CSC336: Assignment 2

Due June 16, 2020 by 11:59pm

General instructions

Please read the following instructions carefully before starting. They contain important information about general expectations, submission instructions, and reminders of course policies.

- Your work is graded on both correctness and clarity of communication. Solutions that are technically correct but poorly written will not receive full marks. Please read over your solutions carefully before submitting them.
- Solutions may be handwritten or typeset as appropriate, but must be submitted as PDFs and must be legible. Any code should be submitted as specified.

  The required filenames for this problem set are listed under additional instructions.
- Your work must be submitted online through MarkUs.
- Your submitted file(s) should not be larger than 9MB. You might exceed this limit if you use a word processor like Microsoft Word to create a PDF; if it does, you should look into PDF compression tools to make your PDF smaller, although please make sure that your PDF is still legible before submitting!
- Submissions must be made *before* the due date on MarkUs. You may submit as many times as you want. Your last submission will be graded.
- The work you submit must be your own.

Additional instructions

For the report questions, submit your solutions in a file called **a2-report.pdf** and your code in **a2-report.py**. Your answers for this question will be graded on both correctness and the quality of presentation of your results and analysis. **It must be typeset using LaTeX, a word processor, or similar (please ask if you aren't sure what is expected)**

For the other question, submit your written (or typed) solutions in **a2.pdf** and your code in **a2.py**.

If you import any non-standard modules (e.g tabulate or pandas) please do so in the if __name__ == '__main__' block of your a2.py if possible (or put the code in a2-report.py)

Please start early and ask questions on Piazza if you need clarification or help with any part of it. The coding questions require you to read some documentation in order to complete them, so don't leave it to the last minute.

1. Report Question - Banded Matrices (Based on Heath Computer Problem 2.17)

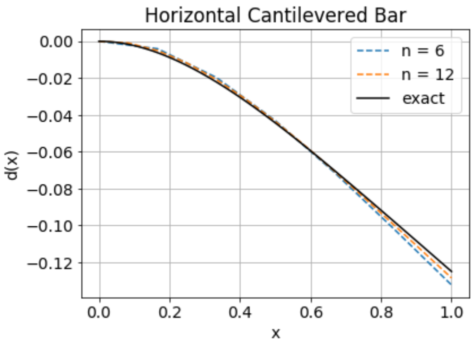
In this question, we will consider the performance of several different scipy linear system solvers when applied to a banded linear system.

We will be solving the $n \times n$ banded linear system:

$$\begin{bmatrix} 9 & -4 & 1 & 0 & \cdots & \cdots & 0 \\ -4 & 6 & -4 & 1 & \ddots & & \vdots \\ 1 & -4 & 6 & -4 & 1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & 1 & -4 & 6 & -4 & 1 \\ \vdots & & \ddots & 1 & -4 & 5 & -2 \\ 0 & \cdots & \cdots & 0 & 1 & -2 & 1 \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \\ \vdots \\ \\ d_{n-1} \\ d_n \end{bmatrix} = \frac{-1}{n^4} \begin{bmatrix} 1 \\ 1 \\ \\ \vdots \\ \\ 1 \\ 1 \end{bmatrix}$$

The physical system being modelled is a horizontal cantilevered beam - the bar is fixed at the left end (i.e at $x = 0$, the displacement in the bar,$d(x)$, is 0) and is free at the right end ($x = 1$). The bar is uniformly loaded (i.e the right hand side vector is constant). The factor of $\frac{1}{n^4}$ is necessary to make the solution comparable as we vary n.

The linear system arises from a discretization of a differential equation (using finite differences similar to what we considered for approximating derivatives in A1). This discretization means that d_i corresponds to $d(x_i)$, $x_i = \frac{i}{n}$, $i = 1, \dots, n$. For example, solving this with different values of n results in the following approximate solutions:



(a) LU

For a given value of n, setup and solve the linear system using **scipy.linalg.solve**. Form the matrix using scipy.sparse.diags as we did in the Week 6 worksheet, but make sure to convert it to its dense format (e.g. if S is a sparse matrix, S.toarray() returns the full matrix)

Time your code using time.perf_counter() as we have done on previous worksheets.

(b) banded LU

Solve the linear system using **scipy.linalg.solve_banded** and time your code. Read the documentation to understand what format the input has to be in.

(c) sparse LU

Solve the linear system using **scipy.sparse.linalg.spsolve** and time your code.

(d) prefactored

From Heath, it turns out that this matrix can be factored as RR^T, where,

$$R = \begin{bmatrix} 2 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & 1 & -2 & 1 \\ \vdots & & & \ddots & 1 & -2 \\ 0 & \cdots & \cdots & \cdots & 0 & 1 \end{bmatrix}$$

Solve the linear system using this factorization.

To do this, form R and use two calls to **scipy.linalg.solve_triangular** with appropriate arguments. Time your code.

(e) banded prefactored

Solve the linear system again using this factorization, but use two calls to **scipy.linalg.solve_banded** with appropriate arguments. Time your code.

(f) sparse prefactored

Repeat the above, but use **scipy.sparse.linalg.spsolve_triangular**. Note, use the optional argument format='csr' in scipy.sparse.diags when forming the sparse R or spsolve_triangular will complain. Time your code.

(g) Cholesky

It turns out that the matrix is symmetric positive definite, so we can also solve the system using the Cholesky decomposition. Use scipy.linalg.cho_factor and scipy.cho_solve to solve the linear system. Time your code.

(h) Experiment

Using the code you have written, perform an appropriate experiment to allow you to compare the performance of these ~~5~~ 7 different solvers. Your experiment should time each method for suitable values of n. Present your results in an appropriate table and/or plot - in whatever way you feel best gets your results across.

For example, your results may look something like:

```
 \         |         |         |         |         |         |         |         |         |
  \ method |    LU   |  banded |sparse LU|    R    | banded R| sparse R|   chol  |
n  \       |         |         |         |         |         |         |         |
----------------------------------------------------------------------------------
    200    |  0.55ms |  0.07ms |  0.19ms |  0.18ms |  0.07ms |  4.42ms |  0.31ms |
    400    |  2.40ms |  0.07ms |  0.23ms |  0.29ms |  0.09ms |  9.64ms |  1.36ms |
    600    |  5.93ms |  0.09ms |  0.31ms |  0.70ms |  0.12ms | 13.35ms |  3.55ms |
    800    | 10.82ms |  0.11ms |  0.40ms |  1.34ms |  0.22ms | 17.44ms |  7.62ms |
    1000   | 19.39ms |  0.13ms |  0.47ms |  2.11ms |  0.17ms | 21.91ms | 16.02ms |
    1200   | 34.74ms |  0.16ms |  0.56ms |  3.17ms |  0.19ms | 26.17ms | 18.94ms |
    1400   | 36.91ms |  0.18ms |  0.63ms |  4.17ms |  0.22ms | 31.76ms | 25.52ms |
    1600   | 56.64ms |  0.20ms |  0.72ms |  6.20ms |  0.24ms | 35.01ms | 39.41ms |
```

Please ask for help on Piazza if you have trouble getting any of the approaches working

When you time your code, don't include the time taken to construct any of the matrices.

You don't have to include it in your report, but please make sure your implementations are correct. If you can't get one of the approaches implemented, clearly indicate this in your work and refer to the above results for that method in your discussion

(i) Discussion

Comment on your results. Make connections to anything we have talked about regarding the costs of the various methods in lecture or discussed in section 2.5 of Heath. We haven't discussed general sparse solvers, so your comments about the two sparse solvers can be limited to what you observe and their relative performance to the other methods.

Note: You might find that spsolve_triangular is slower than you might expect - check the source code (linked in the online documentation [1]) and see if you can figure out why that might be.

Make sure to spend enough time on this part - the discussion of the results is the most important part of this exercise.

2. Report Question - Banded Matrices II (Based on Heath Computer Problem 2.17)

We'll now consider the accuracy of the computed solutions. It turns out that the analytical solution of the mathematical model [2] is $d(x) = \frac{1}{24}\left((1-x)(4-(1-x)^3)-3\right)$, so we will compare against this in our experiment.

(a) Experiment

For the methods in (b) and (e) above, plot the relative error in a loglog plot, where you vary n. Note, increasing n corresponds to refining the discretization of the bar - with a spacing of $h = \frac{1}{n}$ between points on the bar. (Since the left end of the bar is fixed, $d(0) = 0$. And $d(1)$ corresponds to the amount of bend in the far ~~left~~ right of the bar.)

Use values of $n = 16, 32, 64, \ldots, 16384, 32768, 65536$. (i.e 2^i, i = 4, ..., 16).

(b) Discussion

Comment on the accuracy achieved. How does it vary with n? Why might the pre-factored approach be more accurate than the LU approach?

Make sure to spend enough time on this part - the discussion of the results is the most important part of this exercise.

3. Efficiently Translating Math into Code (Based on Exercise 2.21 in Heath)

We'll again consider implementing the formula $x = B^{-1}(2A + \mathbb{1})(C^{-1} + A)b$, where all matrices are $n \times n$, b is a column vector of length n, and $\mathbb{1}$ denotes the identity matrix. On the last two homeworks we considered implementing this in two ways - the first explicitly computed inverses, the second instead solved linear systems.

(a) Give an operation count for the formula as written (**with the inverses explicitly computed and all calculations performed left to right**). Make sure we can easily follow where your total operation count came from.

(b) Give an operation count for the formula implemented **without explicitly computing any inverses and ordering the calculations to reduce the total number of operations performed.** Make sure we can easily follow where your total operation count came from.

(c) Implement the two formulas and perform an appropriate experiment to compare the performance of the two formulas. Clearly present the results of your experiment using a plot or table of values and include it in a2.pdf. **Please ask on Piazza if you aren't sure about your experimental setup or results.**

(d) Comment on how your results compare with your operation count analysis.

[1] see lines 472-607 in `https://github.com/scipy/scipy/blob/v1.4.1/scipy/sparse/linalg/dsolve/linsolve.py`

[2] For example, see slide 17 in `http://web.ncyu.edu.tw/~lanjc/lesson/C3/class/Chap06-A.pdf`

4. Representing Permutation Matrices [autotested]

As we have seen, when we implement linear algebra formulas in code, we often do things to avoid unnecessary work - usually involving zeros in the structure of the matrices involved.

Here we are going to walk through how this can be done for permutation matrices.

An $n \times n$ permutation matrix, P, only contains n non-zero entries, so it is inefficient to explicitly form it and perform a generic matrix multiply to compute PA.

Note: for this question, we'll use the notation $x = [x_0, x_1, \ldots, x_{n-1}]$, so that indices are all zero-based.

One way to represent P is with a vector q of length n, for which $q_i = j$ if $P_{i,j} = 1$ For example, the 4×4 permutation matrix,

$$P = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix},$$

can be represented by the vector, $q = [2, 3, 0, 1]$.

Moreover, in the context of Gaussian Elimination with partial pivoting, on each step of the algorithm, we exchange the current row with one other row, so we only actually need to store which row was swapped with the current row.

To represent the $n - 1$ permutation matrices that arise in partial pivoting, we will use the representation employed by the LAPACK piv vector (see help(scipy.linalg.lu_factor)).

In this representation, we use a vector of length n, call it p, where $p_i = j$ means that on step i of the algorithm, row i was interchanged with row j. Note, there are only $n - 1$ steps, so we always have that $p_{n-1} = n - 1$.

Given this p vector, we can efficiently perform the action of multiplying the $n - 1$ permutation matrices together, to get $P = P_{n-2} \cdots P_1 P_0$, represented in the q vector format.

For example, if we have,

$$P_0 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad P_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}, \text{and} \quad P_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

then $p = [2, 3, 2, 3]$. The product $P_2 P_1 P_0$ can be verified to be P from before (or as a q vector, $q = [2, 3, 0, 1]$.

(a) Write a function in a2.py, p_to_q(p), that takes in a vector p as described above and returns the vector q. Hint: your code should start with $q = [0, 1, \ldots, n - 1]$ and perform a sequence of swaps based on stepping through p (carefully read how we defined p above).

(see the example in help(scipy.linalg.lu_factor) - one of the autotests will do something like this to confirm your function works properly)

Please ask if you aren't sure how to get started here

(b) Write a function in a2.py, solve_plu(A,b), that uses **scipy.linalg.lu_factor**, two calls to **scipy.linalg.solve_triangular**, and your function from (a) to solve $Ax = b$.

Hint: it is maybe a bit confusing, but the only change is that you need to permute the rows in b (see page 73 of Heath) using your q vector (i.e b = b[q]).

Please ask if any of the above is unclear.

Make sure you don't explicitly form any permutation matrices and multiply them - you should only work with the permutation vector.