Computational Linear Algebra 2020/21: Second Coursework

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

This coursework is the second of two 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 three components for the submission.

1. A pdf submitted to the Coursework 2 dropbox, containing written answers to the questions in this coursework.
2. A SHA tagging the revision of your `clacourse-2020` repository, containing your attempts at the exercises so far.
3. A SHA tagging the revision of your `clacourse-2020-cw2` repository, containing your code used to answer questions in this coursework. Where possible you should use the functions that you have completed in your `clacourse-2020` repository.

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

How to create your code repository for this submission

1. Go to this link [`https://classroom.github.com/a/N1tRq8Dv`] to create your Github classroom assignment repository for this coursework (`clacourse-2020-cw2`).
2. Clone the repository on your computer and start working on the questions; in this coursework there is no "skeleton" code and you should just add any functions that you need.
3. Your functions should make use of your additions to the `cla_utils` library from the exercises so far in the course. To make use of these functions, make sure that you activate the virtual environment just like you had to do to attempt the exercises. Then you will be able to use the library after writing `from cla_utils import *`.
4. Don't forget to commit your changes, push them to Github classroom, and record the revisions of both repositories that should be assessed when you submit the coursework on Blackboard.

The coursework marks will be assigned according to:

- 80% for answers to the questions.
- 10% for overall clarity and succinctness of the written submission.
- 10% for code quality: correct use of Git, clear and correct code, sensible use of numpy vector operations, appropriate documentation and comments. Where sensible, organise your code into functions in separate module files (just add your own) with suitable tests of your own devising (just keep everything in the root directory of your cloned `clacourse-2020-cw2` repository.

Please be aware that all three components of the coursework (the PDF report, the exercises repository and the coursework repository) may be checked for plagiarism.

1. (20% of the marks) We will consider the solution of matrix vector systems of the form

$$\begin{pmatrix} c & d & 0 & 0 & \dots & 0 & 0 & 0 \\ d & c & d & 0 & \dots & 0 & 0 & 0 \\ 0 & d & c & d & \dots & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & c & d & 0 \\ 0 & 0 & 0 & 0 & \dots & d & c & d \\ 0 & 0 & 0 & 0 & \dots & 0 & d & c \end{pmatrix} x = b, \tag{1}$$

where c and d are real numbers, i.e. the matrix A has diagonal entries c, and entries d on the first sub and super diagonals, and zeros elsewhere.

(a) Derive and present the specific algorithm for LU factorisation and solution by forward and back substitution for this case. The algorithm should avoid unnecessarily multiplying by, or adding, entries that are known to be zero.

(b) Show that the LU factorisation can be merged with the forward substitution, so that the whole solution procedure can be performed as an algorithm consisting of one iteration forwards through the vector entries and one iteration backwards.

(c) Carry out an operation count analysis for this algorithm and compare it to what is known for LU factorisation for general matrices.

(d) Implement this algorithm as code, providing suitable tests.

2. (20% of the marks) We will consider the application of computational linear algebra to fast numerical solution methods of the partial differential equation,

$$u_{tt} - u_{xx} = 0, \tag{2}$$

in the interval $[0, 1]$ with periodic boundary conditions $u(1, t) = u(0, t)$ (for all t), and initial conditions $u(x, 0) = u_0(x)$, $u_t(x, 0) = u_1(x)$ for some given functions $u_0, u_1 : [0, 1] \mapsto \mathbb{R}$. To solve the equation numerically, we first transform the equation into the first order form,

$$w_t - u_{xx} = 0, \quad u_t - w = 0. \tag{3}$$

Second, we discretise the equation in time using the implicit midpoint rule,

$$w^{n+1} - w^n - \frac{\Delta t}{2}(u^n_{xx} + u^{n+1}_{xx}) = 0, \quad u^{n+1} - u^n - \frac{\Delta t}{2}(w^n + w^{n+1}) = 0, \tag{4}$$

where $w^{n+1}(x)$ is our approximation to $w(x, (n + 1)\Delta t)$, and similar for $u^{n+1}(x)$. Third, we eliminate u^{n+1} to obtain a single equation,

$$w^{n+1} - Cw^{n+1}_{xx} = f, \tag{5}$$

where the constant C and function $f : [0, 1] \to \mathbb{R}$ are to be determined (see later). During each timestep we first solve this equation, and then back substitute to get u^{n+1}.
Finally, we discretise Equation (5) using the central difference formula to obtain

$$w^{n+1}_i - C\left(w^{n+1}_{i+1} - 2w^{n+1}_i + w^{n+1}_{i-1}\right) = f_i, \tag{6}$$

where w^{n+1}_i is our approximation to $w((n + 1)\Delta t, i\Delta x)$, where $\Delta x = 1/M$.

(a) For Equation (6), determine the constant C and the form of f_i in terms of w^n and u^n.

(b) Write Equation (6) in the form of a matrix-vector equation, where the vector contains $(w^{n+1}_1, w^{n+1}_2, \ldots, w^{n+1}_M)$, describing the structure of the matrix A.

(c) Explain why the structure of the matrix means that there is no advantage to using a banded matrix algorithm. Demonstrate this using your LU factorisation codes that you created during the exercises.

(d) Show that $A = T + u_1 v_1^T + u_2 v_2^T$, where T is a tridiagonal matrix (meaning that it has upper and lower bandwidth 1), and describe the form of the vectors u_1, u_2, v_1, v_2. By extending the idea of Exercise 1.13, find a formula for A^{-1} in terms of T^{-1}. (Hint: write $A = L(I + \text{stuff})U$ before inverting.)

(e) Design and describe an algorithm to solve $Ax = b$, using your formula for A^{-1} above. Do not explicitly form A^{-1}, but instead translate your formula for A^{-1} into a formula for the solution of $Ax = b$ that involves the solution of problems $Ty = z$ for suitable y and z, making use of the algorithm implemented in the first part of this coursework to solve them. You should not explicitly form the inverse of any matrices. Make a brief description of the operation count.

(f) Implement your algorithm to solve $Ax = b$ as code, making use of numpy array vector operations where possible, and using your code implemented in the first part of this coursework. Provide suitable tests to demonstrate that it works. Is it faster than using your generic `LU_inplace` function?

(g) Write a script to compute a number of timesteps of this discretisation of (2). You should not store the solution at previous timesteps, but instead provide options to plot or save the solution to disk every few timesteps as specified in the script. Provide suitable tests to demonstrate that it works.

3. (20% of the marks)

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

(b) When computing the QR factorisation for a tridiagonal matrix, it is a waste to use a Householder rotation for the entire below diagonal component of the column as all but one of the entries in that component are already zero. Propose a less wasteful approach that only uses 2×2 Householder rotations on the component of the column containing non-zero entries. Provide a brief explanation of the difference in operation count between the two approaches.

(c) Implement your proposed approach as a Python function `qr_factor_tri`, supported by appropriate tests that you should add. It should avoid multiplication by, or addition of, values that are known to be zero. Further, it should not compute the $m \times m$ matrix Q, but just return all of the 2-dimensional vectors v used to generate the Householder reflections.

(d) Write a new function `qr_alg_tri` implementing the unshifted QR algorithm applied to tridiagonal matrices, using your code implemented in the last step. It should avoid multiplication by, or addition of, values that are known to be zero. It should not explicitly form the Q matrix, but instead work with the Householder reflection generators that are produced in `qr_factor_tri`. Your function should stop 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. Provide tests for your code.

(e) Write a Python script that (1) calls your function from the exercises reducing a symmetric matrix A to Hessenberg form (it will be tridiagonal in this case), (2) calls `qr_alg_tri` until termination, taking $T_{m,m}$ as an eigenvalue, (3) calls `qr_alg_tri` with the $(m-1) \times (m-1)$ submatrix of T consisting of the first $m-1$ rows and columns of T. Modify `qr_alg_tri` so that it returns the values of $|T_{m,m-1}|$ at each iteration in an array. Concatenate these arrays for each call to `qr_alg_tri` with tridiagonal matrices of row sizes $m, m-1, \ldots, 3, 2$, and plot the results when applied to your choice of matrices, including the 5×5 matrix from the previous step. Provide tests for your code and compare with the convergence for the pure QR algorithm.

(f) Modify your code so that it applies the shifted QR algorithm, using the Wilkinson shift,

$$\mu = a - \operatorname{sgn}(\delta) b^2 / (|\delta| + \sqrt{\delta^2 + b^2}), \tag{7}$$

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. Check that it still passes your tests.

(g) 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. (20% of the marks)

(a) Modify your GMRES function so that it has one extra argument, a Python function `apply_pc` that takes in b and returns $\tilde{b}$, the solution of $M\tilde{b} = b$. This way you can provide a specific function that exploits the specific structure in M. Then modify your GMRES function so that it uses `apply_pc` in an implementation of the preconditioned GMRES algorithm. Provide tests for your algorithm.

(b) Assume that the matrix A and the preconditioning matrix M satisfy

$$\|I - M^{-1}A\| = c < 1, \tag{8}$$

where we measured $I - M^{-1}A$ in the operator 2-norm. Show that

$$|1 - \lambda| \leq c, \tag{9}$$

for all eigenvalues λ of $M^{-1}A$.

(c) Hence, explain why $p(z) = (1-z)^n$ provides a good upper bound for the optimal polynomial in the polynomial formulation of GMRES used to derive error estimates, and use it to derive an upper bound for the convergence rate of preconditioned GMRES using M as a preconditioned operator for A.

(d) Investigate this upper bound in the case where $A = I + L$ and L is the graph Laplacian (you may use `scipy.sparse.csgraph.laplacian` for this) for a (not too trivial) graph, $M = cU$, U is the upper triangular part of A (including the diagonal), and $c > 0$. You may use `numpy.linalg.eig` to measure eigenvalues numerically.

5. (20% of the marks) In this section we will extend the approach of the first section, i.e. solving the wave equation with periodic boundary conditions, this time solving a number of timesteps all at once. The method of this section is amenable to parallel computing, but we won't (and you should not) consider parallel computing in your implementation.
The idea is that given initial conditions $w(x,0) = u_t(x,0)$ and $u(x,0)$, we simultaneously solve for u_i^n and w_i^n for all $i = 1,\dots,M$ and all $n = 1,\dots,N$, according to

$$w_i^{n+1} - w_i^n - \frac{\Delta t}{2\Delta x}\left(u_{i-1}^n + u_{i-1}^{n+1} - 2(u_i^n + u_i^{n+1}) + u_{i+1}^n + u_{i-1}^{n+1}\right) = 0, \tag{10}$$

$$u_i^{n+1} - u_i^n - \frac{\Delta t}{2}(w^n + w^{n+1}) = 0, \tag{11}$$

Having computed the solutions for this time segment, we can take u^M and w^M and use them as initial conditions for the next time segment (if we want to compute solutions beyond this point).

(a) Writing

$$U = \begin{pmatrix} p_1 \\ q_1 \\ p_2 \\ q_2 \\ \vdots \\ p_M \\ q_M \end{pmatrix}, \quad p_i = \begin{pmatrix} u_1^i \\ u_2^i \\ \vdots \\ u_N^i \end{pmatrix}, q_i = \begin{pmatrix} w_1^i \\ w_2^i \\ \vdots \\ w_N^i \end{pmatrix}, \quad i = 1,2,\dots,M, \tag{12}$$

show that the matrix vector system equivalent to solving all these equations at once takes the form

$$AU = \left(\begin{pmatrix} I & 0 & 0 & \dots & 0 \\ -I & I & 0 & \dots & 0 \\ 0 & -I & I & \dots & 0 \\ \vdots & & & & \vdots \\ 0 & \dots & 0 & -I & I \end{pmatrix} + \frac{1}{2}\begin{pmatrix} B & 0 & 0 & \dots & 0 \\ B & B & 0 & \dots & 0 \\ 0 & B & B & \dots & 0 \\ \vdots & & & & \vdots \\ 0 & \dots & 0 & B & B \end{pmatrix}\right) U = \begin{pmatrix} r \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \tag{13}$$

where

$$B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \tag{14}$$

describing the form of r, B.

(b) We refer to each p_1, q_2 pair of subvectors as a "time slice" since it gives the values of w and u at a particular time level. We can adopt the compact notation

$$A = C_1 \otimes I + C_2 \otimes B, \tag{15}$$

where

$$C_1 = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ -1 & 1 & 0 & \dots & 0 \\ 0 & -1 & 1 & \dots & 0 \\ \vdots & & & & \vdots \\ 0 & \dots & 0 & -1 & 1 \end{pmatrix}, \quad C_2 = \begin{pmatrix} \frac{1}{2} & 0 & 0 & \dots & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & \dots & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & \dots & 0 \\ \vdots & & & & \vdots \\ 0 & \dots & 0 & \frac{1}{2} & \frac{1}{2} \end{pmatrix}. \tag{16}$$

For an $M \times M$ matrix P and an $N \times N$ matrix Q, the tensor product $P \otimes Q$ produces an $MN \times MN$ matrix where each entry P_{ij} in P is replaced by an $N \times N$ block $P_{ij}Q$.

We now introduce an iterative method for solving (13). Starting from an initial guess U^0 for the solution U, we define an iterative method to compute U^{k+1} from U^k by solving

$$\left(C_1^{(\alpha)} \otimes I + C_2^{(\alpha)} \otimes B\right) U^{k+1} = R = \begin{pmatrix} r + \alpha(-I + B/2) \begin{pmatrix} p_M^k \\ q_M^k \end{pmatrix} \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \tag{17}$$

for U^{k+1}, given $U^k = ((p_1^k)^T, (q_1^k)^T, (p_2^k)^T, (q_2^k)^T, \ldots, (p_M^k)^T, (q_M^k)^T)$, where

$$C_1^{(\alpha)} = \begin{pmatrix} 1 & 0 & 0 & \ldots & -\alpha \\ -1 & 1 & 0 & \ldots & 0 \\ 0 & -1 & 1 & \ldots & 0 \\ \vdots & & & & \vdots \\ 0 & \ldots & 0 & -1 & 1 \end{pmatrix}, \quad C_2^{(\alpha)} = \begin{pmatrix} \frac{1}{2} & 0 & 0 & \ldots & \frac{\alpha}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 & \ldots & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & \ldots & 0 \\ \vdots & & & & \vdots \\ 0 & \ldots & 0 & \frac{1}{2} & \frac{1}{2} \end{pmatrix}, \tag{18}$$

for $\alpha > 0$, i.e. $C_1^{(\alpha)}$ and $C_2^{(\alpha)}$ only differ by the top-right hand entry. Be careful not to get confused here: the k index counts the number of iterations converging towards the solution of the all-at-once problem, and is nothing to do with the time index.

It can be shown (you may assume this) that this iteration will converge for sufficiently small α. Show that if it does converge, then U^k converges to the solution of (13) as $k \to \infty$.

(c) You might ask: what is the point of applying this iterative scheme given that we could just solve (13) directly? The point is that with a clever change of variables, we can make (17) into a block diagonal system, and each block can be solved independently (and in parallel, in principle, but we won't make a parallel implementation).
To derive the change of variables, show that that

$$\begin{pmatrix} 1 \\ \alpha^{-1/M} e^{\frac{2\pi i j}{M}} \\ \alpha^{-2/M} e^{\frac{4\pi i j}{M}} \\ \vdots \\ \alpha^{-(M-1)/M} e^{\frac{2(M-1)\pi i j}{M}} \end{pmatrix} \tag{19}$$

is an eigenvector of both $C_1^{(\alpha)}$ and $C_2^{(\alpha)}$, for $j = 0, 1, 2, \ldots, M-1$. Hence show that

$$C_1^{(\alpha)} \otimes I + C_2^{(\alpha)} \otimes B = (V D_1 V^{-1}) \otimes I + (V D_2 V^{-1}) \otimes B = (V \otimes I)\,(D_1 \otimes I + D_2 \otimes B)\,(V^{-1} \otimes I), \tag{20}$$

for $M \times M$ matrices V and D_1.

(d) Given a vector U, show that $(V \otimes I)U$ and $(V^{-1} \otimes I)U$ can be computed by independently applying a composition of discrete Fourier (and inverse Fourier) transforms with multiplication by certain diagonal matrices to each time slice of U.

(e) Hence show that we can solve (17) by the following steps:

i. Compute $\hat{R} = (V \otimes I)R$.

ii. Solve

$$(d_{1,k} I + d_{2,k} B) \begin{pmatrix} \hat{p}_k \\ \hat{q}_k \end{pmatrix} = \hat{r}_k, \tag{21}$$

where $\hat{p}_k, \hat{q}_k$ is the k-th time slice of $\hat{U}$, for $k = 1, 2, \ldots, M$.

iii. Compute $U = (V^{-1} \otimes I)\hat{U}$.

(f) Implement this algorithm in code. (Do not attempt to make a parallel code as this is beyond the scope of the course.) To implement steps 1 and 3, note that you can use `reshape` to transform an NM dimensional array into an $N \times M$ dimensional array, and then you can use `numpy.fft.fft` and `numpy.fft.ifft` to apply the forward and inverse discrete Fourier transform (respectively) simultaneously to each row (or

column). Beware that the FFT and IFFT are not normalised, so applying FFT and then IFFT produces a scaling of the original array which you will need to correct afterwards. To implement step 2, you should follow the method of the previous section, namely eliminating $\hat{p}_k$ to get a nearly tridiagonal system for $\hat{q}_k$ which you can solve using the code that you wrote there (you may need to revisit it to ensure that it works for complex-valued arrays). Present some results and provide some tests that demonstrate that the code works.