Computational Linear Algebra 2021/22: 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, will consist of two components for the submission.

1. A pdf submitted to the Coursework 3 dropbox, containing written answers to the questions in this coursework.
2. A SHA tagging the revision of your repository, i.e. the repository for your weekly exercises that you have attempted so far. The SHA is the (nearly) unique hexadecimal code tagging each git revision. It is possible to find the SHA of a commit locally, in the copy of the repository on your computer. However this is a dangerous practice, because you might not have pushed that commit to GitHub, so you risk sending someone on a wild goose chase for a commit that they will never find. It is therefore a much better idea to grab the commit SHA for the commit you want directly from the GitHub web interface. See the link below for information on how to do this.
   `https://imperial-fons-computing.github.io/git.html#reporting-the-commit-hash`

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

The coursework marks will be assigned according to:

- 35% for the code for the weekly exercises from Sections 5 and 6. Only the exercises where you are asked to complete skeleton functions will be marked.
- 65% for the code and written answers to the project work here.

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** tell us which git commit is the one you want us to mark.
- **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.
- **do** go onto Github and check that you haven’t pushed any non-code files such as the content of your venv or the .pyc files that are automatically generated when running Python scripts.

- **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 difficulties affecting your 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. It is fine to work together on the exercises and discuss your answers to project questions but you should write your own code and text, answering the project questions yourself.

Coursework questions

1. (35% of the marks)
   Complete the weekly exercises from Sections 5 and 6. Make sure that the code is committed your git repository and pushed to Github Classroom. Only the exercises where you are asked to complete skeleton functions will be marked.
2. (20% of the marks) Consider the $2n \times 2n$ matrix A with

$$A_{ij} = \begin{cases} -1 & j = i-1, \\ 1 & j = i+1, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

   (a) (5%) Apply the pure QR algorithm to A for various matrix sizes. What do you observe? Why is this happening? Why can't we read off the eigenvalues in this case?

   (b) (5%) Devise and implement a method to efficiently obtain the eigenvalues of A, using the output of the QR algorithm. Describe your method, explain how it works and why.

   (c) (5%) Implement your method it as code, with appropriate automated testing. Demonstrate your methods with numerical experiments for different values of n.

   (d) (5%) Extend this technique to the $2n \times 2n$ matrix B with

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

   The following identity is useful for this,

$$\det\left(\begin{pmatrix} A & B \\ 0 & D \end{pmatrix}\right) = \det(A)\det(D), \tag{3}$$

   where A is an $m \times m$ matrix, B is an $m \times q$ matrix, D is an $q \times q$ matrix.
   Implement your extended method as code, with appropriate tests, and demonstrate it with numerical experiments.

3. (20% of the marks) In this question we will make some modifications of `exercises9.pure_QR`. You should make modifications in that function rather than creating a new one. To ensure that it still passes the original tests, you will probably need to make use of optional arguments in the function definition.

   (a) (4%) Here we consider the QR algorithm when combined with the reduction to tridiagonal form (for symmetric matrices). Prove that the steps of the QR algorithm preserve the symmetric tridiagonal structure.

   (b) (4%) Modify the termination criteria for your implementation of the QR algorithm so that it stops when the $m, m-1$ element of the $m \times m$ tridiagonal matrix T satisfies $|T_{m,m-1}| < 10^{-12}$. Apply your program to the 5×5 matrix $A_{ij} = 1/(i+j+1)$ and discuss the results in your report - which of the eigenvalues are converged? Provide tests for your code.

   (c) (4%) Write a Python script implementing the following steps:

      i. Call your function from the exercises reducing a symmetric matrix A to Hessenberg form T (it will be tridiagonal in this case).

      ii. For $k = m$ to 1 downwards:

         A. Call your modified QR algorithm function applied to T until termination with the new criteria.

         B. Record $T_{k,k}$ as an eigenvalue.

         C. Replace T with the $(k-1) \times (k-1)$ submatrix of T consisting of the first $k-1$ rows and columns of T.

   Modify your QR algorithm so that it returns an array containing the values of $|T_{k,k-1}|$ at each QR iteration. Concatenate these arrays for each call to your modified QR algorithm function with tridiagonal matrices of row sizes $k = m, m-1, \ldots, 3, 2$. Plot the results when applied to your choice of matrices, including the 5×5 matrix from the previous step. Describe the results and explain what is happening. Provide tests for your code and compare with the convergence for the unmodified QR algorithm.

   (d) (4%) Modify your QR algorithm code so that it optionally applies the shifted QR algorithm, using the Wilkinson shift,

   $$\mu = a - \operatorname{sgn}(\delta) b^2 / (|\delta| + \sqrt{\delta^2 + b^2}), \tag{4}$$

   where $a = T_{m,m}$, $b = T_{m,m-1}$, $\delta = (T_{m-1,m-1} - T_{m,m})/2$. Compare your plots of values of $|T_{m,m-1}|$, including for the 5×5 matrix from previous steps. What is causing this? Check that it still passes your tests.

   (e) (4%) Compare your plots for the 5×5 matrix to the results from the matrix $A = D + O$ where D is the diagonal matrix with diagonals $(15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1)$ and O is the matrix with every entry equal to 1. Make the comparison for plots corresponding to shifts and no shifts. What do you observe?

4. (25% of the marks) In this question we consider an image denoising algorithm. The monochrome image is stored as a set of floating point pixel values from 0 (dark) to 1 (bright), in a two dimensional array $u_{i,j} = u(i\Delta x, j\Delta x)$, $0 < i < n+1$, $0 < j < n+1$, where the function $0 \leq u(x, y) \leq 1$ gives the image brightness, and $\Delta x = \Delta y = 1/(n+1)$. We assume that the image is dark around the boundary, so that $u_{i,j} = 0$ for $i = 0$, $i = n+1$, $j = 0$, and $j = n+1$ (and hence we don't need to solve for these values, as they are given). We assume that the image is noisy (we can model this by adding independent normal random variables to the brightness values) and we would like to remove this noise without blurring the image, assuming that the image has some sharp interfaces. For example, the image might be a silhouette that has values 1 on the inside of some shape, and 0 on the outside.

   One way to denoise the image is to try to find the minimum

   $$\min_u \|u\|_{BV} + \frac{\mu}{2}\|u - \hat{u}\|_2^2, \tag{5}$$

   where $\hat{u}$ is the noisy image, the bounded variation norm is defined as

   $$\|u\|_{BV} = \sum_{i=1}^{n+1} \sum_{j=1}^{n} |u_{i+1,j} - u_{i,j}| + \sum_{i=1}^{n} \sum_{j=1}^{n+1} |u_{i,j+1} - u_{i,j}|, \tag{6}$$

and the 2-norm is defined as

$$\|u\|_2^2 = \sum_{i=1}^{n}\sum_{j=1}^{n} u_{ij}^2. \tag{7}$$

The BV term tries to penalise smooth jumps in the solution, preferring sharp jumps from one value to another over a smooth transition, and the 2-norm term penalises deviations too far from $\hat{u}$. The parameter $\mu > 0$ is a trade-off parameter that balances between these two requirements.

We consider an iterative algorithm to solve this problem, which involves the introduction of additional variables $(d_{x;i,j}, b_{x;i,j})$, $1 \leq i \leq n+1$, $1 \leq j \leq n$ and $(d_{y;i,j}, b_{y;i,j})$, $1 \leq i \leq n$, $1 \leq j \leq n+1$. When the algorithm is converged, we will have $d_{x;i,j} = u_{i,j} - u_{i-1,j}$ and $d_{y;i,j} = u_{i,j} - u_{i,j-1}$; the b-values will be chosen to ensure this constraint.

Then we introduce iterative sequences u^0, u^1, u^2, ..., and similar for the d-values and b-values such that u^k should converge to the solution of the minimisation problem in the limit as k tends to infinity. We initialise $u^0 = \hat{u}$ (the noisy image) and set the d-values and the b-values to zero.

The iterative update from k to $k+1$ is defined as follows:

- Solve

  $$(\mu I + \lambda A)v^{k+1} = \mu\hat{v} + \lambda L_x^T(p_x^k - q_x^k) + \lambda L_y^T(p_y^k - q_y^k), \tag{8}$$

  where $\lambda > 0$ is a chosen parameter for the iterative algorithm, v^{k+1} is the "serialised" vector of u values, i.e. u reshaped into a 1D vector, $\hat{v}$ is the serialised vector of $\hat{u}$ values, and p_x^k, q_x^k, p_y^k, q_y^k are serialised vectors of d_x^k, b_x^k, d_y^k, and b_y^k, respectively. Further, L_x is the matrix that maps u values to differences $u_{i+1,j} - u_{i,j}$ (in serialised form throughout), L_y is the equivalent matrix for the differences $u_{i,j+1} - u_{i,j}$, and $A = L_x^T L_x + L_y^T L_y$.
  Application of A to v should be equivalent (after transforming back to a 2D array) to the central difference formula for the minus Laplacian (without the Δx^2 scaling),

  $$4u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1}. \tag{9}$$

- Update

  $$p_x^{k+1} = \mathrm{shrink}(L_x v^{k+1} + q_x^k, 1/\lambda), \quad p_y^{k+1} = \mathrm{shrink}(L_y v^{k+1} + q_y^k, 1/\lambda), \tag{10}$$

  where $\mathrm{shrink} : \mathbb{R}^M \times \mathbb{R} \to \mathbb{R}^M$ is defined as the pointwise operation

  $$(\mathrm{shrink}(x,\alpha))_i = \frac{x_i}{|x_i|}\max(|x|_i - \alpha, 0), \quad i = 1, \ldots, M. \tag{11}$$

- Update

  $$q_x^{k+1} = q_x^k + L_x v^{k+1} - p_x^{k+1}, \quad q_y^{k+1} = q_y^k + L_y v^{k+1} - p_y^{k+1}. \tag{12}$$

Our task here is to implement this algorithm by using an iterative method to solve (8). We will do this using GMRES, preconditioned by the alternating iteration that was considered in coursework 2 (for those of you who did not manage it, there are detailed instructions below). The idea is that we will avoid explicitly forming the matrices A, L_x and L_y, and just consider their actions on the entries of u.

(a) (3%) Write a function that takes in the two-dimensional array of u values and returns the equivalent serialised one-dimensional vector v. Write similar functions that can be applied to d_x and d_y (since these have different dimensions). Write two more functions that apply the inverse transformation. Provide suitable tests.

(b) (3%) Write a function `H_apply` that takes in v and applies $H = \mu I + \lambda A$. This should be done in a "matrix-free" manner as follows.
   - Transform v to the two-dimensional array form using the appropriate function above.
   - Compute the negative Laplacian by doing slice operations for the $x-$ and $y-$ differences seperately. This should not require any loops, just vectorised slice operations.
   - Scale the negative Laplacian by λ and add μu.
   - Transform the result back using the appropriate function above.

Provide suitable tests (one useful test is that the sum over the indices of the negative Laplacian should produce zero).

(c) (4%) Modify your GMRES implementation in the exercises library so that you can optionally provide a function to apply the matrix instead of providing the matrix itself (since GMRES only needs to apply the matrix, not do anything else with it).
Provide suitable tests. *Important Python fact: you can pass functions to other functions in Python, just as you pass other variables.*

(d) (3%) Write a function `M_solve` that applies one iteration of the following sweeping algorithm:

- The input is a vector x (of the dimension of the serialised vector v).
- Solve
$$(\mu I + \lambda L_x^T L_x)\hat{x} = x, \tag{13}$$
followed by
$$(\mu I + \lambda L_y^T L_y)y = -\lambda L_x^T L_x \hat{x} + x. \tag{14}$$
- Return y.

Here, the matrix $L_x^T L_x$ is equivalent to the centred difference formula for the x-derivative (without the factor of Δx: $2u_{i,j} - u_{i-1,j} - u_{i+1,j}$. Similarly, the matrix $L_y^T L_y$ is equivalent to the centred difference formula for the y-derivative: $2u_{i,j} - u_{i,j-1} - u_{i,j+1}$. Hence, the first system decouples into n independent tridiagonal solves, one per row of the image. These can be executed one-by-one by the following steps:

- Use the appropriate function to transform x to the equivalent 2-dimensional array.
- Use your banded matrix solver to efficiently solve the tridiagonal solves one by one, with each right-hand side given by a row of the 2-dimensional array version of x, replacing that row with the output of the tridiagonal solve. You should use slice notation to extract and replace rows of x.

Similary the second system decouples into n independent tridiagonal solve, one per row. You should apply the same procedure but now extracting and replacing rows rather than columns. Finally, transform the result back to serialised form to return.

(e) (4%) Modify GMRES to take another optional argument, a function to solve $Mz = y$, taking in y and returning z. If the argument is present, the GMRES function should implement the preconditioned GMRES algorithm, with the preconditioning step implemented by the provided function.
Provide suitable tests.

(f) (4%) Apply your modified GMRES code to solve (8), using `A_apply` and the sweeping preconditioner above. Provide suitable tests. Experiment with how the convergence rate of GMRES depends on n, μ and λ.

(g) (4%) Implement the denoising algorithm using this GMRES solver, and demonstrate it using some silhouette image with noise added. Experiment with λ and μ to get the best and fastest results.