CS/MATH 375, Fall 2021 — HOMEWORK # 6
Due : Oct. 5 at 6pm on UNM Learn

Instructions

– **Report:** In general, your report needs to read coherently. That is, start off by answering question 1. Fully answer the question, and provide all the information needed to understand your answer. If Matlab code or output is part of the question, include that code or output (e.g., screenshot) alongside your narrative answer. If discussion is required for a question, include that. *Your report should include your Matlab scripts, code output, and any figures.*

– **What to hand in:** Submission must be one **single PDF** document submitted on UNM Learn.

  Overall, your report is your narrative explanation of what was done, your answers to the specific questions, and how you arrived at your answers. If discussion is required for a question, include that. *Your report should include your Matlab scripts, code output, and any figures.*

– **Partners:** You are strongly encouraged to **work in pairs**. If you work with a partner, only one partner should submit a homework, but write both collaborators' names at the top. Groups of more than 2 students are not allowed.

– **Typesetting:** If you write your answers by hand, then make sure that your handwriting is readable. Otherwise, I cannot grade it.

– **Plots:** All plots/figures in the report must be generated in Matlab or Python and not hand drawn (unless otherwise specified in the homework question).

  In general, make sure to (1) title figures, (2) label both axes, and (3) include a legend for the plotted data sets. The font-size of all text in your figures must be large and easily readable.

– **Generating PDFs:** See the UNM Learn homework page for tips.

Iterative Solvers for the Poisson Equation

The Poisson equation, modeling a large number of physical phenomena, is

$$\begin{aligned} -\nabla \cdot K \nabla u(\mathbf{x}) &= f(\mathbf{x}) && \mathbf{x} \in \Omega, \\ u(\mathbf{x}) &= g(\mathbf{x}) && \mathbf{x} \in \partial\Omega, \end{aligned}$$

where $u(\mathbf{x})$ is the solution, $\Omega \subset \mathbb{R}^d$, $\partial\Omega$ is the boundary of Ω, K is an SPD $d \times d$ matrix and g is a function specifying the boundary conditions. For this homework, we will use the built-in Matlab function `gallery` to generate matrices associated with a finite difference approximation of the Poisson equation in 2D with $K = \mathbb{I}$. To create a matrix, define P to be a positive integer, e.g.,

```
>> P = 10;
```

and then type

```
>> A = gallery('poisson',P);
```

Note that A is a sparse, banded SPD $n \times n$ matrix where $n = P^2$ (P denotes the number of sub-intervals in each spatial dimension). For the right hand side, we will simply use a vector of ones:

```
>> b = ones(P^2,1);
```

The problems involving Matlab will use these methods to construct A and b.

---

1. In class, we studied the "smart" version of the Jacobi and Gauss-Siedel algorithms. Let

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & & \cdots & a_{nn} \end{bmatrix}, \quad x^{(k)} = \begin{bmatrix} x_1^{(k)} \\ x_2^{(k)} \\ \vdots \\ x_n^{(k)} \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

   Explain how the component-wise update formula for Jacobi given on the course slides is equivalent to the vector-update version. That is, for the Jacobi algorithm show that the vector-update version

$$x^{(k)} = D^{-1}(C_L + C_U)x^{(k-1)} + D^{-1}b$$

   is the same as the component-wise update,

$$x_i^{(k)} = -\sum_{j=1, j\neq i}^{n} \left(\frac{a_{ij}}{a_{ii}}\right) x_j^{(k-1)} + \frac{b_i}{a_{ii}}$$

   for $1 \leq i \leq n$.

2. *Jacobi Algorithms*

   (a) Write a Matlab function, `function x = my_jacobi(A,b, tot_it)`, which takes as input the matrix A, the vector b, number of iterations `tot_it` and outputs the Jacobi solution after `tot_it` iterations. For this part, code up the component-wise version of the "Smart Jacobi Algorithm".

(b) Now create another function, `function x = my_vector_jacobi(A,b, tot_it)`, which also takes as input the matrix A, the vector b, number of iterations `tot_it` and outputs the Jacobi solution after `tot_it` iterations. For this part, code up the vector-update version of the "Smart Jacobi Algorithm". Here, we are willing to sacrifice some memory for the sake of speed, so go ahead and extract D and use "backslash" for D^{-1}. Verify that you get the same residuals and errors using the component-wise and vector-update versions with $P = 20$ and 100 iterations.

Hint: Use the `diag` function in Matlab to extract the main diagonal of the matrix: `diag(diag(A))`.

(c) Which version of the Jacobi algorithm is faster? Compare the time it takes to perform 100 iterations with $P = 20$.

(d) Now, we will investigate the error in the Jacobi solution by varying the size of the linear systems, and keeping the number of Jacobi iterations fixed at 100. We compare the error in the Jacobi solution with the "true solution" obtained from the Matlab "backslash" operator. Make a table showing how the 2-norm in the relative error in the Jacobi solution,

$$\frac{\|x_{\text{true}} - x\|_2}{\|x_{\text{true}}\|_2},$$

varies with increasing size of the linear system. Use $P = 10, 20, 40, 80, 160$ which corresponds to $n = 100, 400, 1600, 6400, 25600$. Use whichever version (component-wise or vector-update) of Jacobi is faster. Does the relative error increase or decrease as the system size increases?

3. *Comparing Algorithms*

(a) Create a function, `function x = my_gauss_seidel(A,b, tot_it)`, which also takes as input the matrix A, the vector b, number of iterations `tot_it` and outputs the Gauss-Seidel solution after `tot_it` iterations. You may choose either the component-wise or the vector-update of the "smart version" version.

Hint: If you code the vector-update version, you can use `triu` to get the upper triangular portion and `tril` to get the lower triangular portion of A.

(b) Create a function, `function x = my_CG(A,b, tot_it)`, which also takes as input the matrix A, the vector b, number of iterations `tot_it` and outputs the Conjugate Gradient solution after `tot_it` iterations.

(c) We now investigate the performance of these three (Jacobi, GS and CG) algorithms. Compute the error in the solutions obtained from Jacobi, GS and CG as the number of iterations is varied for a fixed system size (use $P = 160$). Make a **single** table showing the 2-norm in the relative error as the number of iterations is varied. Use `tot_it = 50, 100, 200, 400, 800, 1600`. This table will have four columns: Num Iterations, Error Jacobi, Error GS and Error CG. Comment on the performance of different algorithms in reducing the relative error.

4. *Comparing the cost with direct solvers*

   We see from the above results that the Jacobi algorithm reduces the error quite slowly in each iteration. For a general matrix of size $n \times n$, Gaussian elimination (or Cholesky factorization) costs $O(n^3)$ whereas each iteration of Jacobi costs $O(n^2)$. So, for a system of size $n = 100$, if we perform more than 100 Jacobi iterations, we are doing more work than the direct solvers (Gaussian/Cholesky) — Ouch! However, we now note that the matrix A has a special structure.

   (a) How many non-zeros does the matrix A have in each row?

   (b) Based on the answer to part (a), how does the cost (in big-Oh notation) of performing the Jacobi algorithm change if you could take advantage of this sparsity? Do you think either of your implementations (component-wise or vector-update) takes advantage of the sparsity?

   (c) A is an m-banded matrix, where $m = 2P = 2\sqrt{n}$. What is the cost in constructing an LU factorization using the special algorithm for an m-banded matrix? Give your answer in terms of $\mathcal{O}(n^\alpha)$ and justify your value of α.

   (d) Based on your result for parts (b) and (c), what is the number of iterations for which the faster Jacobi algorithm (from part (b)) becomes more expensive than the fast direct solver (from part (c)) if $n = 100$?