AMATH740/CM770/CS770
Fall 2020: Computational Assignment A

Instr.: Hans De Sterck e-mail: *hdesterck@uwaterloo.ca* Office hrs: Wed 9:00-9:45am, Thu 11am-12
TA.: Mohammad Aali e-mail: *mohammad.aali@uwaterloo.ca* Office hrs: Tue 11am-12, Thu 9-10am

Assignment due date: Friday October 9, 5:00pm (Waterloo time), Crowdmark (for the answer package) and LEARN (for the program code files)

Submission instructions: Dual submission

- **LEARN submission**: Please submit your program code files to the Dropbox on LEARN (only the program code files; no program output or answers to questions).
- **Crowdmark submission**: Please submit the answers to each assignment question on Crowdmark, including written or typed answers to the questions asked, with relevant computer output (plots, tables or number output) as requested in the assignment questions. Please also include, for each question, a pdf file or screenshot with your program code (to facilitate TA feedback on your code).
- one common submission per group
- MATLAB is the recommended computer language, but you can choose a different language like python or Julia or C or C++ (contact the instructor if you want to choose a language not listed before; Maple or Mathematica are not suitable).
- The questions are written assuming you use MATLAB, and some short MATLAB code fragments are provided on LEARN for download, but it should not be difficult to translate this to another language like python, and I don't think there should be much extra work when using another language; let me know if you encounter issues or have questions about this
- there are some potentially useful tools for programming in teams that you may consider to use, such as version control tools like git (e.g., you can create a git repository for your group on github or bitbucket) (but the computer code you will write for the programming questions will generally not be long or complicated, so it is by no means necessary to use that kind of tools); collaboration tools like slack could also be useful (or you can communicate and share files via tools from Google, or MS Teams, etc.)

1. LU decomposition. (10 marks)
   Implement each of the following linear algebra algorithms in MATLAB using only primitive commands (*i.e.* do not appeal to the built-in linear algebra functionality in MATLAB). You should use the function header given for each below and submit the code for each in a seperate m-file to the LEARN drop box on the course webpage. Also, include a copy of your code in the assignment submission on Crowdmark.

   (a) LU factorization. When implementing your algorithm, do not worry about pivoting but check for a zero pivot element and stop. You should return the matrices L and U stored in one matrix T (the diagonal of L is not stored; you know it consists of all ones). You are allowed to use vectorization of the inner loop (or the two innermost loops) to make your code run faster for large matrices.

```
function T = LUFactorization(A)
% Usage: T = LUFactorization(A)
% Compute the LU factorization of matrix A
% and return the resulting factorized matrix.
```

(b) Forward Substitution. This function should take as input the matrix output using LU-Factorization from part a).

```
function y = ForwardSubstitution(T, b)
% Usage: y = ForwardSubstitution(T, b)
% Perform forward substitution using the unit
% lower triangular portion of the matrix T.
```

(c) Backward Substitution. This function should take as input the matrix output using LUFactorization from part a).

```
function x = BackwardSubstitution(T, y)
% Usage: x = BackwardSubstitution(T, y)
% Perform backward substitution using the
% upper triangular portion of the matrix T.
```

Please download `VerifyLU.m` from the course website and report on the output you obtain.

2. Tridiagonal solver. (20 marks)
Consider the following tridiagonal matrix

$$A_t = \begin{bmatrix} b_1 & c_1 & 0 & & \cdots & 0 & 0 \\ a_2 & b_2 & c_2 & & & 0 & 0 \\ 0 & a_3 & b_3 & c_3 & & 0 & 0 \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & \\ 0 & 0 & 0 & & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & 0 & 0 & & & a_n & b_n \end{bmatrix} \in \mathbb{R}^{n \times n}$$

which comes up in problems such as spline interpolation, or solving ordinary differential equations that model elastic beams or electric potentials.

(a) When A is tridiagonal (and assuming that pivoting is not required), the L and U factors in the decomposition $A_t = L\,U$ take on the form

$$L = \begin{bmatrix} 1 & 0 & & \cdots & 0 & 0 \\ l_2 & 1 & 0 & & 0 & 0 \\ 0 & l_3 & 1 & & & 0 \\ & & \ddots & \ddots & & \\ 0 & 0 & & & 1 & 0 \\ 0 & 0 & & & l_n & 1 \end{bmatrix} \quad U = \begin{bmatrix} d_1 & u_1 & & \cdots & 0 & 0 \\ 0 & d_2 & u_2 & & & 0 \\ 0 & 0 & d_3 & u_3 & & \\ & & & \ddots & \ddots & \\ 0 & 0 & & & d_{n-1} & u_{n-1} \\ 0 & 0 & & & 0 & d_n \end{bmatrix}$$

Derive the recurrence relations for l_i , d_i, and u_i in terms of a_i, b_i and c_i. (It may be helpful to consider the cases $i = 1$ and $i = n$ separately from the general case $2 \leq i \leq n-1$.) (Note that this is equivalent to a banded LU solver with bandwidth 1, but you are asked here to derive these formulas explicitly from scratch, and implement them directly in the next question part.)

(b) Implement the algorithm you have derived in (a) into a MATLAB file `TriDiagonalSolve.m`, for solving $A_t\vec{x} = \vec{f}$. Use only primitive commands (*i.e.* do not appeal to the built-in linear algebra functionality in MATLAB). Use the function header given below. The file should first compute the L and U factors, and should then perform forward and backward substitution. (Note: Don't use standard forward and backward substitution code, because that will be too expensive in this case!)

```
function x = TriDiagonalSolve(a, b, c, k)
% Usage: x = TriDiagonalSolve(a, b, c, k)
% Completely solve the system Ax = k, where
% A is a tridiagonal matrix.  The column vectors
% a, b, c describe the diagonal entries of the
% matrix.  a(1) and c(n) are ignored here.
```

(c) Test the accuracy of your implementation in `TriDiagonalSolve.m` using `VerifyTriDiagonalLU.m`, which you can donwload from LEARN. Here, `VerifyTriDiagonalLU.m` compares the result with MATLAB's built-in solver. Report the difference between the MATLAB solution and your solution.

(d) What is the asymptotic computational cost of your algorithm for large problem size n? Explain.

(e) Apply the algorithm derived above to the finite difference discretization of the BVP

$$\begin{aligned} u'' &= f(t) \\ u(0) &= 0 \\ u(1) &= 0 \end{aligned} \tag{1}$$

with

$$f(t) = 16\pi\cos(8\pi t^2) - 256\pi^2 t^2 \sin(8\pi t^2), \qquad t \in [0,1]. \tag{2}$$

The unique exact solution to this problem is given by $u(t) = \sin(8\pi t^2)$. By approximating the exact solution u to this system at a discrete number of points, we construct an approximation $\{v_i\}_{i=0}^{N+1}$ such that $v_0 = u_0 = 0$ when $t_0 = 0$ and $v_{N+1} = u_{N+1} = 0$ when $t_{N+1} = 1$. The step size is $h = 1/(N+1)$.

Write a MATLAB script `CompareBVP.m` that compares the solutions and the execution times for solving the linear system using the full LU decomposition from Question 1, and the specialized tridiagonal solver from this question, producing the following output:

(e1) For N =20 interior points, make a plot that shows the approximate solutions obtained by the full LU and the tridiagonal LU solvers (they should essentially be the same), compared with the exact solution (plotted with, for example, 1000 points). Put the three curves in the same plot so they can be compared easily.

(e2) Provide tables showing the execution times of the full LU decomposition from Question 1, and the specialized tridiagonal solver from this question, for different values of N. (You may use `tic` and `toc` to measure elapsed time. Use `format long` to see a sufficient amount of digits in the number format.) For LU, you can use $N = 100, 200, 400, 800, 1600$ (or larger numbers, increasing by a factor of 2 every time, if you have used vectorization in your LU code which may give faster execution times). (Note also: in MATLAB, do NOT use sparse storage format for the 1D

model problem matrix, because your LU code is implemented assuming a dense matrix, and sparse format would make it run much slower due to the increased cost of memory access for the sparse format.) For the tridiagonal solver, use problem sizes $N = 1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000, 256000, 512000$. (Make sure to not actually form the matrix, because you will run out of memory!)

(e3) For each of the two methods, include a table column where you show the ratio of the compute time between problem size N and $N/2$. What do you expect this ratio to be, asymptotically for large N, for each of these methods? Comment on whether you see this confirmed in your performance tests. (Note that you may not really get the expected ratios, even for the large problem sizes, due to variations in the load of your computer, and due to runtime code optimizations that may be performed by the software. Still, you should be able to see that the ratios you get for the tridiagonal solver are much smaller than for full LU, roughly in line with the difference between the theoretical asymptotic ratios.)

Make sure to submit all code files on LEARN, and to include a pdf or screenshot copy of your code in the assignment submission on Crowdmark.

3. 2D model problem. (20 marks)

(a) Implement a MATLAB function with header

```
function A=build_laplace_2D(N)
```

that constructs a sparse matrix `A` containing the 2D Laplacian matrix as defined in class and in the lecture notes. Here, `N` is the number of interior points per direction, i.e., the number of rows and columns of `A` is $n = N^2$.
Test your program using `A=build_laplace_2D(5)` and `spy(A)`, and submit the spy plot as part of your submission.
Programming notes:

- Use a sparse matrix to store `A`. For example, some of the MATLAB commands `sparse, speye, spdiags` could be useful. (Use the `help` command to learn about them.)
- Don't use the MATLAB Kronecker product command `kron` here, but use more elementary MATLAB commands to put the $N \times N$ blocks into `A`.

(b) Write a MATLAB script `laplace_zeroBC.m` to solve the PDE boundary value problem

$$\text{BVP} \quad \begin{cases} \dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} = -20\pi^2 \sin(2\pi x)\sin(4\pi y) \\ (x, y) \in \Omega = [0, 1] \times [0, 1] \\ u(x, y) = 0 \quad \text{on } \partial\Omega. \end{cases}$$

The exact solution of this problem (with zero boundary conditions) can be found in closed form and is given by

$$u(x, y) = \sin(2\pi x)\sin(4\pi y).$$

Use the MATLAB function `build_laplace_2D.m` from part (a) to build the matrix A, or you can download and use the simpler function `build_laplace_2D_kron.m` from LEARN

(which generates the 2D Laplacian matrix using a Kronecker product approach) if you did not get `build_laplace_2D.m` to work.

Test your program using $N = 32$ interior points in each direction, and submit `mesh` plots of the exact solution, the approximate solution, and the difference between the exact and approximate solution in each grid point.

Programming notes:

- Since the boundary conditions are zero, the boundary values do not need to be added to the RHS of the linear system.
- Make sure to consistently use *row lexicographic ordering*, e.g., when putting the RHS of the PDE into the RHS vector of the linear system (don't forget the h^2 factor), or when taking the solution vector of the linear system and turning it into a matrix corresponding to the grid points for plotting with the `mesh` command. (You can also use the `reshape` command to reshape a lexicographically ordered vector into a matrix.)
- Once you have set up the matrix and the RHS of the linear system, you can use MATLAB's built-in *backslash* operator to solve the linear system: solve the system $A\vec{v} = \vec{f}$ using `v=A\f`. (Or you can use your LU solver from Question 1, even though that may be much slower.)
- Unknowns for the boundary points, with values zero, are not included in your linear systems, but it is best to add these zero values to the arrays that you use to plot the results, such that the zero boundary values appear in the result plots.
- To generate the grids for plotting, for example, the exact solution, you can use this sequence of commands: `x=(0:h:1); y=(0:h:1); [X Y]=meshgrid(x,y); u=(X-X.*X).*(Y-Y.*Y); mesh(X,Y,u)` (after defining $h = 1/(N+1)$). (Here `X` and `Y` are matrices of size $(N+2) \times (N+2)$ that contain the x and y coordinates of the grid points, respectively.)

(c) Write a MATLAB script `laplace_heat.m` that uses your MATLAB function `build_laplace_2D.m` to solve a heat conduction PDE boundary value problem (similar to the 2D problem from Section 1.2 in the course notes, but some of the parameters are a bit different):

$$\text{BVP} \quad \begin{cases} \dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} = -g(x,y) \\ (x,y) \in \Omega = [0,1] \times [0,1] \\ u(x,y) = u_0 \quad \text{on } \partial\Omega, \end{cases}$$

with $u_0 = 300$ Kelvin, and with heat source

$$g(x,y) = 5{,}000 \exp\left(-\frac{(x-1/4)^2 + (y-3/4)^2}{0.01} \right).$$

Provide a `mesh` plot and a `contour` plot of your approximate solution with $N = 64$ (to be compared with the plots in the course notes).

What is the maximum temperature in your solution? (Provide the value with at least 10 digits of accuracy; use the `format long` and `max` commands.)

Programming notes:

- The difference with (b) is that you now have to add the boundary values to the RHS of the linear system.

– You can first try this with the parameters from Section 1.2 in the course notes, to see if you get the same solution as in the notes.

Make sure to submit all code files on LEARN, and to include a pdf or screenshot copy of your code in the assignment submission on Crowdmark.

4. Conditioning of linear systems. (10 marks)

(a) Let $A \in \mathbb{R}^{n\times n}$ be an orthogonal matrix (i.e, $AA^T = I$). Show that

$$\kappa_2(A) = \|A\|_2 \, \|A^{-1}\|_2 = 1.$$

(This is very short.)

Write a MATLAB script `Matrix_conditioning.m` that does the computations for the following questions.

(b) Consider linear system

$$A\vec{x} = \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix},$$

the solution of which is the intersection in the 2-dimensional plane of the two lines

$$\begin{aligned} x - y &= 1 \\ x + y &= 1, \end{aligned}$$

or

$$\begin{aligned} y &= x - 1 \\ y &= 1 - x, \end{aligned}$$

which is clearly the point $[1 \;\; 0]^T$. Is the matrix orthogonal? (Note that the two lines are orthogonal!) Conclude about $\kappa_2(A)$. Is the matrix well-conditioned? The RHS vector determines where the lines intersect the x and y axis. Compute the solution of the system if you slightly perturb the RHS to $[1 + 10^{-3} \;\; 1]^T$. (This is a small perturbation relative to the size of the RHS.) Interpret the size of the change in solution relative to the size of the change in RHS (compute the relative condition number for this problem and perturbation in the 2-norm), and relate this to the 2-norm matrix condition number.

(c) Consider linear system

$$B\vec{x} = \begin{bmatrix} 1 & -1+\delta \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

with $\delta = 10^{-10}$, the solution of which is the intersection of the two lines

$$\begin{aligned} x + (-1+\delta)y &= 1 \\ x - y &= 1, \end{aligned}$$

which is also clearly the point $[1 \;\; 0]^T$. Is the matrix orthogonal? (Note that the two lines are almost parallel now ...) Compute the inverse of B and $\kappa_2(B)$ using MATLAB's `inv` and `cond`. Note that B^{-1} has very large matrix elements. Is B well-conditioned?

Compute the solution of the system if you slightly perturb the RHS to $[1 + 10^{-3} \quad 1]^T$. (This is a small perturbation relative to the size of the RHS.) Interpret the size of the change in solution relative to the size of the change in RHS (compute the relative condition number for this problem and perturbation in the 2-norm), and relate this to the 2-norm matrix condition number.

Make sure to submit all code files on LEARN, and to include a pdf or screenshot copy of your code in the assignment submission on Crowdmark.

5. Determining the base of a floating point number system. (10 marks)
Download the MATLAB function `determine_b.m` from LEARN. Verify by running the code that this function correctly determines the base of the floating point number system (with rounding) on the computer you use. What is the correct base? (This is an obvious question, of course.)

   Then explain how the algorithm works by answering the following questions.

   Consider the steps the algorithm would take to determine the base of floating point number system $F(\beta = 10, t = 2, L = -5, U = 5)$. Assume that your 'decimal' computer uses rounding-to-nearest, tie-to-even.

   (1) Give the successive values that the exact a and the rounded $\bar{a} = \texttt{fl}(a)$ would take on in the first phase of the algorithm. What is the event that triggers the end condition for the first phase?
   (2) Write down the values that i and $\bar{a} + i$ and $\texttt{fl}(\bar{a} + i)$ would take on in the second phase. (Note: we have assumed a rounding rule that would round 135 (which lies exactly in the middle between 130 and 140) to 140.) What is the event that triggers the end condition for the second phase?
   (3) Write down the final value of $\texttt{fl}(\bar{a} + i)$, $\bar{a}$ and b.

   (No need to submit any code for this question.)

6. QR decomposition of $A \in R^{m \times n}$ by Gram-Schmidt and Householder. (20 marks)

   (a) Download the file `myGramSchmidt.m` from LEARN; it contains an implementation of the classical Gram-Schmidt algorithm (Algorithm 3.2).
   Make a modified version `myGramSchmidtMod.m` that implements the more stable modified Gram-Schmidt algorithm (Algorithm 3.3). (It is, indeed, just a tiny change.)
   (b) Implement the QR algorithm using Householder reflections in MATLAB according to the pseudocode discussed in class. Your implementation `myHouseholder.m` should return the full versions of the factors, *i.e.* $Q \in R^{m \times m}$ and $R \in R^{m \times n}$. You should use the function header given below and submit the code to the LEARN drop box on the course webpage. Also, include a pdf or screenshot image of your code in the assignment submission on Crowdmark.

```
function [Q,R] = myHouseholder(A)
% Usage: [Q,R] = myHouseholder(A)
% Compute the QR factorization of matrix A using
% Householder reflections and return the resulting
% factors Q and R.
```

(c) Make a script `test_QR_1.m` that compares the three versions:

```
[Qgs,Rgs]=myGramSchmidt(A);
[Qmodgs,Rmodgs]=myGramSchmidtMod(A);
[Qhouse,Rhouse]=myHouseholder(A);
```

for a random matrix

```
A=rand(1000,500);
```

Compare the orthogonality and the accuracy by displaying

```
norm(Q'*Q-eye(n))
norm(A-Q*R)
```

for each version. (Include this result in your assignment answer with a brief discussion.) You will see that the computed Q factors are nicely orthogonal for all three methods, and QR accurately approximates A.

(d) Now generate the following *Vandermonde* matrix:

```
m=100;
n=15;
t=(0:m-1)'/(m-1);
A=[];
for i=1:n
    A = [A t.^(i-1)];
end
```

This matrix arises in the context of polynomial interpolation and is notoriously ill-conditioned. Its condition number (the ratio of the largest singular value to the smallest) is about $2.2\,10^{10}$. Make a new script `test_QR_2.m` that modifies `test_QR_1.m` to repeat the tests from (c) for this matrix. You will see that the classical Gram-Schmidt algorithm loses all orthogonality, the modified version is substantially better, and the Householder version maintains orthogonality close to machine precision (also have a look at the actual matrix elements of `Q'*Q` to see how many digits of accuracy they retain for each method; no need to report on this). Interestingly, though, all methods are still accurate for $A - QR$. (Understanding this fully is quite complicated and the ramifications of this are quite involved; see, e.g., the book "Numerical linear algebra" by Trefethen and Bau p. 67, p. 115, p. 137.)

Make sure to submit all code files on LEARN, and to include a pdf or screenshot copy of your code in the assignment submission on Crowdmark.