Master of Science (Business Analytics)
—
Numerical Analytics and Software
—
Programming Assignment: Linear equations

Due date and time: 5:00pm, Monday 21st November, 2016.

1. Introduction

There are two main purposes in doing this assignment:

(*i*) to develop an appreciation of applications of numerical algorithms; and
(*ii*) to further develop your programming skills.

I want to see evidence that you have thought about what you are doing, and the results that you have found. 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).

The assignment will be done in a team of three people. Form teams yourselves and email them to me by 2:00pm Tuesday 1st November, 2016. Use "NAS Programming Assignment 2 Team" as the email subject.

The assignment is worth 20% of the course mark, broken down thus:

- 12% for your program;
- 8% for your documentation;

It will be graded according to the following criteria:

(1) Correctness;
(2) Completeness;
(3) Generality;
(4) Adequacy of design;
(5) Adequacy of testing;
(6) Discussion, Summary and Conclusions.

Late assignments will be penalised:

- losing 10% of the assignment mark if up to one week late;
- losing 20% if between 1 and 2 weeks late; and
- *cannot* be accepted if more than 2 weeks late.

These are general UCD regulations, outside my control.

Plagiarism will incur a more significant penalty, ranging from a 25% penalty for minor cases, to no grade for this assessment in serious cases. Again, please see the University policy on plagiarism.

These UCD policies are at `http://www.ucd.ie/registry/academicsecretariat/policyd.htm`: look under L for "Late Submission of Coursework" and P for "Plagiarism", respectively.

1.1. **Program Code.** Develop your assigned program code in Python, according to good programming standards and styles.

All of your code and documentation should be put under a top level directory (folder) called

`nas_Surname1_Surname2_Surname3_prog2`

Here,

- `nas` means Numerical Analytics and Software
- `Surname1` is the surname of the first team member
- `Surname2` is the surname of the second team member
- `Surname3` is the surname of the third team member

Give the surnames in alphabetical order. Thus, the project name `nas_Bloggs_Jones_Smith_prog2` is valid, while `nas_Jones_Bloggs_Smith_prog2` is not. Create a directory (folder) structure as follows:

- create the top level directory `nas_Surname1_Surname2_Surname3_prog2`;
- under this top level directory, create two subdirectories:
  - `code`
  - `docs`

As you have much more freedom of design compared to the first programning assignment, you do not need to adhere to a fixed set of subdirectories. The only requirements are:

- the main executable program(s) should be in the subdirectory `code` of the top level directory `nas_Surname1_Surname2_Surname3_prog2` (you will probably have one for carrying out SOR on a given matrix, and one for solving the Black-Scholes-Merton problem).
- the program for solving $Ax = b$ using SOR should accept input file `nas_Sor.in` and write output file `nas_Sor.out` (see below), both located in the subdirectory `code` of the top level directory `nas_Surname1_Surname2_Surname3_prog2`

2. PROBLEM

Write a program to implement the *Sparse-SOR* algorithm for solving a system of n linear equations in n unknowns, $Ax = b$, with $A \in M_n\mathbb{R}$ and $x, b \in \mathbb{R}^n$.

3. SPECIFICATION

Your program should

- read data from an input file containing A and b
- solve (if possible) $Ax = b$ using the *Sparse-SOR* algorithm and the sparse matrix data structure for *SOR*, as given in lectures
- write the computed solution vector x, together with the reason for stopping and other information, to an output file.

The exact formats of the files are given in §3.1; the possible stopping reasons are given in §3.2.

Apart from the above requirements, this specification makes no particular demands on how you implement *SOR*. Most of the details are left up to you to work out in your detailed design document.

For example, consider the issue that since n is not known initially, you do not know initially how much storage you need. In your detailed design document, discuss this issue and your approach to solving it, with your reasons for taking the approach. Also discuss, giving your reasoning, your approach to the implementation issues given below as bullet points, and any other implementation considerations that you may think of.

Recall from the mathematical discussion of *SOR* that it

- is only defined for a matrix A with no zeros on the diagonal
- will only converge if the matrix $C := -(D + L)^{-1}U$ has spectral radius $r_\sigma(C) < 1$; a *sufficient* condition for this is that A be either strictly row or column diagonally dominant (recall that D, L and U are the diagonal, subdiagonal and superdiagonal parts of A, respectively).

These points, and others, must be taken account of in your implementation:

- check matrix diagonal values for zeros
- check for divergence, *i.e.*, cycling or $\|x^{(k)} - x^{(k-1)}\|$ increasing with each iteration
- you will need a function to calculate vector norms for the divergence check in the last point
- compare norms with appropriate tolerances (NB Chapter 4, §3, §4), for:
  - $\|x^{(k)} - x^{(k-1)}\|$, the difference in successive x approximations
  - $\|r^{(k)}\|$, where $r^{(k)} := b - Ax^{(k)}$ is the k^{th} residual (see §6.6 of Ch. 4); comment on the usefulness of this test and what tolerance you consider appropriate: *e.g.*, would you use tolerance 0?)
- stop when `maxits` reached.

For your own benefit you may like to write the results to screen as well while running the program but this is not necessary.

3.1. **File handling.** Both the input and output files should be human-readable.

If the user provides an argument to the program, use that as the input filename; otherwise use the default filename `nas_Sor.in`. A suggested format for the input file is (a total of $n + 2$ lines, in the order given):

(1) the first line of the file contains just an `int` n, the size of the matrix A;
(2) The next n lines each contain n `double`s: these are the entries of the $n \times n$ real matrix A; so the first line of n entries in the file comprises the first row of the matrix, the next line of n entries comprises the second row and so on;
(3) The last line contains n `double`s, the entries of the n-column vector b.

You may use another (*e.g.*, more efficient) input format for A in (2) above if you wish, but you will have to clearly define the required format and be able to correctly populate your *Sparse-SOR* data structure (*val*, *col* and *rowStart*) from your input file. You will still need n and b as in (1) and (3) above. For example, you could use a format from `http://math.nist.gov/MatrixMarket`.

If the user provided two arguments to the program, use the second one as the output filename; otherwise use the default filename `nas_Sor.out`. The output file consists of three lines.

(1) Line 1 gives heading titles *Stopping reason*, *Max num of iterations*, *Number of iterations*, *Machine epsilon*, *X seq tolerance*, *Residual seq tolerance* (if used) for the values on line 2.
(2) The second line consists of the following, *aligned* under the headings in line 1:
   - one of the possible stopping reasons as given below [text string]
   - the value of `maxIts` [integer]
   - the actual number of iterations taken [integer]
   - the value of machine epsilon used, ε_m
   - the x sequence tolerance used
   - the residual sequence tolerance used (if you decide to use this).
(3) If the stopping reason is one of
   - `"Max Iterations reached"`,
   - `"x Sequence convergence"`, or
   - `"Residual convergence"` (if this test is used),

   then the third line consists of n `double`s, the components of the solution vector x. Otherwise (no convergence), the third line is left blank.

   You may also display the output to screen if you wish.

3.2. **Stopping rule conventions.** Among other things, you will need to test for the following situations and set `stopReason` appropriately.

Situation	`stopReason`
Convergence of x-sequence	`"x Sequence convergence"`
Convergence of residual (if used)	`"Residual convergence"`
`maxIts` reached without x convergence	`"Max Iterations reached"`
Divergence of x sequence	`"x Sequence divergence"`
Zero on diagonal of A (will cause SOR divide by zero)	`"Zero on diagonal"`
Other cases of divide by zero or unrecoverable exception	`"Cannot proceed"`

The stopping reason should be written to the output file (see above). Good exception handling will be useful here, and will be rewarded.

4. Running and testing.

Even if your program is not working perfectly, still submit it. However, in this case, document what does not work and why you suspect it doesn't.

Each program must be adequately *Unit tested* before being handed up. This means:

(1) Construct a test plan — list all the important different cases that can arise in execution of the program. Going through the important paths along which the program branches will identify these different *Test Cases.* At a bare minimum you can treat your program as a black box, knowing what outputs it should produce for given inputs[1]; but as you are the writer of the code and so know the internals of it, you should also be developing tests that ensure the most important of the internals get tested too.
(2) For each test case,
   (a) Construct *Test Data* that will cause execution along that path.
   (b) Document the program output this data should give, *i.e.*, the *Expected Result.*
   (c) Carry out the test run and compare the *Actual Result* with the expected result.
(3) If necessary, modify code and retest.

You are encouraged to test your program with different size matrices, and different examples of A and b. In particular, test with the following. Here, "small" means $n \leq 5$. Recall that we denote the diagonal, subdiagonal and superdiagonal parts of A by D, L and U respectively, and $C := -(D+L)^{-1}U$. *Note*: C is not invertible since U has a zero row, giving $\det U = 0$.

- A small matrix A having a 0 on the diagonal;
- A small diagonally dominant matrix A;
- A small matrix A such that
  - A has no 0 on the main diagonal
  - A is *not* diagonally dominant
  - all of the eigenvalues of C have absolute value < 1.

  What do you expect to happen in this case?
- A small matrix A such that
  - A has no 0 on the main diagonal
  - A is *not* diagonally dominant
  - one or more of the eigenvalues of C have absolute value > 1.

  What do you expect to happen in this case?
- A small matrix A such that
  - A has no 0 on the main diagonal
  - A is *not* diagonally dominant

[1] This is really Integration Testing.

- ◦ all of C^tC's eigenvalues[2] have absolute value < 1.
  [That is, the spectral radius $0 \le r_\sigma(C^tC) < 1$. This means $\|C\|_2 := \sqrt{r_\sigma(C^tC)} < 1$.]

  What do you expect to happen in this case?
- A large (say $n = 1000$–10000) sparse diagonally dominant matrix.

Your program should converge for some of these cases but not for others — can you predict which? Try the website `http://math.nist.gov/MatrixMarket` for examples of matrices to test with. This site provides browser routines to generate matrices according to requirements you give, *e.g.*, sparse, diagonally dominant, etc. You can also construct matrices with pre-chosen eigenvalues and eigenvectors using MATLAB or the freely downloadable GNU Octave. Also, do some extra testing, *e.g.*, use different values of tolerance to see the effect on number of iterations. Be imaginative.

What effect do you expect an ill-conditioned matrix to have on the running of the program? Try running with an ill-conditioned matrix to see what happens.

4.1. **The Black-Scholes-Merton option pricing problem.** Read the background document to this assignment and numerically solve the Black-Scholes-Merton equation described there. This essentially means solving a system of linear equations for each timestep m from $M-1$ down to 0. For a given timestep m,

$$-\frac{nk}{2}(n\sigma^2 - r)f_{n-1,m} + (1 + kr + k\sigma^2 n^2)f_{n,m} - \frac{nk}{2}(n\sigma^2 + r)f_{n+1,m} = f_{n,m+1} \tag{1}$$

for each $n = 1, \ldots, N-1$. At the time you are solving for timestep m, you will have already got the $f_{n,m+1}$ for $n = 1, \ldots, N-1$, so they will be known, they give the right-hand vector b in $Ax = b$. (Here, k is the grid separation in the time direction, *i.e.*, the length of time between timesteps.) From the boundary conditions you have $f_{0,m} = X$, the strike price, for all m; and $f_{N,m} = 0$ for all m. Now, when $n = 1$, this gives $f_{n-1,m} = f_{0,m} = X$; and when $n = N-1$ you have $f_{n+1,m} = f_{N,m} = 0$. This system of linear equations (1) can be written using a tridiagonal $(N-1) \times (N-1)$ matrix $A =:$

$$\begin{bmatrix} 1 + kr + k\sigma^2 1^2 & \frac{-k}{2}(\sigma^2 + r) & 0 & \cdots & 0 & 0 \\ \frac{-2k}{2}(2\sigma^2 - r) & 1 + kr + k\sigma^2 2^2 & \frac{-2k}{2}(2\sigma^2 + r) & \cdots & 0 & 0 \\ 0 & \frac{-3k}{2}(3\sigma^2 - r) & 1 + kr + k\sigma^2 3^2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 + kr + k\sigma^2(N-2)^2 & \frac{-(N-2)k}{2}((N-2)\sigma^2 + r) \\ 0 & 0 & 0 & \cdots & \frac{-(N-1)k}{2}((N-1)\sigma^2 - r) & 1 + kr + k\sigma^2(N-1)^2 \end{bmatrix}$$

This gives the system of linear equations $$A \begin{bmatrix} f_{1,m} \\ f_{2,m} \\ f_{3,m} \\ \vdots \\ f_{N-2,m} \\ f_{N-1,m} \end{bmatrix} = \begin{bmatrix} f_{1,m+1} + \frac{k}{2}(\sigma^2 - r)X \\ f_{2,m+1} \\ f_{3,m+1} \\ \vdots \\ f_{N-2,m+1} \\ f_{N-1,m+1} \end{bmatrix}.$$

In your numerical solution, use reasonable values for the Black-Scholes-Merton parameters r (say 0.01 to 0.03) and σ (say 0.2 to 0.4). The maturity date T and strike price X can have any realistic value (*e.g.*, one month in the future, and \$10). Can you construct the matrix entries just from these, once you know the grid spacing in each dimension and the number of steps in each direction? First try a few, then a few dozen, then a few hundred grid points in each direction and see what happens. That is, vary $k = T/M$ and see what effect there is on the solutions you get. The video at `http://demonstrations.wolfram.com/EuropeanOptionPricesAndGreeksIn3D` should give you some intuition on the shape of the surface we are trying to find, and indeed how it varies with the parameters r, σ, etc.

[2] A matrix M is called *positive semidefinite* if $v^t M v \ge 0$ for all $v \in \mathbb{R}^n$. All of its eigenvalues are real and ≥ 0. For any matrix C (invertible or not) C^tC is positive semidefinite, so all eigenvalues of C^tC must be ≥ 0.

5. Deliverables and Conclusions.

Apart from the Python code, you must provide documentation, consisting of cover/title pages and three major sections, as follows:

(1) A standard cover page stating that this is all your own work, signed by all team members (scanned signatures will suffice for uploading to Blackboard);
(2) A title page, containing
  - Title and handup date of assignment
  - Full name, student number and class (MSc(BA) FT/PT) of first team member
  - Full name, student number and class of second team member
  - Full name, student number and class of third team member

(3) Overview, where you discuss the mathematical issues, discuss possible applications (*e.g.*, numerical solution of PDEs arising in finance), and answer the questions raised above:
  - Is it useful to check for convergence of the residual to 0? If so, what tolerance do you consider appropriate, *e.g.*, would you use tolerance 0?
  - What matrix norm(s) might be useful and why?
  - Would scaling be helpful?
  - For which of the example matrices in §4 would you expect convergence, and for which would you not expect convergence? Why?
  - What is the effect of using different values of tolerance?
  - What effect do you expect an ill-conditioned matrix to have?

  Here you should also summarise your results from this project, and draw conclusions.

(4) Detailed design, where you address the implementation issues (as mentioned in §3), make your design decisions and justify these decisions.
(5) Unit test plan and results, detailing
  - test cases,
  - test data to direct program flow to those cases,
  - expected results, and
  - actual results.

The body of the document may consist of at most ten A4 pages of text in 10 point font. This ten page limit does not include the cover page, the title page or any pages consisting only of diagrams, figures or tables (put such pages in an appendix at the end of the document). Use these ten pages wisely and write concisely.

You may use Word, Openoffice.org, pdf for this document. Name[3] your document

`nas_Surname1_Surname2_Surname3_prog2.pdf` (or `.docx`, `.odt`, etc.)

and put it in the subdirectory `docs` of `nas_Surname1_Surname2_Surname3_prog2`.

Zip `nas_Surname1_Surname2_Surname3_prog2` with its subdirectories and their contents into one zipfile named `nas_Surname1_Surname2_Surname3_prog2.zip`. When I unzip this, I should see one top-level directory, `nas_Surname1_Surname2_Surname3_prog2`, and inside this directory are your `code` and `docs` subdirectory structure. NB: use `.zip` files as opposed to `.rar` files. Submitting a `.rar` or other form of compressed file will mean you get zero marks.

5.1. **Submitting.** One team member submits the zipfile through Blackboard SafeAssign: a second submits the document only. Only one team member should submit each deliverable to avoid false positives from the plagiarism detection tool. Also, as a backup, email both the zipped code and documentation to `sean.mcgarraghy@ucd.ie`. The subject of the email must be your project name `nas_Surname1_Surname2_Surname3_prog2`. (Some people did not use the right email subject for Programming Assignment 1!) Ensure that all team members are CCed on the email.

[3]Of course, you replace `Surname1` etc. with your own names!