Lab #13

Physics 91SI Spring 2013

Objective: In this lab, you will use optimization procedures to substantially speed up the execution of several operations.

As usual, log on to `corn` and clone over the starter repository:

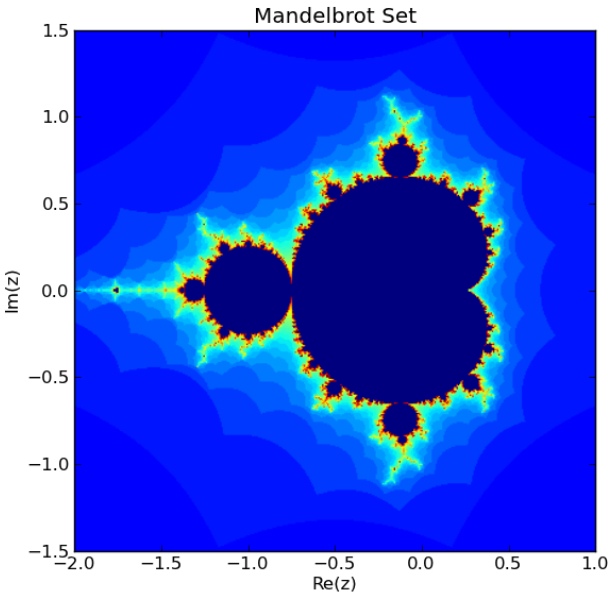
```
hg clone /afs/ir.stanford.edu/class/physics91si/src/lab13 lab13
```

We have again included a `Makefile` for you, so all you need to do to compile your code for Part 2 is type `make` at the command line. You can then type `./matrices <size>` to run it.

Remember to `hg commit` often to save your changes, and submit your code at the end of class.

Part 1: Integrating Python and C

The Mandelbrot Set is a distinctive and easily-recognized fractal, shown at right, with an elaborate boundary structure that incorporates smaller versions of the main shape as well as a diverse array of different structures. The set is defined as the complex numbers c for which the sequence $z_{n+1} = z_n^2 + c$ (with $z_0 = 0$) remains bounded, i.e. does not diverge to infinity. It is easily computed by calculating this sequence for a grid of starting points, and it can be visualized by plotting a color to represent the number of iterations before each point "escapes" to infinity.



The starter code in `mandelbrot.py` contains Python code to generate an image of the set. Run it with `./mandelbrot.py -p`, and note the execution time (the script will automatically print it for you). This isn't bad for a low resolution image, but will be painfully slow for more detailed views of the set.

Your task is to improve on this, and implement the C++ code in the `weave_test()` function. This code should do exactly the same thing as the Python code, so that they can be directly compared. You will need to use some of the C++ standard library functions; google "C++ std::complex" for a handy reference.

When you've implemented your code, run the program in weave mode with `./mandelbrot.py -w`. By what factor does your performance increase? Also, what happens when you run it a second time?

Part 2: Revisiting Matrices in C

In this part we will revisit a program that performs common matrix operations such as matrix addition and matrix multiplication. This time, we have already written the functions for you that perform these operations. It is now your job to use everything you've learned in lecture to speed up the computation.

The source code is in the file `matrices.c`. The program is described in detail in the comments, but roughly, it does the following: The program is called with a single command line argument, an integer giving the size of the square matrices involved. The `main()` function then fills 2 square matrices of that size with random numbers. It then multiplies and/or sums (depending on which function calls you comment/uncomment) the two matrices into a result matrix. The program is written to time the sum/multiplication operation and print the time spent after execution. This will allow you to observe the change in performance as you tweak the program.

We have also included two files, `fast_mul` and `fast_sum`, that were written using some simple optimizations. They behave exactly like the `matrices` file, in that they take one command line argument (the matrix size) and perform either the multiplication (`fast_mul`) or addition (`fast_sum`). You should be able to get close to their performance or better using only simple modifications of the code in `matrices.c`.

First, have a look at the sum function: `matsum()`. This function takes two matrices and adds them. Run the program a couple of times with different matrix sizes. Observe the scaling. Is it what you expect to be? From what you have learned in lecture, how would you go about speeding up the execution? There are several ways in which you could speed up the execution of that function. *Hint: Think about how the loops iterate over the arrays and how efficiently the cache works with that scheme. Also, try to eliminate all redundant computation.*

Next, consider the function `matmul()`. This function also takes two matrices, but it multiplies them using standard matrix multiplication. Go to the main function and comment the call to `matsum()` and uncomment that to `matmul()`. Recompile, and run the program for a few arguments. Does the scaling behave as you would expect? What *should* the scaling be? Now, go back to the source code and see if you can optimize the code. Can you use the same trick as you did for `matsum()`? You might have to rearrange your data or use temporary storage in order to do it. How much can you speed up your code? Compare the codes at large matrix sizes (~ 800 or more) for the differences to become pronounced.