# 15-859E Assignment 2: Multigrid

Franklin Chen

October 21, 1998

## Contents

## List of Figures

## 1 Equation

I used the *Poisson* equation in 2d:

$$\nabla^2 \mathbf{u} = \mathbf{f}$$

This was suggested in the assignment, and was also discussed in Briggs.

### 1.1 Source

I tried some sample $\mathbf{f}$ for the equation. One was $\mathbf{f} = \mathbf{0}$ (Laplace). I settled on another constant, just for the heck of it.

I did not provide a user interface for plugging in various f, though that would be easy, e.g., parse some stream of numbers from a file.

### 1.2 Boundary values

The *Dirichlet boundary values* I chose were to take the 2d unit square as the boundaries, and have all sides be 0 except a portion of sine wave on one side, in order to get a simple but not completely trivial solution surface.

## 2 Implementation language

I implemented everything in *Objective Caml* (O'Caml), a dialect of ML. I hadn't done much with this dialect, but figured now was as good a time as any to experiment with it, instead of using *Standard ML*.

Since O'Caml supports arrays of unboxed floating point numbers, this gave a fighting chance of not being too inefficient, though I suspect that I could write C or C++ code that does significantly better.

Because I lacked time, I did not experiment with using a language with high level support of arrays, such as ZPL, SAC, or FiSH. I am interested in trying this at some point.

Note: I use arrays of arrays for matrices. This obviously misses out on the memory locality and opportunities for good caching and loop unrolling possible when using a block of contiguous memory as a matrix (as the `svl` library in C++ attempts to do, as does the SML/NJ library for Standard ML).

My first implementation was written purely functionally, with no side effects in the computation, e.g., arrays were never modified. I reimplemented it to use mutation, and the speedup wasn't all that large, actually. I stuck to the mutative version though.

My primary source of equations was Briggs. I also looked at *Numerical Recipes in C* earlier, did not consult it while in the process of writing my own, since I first wrote all my code purely functionally, then converted it in steps to use mutation.

## 3 Platform

I developed my program on my home PC, a Pentium 200 with 32 MB RAM running Red Hat Linux 5.1. I did some runs on my office PC, a Pentium II 400 with 128 MB RAM, and performance appeared to be around four times greater.

# 4 Algorithms

## 4.1 Relaxation

I implemented both *Gauss-Seidel* (red-black) and *Jacobi*, both of them with optional weighting ($\omega$) for overrelaxation.

I made no effort to make these crucial steps fast. A lot of interesting cache-related optimizations could be performed, but I could only do that if I were using something as low level as C.

## 4.2 Interpolation

I used 2d linear interpolation, as suggested by Briggs.

## 4.3 Restriction

I used full-weighted 2d restriction, as suggested by Briggs.

## 4.4 Multigrid

I implemented the $\mu$-cycle generalization of the V-cycle, with fixed numbers of iterations, specified for the duration of a run from the command line.

## 5 Command line arguments

Many parameters can be set for the duration of a run by means of command line arguments.

```
Usage:
  -escape Escape from algorithm when relative residual small?
        Default = false
  -debug Debugging level
        Default = 0
        0 = no debug
        1 = output initial and final states
        2 = output after every relaxation step
        3 = output after each read and black step of Gauss-Seidel
        4 = output full multigrid residual and its restriction
  -nu0 Number of mu-cycles for each phase of full multigrid
        Default = 1
  -nu1 Number of pre-cycle relaxation steps in mu-cycle
        Default = 1
  -nu2 Number of post-cycle relaxation steps in mu-cycle
        Default = 1
  -mu Number of recursive calls in a mu-cycle
        Default = 1
  -coarse Size of coarsest grid base case (2^k+1)
        Default = 3
  -size Size of finest grid base case (2^k+1)
        Default = 9
  -omega Weight for relaxation method (in (0, 1])
        Default = 0.666666666667
  -relax Relaxation method (gs or jacobi)
        Default = gs
        gs = Gauss-Seidel
        jacobi = Jacobi
  -multigrid Overall method
        Default = full
        full = full multigrid
        mucycle = mu-cycle
        gs = Gauss-Seidel
        jacobi = Jacobi

  -iterations Number of iterations of relaxation
        Default = 1000
        (Only applicable to -multigrid gs|jacobi)
```

## 6 Timing, accuracy, speed

Roughly, I was able to get full multigrid running to the desired accuracy suggested (relative residual $< 10^{-6}$) on a $256 \times 256$ grid in about 9 seconds on my home PC, and about 2 seconds on my office PC.

Attempts to get fine-grained data for algorithms other than full multigrid were not very successful, because for larger $n$ things would take forever, and furthermore, not achieve good accuracy.

I generated a bunch of runs by trying to guess parameters. I tried to be more scientific by adding code to detect when the relative residual was small enough, but this expensive computation slowed things down significantly. I didn't try to just periodically do the check instead.

## 7 Visual presentations

### 7.1 Plot

In Figure 1, we show the results for

- Weighted red-black Gauss-Seidel, 1300 iterations (note last two grid sizes did not complete in a reasonable time).
- $\mu$-cycle, actually just a V-cycle, with 500 iterations pre and post.
- Full multigrid, with parameters 4, 2, 2.

### 7.2 Picture

I didn't have time to do anything fancy with graphics, an area I don't really know much about (but wish to). My program will generate a simplistic VRML file, however, treating the solution field as an `ElevationGrid`. An example is in `dataset.data` (see `Makefile` for options used).

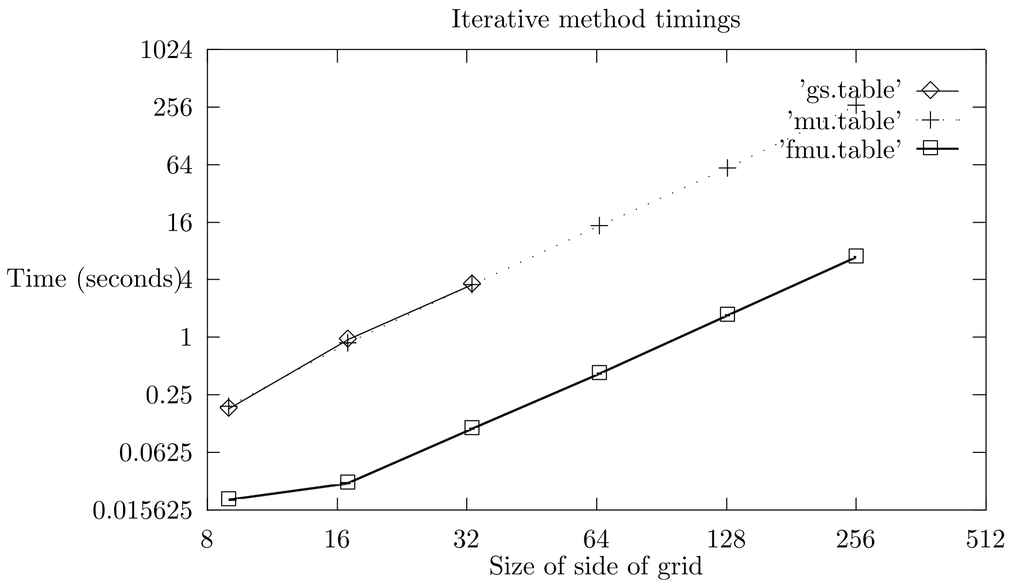

Figure 1: Iterative method performances