Mathematical Issues:

What matrix norm(s) might be useful and why? - Peter

Discuss applications of numerical solutions of PDEs arising in Finance - Peter

Personal thoughts - All

I want to hear in your own words what you thought the assignment was useful (or

not useful!) for, what you learned from it, how it helped you and whether you think it could be improved. “I also want you to identify the most important aspects of the assignment (as you see

them).”

Peter:

I personally thought the assignment was very useful. I was quite competent with python, git and test driven development coming into the project, but teaching these concepts (especially git) to Andy and Kieron helped me solidify my understanding. Development experience in a collaborative environment is difficult to obtain, and this project allowed us to practice agile project management practices and see what worked for us.

Researching the implementation of mathematical concepts is also a new skill, as was structuring and testing such an integrated project. Also, the 4-step process of Test Driven Development, Pair Programming, writing documentation, and writing design criteria was very eye opening. The number of errors, logical fallacies or unjustified decisions caught at each stage was surprising, and the multi-layer structure of this review process worked very well.

As such, I think the most important aspects of the project were teamwork and project management, multi-level and iterative review processes, and the application of mathematical concepts through code. All of these will be critical in our careers, and being able to practice them on such an interesting assignment is invaluable.

Detailed Design Document

The initial planning meeting saw the project split into 3 parts:

* Input / Output;
* SOR; and
* Black-Scholes.

While SOR could not be properly tested without input data, and Black-Scholes could not be solved without SOR, we got around this by setting out rigid data structures to span between components. As such, SOR could be completed with test data that was of exactly the same format as would be provided by the input modules. The Black-Scholes matrix and solution vector could be tested using traditional matrix multiplication methods until SOR was ready.

This agile approach allowed each member to create their sections simultaneously, and was very successful. When the first working iteration of each component was complete, they were integrated and worked with very little manipulation.

**Input**

The input components consisted of reading in files, checking their format, converting to Compressed Sparse Row (CSR) format, and the checking for strict row / column diagonal dominance. A number of standard errors were identified and tested for, and if they were found to occur, we could return advice to the user on how to fix them. Four numpy arrays were passed to the SOR section, 3 containing the input matrix in CSR form and 1 containing the solution vector .

Input and output files names are specified when calling the function on the command line, but the check\_CM\_args() function exists to check these files exist. If not, the function prompts the user to input another file name until the file is found. This process is robust and ensures that file name errors are caught and the user notified and given an opportunity to correct without exiting the program.

3 file formats are supported. The first, a dense matrix format as described in the assignment outline. The second, a CSR format containing 4 lines of data (the non-zero matrix values, the column indices, the row start indices, and the solution vector ) was included as it is an efficient storage method for sparse matrices. Finally, a .mtx format was included to make testing with ‘matrix market’ matrices easier.

A raw input check was conducted on dense and CSR files to ensure that non-digit entries were not present. This was done line by line using Regular Expressions to minimize memory usage. As .mtx files were external to the core purpose of the program, explicit testing of their contents was not conducted. Errors were handled using try/except statements when importing data.

Matrix Market data was read and converted to CSR format using the scipy.io package. Dense and CSR data was read into memory line by line to minimize the risk of running out of memory when reading in massive matrices. The format of the data was confirmed (correct number of entries, rows and columns) and converted (if necessary) to CSR. This was to minimize the size of numpy arrays stored on disk.

While numpy arrays have many advantages over traditional python lists, they are memory intensive, especially when appending values to them. As such, the vectors were compiled in a list format, and converted to a numpy array when complete.

The final purpose of the input components was to check for zeros on the diagonal, and for row / column diagonal dominance. This is conceptually more difficult with CSR data than with traditional dense data, but actually faster. Any errors found are returned to the user, and an output file generated if required.

The input components pass 4 correctly formatted and tested numpy arrays to the SOR section.

**SOR**

Successive Over-Relaxation was implemented in a similar fashion to that described in the lecture notes. One major design issue was the difference between Python (0-indexing) and the pseudo-code (1-indexing). This was resolved by changing the CSR arrays to 0-indexed and iterating from to , and .

At the end of each iteration, the residual between the calculated vector and solution vector for convergence to within a specified tolerance, and the Euclidean vector norm of the difference between vectors in successive iterations is checked for divergence.

The residual tolerance…

The SOR function returns the calculated solution vector (if found), the stopping reason, the iteration limit, the actual number of iterations performed before convergence, the -sequence tolerance and the residual tolerance.

**Output**

The output file is created using the -vector and values returned from SOR, spaced using the ljust() function and list comprehension.

**Black-Scholes**

**Optimisation**

Throughout the programming, optimisation was at the forefront of our approach, in two ways. The first was to exploit the natural ‘fastest method’ of operations in Python. One example of this was the decision to build vectors by appending to lists, and then converting to numpy arrays. This minimises array creation operations, which are slow and memory intensive. Another was the use of the elementwise operation of numpy arrays, which allow fast and complete operations to be performed without iterating through lists. Finally, we use of renaming numpy arrays (without creating another in memory) allowed our SOR modules to minimise the number of arrays kept in storage.

The second optimisation approach was algorithm based, around what we could do to help speed up the process. A number of ideas were presented to optimise the relaxation parameter and initial guess of the vector . These ideas were harder to quantify, so when the main modules were completed, small test modules were created and run on a large Matrix Market .mtx file () with a standard (using ) random solution vector (as Matrix Market does not supply solution vectors).

The two options for initial optimisation were to create a random numpy array, or create a range of random numpy arrays, check their ‘distance’ from the solution vector (using the difference of Euclidean norms), and choose the closest.

Discussion and times

There were two optimisation approaches suggested for that were compared to the baseline approach of a guess of . The first was to solve using values of 1.2, 1.3 and 1.4 and then choose the fastest, but this was dismissed for being needlessly computationally heavy. The second was to compare the convergence of the vector as a ratio of the convergence on the current iteration over the convergence on the previous iteration. If this is faster than ‘usual’, is decreased, otherwise is increased.

Divergence: Row / diag dominance – do before for divergence, or just let it run through 2 iterations. Seen to be much slower to do before

**Collaboration Issues**

A project of this size led to a number of housekeeping issues. Individual coding styles created a challenge to present a cohesive body of code, although this was mostly resolved with careful planning. Separating code out into small modules helped with organisation, but did create issues when testing.

This was also the first time for us working on a collaborative coding project across multiple operating systems. Implementing solutions like ‘os.path.join’ to get around the difference between ‘/’ and ‘\’ between Unix and Windows was a key lesson learned. While we had varying individual experience and confidence with Git, the problems presented by local environment files being shared led to issues with our chosen IDE, PyCharm. Research led us to fully utilize the ‘.gitignore’ file to ensure that local files were not pushed to GitHub.

Implementing Test Driven Development is an ideal that supposes a certain level of development prowess that was not evident across the whole team. As such, tests were planned but not created before coding commenced. As the project progressed, all team members learned the language of testing, and saw the value in writing tests first.

As modules were separated for housekeeping, our given testing package, ‘nosetests’, started failing to import modules and showing other errors. One unanticipated side-effect of housekeeping was tracking dependencies across all modules. These issues led us to the ‘nosetests’ documentation which in general was sparse and unhelpful. A lot of reading and trial and error eventually fixed our problems, but we definitely learned the value of proper documentation.