. Compile c program in MATLAB using the follwing command.

   ```
   mex mxMG_2D.c
   ```

2. Run `mg_2d_main.m` in MATLAB.

Appendix 2. C-program header file

```c
#ifndef mg_2d
#define mg_2d

/* TYPEDEFS STRUCTS AND MACROS */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>


/* In order to be able to change precision for the entire
 * computation we used a typedef data_t.
 *
 * pow_t is used instead of int to make sure there wont be a datatype
 * overflow in any of the loops.
 */

typedef double data_t;
typedef unsigned long pow_t;

/* Struct containing the entire problem. */
typedef struct{
    pow_t k;         //(2^k-1)^2 grid points for the finest grid
    pow_t c;         // current grid
    pow_t n;         //current grid as 2^c-1
    pow_t r;         //roughest grid
    pow_t sum;         // Size of allocated memory, used in debugging
    data_t* f;         // Left-hand side
    data_t* bc0;     // beginning conditions,
    data_t* bc1;     // beginning conditions,
    data_t* bc2;     // beginning conditions,
    data_t* bc3;     // beginning conditions,
    data_t* g;       // the u grid
    data_t* rf;      // the Residual grid
```

```
35
36  #if 0
37          ----bc0----
38          |          |
39          b          b
40          c          c
41          1          2
42          |          |
43          ----bc3----
44  #endif
45  } grid_t;
46
47  /* ---------------------- MAIN-FUNCTIONS  ---------------------- */
48
49  /* Recursive function */
50  void FMGVh(grid_t* g, data_t gamma);
51  /* Makes first pre-smooting and calls the recursive function. */
52  void FMGV(grid_t* g, data_t gamma);
53
54  /*  ------------------------- DRIVERS ------------------------- */
55
56  /* Allocates memory for the problem and sets pointers to correct
57   * initial positions.
58   */
59  grid_t* initiate_grid(long k, long r);
60
61  /* First guess of v
62   */
63  void guess_g(grid_t* g, data_t gamma);
64
65  /* Makes the calculations before the recursive call:
66      rf = Tdx*v - f;          // Compute residual
67      v = v - omega*rf/Ddx;    // Pre-smoothing Jacobi
68      rf = Tdx*v - f;            // Compute residual
69   */
70  void prereq(grid_t* g, data_t gamma);
71
72  /* Makes the calculations after the recursive call
73      v = v - ef;                // Correct: remove error
74      rf = Tdx*v - f;            // Compute residual
75      v = v - omega*rf/Ddx;    // Post-smoothing Jacobi
76   */
77  void postreq(grid_t* g, data_t gamma);
78
79  /*  ----------------------- ROUTINES  ----------------------- */
80
```

```
81  /* 2D convolution with the following kernel layout:
82          k3 k1 k3
83          k1 k2 k1
84          k3 k1 k3
85   */
86  void conv2(grid_t* g, data_t* pt_org, data_t* pt_dest, data_t k1, data_t k2, data_t k3);
87
88  /* Prolong grid - used in inital guess of v.
89   */
90  void prolong_init(grid_t* g);
91
92  /* Iterative conjugate-gradient method for solving
93   * the linear system on the smallest grid.
94   */
95  void conjgrad(grid_t* g, data_t gamma);
96
97  /* Lowpass and restrict grid
98   */
99  void lp_rest2(grid_t* g);
100
101 /* Prolong grid
102  */
103 void prolong_grid_2d(grid_t* g);
104
105 /*  ---------------------- SUBROUTINES  ---------------------- */
106
107 /* Returns the index of an element on a finer grid.
108 */
109 pow_t lthmap(pow_t k, pow_t n, pow_t N);
110
111 /* Get element (x, y) from a.
112  */
113 data_t* eget(data_t* a, pow_t c, pow_t x, pow_t y);
114
115 pow_t bcmap(pow_t i, pow_t d);
116
117 void vecsub2(data_t* a, data_t* b, pow_t c);
118
119 void vecadd(data_t* a, data_t* b, pow_t c);
120
121 /*  ---------------------- DEBUGGING ---------------------- */
122
123 void print_2d(grid_t* g);
124
125 void print_all(grid_t* g);
126 #endif
```

Appendix 3. Profiling description

```
 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank. Recursive
           calls are not seen.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.
```