Contents

List of Figures

1 Introduction

Boundary Value problems history and motivation

Types of boundaries

2 Methods

Ultimately, the heavy lifting in solving a potential problem involves solving a partial differential equation – in this case, Poisson's equation. Here we will be considering the 2-dimensional Poisson equation in Euclidean space,

$$\nabla^2 \phi = f,$$

where f is a real-valued function inside the space and ϕ is the steady-state solution for the potential in the space. However, since this solution will be calcuated numerically on a computer, we must discretize the equation so that it can be used to represent a solution on a 2-dimensional grid.

2.1 Discreetizing Poisson's Equation

Consider a grid of size n by n[1], with values represented by $u_{i,j}$. We want to apply the Poisson equation to this u as follows:

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial x^2} = f,$$

however, u is not continuous, so the partial derivative for x becomes

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2},$$

at a grid point (i, j), where $h = 1/n$ is the width and height of a grid cell. The partial derivative for y is similar, using j instead of i. This gives the full discrete Poisson equation as,

$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} = f_{i,j}.$$

Since $u_{i,j}$ is what we are interested in, this is more useful after some rearranging:

$$h^2 f_{i,j} = -4u_{i,j} + u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1},$$

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

It can be seen from the top equation that this is now a linear algebra problem, requiring a method of solving n^2 dependent linear equations.

[1] In general, it is not difficult to allow a rectangular grid, but for simplicity we will require the grid to be square.

2.2 Iterative Methods

Since we are using a computer, an iterative method is feasible to be used as a numerical solution method. In this program, I have implemented two methods for solving these problems: Jacobi iteration, and successive over-relaxation (SOR).

2.2.1 Jacobi Iteration

This method is perhaps the simplest method of doing an iterative solution to this linear algebra problem. The procedure involves treating each equation as independent, and solving iteratively. This turns Equation 1 into

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

where k is the iteration number. This is saying that to calculate u everywhere, we must start with a guess for u, and then iteratively solve each element in u, which depends on its four nearest neighbors. One immediate unfortunate reality with Jacobi iteration presents itself: we must store a secondary copy of the entire grid because each calculation of $u_{i,j}^{k+1}$ strictly requires values from the previous iteration. Since we iterate over the grid in order to solve it, we cannot write the new value for the $k+1$ iteration into that element of u, since we would be overwriting a value that is needed later. This unforunately but necessarily increases the space required by a factor of n^2.

The above equation does not yet include any consideration of boundary conditions. In fact, this is only valid for a region that is unbounded. The primary functionality of supporting solving statics problems requires support for dealing with Dirichlet boundary conditions. Dirichlet boundary conditions are defined such that the solution ϕ of a partial differential equations in a region Ω has a value $\phi(\mathbf{r}) = y(\mathbf{r})\ \forall \mathbf{r} \in \partial\Omega$. For this case, this means that if a Dirichlet boundary value condition is specified for a particular grid cell (x, y), then its value is simply fixed to the value of the Dirichlet condition, and that cell does not need to be recalculated on successive iterations.

Another consideration is that the grid is of finite size. This means that when calculating $u_{i=0,j}$, for example, we cannot incorporate $ui-1, j$ as Equation 2 would dictate. This is because that would try to get the -1st row the the matrix u, which does not exist. A common method of dealing with this problem is "wrap around " to the other side – essentially considering the region Ω to the surface of a sphere, or to consider the space to be periodic. Because of the nature of the problems that I will be considering, I have chosen to consider an implicit Dirichlet boundary condition of zero everywhere outside of Ω, thus effectively limiting the calculation to be inside the region while clamping any values "outside" of Ω to zero. This is easy to implement and visualize as well: any value of $u_{i,j}$ for $i >= n$, $i < 0$, $j >= n$, or $j < 0$ is simply zero, always.

2.2.2 Successive Over-Relaxation

An alternate approach to iteratively solving this problem numerically is called successive over-relaxation (or SOR), and it has advantages over simple Jacobi iteration. It works by reusing previous calculations for the next iteration when calculating a given cell. It transforms Equation 2 into,

$$u_{i,j}^{k+1} = (1-\omega) + \frac{\omega}{4}\left(u_{i-1,j}^{k+1} + u_{i+1,j}^{k} + u_{i,j-1}^{k+1} + u_{i,j+1}^{k} - h^2 f_{i,j}\right), \tag{3}$$

where ω is a tunable parameter called the *over-relaxtion parameter*. For a square lattice, a value of

$$\omega = \frac{2}{1+\frac{\pi}{n}}$$

has the best rate of convergence. There are two terms on the right-hand-side of equation 3 which refer to the $k+1$ iteration. What this is doing is using the new values for the cells that have already been updated on this iteration in order to speed up the convergence.

TODO Determining when to halt

2.3 Design of Solver

The solving program is written in C in order to leverage the full performance of modern computers and is implemented as a shared library (on UNIX) such that it can be linked with another program which sets up the initial conditions before calling the solving function. The C code that does the solving is hereby refered to as the engine, while any code that interfaces with the engine and provides a usable user interface is referred to as the frontend.

I have written an example frontend in Python, which is capable to parsing a configuration file which describes the values of f, the initial boundary conditions, the grid size, and a number of other details about a given desired setup (hereby collectively reffered to as a configuration). I have written several example configurations which I have used to confirm the correctness of the solving program. For a more detailed technical description of the implementation, see Appendix C.

2.4 Configurations and Mapping to Electrostatics

TODO also talk about how the configuration is written (include advance features like circles)

2.5 Verifying Correctness

The clearest way to verify that the solver is working correctly is to test it out on a number of different configurations for which the correct solution is known. My first choice here was a simple electrostatics problem straight out of a textbook:

> Consider a square region with the bottom left corner at the origin with each side of length L. Let the potential along each side of the region be fixed at 0, except for the bottom side, which has a potential $V(x) = V_0 \sin(x\pi/L)$. What is the potential V inside the region?

This problem describes solving Laplace's equation (which is Poisson's equation with $f = 0$), given some initial boundary conditions. It can be solved analytically using a standard separation of variables trick. Assume

$$\phi = X(x)Y(y),$$

then

$$X''(x)/X(x) = -Y''(y)/Y(y) = -\lambda,$$

and by applying the boundary conditions, one comes to the solution

$$V(x, y) = \frac{\sinh(y\pi/L)}{\sinh(\pi/L)} \sin(x\pi/L).$$

In order to compare the result of my program with the analytical solution, I added a feature to the Python frontend to accept a pre-made file containing a precalculated grid of the correct output by using the analytic solution. The frontend then reads this file and compares it to the result of the solver.

2.6 Potential From Electric Dipole

Another correctness check is a simulation of an electric dipole. The potential from an electric dipole is well understood. If two equal and opposite charges are separated on an axis by distance d, then at a point p far away, the potential is,

$$V = C\left(\frac{1}{r_0} - \frac{1}{r_1}\right),$$

where r_0 and r_1 are the distances from each charge, and C is a constant. With some rearranging,

$$V = C\left(\frac{r_0 - r_1}{r_0 r_1}\right) = \frac{Cd\cos(\theta)}{r^2},$$

where r is the distance from the center of charge to the point, and θ is the angle that the line from the center of charge forms with the axis that the dipole is on. The potential from

an electric dipole falls off as $1/r^2$, which should be reflected in the solution given by the program if it is given a configuration describing a dipole.

If the solving program is correct, then if two charges are placed near each other, the numerically calculated potential should display this $1/r^2$ behavior. I wrote such a configuration and had the frontend output the potential at points increasing in distance from the center of the dipole, allowing me to fit the data to a $1/r^2$ curve.

Correctness Checks - Solution to sin(x) along wall potential – non-discreet. - Electric dipole solution for far away

Calculating correctness automatically

2.7 Optimization and Multithreading

The purpose of this program is to not only be correct, but to be correct quickly. In order to achieve this, I have tried three things in combination:

1. General program optimization, targeted towards 64-bit x86 machines
2. The use of the x86 SIMD instruction set
3. The use of multithreading to split up the work among CPU cores

The first of these is pretty basic. The main part of the solver is written in C so that it can run directly on the processor. When the program is compiled, I instruct the compiler to optimize the program to the best of its ability, and the program is written in such a way that it can be optimized well and does not waste time doing unneccesary work.

As an example of the last point, consider a 2-dimensional grid (2-D array), which must be iterated over (much like this program does). There are two ways to do this: either iterate row-by-row, or column-by-column. The end result is the same, and one might think the choice is arbitrary. However, when I tested the two different approaches, I found the row-by-row approach to be around 5 times faster. This is because of caching effects and the way that the CPU fetches data from RAM. For a more detailed description of this, see Appendix B.

Another example involves data structure organization. Say a program needs to store an array of objects, and each object has data associated with it (for example, current value, previous value, error, initial state, etc). One might be tempted to write something like in Listing 1. The problem here is that if each object's CUR_VAL are all calculated together (as is often the case, and is definitely the case for this solving program), then the location of the values that need to be loaded in succession by the processor are far apart. This, again, comes down to caching effects which are further explained in Appendix B.

```
struct {
  double cur_val, prev_val;
  float err;
  int initial_state;
} objects[N];
```

Listing 1: Array of structures organization.

Fortunately, this also has an easy to make change that drastically improves performance: change to structure of arrays organization, thus reorganizing the memory layout so that related and commonly accessed-together values appear adjacent in memory. This is shown in listing 2. While it may be a less intuitive way to describe and program a simulation, the result is significantly faster code the majority of the time.

```
struct {
  double cur_val[N];
  double prev_val[N];
  float err[N];
  int initial_state[N];
} objects;
```

Listing 2: Structure of arrays organization.

2.8 SIMD Instructions

I have also made use of the SIMD (Single Instruction Multiple Data) instruction set. These instructions are available on modern 64-bit x86 processors[2]. Their purpose is to enable whats known as vector processing of data on x86 machines. This provides the ability to apply a mathematical operation to one or more arrays, and have that operation by applied to each element. For example, I can have a 128-bit register filled with 4 values, $x = (1, 2, 3, 4)$, and other register like it, $y = (10, 10, 11, 11)$. I can then add them together to get $(11, 12, 14, 15)$, and this addition is issued with a single instruction. Furthermore, this takes the same amount of time as a single addition, because the processor can do the 4 separate addition operations in parallel.

I have written in the use of SIMD instructions in the solver program in order to both reduce branches and to parallelize the calculations required when calculating the next iteration of a given cell. I provided the ability to disable this functionality and instead do a non-SIMD style calculation in order to get benchmarks. A requirement of the SIMD code was that it give exactly the same result as the non-SIMD version (otherwise the implementation of the

[2]More instructions are added in most new releases of CPUs. There are different SIMD instruction sets available on different processor models. I have used the sse through avx instruction sets in this code.

math would be incorrect). For this reason, when talking about the results of a simulation, it does not matter if SIMD was used or not; the use of SIMD only affects performance.

An important note is that when fully optimizing code for a machine which it knows about, `gcc` may choose to output SIMD instructions in order to do computation even if the user does not do so explicitly. For this reason, in places where I refer to the mode of the simulation as being "non-SIMD", what I mean is that it does not use my manual SIMD instructions but it may still use compiler-generated SIMD instructions.

2.9 Multithreading

Finally, I have added multithreading support to the solver. This is done by spitting up the grid into subregions which are contiguous in memory, and spawning N threads via the pthreads API. Each thread is then given a region to work on, and they coordinate through shared state in order to decide when the solution has been reached. The initial implementation showed extremely poor results with multithreading (often being drastically slowed than the single threaded version) due to the use of atomic shared variables. It was then re-written such that the code was unsafe but much faster. This is because each thread modifies a shared non-atomic variable that keeps track of the current total change per iteration after that thread itself completes a number of iterations over its region. This kind of shared memory access is much faster than coordinating the threads, but it is also technically undefined behavior in C. Fortunately, on x86, it is actually safe, however there is potential information loss because a thread could overwrite another thread if they are trying to update the value at the same time. A more detailed discussion of the code and why it is unsafe can be found in Appendix B.

2.10 Benchmarking Performance

Benchmarks were done by placing two calls to `gettimeofday(3)` around the core of the solver. This function has a granularity of 1 microsecond CITE ME. This code returns both the number of iterations done and the difference between the two gettimeofday calls, which are then used to calculate the iterations per second. In order to take into account error from unpredicatble scheduling effect from the operating system, each benchmark was run 300 times, and the standard deviation and mean were calculated, along with 95% confidence intervals. Several different computers were used: computer A was an Intel Core i5-3570K with 4 cores running at 3.4GHz with 16GB of RAM. Computer B was an Intel Xeon E5620 with 4 cores with 2 threads each, running at 2.4GHz, with 24GB of RAM. Computer B did not support all of the SIMD instructions that I used, so it was only able to benchmark the non-SIMD version of the code.

3 Results

All results describing the correctness of the simulation or discussing the closeness of the simulation to an analytical result are done using the single-threaded mode. This is because this makes the result of the simulation deterministic, and so it does not matter which machine the test was run on. For results that measure performance, the machine and options for the simulation used will be specified. There are times when the number of threads affected the correctness of the result.

3.1 The $\sin(x)$ Potential

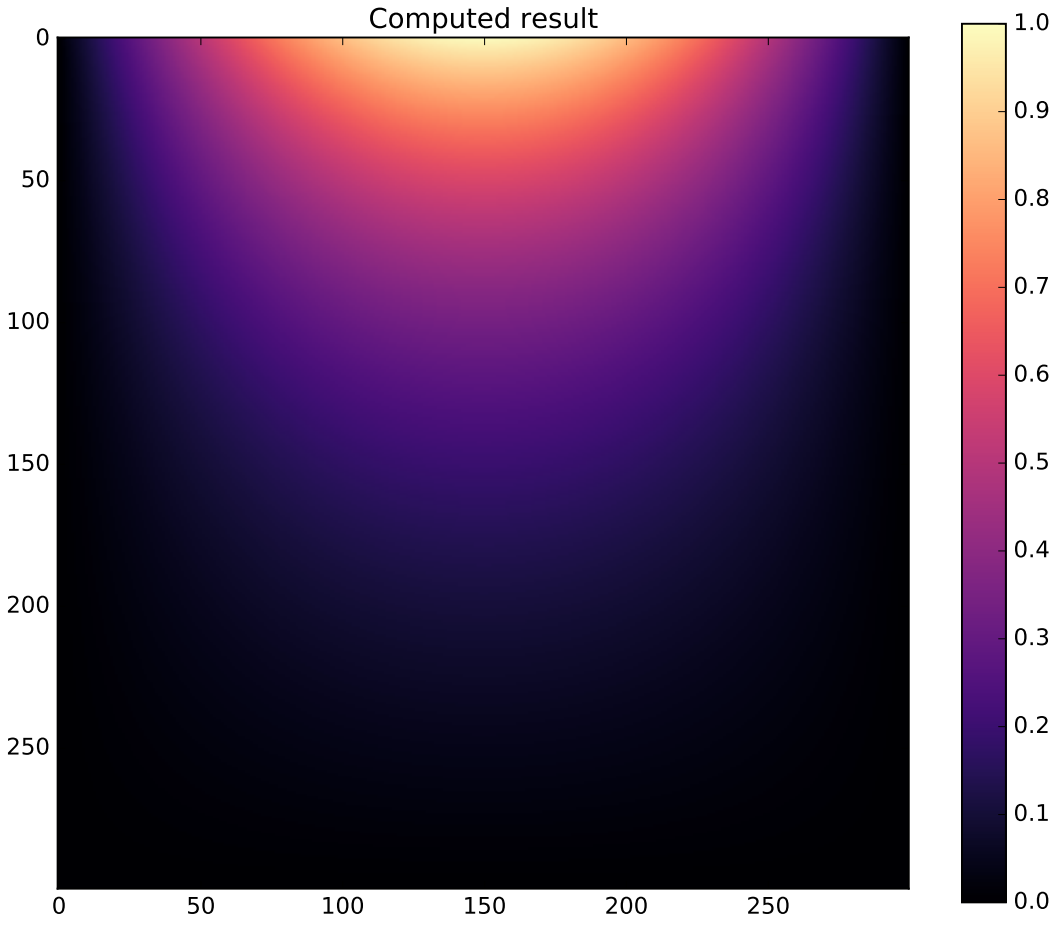


Figure 1: Simplistic view of CPU and RAM.

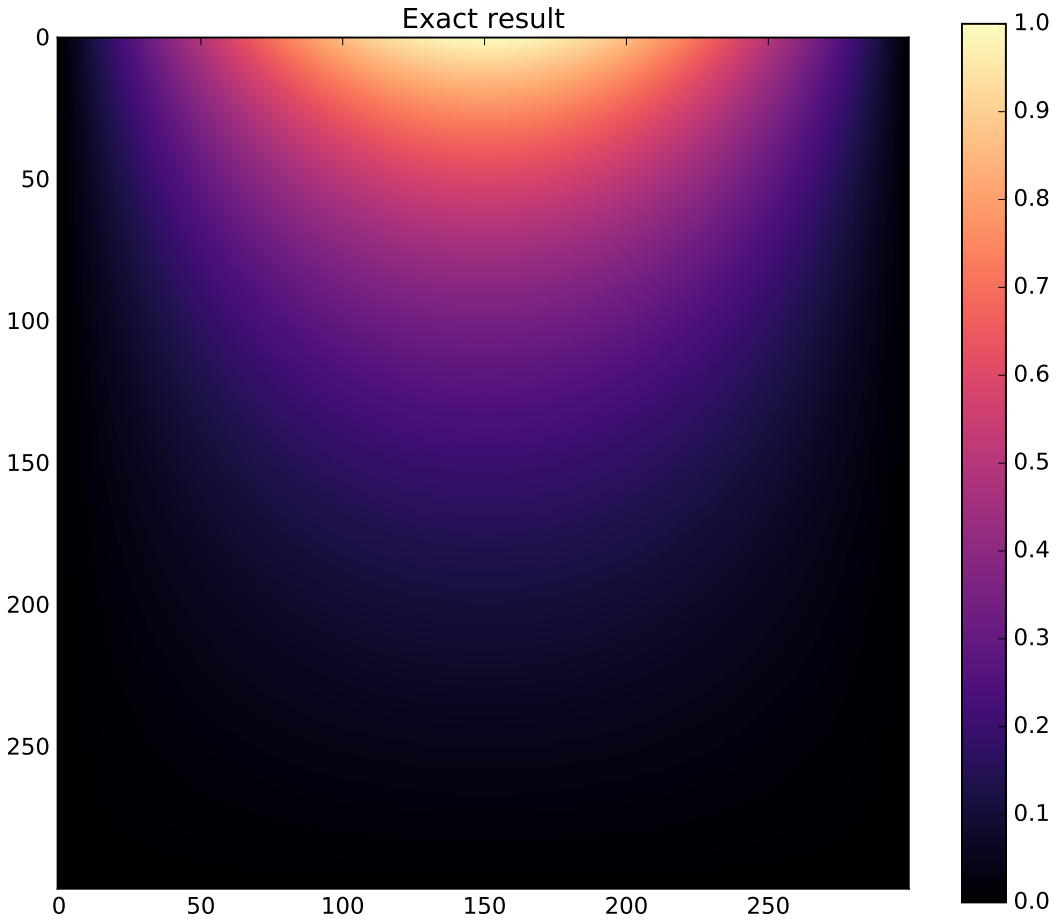


Figure 2: More realistic view of CPU and RAM.

Figure 1 shows the output of the solver program in single-threaded mode when given the $\sin(x)$ along one side potential, plotted with the Python library matplotlib. The configuration has a grid size of 300 by 300. It matches very well with the analytical result, which is

$$\frac{\sinh(y\pi/n)}{\sinh(\pi)}\sin(x\pi/n),$$

and is shown plotted in a similar way in figure 2. The difference between the calculated result and the analytical result is shown in figure 3. The values shown in the difference map are small compared to the values in the result in all locations except the corners at the top, where the simulation does diverge somewhat, but never reaches values which are hugely divergence from the analytical result. Using the root-mean-square error statistic described earlier, we have calculated that the RMS error in this simulation to be SOME NUMBER.

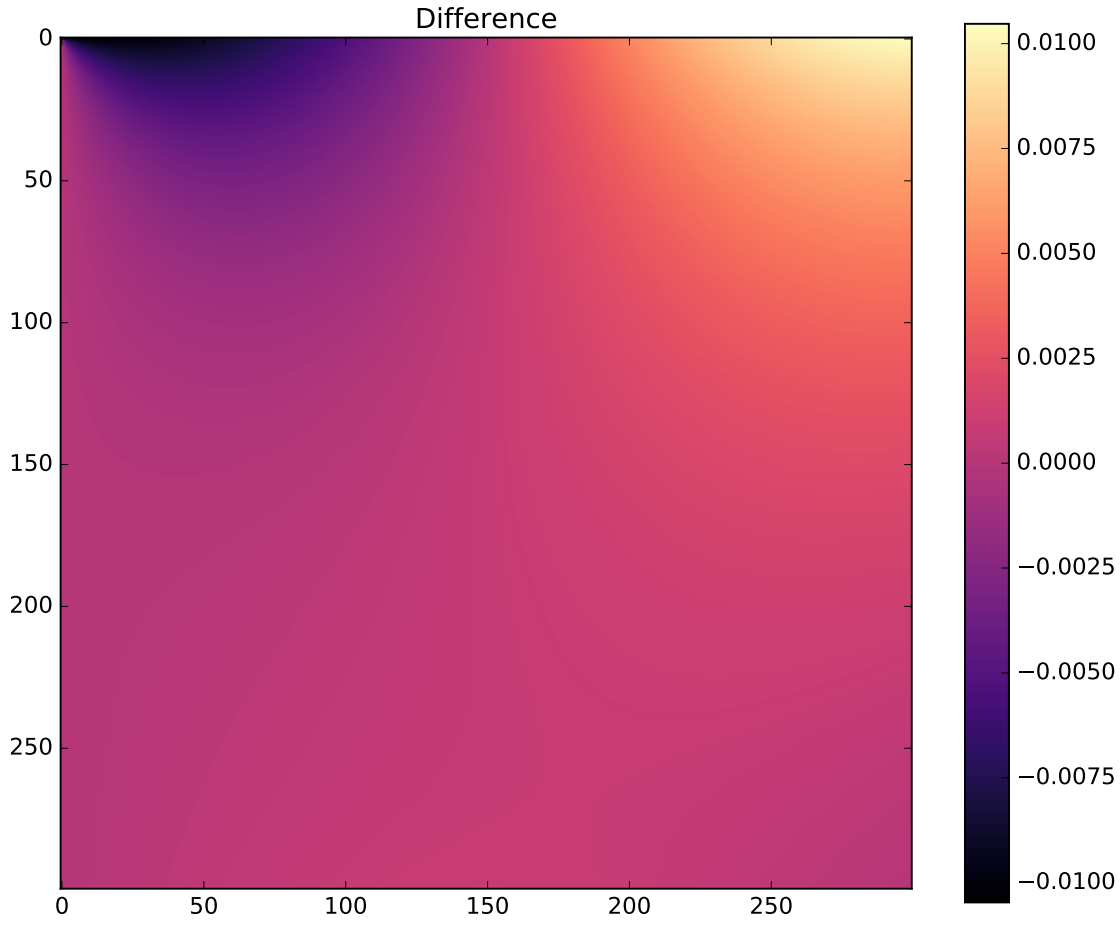


Figure 3: More realistic view of CPU and RAM.

3.2 Fitting to a $1/r^2$ Dipole Potential

The dipole configuration had a grid size of 400 by 400, with all walls given a Dirichlet boundary condition of zero. Two charges were placed near the center 10 cells apart with an arbitrary but equal and opposite charge. The result of this simulation is shown in figure 4, with the addition of constant-potential contour lines. The simulation shows two opposite charges which near perfectly mirror each other. The vertical line in the center represents a constant potential contour with a value of zero, which is expected from an electric dipole. This is another excellent verifier that the simulation is producing correct results.

The potential was then measured at locations increasing in distance from the dipole in an arbitrary direction. These data were then plotted using gnuplot, and a line of the form

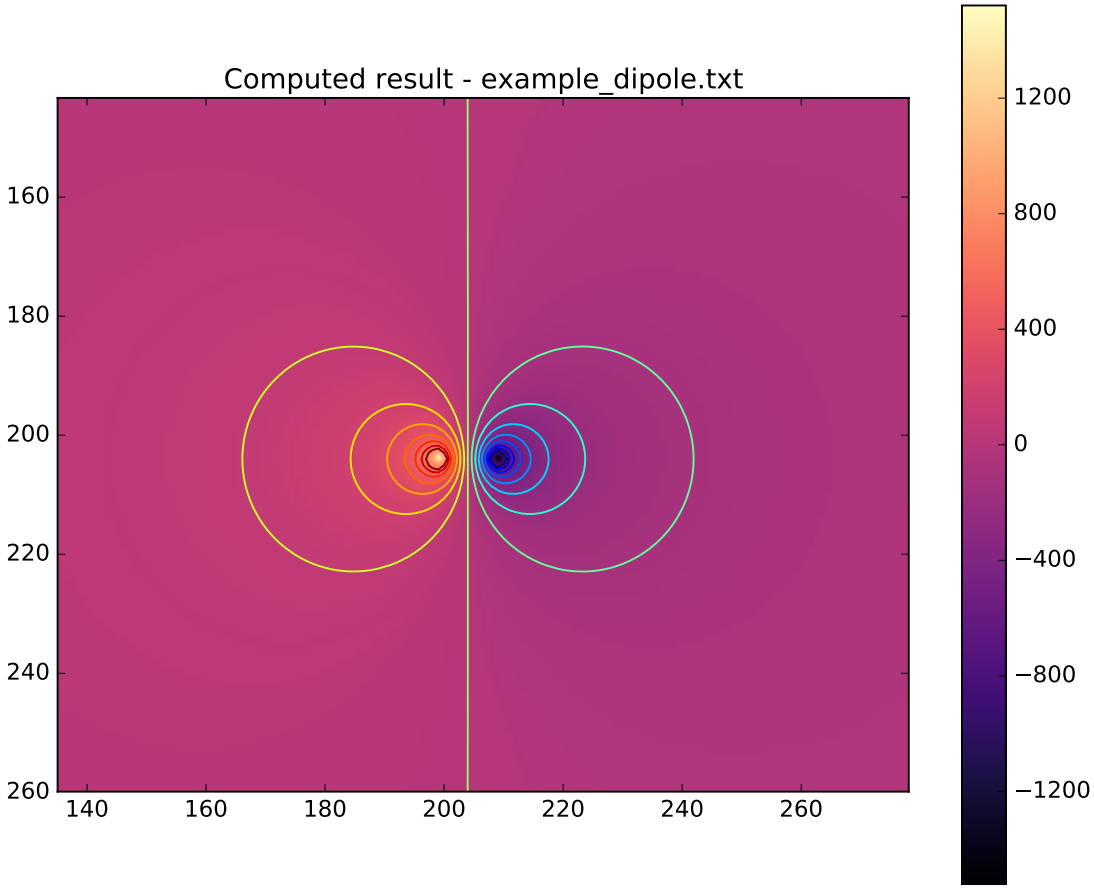


Figure 4: Fitting a $1/r^2$ curve to the calculated dipole potential.

$y(x) = A + B/(r - r_0)^2$ was fit to the data, which is shown in figure 5 The fit is excellent, confirming that the program had properly simulated an electric dipole. The error bars on the data points come from the difference between successive iterations at the end of the simulation of the dipole. They are very small, but non-zero.

The dipole is an example which lends itself to presenting another feature of the solver program. If passed the `-V` option, it will take the negative gradient of the result and plot that as a vector field. This has the effect of plotting the electric field of the result of the simulation, which for the dipole is shown in figure 6.

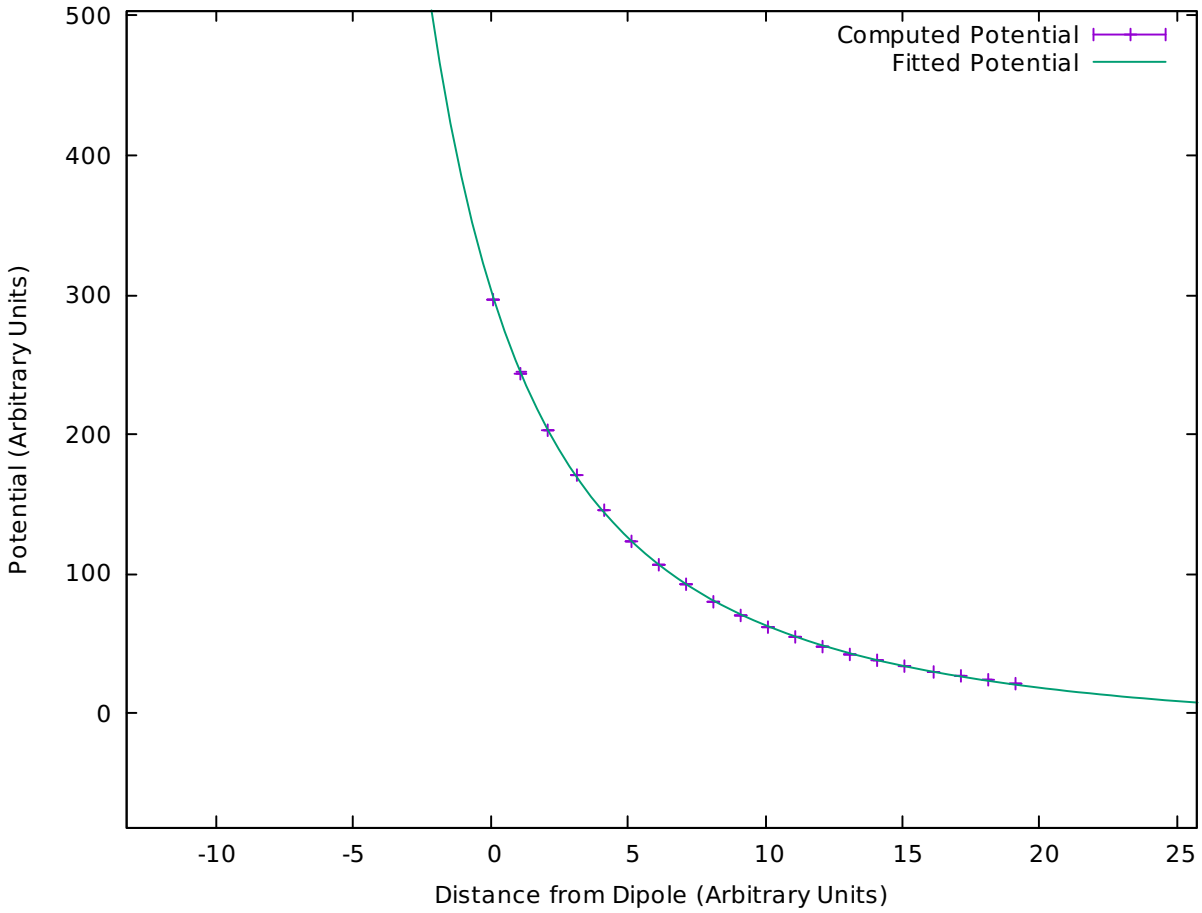


Figure 5: Fitting a $1/r^2$ curve to the calculated dipole potential.

3.3 Performance

Figure 7 shows the performance in iterations per second for a varying grid size for given configuration, in this case the $\sin(x)$ along a wall potential. The performance tests were run square grids of size 50 by 50, 300 by 300, and 1000 by 1000. For each grid size, 4 different simulation options were given: single-threaded non-SIMD, single-threaded SIMD, multi-threaded with 4 threads non-SIMD, and multi-threaded with 4 threads and SIMD.

In the case of the 50 by 50 grid, the performance was similar for all cases except for multi-threaded with SIMD. I believe the reason that the SIMD multi-threaded version was slower was due to the manual SIMD code being less cache-friendly, resulting in the threaded versions having more contention. Both the 300 by 300 case and 1000 by 1000 case display similar performance behavior: the SIMD case is faster than the non-SIMD case when single threaded, and the SIMD case is slower than the non-SIMD case when multi-threaded. This can be explained in the same was as it was for the 50 by 50 case. Another characteristic is that the multi-threaded case (with 4 threads) is faster than the single threaded case. In fact, it is better than I expected it to be.

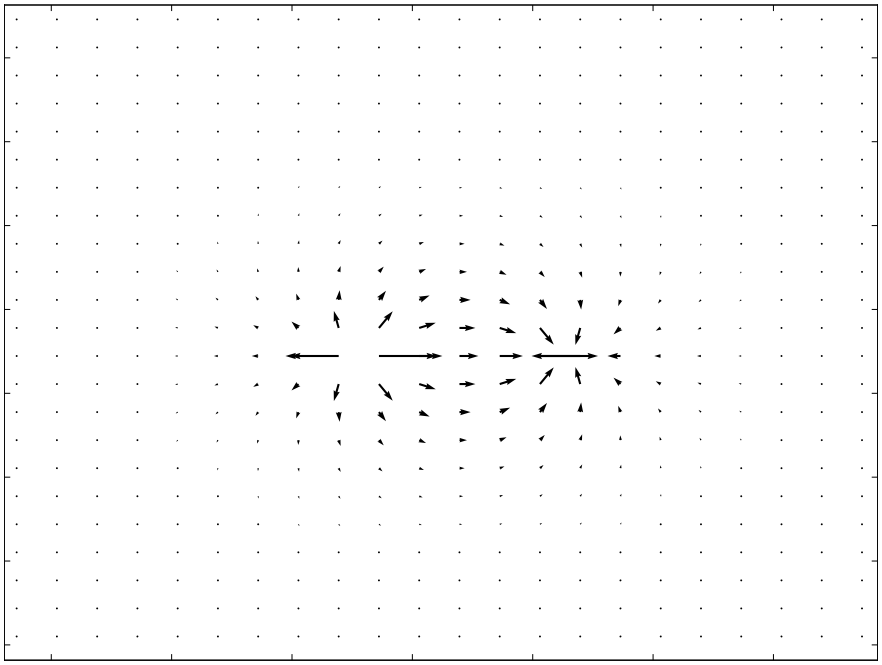


Figure 6: Fitting a $1/r^2$ curve to the calculated dipole potential.

Additionally, the performance depends heavily on the grid size. This makes sense, as the amount of work needed to be done per iteration is proportional to the number of cells in the grid, and the number of cells in the grid is n^2. Another note is that the variance on the performance is significantly higher for all of the multi-threaded cases. This is because having multiple threads increases the program's dependence on the scheduling quirks of the operating system. Finally, the variance for the 50 by 50 case multi-threaded versions are significantly higher than all other variances for the other multi-threaded cases. I believe that this is because the overhead of managing threads compared to how much work they actually have to do is large in this case compared to the other cases.

Figure 8 shows the performance in iterations per second of the solver when run with the $\sin(x)$ potential with a grid size of 300 by 300. This was done on computer B, and so was done only in non-SIMD mode. The results show a significant improvement in performance when using multiple threads compared to the single threaded case. One may naïvely have expected the performance to scale linearly with the number of threads; however, this would be very unlikely. As the number of threads goes up, the amount of work that can be done in parallel would seem to scale with the number of threads, until one considers processor cache effects (specifically cache coherency and cache size limitations). This is seen in the case of 4 threads, as it is not 4 times as fast as the single threaded case, only about 3.2 times as fast. Increasing the number of threads quickly starts giving diminishing returns, as the 8 threads case is now only slightly better than the 4 threads case. Recall that computer B, on which these tests are run, has 4 cores with 2 threads each, totally 8 possible threads running in parallel. After the 8 threads case, we start to see a drop in performance; the 16 and 32 threads case are worse than the 8 threads case. This is because the machine cannot run

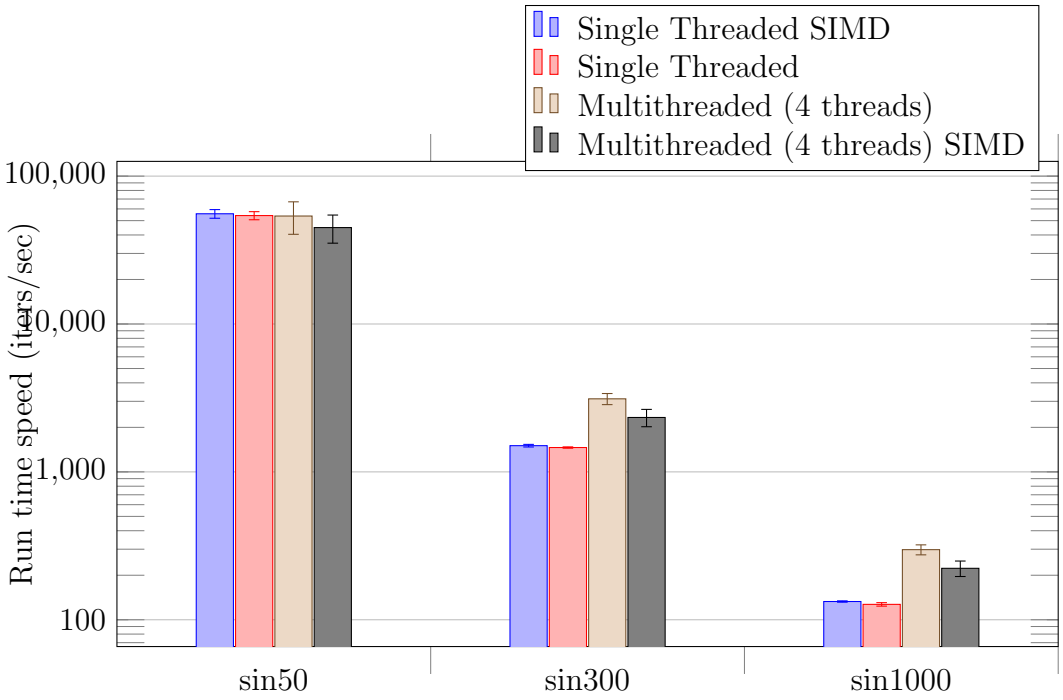


Figure 7: Different grid sizes for the $\sin(x)$ configuration, and their performance characteristics.

more than 8 threads in parallel, so in order to have 16 threads it must rely on the scheduling of the operating system in order to get work done. In a case of many threads, each doing heave computation and memory accesses (as is the case here), it does not usually help to increase the number of threads beyond what the machine can handle, as is shown here.

Furthermore, figure 9 shows that it is disadventagous to run the simulation with a large number of threads. The RMS error of the simulation compared to the analytical result increases rapidly with the number of threads. This is also reflected in the result of the simulation, which is noticably wrong in the case of 16 or higher threads. I believe this to be the result of the way the threads are synchronized, or more accurately, how they are not. The threads run mostly in isolation, sharing the grid and doing operations on it without waiting for other threads to complete any of their work. This means that if one threads runs faster than the others, it may complete a different number of iterations than the other threads in a given amount of time. This would result in some sections of the grid recieving more iterations than others, which could have the effect of invalidating the simulation. The alternative method would be to synchronize the threads, and have them wait for each thread to complete a given iteration before moving on to the next one. Indeed, this was the original design of the system, and I found it to be so slow compared to single threaded that I changed the design around to the less accurate but much faster design that is shown here.

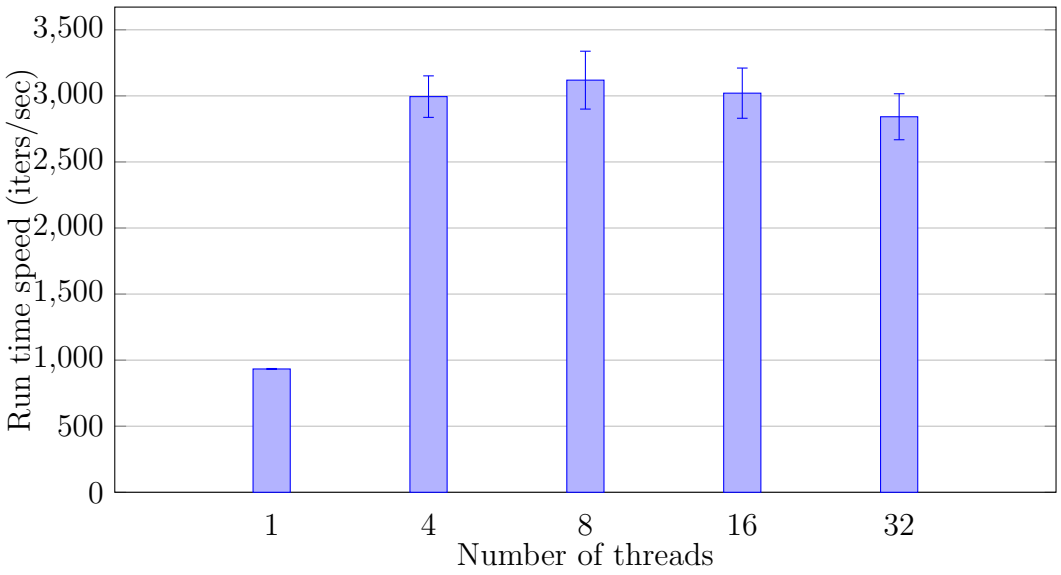


Figure 8: Performance results for a varying number of threads in non-SIMD mode. These tests were done on computer B, using the $\sin(x)$ potential with a grid size of 300 by 300.

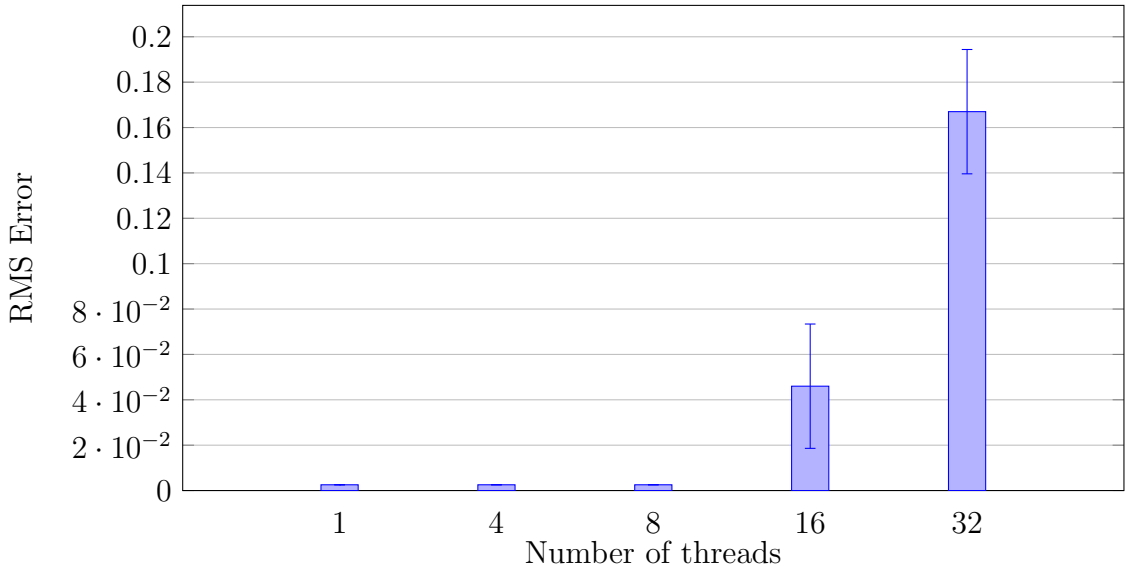


Figure 9: Error results for a varying number of threads in non-SIMD mode. These tests were done on computer B, using the $\sin(x)$ potential with a grid size of 300 by 300.

4 Conclusion

This program correctly solves certain boundary value problems, enabling simulation of complex electrostatics situations. It does this in an efficient manner by leveraging proper programming languages and techniques and utilizing the basics of modern processor design. There were numerous surprises found in developing this software and looking at the resulting physics that it described.

4.1 Correctness

The most important aspect of this program is that is produces an accurate result, no matter how fast it runs.

4.2 Performance

Since the simulation is correct, I was able to then turn my attention towards improving the performance without reducing the accuracy of the simulation. Most of this was easy to do without reducing accuracy, as it was simply transforming one valid implementation of the code into another and measuring to ensure that my new implementation did in fact cause an improvement. Most of this was done by improving cache locality and reducing memory accesses. I have left out a lot of this process for brevity, however an overview of it and the concepts used can be found in appendix B.

The next significant performance gain attempt that I had made was the use of SIMD instructions. I was hoping for a much more significant performance boost than I saw, which was rather miniscule in reality. The only small improvement was not surprising in the single threaded case because I knew that `gcc` utilizes SIMD when it can, but I did not expect to see the manual SIMD instructions degrade the performance in the multi-threaded case. I speculate that this is caused by my manual SIMD code being less cache friendly, which is okay in the single threaded case and allowed a small performance improvement, but in the multithreaded case the threads are all contending for limited cache space, so the additional memory usage slowed them down.

4.3 Future Work

Appendices

A Program Usage

Compiling the engine (the C library) requires make and gcc (or clang, though clang is untested), and can be done by simply invoking `make`. This will also create an executable file, `solve`. This is a script that invokes the python example frontend so that it may be used.

For more details on using the python frontend or the C backend engine, see the man pages (reproduced on the following pages). There are several example files, including `example_sin.txt` and `example_dipole.txt`. For more details on writing configurations for a desired simulation or experiment, see the man pages.

The source code also contains a directory named `tests`, which contains python code to generate verifier data files for the `example_sin` configuration. Finally, all of the data written on in this document are present in the source directory, under `results`. The source code for the C library is in `engine`, and the source code for the python example frontend is under `frontend`. The source code and git revision history is available at `http://github.com/dbittman/statics-solver`.

NAME

solve - Solve Discreet Poisson Equation

SYNOPSIS

`solve [-V] [-m` *`method`*`] [-v` *`verification-data`*`]` *`configuration-file`*

DESCRIPTION

Solve Poisson Equation given boundary conditions on a discreet grid. Capable of solving physics problems that reduce to a boundary value problem. Operates on the provided configuation file, producing a 2-D array describing the calculated potential.

`-V`

Produce a vector plot, where the vectors are the negative gradient of the potential.

`-m` *`method`*

Use method *method* when solving. Current supported values are `jacobi` for Jacobi Iteration, and `sor` for Successive Over-Relaxation.

`-v` *`verification-data`*

Use the data file *verification-data* to compare with the generated potential. The format for this file must be that of the python library pickle serializing a 2-D array.

CONFIGURATION

The configuration file format is specified by a series of commands on lines. A single line can contain at most one command. A command must be contained within one line. Comments begin with a #, and comment out the rest of that line. Valid commands are as follows, where italics indicates something to be replaced by one token:

`gridsize` *`size`*

Set the size of the grid. The grid is always a square, and this command **must** come before any other.

`cell` *`coords`* `initial` *`value`*

Specify initial value of a cell inside the grid. Analogous to a point charge.

`dirichlet` *`coords coords`* `=` *`value`*

Specify a diriclet boundary condition along the interpolated straight line from the first set of coordinates to the second with value *value*.

`neumann` *`coords coords direction = value`*

Specify a diriclet boundary condition along the interpolated straight line from the first set of coordinates to the second with value *value* across the boundary of the cells specified by *direction*, which may be one of `left, up, right, down`.

The values of *`coords, size, direction`* must all be one token, that is, they may not contain any whitespace. The contents of *`value`* and *`coords`* are as follows:

`value` is an `expression`.

`coords` is an `expression` followed by a comma, followed by an `expression`.

An `expression` is a valid python expression, with access to the python math library, the current grid class, and the current position in the grid as specified by `x` and `y`. For example, `math.sin(x*math.pi/grid.len)` is a valid `expression`.

AUTHOR

Written by Daniel Bittman (`danielbittman1@gmail.com`). Please submit any bug reports to this email address.

COPYRIGHT

NAME

solver - Library to solve Discreet Poisson Equation

SYNOPSIS

```
solver.h
     SOLVE_METHOD_JACOBI
     SOLVE_METHOD_SOR
double solve(struct grid *grid, int method);
void init_grid(struct grid *grid);
struct grid {
     int len;
     int iters;
     float **values;
     float **value_prevs;
     float **initials;
     uint8_t **dirichlet_presents;
     float **dirichlets;
     uint8_t **neumann_presents;
     float **neumanns[4];
};
```

DESCRIPTION

This library provides fast solving of a 2-D discreet boundary value problem using the Poisson equation. The full process is to define a `struct grid`, and set its `len` field. Then call `init_grid` on the grid to initialize the 2-D contiguous arrays. After that, call `solve` and pass it the grid and a method, either `SOLVE_METHOD_JACOBI` or `SOLVE_METHOD_SOR`.

The `solve` function will return when complete, returning back a value describing how confident it is in its result (values closer to zero are better). The `iters` field in the grid will have been updated to indicate how many iterations the program took. The `values` field points to a 2-D array of size `len` by `len` containing the solution.

AUTHOR

Written by Daniel Bittman (`danielbittman1@gmail.com`). Please submit any bug reports to this email address.

COPYRIGHT

B Computer Program Optimization

When optimizing a program, there is an important cardinal rule to follow which at times is easy to forget in the face of increasingly interesting and complex minor optimizations designed to squeeze every bit of performance out of your computer. Unfortunately, due to the complexity of modern computer hardware, it is very difficult to predict if a given micro-optimization will actually be benificial or not. For this reason, the most important rule when taking a working program and making it fast is[3] *benchmark everything.*

The field of program optimization is huge. There is a small number of people who, for a given platform, are really *good* at optimization. There is an even smaller number who are truly experts. I do not fit into either of these categories. For this reason, the program that I have written here is *decently* optimized. I believe that the maximum throughput is not orders of magnitude better than what I have achieved, however, there is still a lot of work that can be done (including a potentially significant optimization that I have neglected, and will talk about later).

B.1 Freebies

There are some optimizations which can be used in high confidence and are also very easy to do. The first, and most obvious, is to use a fast language. While writing in `C` may be a chore compared to writing in python, the resulting code will be much faster. This is the reason behind the organization of this program: the frontend (which has no performance requirements) is written in python because parsing the configuration file is much easier. The backend needs to be fast, so it is implemented in `C`.

A quick and rough experiement can easily convince you that this is true (remember how I said to benchmark everything?). Credit to my friend Chris Milke for doing this test. Write the following code in both python and in C: initialize a variable to zero. Then loop from zero to some large number (we used 123,456,789), and add that iteration to the variable. After the loop, print the variable. The resultant python program took 14.25 seconds to run on my computer, but the C code completed *instantaneously.* Why is python slower than C in general? Because python is an interpreted language. The code that you write is read by another program and executed by that program. In contrast, C code must be compiled into machine code. This is then run directly on the processor, which cuts out the middle-man of the interpreter.

When using C, there are some more things that you can get for free: compiler optimizations. When compiling a C program, the command looks something like this: `gcc -Wall -Wextra`

[3] besides *don't break it*, of course. An incorrect program is worthless, no matter how fast it is.

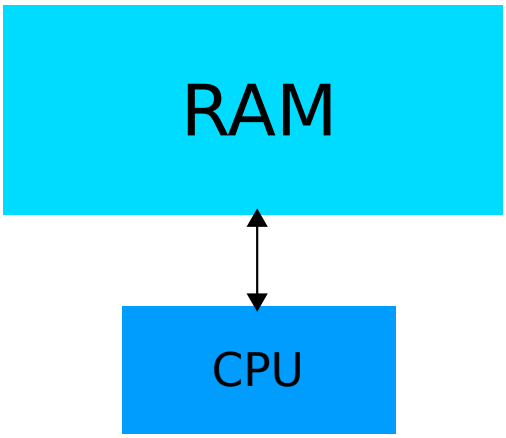
`foo.c`[4]. This will result in a compiled C program (a binary file, containing the machine code), but it will be unoptimized. There are reasons why one may want an unoptimized binary (they are typically easier to debug, for example), but generally when you compile your program for actual usage you will want to optimize it. An aggressive optimization compilation command may look something like: `gcc -Wall -Wextra -O3 -ffast-math -mavx`, as a start. The main flag here is `-O3`, which enables almost every optimization that gcc can do.

Going back to the example from earlier, the optimized version of the C program completed instantly, but the unoptimized version took 0.33 seconds. The unoptimized version is still 43 times faster than the python code, but what was the optimizing compiler doing to make it so much faster with `-O3` on? Part of optimizing is understanding why something is faster, so lets look at the assembly code. On UNIX, this can be done with the objdump command. Table 1 shows the annotated assembly from the function `main`. If you don't know x86_64 assembly language, you will at least notice that the optimized version is much shorter. If you do know how to read the assembly, you'll notice that the unoptimized version is doing almost literally what the C code says to do: set a variable to zero, and iterate from zero to a big number, adding that iteration to the variable, and finally printing the variable. The optimized code just prints a large number – which happens to be the result of the sum.

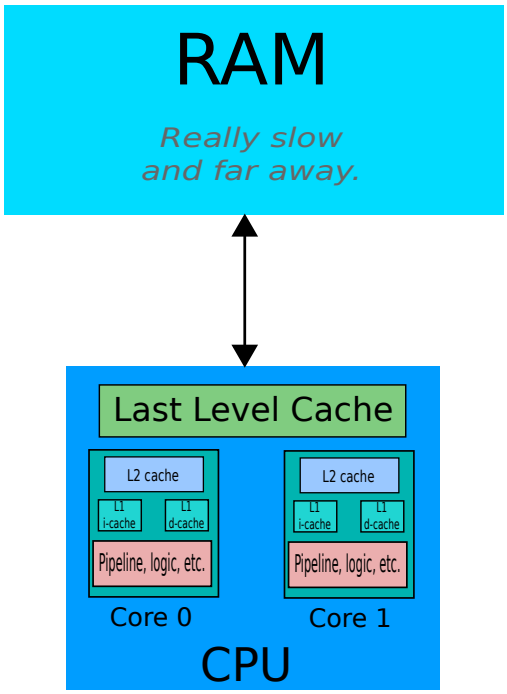
Unoptimized	**Optimized**
`mov [rbp-0x10], 0x0`	`movabs rsi,0x1b131147ee6b52`
`jmp .loop_end`	`mov edi,0x400594`
`.loop_top:`	`xor eax,eax`
`mov rax, [rbp-0x10]`	`jmp 4003c0 <printf@plt>`
`add [rbp-0x8], rax`	
`add [rbp-0x10], 0x1`	
`.loop_end:`	
`cmp [rbp-0x10],0x75bcd14`	
`jle loop_top`	
`mov rsi,[rbp-0x8]`	
`mov edi, 0x4005b4`	
`mov eax, 0`	
`call 4003c0 <printf@plt>`	

Table 1: Unoptimzed and optimized assembly from the sum-a-lot-of-numbers example.

This is just a simple example, but a good C compiler can drastically improve the performance of a program. Here, it made a program that took a third of a second (which is already drastically faster than the interpreted python version!) finish instantly.



(a) Simplistic view of CPU and RAM.



(b) More realistic view of CPU and RAM.

Figure 10: View of programming that programmers like to have (a), versus the the more realistic and complicated view.

B.2 A Model of Processor Design

B.3 Structure of Arrays

B.4 Multithreading

B.5 Vectorization

B.5.1 Branch Reduction

[4]as an aside, you should **always** compile with -Wall and -Wextra, and eliminate all warnings from your program. Warnings are warnings for a reason, and should rarely be ignored.

C Program Implementation