AM 213A, Winter 2024
Final Coding Project (100 points)

Posted on Tue, Mar 12, 2024
Due 11:59 pm, Tue, Mar 19, 2024

Submit your work to Canvas (not to Gradescope)

- Use LaTeX or MS-words like text editors for the project. A scanned copy of handwritten solutions will be acceptable on an exceptional case-by-case only with permission from the instructor.
- Your report needs to have relevant discussions on each problem to describe what you demonstrate. In this coursework, do not simply copy and paste any screen outputs (e.g., no screenshots) from your code execution and provide them as answers. Instead, discuss the code results required for each problem and display them concisely with logical justification. For all coding problems, showing screen outputs only from your code execution is insufficient and will lose points.
- To disprove, you need to provide a counter-example.
- All homework submissions should meet the deadline. Late homework will be accepted under emergencies with permission from the instructor.
- Submit all codes and reports using the following naming conventions:
  - Report: All written solutions should be together in one single PDF named as `LastnameFirstname_Report_finalProject.pdf`
  - Codes: The supporting codes for *each* problem should be provided in a single compressed file. As there are two problems (SVD and iterative methods), prepare two sets of compressed files, such as
    - `LastnameFirstname_Code_finalProject_SVD.tar.gz` (or `.zip`)
    - `LastnameFirstname_Code_finalProject_Iterative.tar.gz` (or `.zip`)
    - Include only source files (e.g. `*.f90`, `*.c`, etc.) and the needed Makefile
    - Do **not** include object files, module files, executables, or data files.

Project Descriptions

There are two coding problems, (a) SVD and (b) iterative methods. For each problem, specify the inputs and outputs of your codes in your report. Describe the behaviors of your codes sufficiently in each problem. Clarify what parts of your code implementations correspond to each individual problem.

IMPORTANT: Any code submissions *without* a Makefile will not be graded.

(a) SVD for image compression (50 points)

Download the black-and-white image file `dog_bw_data.dat` and implement Fortran routines to compress the given image to lower resolutions using SVD. You will need to implement separate plotting routine(s) to visualize both the original and compressed images using either MATLAB or Python.

The data stored in `dog_bw_data.dat` is in double-precision ASCII format, with the values ranging from 0 (maps to black) to 255 (maps to white). The main goal is to conduct the following sequence of tasks:

1. Read-in `dog_bw_data.dat` as input from your Fortran/C routine. Call `dgesvd` routine in the LAPACK linear algebra library to perform a series of singular value approximations $\mathbf{A}_{\sigma_k}$ of the original data $\mathbf{A}$ stored in `dog_bw_data.dat`, i.e., the data in `dog_bw_data.dat` is treated as an $m \times n$ matrix $\mathbf{A}$. The pixel size, measured as "width $\times$ height," of the original image is 1920×1279, which should be transposed in linear algebra to conform the conventional row $\times$ column dimension that gives $m \times n = 1279 \times 1920$. It is, though confusing, in actual implementation depending on the column-major (e.g., Fortran) vs. row-major (e.g., C) formats. At this point, students should read `ref_8.pdf` under "Other Resources" for more details. The rank of $\mathbf{A}$ should be at most 1279 mathematically.

   Important Tip: One easy way to avoid confusion in Fortran is to read in the `dog_bw_data.dat` data as a matrix of size 1920×1279 instead of 1279×1920, and perform the SVD to get the results. When you do this, your data of the low-rank approximations would appear as transposed matrix data, in which case you can simply rotate your figures once plotted, or transpose the approximation data before plotting.

2. Given $\mathbf{A}$, call `dgesvd` to compute the singular value decomposition of $\mathbf{A}$,

$$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T, \tag{1}$$

where $\mathbf{A} \in \mathbb{R}^{m\times n}$, $\mathbf{U} \in \mathbb{R}^{m\times m}$ and $\mathbf{V} \in \mathbb{R}^{n\times n}$ are the left and right singular matrices, respectively. The full singular value matrix Σ is given as

$$\Sigma = \begin{pmatrix} \sigma_1 & & & & & & \\ & \sigma_2 & & & & & \\ & & \ddots & & & & \\ & & & \ddots & & & \\ & & & & \sigma_r & & \\ & & & & & \ddots & \\ & & & & & & 0 \end{pmatrix}. \tag{2}$$

To do this, you will first need to install LAPACK on your machine. The detailed instruction is available in the AM 129/209 online lecture note from F22. Go to the section "External Libraries for Scientific Computing" to learn how to install the library.

For those of you who have been using C, take a look at the past email communication (see the document `ref_1.pdf` under "Other Resources" on the Final Project page on Canvas) on useful guidelines and tips provided by TA Ian May last year.

There are other useful resources as well, including some plotting issues (`ref_4.txt`), the different conventions on how to refer to image sizes in the digital community vs. array sizes in linear algebra (read `ref_8.pdf`), and how to run LAPACK on WSL (`ref_3.pdf`) and Hummingbird (`ref_7.pdf`). These are all very useful; read them as needed.

To successfully implement a driver routine that calls `dgesvd`, it is necessary to learn the structure of the routine. In particular, pay careful attention to the data types of the input and output arguments. You are welcome to use any resources (online or offline) to study them. One reference is:

- Ref. 1

Note: some may encounter an error message, saying *"Warning: Procedure 'dgesvd' called with an implicit interface at (1) [-Wimplicit-interface].* It is optional to remove this warning message, but in case this annoys you, you can include an interface block for `dgesvd` immediately after `implicit none` declaration. Make sure you first use appropriate debugging flags.

3. Reconstruct eight separate compressed (or low-rank approximated) images (i.e., $\mathbf{A}_{\sigma_k}$, $k = 10, 20, 40, 80, 160, 320, 640, 1279$). Each k-SVD approximation is defined by

$$\mathbf{A}_{\sigma_k} = \mathbf{U}\Sigma_{\sigma_k}\mathbf{V}^T, \tag{3}$$

where each $\mathbf{A}_{\sigma_k} \in \mathbb{R}^{m\times n}$ is a newly reconstructed compressed data from the original data $\mathbf{A}$. Each singular value matrix $\Sigma_{\sigma_k} \in \mathbb{R}^{m\times n}$ is a reduced

matrix of the full diagonal matrix Σ containing only up to the first k largest singular values with $k \leq r = \text{rank}(\mathbf{A}) = 1279$, given as

$$\Sigma_{\sigma_k} = \begin{pmatrix} \sigma_1 & & & & & & \\ & \sigma_2 & & & & & \\ & & \ddots & & & & \\ & & & \sigma_k & & & \\ & & & & 0 & & \\ & & & & & \ddots & \\ & & & & & & 0 \end{pmatrix}. \tag{4}$$

Also compute the case with $k = 1279$, which should correspond to a full-rank approximation. In Problem 6 below, you will confirm the full-rankness with $k = 1279$ by checking if its error estimation in Eq. (5) is an order of machine accuracy.

4. Save each of the eight data $\mathbf{A}_{\sigma_k}$ in an ascii format data file. Name your data file with reasonable file names such as `Image_appn_1xxxxx.dat` where the file index "`1xxxxx`" reflects the number of singular value used in each calculation, e.g., `100640` for $\Sigma_{\sigma_{640}}$.

5. Implement a plotting routine using either MATLAB or Python to visualize the compressed data. A good example of computing a similar image compression and displaying results in MATLAB is available here. Note that you should convert your double-precision data to an unsigned 8-bit data type (e.g., `uint8` in the MATLAB example) to properly display the results. Displaying data in double-precision in plotting will result in overflow errors.

6. Compute the corresponding errors of each case using the averaged Frobenius norm, $||\cdot||_F$,

$$E_k = \frac{||\mathbf{A} - \mathbf{A}_{\sigma_k}||_F}{mn}, \tag{5}$$

save the results to an ascii file, e.g., "`errors.dat`", and plot the results using MATLAB or Python as a function of the number of singular values.

In your PDF document:

- Report the first 10 largest singular values from σ_1 to σ_{10}. Also report the rest singular values of σ_k for $k = 10, 20, 40, 80, 160, 320, 640, 1279$.
- Display all eight compressed images along with the original image. Identify figures clearly with the number of singular values used for each.
- Discuss what you see as you increase/decrease the number of singular values in calculations. Among the eight selected k values, report at which k value you obtain an error lower than 10^{-3}.

(b) Iterative methods (50 points)

(b.1) Gauss-Jacobi and Gauss-Seidel (25 points)

In your `LinAl.f90` module, create

- a routine that solves the equation $\mathbf{Ax} = \mathbf{b}$ for a given $m \times m$ matrix $\mathbf{A}$, and an $m-$long vector $\mathbf{b}$ using the Gauss-Jacobi algorithm.
- a routine that solves the equation $\mathbf{Ax} = \mathbf{b}$ for a given $m \times m$ matrix $\mathbf{A}$, and an $m-$long vector $\mathbf{b}$ using the Gauss-Seidel algorithm.

Both routines should take as argument the required accuracy, and return the solution $\mathbf{x}$ once $||\mathbf{b} - \mathbf{Ax}||_2$ is smaller than the required accuracy. You are otherwise free to select the argument list, but please document it thoroughly. Within each routine, you need to print to a file the error $||\mathbf{b} - \mathbf{Ax}||_2$ at each iteration. Then, create a program that

- generates a matrix $\mathbf{A}$ full of ones, except on the diagonal where $a_{ii} = D$ where D is a value entered by the user (either as argument to the program, or as a prompt to the user),
- generates a vector $\mathbf{b}$ such that $b_i = i$,
- asks the user whether to use Gauss-Jacobi or Gauss-Seidel, and finally
- calls the right routine to solve $\mathbf{Ax} = \mathbf{b}$, with accuracy 10^{-5}, and prints the answer to the screen.

In your PDF document:

- Explain briefly what the Gauss-Jacobi and Gauss-Seidel algorithms do, how they differ, and what the convergence criterion for the algorithm is.
- Run the code for a 10×10 matrix $\mathbf{A}$ with $D = 2$, $D = 5$, $D = 10$, $D = 100$ and $D = 1000$. For each value of D where the algorithm converges, produce a figure showing on a log-linear plot, $||\mathbf{b} - \mathbf{Ax}||_2$ as a function of the iteration number (i.e. by plotting the file that you saved) for both Gauss-Jacobi and Gauss-Seidel (on the same plot).
- Discuss why some of the cases didn't converge, and compare the convergence rates of the two algorithms in light of the theoretical predictions discussed in the lectures.
- Finally modify the program so that $\mathbf{A}$ is the matrix $\mathbf{A}$ full of ones, except on the diagonal where $a_{ii} = i$. Run the program for a 10×10 matrix $\mathbf{A}$. Do either of the algorithms converge?

(b.2) Conjugate Gradient algorithm (25 points)

In your `LinAl.f90` module, create the smart conjugate gradient algorithm without pre-conditioning to solve the problem $\mathbf{Ax} = \mathbf{b}$. The argument list should contain

- The input matrix $\mathbf{A}$ and input vector $\mathbf{b}$, as well as their dimension m (inputs).
- The desired accuracy (input).
- The solution (output).

The routine should check that the input matrix is symmetric. Modify your

program from **(b.1)** as follow:

- Go back to using D as the diagonal elements.
- Add an option to use the Conjugate Gradient as the solver.

In your PDF document:

- Explain briefly what the Conjugate Gradient algorithm does and why it is guaranteed to converge in a finite number of iterations.
- Prove that the smart conjugate gradient algorithm is equivalent to the basic conjugate gradient algorithm (see notes).
- Run the code using the CG algorithm for a 10×10 matrix $\mathbf{A}$ with $D = 2$, $D = 5$, $D = 10$, $D = 100$ and $D = 1000$. What happens this time? Make a table comparing the number of iterations until complete convergence between the 3 algorithms, for each value of D. Explain the trends that you see.
- Explain why the diagonal pre-conditioner is not very useful for these matrices.
- Finally modify the program so that $\mathbf{A}$ is the matrix $\mathbf{A}$ full of ones, except on the diagonal where $a_{ii} = i$. Run the program for a 10×10 matrix $\mathbf{A}$, and a 100×100 matrix. In each case, does the CG algorithm converge? If yes, in how many iterations?
- Implement the diagonal pre-conditioning, and re-do the last bullet point question above. How much faster is the convergence this time?