Computational Linear Algebra 2022/23: Second Coursework

Prof. Colin Cotter, Department of Mathematics, Imperial College London

This coursework is the second of three courseworks (plus a mastery component for MSc/MRes/4th year MSci students). Your submission, which must arrive before the deadline specified on Blackboard and Github Classroom, will consist of two components for the submission.

1. A pdf submitted on Blackboard to the Coursework 2 dropbox, containing written answers to the questions in this coursework. **This pdf should have the title** `CID_Coursework2.pdf` **replacing** `CID` **with your CID. This enables anonymous marking of your work.**

2. Your code committed and pushed to your Github Classroom repository on the master branch. You can and should push at any time, but we will filter out commits made after the submission deadline (note that the time of the push is not relevant but please remember to push).

If you have any questions about this please ask them in the Ed Discussion Forum.

In answering the project work here, you will need to write additional code. This code should be added to your git repository in the cw2 directory. It is expected that it will just be run from that directory, so no need to deal with making it accessible through the installed module.

Some dos and don'ts for the coursework:

- **don't** attach a declaration: it is assumed that it is all of your own work unless indicated otherwise, and you commit to this by enrolling on our degree programmes.
- **do** type the report and upload it as a machine-readable pdf (i.e. the text can be copied and pasted). This can be done by LaTeX or by exporting a PDF from Microsoft Word (if you really must). This is necessary to enable automated plagiarism checks.
- **don't** post-process the pdf (e.g. by merging pdfs together) as this causes problems for the automated checks, and makes the resulting documents very large.
- **don't** write anything in the document about the weekly exercises, we will just be checking the code.
- **don't** include code in the report (we will access it from your repository).
- **do** describe in your answers in the report where to find the relevant code in your repository.
- **do** make regular commits and pushes, so that you have a good record of your changes.
- **don't** submit Jupyter notebooks as code submissions. Instead, **do** submit your code as .py modules and scripts.
- **do** remember to "git add" any new files that you add.
- **don't** forget to git push your final commit!
- **don't** use "git add ." or add files that are reproducible from running your code (such as stored matrices, or .pyc files, etc.)
- **don't** use screenshots of code output. Instead, paste and format it as text.
- **do** document functions using docstrings including function arguments.
- **do** add tests for your code, executable using pytest, to help you verify that your code is a correct implementation of the maths.

- **do** write your report as clearly and succinctly as possible. You do not need to write it as a formal report with introduction/conclusions, just address the questions and tasks in this document.
- **do** label and caption all of your figures and tables, and refer to them from the text by label (e.g. Figure 23) rather than relying on their position within the text (don't e.g. write "in the figure below). This is a good habit as this is a standard requirement for scientific writing.
- **don't** hide your answers to the questions in the code. The code is just there to show how you got your answers. Write everything in the report, assuming that the marker will only run your code to check that things are working.
- If you have any personal issues that are affecting your ability to work on this course please **do** raise them with the course lecturer by email as soon as possible.

Please be aware that both components of the coursework may be checked for plagiarism.

working on real matrices in this coursework

Coursework questions The coursework is organised so that it should be possible to reach a grade of 75% in the first two questions, and the third question is designed to be much more challenging. It is not expected that all candidates will attempt the third question.

1. (30% of the marks)
   In this question we will investigate some stability properties of pivoted LU factorisation.

   (a) (**7**%) Consider the matrix $A^{(n)}$, which is an $n \times n$ matrix with

   $$A^{(n)}_{ij} = \begin{cases} 1 & \text{if } i = j, \\ 1 & \text{if } j = n, \\ -1 & \text{if } i > j, \end{cases} \tag{1}$$

   For example,

   $$A^{(6)} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 0 & 1 \\ -1 & -1 & -1 & 1 & 0 & 1 \\ -1 & -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & -1 & 1 \end{pmatrix} \tag{2}$$

   1b you may go through LU decomposition or simply show LU = A because LU factorisation with 1s on diagonal is unique

   Apply `cla_utils/LUP_inplace` to $A^{(6)}$. What do you observe? What is the growth factor ρ for this matrix, as defined in the upper bound for the error in the stability theorem for partial pivotted Gaussian elimination in section 4.3?

   (b) (**7**%) What is the general form of the LU factorisation for $A^{(n)}$? Provide a brief proof of this form. How does the growth factor depend on n? should be LUP

   (c) (**8**%) Demonstrate the impacts of this growth factor on backward stability of `cla_utils/LUP_inplace` and `cla_utils/solve_LUP` applied to $A^{(60)}$, using `float64` numbers.

   1c no need to find error bound for solve_LUP

   (d) (**8**%) This rapid growth of the growth factor with the dimension n in this example, and the fact that the upper bound can be reached, looks like really bad news for Gaussian elimination. To investigate whether this situation is generic, we will look at random matrices with individually independently distributed random variables drawn from the uniform distribution on the interval [-1/n,1/n]. every entry drawn from Unif[-1/n, 1/n] Conduct some experiments, describing your methodology and presenting code in `cw2`. Describe what you observe about the growth rates of these random matrices. Is it bad news?

   1d don't worry about errors. Graphs required for rho, provide some indicative lines to show how fast the bound for rho is growing. (but no need to do linear regression)

2. (45% of the marks)
   In this question we shall perform some experiments using a matrix stored as an array available at
   `https://raw.githubusercontent.com/comp-lin-alg/cw-data/main/A2.dat`
   Save this data to a file called `A2.dat`. You can read this file into a numpy array by using
   `A2 = readtxt('A2.dat')`
   It should be a 100×20 array.
   We are going to look at using the reduced QR factorisation from the modified Gram-Schmidt algorithm using your `cla_utils` implementation.

(a) (**7**%) Add a function to `cw2` called `MGS_solve_ls` that implements the following algorithm for finding the least squares solution of $Ax = b$. assume b is vector for this question

- Find the reduced QR factorisation $\hat{Q}\hat{R} = A$ using your modified Gram-Schmidt implementation `cla_utils.exercises2.GS_modified`.
- Multiply $z = \hat{Q}^* b$.
- Solve $\hat{R}x = z$.

Provide tests automatable with `pytest` that verify that the code is a correct implementation of the algorithm.

(b) (**7**%) Investigate the error in this algorithm as follows. Choose an exact solution x^*, and then compute $b = Ax^*$, using the A_2 matrix describe above. Then, x^* will be the least squares solution of $Ax = b$. Now apply the above algorithm to obtain your numerical solution $\hat{x}$, and measure the error $\|\hat{x} - x^*\|$. Apply this a few times with different x^* and collate the results. Compare them with your results from `householder_ls`. Place your code making comparisons in `cw2`. collate = present the results

(c) (**7**%) The problem with the algorithm as written is that the multiplication with $\hat{Q}^*$ introduces additional round-off error. In particular, there is a loss of orthogonality with $\|\tilde{Q}^*\tilde{Q} - I\| \approx \varepsilon\kappa(A)$, where the condition number for rectangular matrices is defined as $\sqrt{\|A^TA\|/\|(A^TA)^{-1}\|}$, and $\tilde{Q}$ is the approximate orthogonal matrix obtained from modified Gram-Schmidt using floating point numbers. This means that b is not properly projected into the range space of A, introducing a similar error.
To solve this we introduce an alternative, more stable algorithm. We build the augmented matrix

$$A_+ = \left(A|b\right), \tag{3}$$

i.e. we add b as an extra column on the right. The reduced QR factorisation of A_+ is

first find b = … (using Q, R, q_{n+1}, rho)
then find z = …
The goal is to understand how z can be used to solve the least squares equation.

$$A_+ = \left(\hat{Q}|q_{n+1}\right)\begin{pmatrix}\hat{R} & z\\ 0 & \rho\end{pmatrix}, \tag{4}$$

where $A = QR$.
Find formulas for b and z in terms of $\hat{Q}$, $\hat{R}$, q_{n+1} and ρ.

(d) (**8**%) This approach means that we can indirectly compute Q^*b without first computing Q. This means that the same approximate column space is being used to find R and Q^*b reduces the issues with round-off.
Propose a modified algorithm based upon this idea, and briefly discuss the operation count (leading order coefficients are not necessary e.g. $\mathcal{O}(n)$ is fine instead of $\mathcal{O}(3n)$. Implement it as `MGS_solve_ls_modified` in `cw2`. Provide appropriate automatable tests that check that the code is a correct implementation of the algorithm.

(e) (**8**%) Apply your code to the provided matrix A_2, and compare the error with the previous results. Provide example code in `cw2`. this refers to the original MGS only, no need to do Householder

(f) (**8**%) In our example, we considered a rather special case, when b is in the range space of A. Make a modified version example where b is outside the range space, by choosing the x^* and picking $b = Ax^* + r$, so that the minimum of $\|Ax - b\|$ is greater than zero. keep use A2 for part 2f
Demonstrate whether the improvement in `MGS_solve_ls_modified` is still evident.
describe how you choose r and why it would be a good test (i.e. will not alter the solution x*)
demonstrate means present the results (using figures, tables etc.)

3. (25% of the marks)
In statistical image analysis, a common tool is the simulation of random pixellated images that have some correlation between pixels. Here we will explore some computational linear algebra aspects of this.

In our example, we will consider greyscale pixels on an $N \times N$ grid, with $u_{i,j}$ being the value of the pixel at grid location (i, j), ~~with value 1 for white and 0 for black~~, and shades of grey in between. We assume that the image is black for all points $i < 1, j < 1, i > N, j > N$. ignore this scale, grey scale can be plotted for any interval [a, b]

To describe the linear algebra operations required, we need to reorder the array u into a one dimensional vector $\hat{u}$ with dimension N^2. This is reordering is defined by the mapping $P\hat{u} = u$, with

$$\hat{u}_{(i-1)N+j} = u_{i,j}. \tag{5}$$

We choose $\hat{u}$ to be a multivariate Normal random variable with mean 0 and covariance $(D^2)^{-1}$, where D is the matrix such that

$$(PD\hat{u})_{i,j} = u_{i,j} + s^2\left(-u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1} + 4u_{i,j}\right), \tag{6}$$

where $s > 0$ is a parameter that governs the strength of correlations between nearby points.

Samples from this distribution can be computed by solving

$$D\hat{u} = w, \tag{7}$$

D is N²xN² matrix

where w is a vector of independent and indentically distributed random Normal random variables with mean 0 and variance 1. We can use ~~`cla_utils`~~`.random.randn` to simulate these random variables.

numpy

no need to prove, just state formulae for D

(a) (**3**%) Define a formula for the matrix D. It is a banded matrix; what is the bandwidth?

(b) (**3**%) Modify `cla_utils.solve_LU` so that it has optional arguments bl and bu for the lower and upper bandwidth, respectively. When present, the code should use the banded version of LU factorisation described in the lecture notes. This will require additionally modifying the other functions called by `solve_LU`, i.e. `LU_inplace`, `solve_U` and `solve_L`. Ensure that the original tests still pass, and provide new tests to the test directory to verify that the banded algorithm works.

3b write a new function, solve_LU was not in cla_utils

3c you may explicitly construct D here, but not 3g

(c) (**3**%) Write a function in `cw2` that simulates random images as described above, with options to use the banded solver, or your original LU solver. Present some figures in your report for various s for $N = 50$, covering the important cases.

this means you have to investigate for sufficiently large and sufficiently small values of s so that the image appearance clearly changes

(d) (**3**%) Compare the timings between the two solvers for various N.[1] What do you expect for solve times from the operation count estimates? Do you see something consistent with that?

(e) (**3**%) The large operation count and memory requirements of banded LU for large N for this matrix means that it can be better to resort to iterative methods (particularly for applications where it is not necessary to obtain the exact solution, such as when random numbers are present). One such iterative method requires the definition of two new matrices, H and V, defined by

no need to prove formulae for H, V

$$(PH\hat{u})_{i,j} = 2u_{i,j} - u_{i-1,j} - u_{i+1,j}, \quad (PV\hat{u})_{i,j} = 2u_{i,j} - u_{i,j-1} - u_{i,j+1}. \tag{8}$$

Starting from $\hat{u}^0 = 0$, we define the iterative algorithm

u^{n+1/2} means the value at intermediate step, n+1/2 is not an exponent

solve 9, 10 as linear systems don't do inversion

$$((1+\rho)I + s^2H)\,\hat{u}^{n+1/2} = (\rho I - s^2V)\,\hat{u}^n + b, \tag{9}$$

$$((1+\nu)I + s^2V)\,\hat{u}^{n+1} = (\nu I - s^2H)\,\hat{u}^{n+1/2} + b, \tag{10}$$

(11)

where $\rho, \nu > 0$ are chosen parameters of the method.

Show that if this algorithm converges to a limit u^*, then $Du^* = w$.

3f find something about the structure so that you can work with smaller matrices. don't explicitly construct H, V

(f) (**3**%) In practice, this algorithm is terminated when $\|D\hat{u}^n - w\| < \epsilon\|w\|$ for some chosen tolerance ϵ. Describe an efficient algorithm for solving the equations (9 -10). Estimate the operation count for this algorithm and show that it is smaller than the operation count for solving $D\hat{u} = w$ using a banded algorithm. Explain why it requires less memory than solving $D\hat{u} = w$ directly.

memory = mean size of arrays used in algorithm

3g you should choose ρ, ν

(g) (**3**%) Write a function in `cw2` that implements the iterative method (8), using your proposed efficient algorithm for solving equations (9-10). It should be making use of your banded LU factorisation implementation. Provide tests to verify that it is a correct implementation.

try various ρ, ν

(h) (**2**%) Make a numerical exploration of the values ρ and ~~μ~~ ν that lead to the fastest convergence of the iterative scheme for various N, describing your methodology and conclusions, and placing code used in `cw2`. How many iterations are typically required to obtain a converged solution with $\epsilon = 1.0e-6$ using optimal choices? What does this mean for the typical operation count for the algorithm as a function of N?

(i) (**2**%) Comment on how a parallel computer that can execute multiple operations at once might be exploited to accelerate this algorithm.

[1] The easiest way to get timings is to use CProfile, part of the standard Python library. If you write a script that calls your function, then you can execute the script with `python -m cProfile myscript.py`. It will provide a table of all of the Python function calls, and how long Python spends in each of them per call and in total.

no need to use this timing, you may use your own timing method