Computational Linear Algebra 2022/23: Third Coursework

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

This coursework is the third 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 3 dropbox, containing written answers to the questions in this coursework.
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 cw3 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.

Coursework questions The coursework is organised so that it should be possible to reach a grade of 80% 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. (60% of the marks)

   (a) (**5**%) Apply `cla_utils.pure_QR` to the matrix A_3, stored as an array at `https://raw.githubusercontent.com/comp-lin-alg/cw-data/main/A3.dat`.
   Demonstrate the convergence of the algorithm by plotting $\|A_S\|$ versus iteration number, where A_S is the result of setting all entries of A on or above the diagonal to zero. You should use a logarithmic scale for $\|A_S\|$ but not the iteration number. (Use `matplotlib.pyplot.semilogy`.)
   Verify that you have obtained the eigenvalues of A. You can do this by performing inverse iteration using a shift for each eigenvalue to recover the corresponding eigenvector. Put the script you use for this in `cw3`.

   (b) (**5**%) For each diagonal entry of the transformed matrix, examine the convergence to an eigenvalue of A. In which order are the eigenvalues converging, from fastest to slowest? In which position do they appear in the matrix diagonal?
   Demonstrate this by plotting the convergence of the diagonal entries to eigenvalues.

   (c) (**5**%) Apply `cla_utils.pure_QR` to the matrix A_4, stored as an array at `https://raw.githubusercontent.com/comp-lin-alg/cw-data/main/A4.dat`. Show what you get. Why is it that the pure QR algorithm will never converge to the Schur decomposition of A_4?

   (d) (**6**%) In general, if A is real and does not have repeated eigenvalues, then the pure QR algorithm will converge to the form,

   $$A' = Q^T A Q = \begin{pmatrix} R_{11} & R_{12} & \dots & R_{1r} \\ 0 & R_{22} & \dots & R_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & R_{rr} \end{pmatrix}, \tag{1}$$

   where the submatrices R_{kk} are either 1×1 or 2×2 in each case for $1 \leq k \leq r$. The convergence is rather complicated: it is not the case that the pure QR algorithm converges to a limit. It is rather that that the matrices produced by the QR algorithm get closer and closer to matrices of the form A'. We need to deal with this aspect in our termination criteria.
   Using the fact that matrices of the form A' satisfy the following,

   $$\det(A') = \prod_{i=1}^{r} \det(R_{ii}), \tag{2}$$

   find, with justification, a method for extracting the eigenvalues of A.
   To implement this, you will need to modify the convergence criteria for `cla_utils.pure_QR`. Add one extra test to ensure that the additional case is covered (i.e. that you are stress-testing your termination criteria).
   Use it to compute the eigenvalues of A_4, and verify them using inverse iteration. Put the script you use for this in `cw3`, and present your results.

(e) (**5**%) Write a function in `cw3` that finds eigenvalues of real matrices with a similar pattern to A_4, using the same method. Demonstrate that it works on matrices of varying sizes, briefly providing and explaining examples in your report, and provide tests that quickly verify that it works.

(f) (**5**%) This approach produces good results in general (as long as matrices with repeated eigenvalues are avoided, as extra modifications are required for those), but converges rather slowly. As described in the course, a practical algorithm speeds up the QR algorithm using three techniques: (1) reduction to Hessenberg form (and subsequent use of special orthogonal transformations that preserve it), (2) shifts to accelerate convergence of chosen eigenvalues, (3) deflation to reduce the size of the matrix once those eigenvalues have sufficiently converged. In the remainder of this question we will concentrate on (2).
For symmetric matrices, we already observed that the diagonal entry in the final column is converging fastest to an eigenvalue. We can accelerate this by the following procedure.

- Take the current value of the diagonal entry from the final column, and call it $\hat{\mu}$.
- Form $QR = A - \mu I$.
- Set $A = RQ + \mu I$.

Explain why this procedure preserves the eigenvalues of A.

(g) (**5**%) (Continuing to specialise to symmetric matrices.) Once we have converged this eigenvalue to our satisfaction, we can move onto the last-but-one column, using the same technique.
To check if a diagonal entry $A^{(k)}_{jj}$ has converged enough, we check if $A^{(k)}_{j-1,j}$ is small. Explain why this works as a convergence test for that eigenvalue.

(h) (**6**%) Implement this modified algorithm for symmetric matrices with shifts as a new function in `cw3` (it should be separate from the code for parts (a-e)), and demonstrate it on A_3. Provide an automated verification that it works.

(i) (**6**%) For larger symmetric matrices demonstrate that the shifting technique accelerates the convergence, presenting results in your report.

(j) (**6**%) Describe how the deflation strategy would work and how it would accelerate the computation in this procedure. You should not implement this.

(k) (**6**%) Apply your implementation from part (h) to A_4. What goes wrong? Describe, but do not implement, a suitable fix. What would be the consequences of such a fix on the design of the QR factorisation algorithm that would be required for this modifed QR algorithm?[1]

2. (20% of the marks)
In this question we will focus on the preconditioned GMRES algorithm described right at the end of the course notes.

(a) (**3**%) Add an optional argument to `cla_utils/gmres`, `callback`, which should default to `None`. When `callback` is present, it should be a function[2] that takes in the current best estimate of the solution, `x`, and returns no arguments. We can use this callback interface to to compute the error at each iteration (or any other diagnostics that we might be interested in).
Tip: If you want to easily change the solution, you can "curry" it into `callback` like this:

```
def get_callback(x_sol):
   def callback(x):
      <do stuff with x and x_sol>
   return callback
```

Calling `callback = get_callback(x_sol)` gets a you a function that you can them pass to your `GMRES`. Check that the original tests still pass, and provide a new test for this additional feature.

[1] It will hopefully have become clear after this exercise that the general practical QR algorithm used in e.g. `numpy.linalg.qr` contains a great many adaptations and tricks to produce an efficient and robust implementation!

[2] Yes! Functions can be passed to functions in Python.

(b) (**3**%) We will experiment with this by using $m \times m$ matrices of the form

$$a_{ij} = \begin{cases} -2 & \text{if } i = j, \\ 1 & \text{if } i = j-1 \text{ or } i = j+1, \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

Assuming that the eigenvectors u of our matrix A are all of the form

$$u_k^{(l)} = \sin(kl\pi/(m+1)), \quad k = 1, \ldots, m, \tag{4}$$

with one eigenvector for each value of $l = l, \ldots, m$, find a formula for the eigenvalues of A for general m.

(c) (**3**%) Verify this formula using your QR factorisation code for various m, providing code for the verification and presenting results in the report.

(d) (**4**%) Examine the distribution of the eigenvalues for various m. What properties does the polynomial have to have to make the residual estimate small for large m?
What does this tell you about the convergence of GMRES as m increases?

(e) (**5**%) When matrices are symmetric and positive definite, the residual estimate presented in lectures can be used to derive

$$\|r_n\| \leq \left(\frac{\kappa^2 - 1}{\kappa^2}\right)^{n/2} \|r_0\|, \tag{5}$$

where κ is the condition number of A.
Use your eigenvalue formula to write the condition number of A for general m.

Find an approximate power law in m for the condition number for large m, by using a formal power series expansion in $1/m$.

What does it tell you about the convergence of GMRES as m increases?
Compare this estimate with errors obtained from your callback.

(f) (**2**%) The solution to poor GMRES convergence is to find a good preconditioner. One cheap option for a preconditioner is to use the diagonal part of A. Why is this a cheap option?
Without implementing preconditioned GMRES, explain what we expect to happen if this preconditioner is used for solving $Ax = b$ with preconditioned GMRES for our matrix A.

3. (20% of the marks)
In this question we will examine algorithms for forming one of the most general matrix decompositions of $m \times m$ matrices A, given by $A = UDV^*$ where U $(m \times m)$ and V $(m \times m)$ are unitary matrices and D is diagonal with non-negative entries.

(a) (**4**%) Show how to obtain this decomposition from the eigenvalue decomposition of A^*A, justifying the steps.

(b) (**4**%) The problem with this approach is that if A has a large condition number σ, A^*A has an even larger condition number σ^2, which is bad news for numerical stability. As an alternative, consider the matrix

$$H = \begin{pmatrix} 0 & A^* \\ A & 0 \end{pmatrix}. \tag{6}$$

Show that if $A = U^*DV$ then

$$H \begin{pmatrix} V & V \\ U & -U \end{pmatrix} = \begin{pmatrix} V & V \\ U & -U \end{pmatrix} \begin{pmatrix} D & 0 \\ 0 & -D \end{pmatrix}. \tag{7}$$

Hence, propose an algorithm for finding U, D and V based on computing the eigendecomposition of H. Why is this advantageous?

(c) (**4**%) Use your existing code in `cla_utils` to assemble the part of this algorithm necessary to compute D, placing the code in `cw3`. Provide suitable automated tests. Demonstrate your implementation on some example matrices.

(d) (**4**%) The QR algorithm produces eigenvalues by reading from the diagonal, but not eigenvectors. A good numerically stable approach is to apply inverse iteration using each of the numerically obtained eigenvalues as a shift.

Implement a function in `cw3` to compute all the eigenvectors using inverse iteration, and provide tests.

(e) (**4**%) Apply your implementation to the problem of finding the least squares solution of $Ax = b$ with minimal solution $\|x\|$ when A is a rank-deficient square matrix.

Place a function to do this `cw3` and provide suitable tests.