ME 5401: Scientific Computing using Python

Tutorial 2

Duration: 3 hours

Topics covered

Linear system of equations, Cramer's rule, Gaussian elimination with and without partial pivoting, LU factorization

Setup

Use Python with NumPy, Matplotlib and SciPy.

```
import numpy as np
import matplotlib.pyplot as plt
from time import perf_counter
```

Throughout this tutorial, solve systems of the form

$$Ax = b,$$

where $A \in \mathbb{R}^{n \times n}$ and $x, b \in \mathbb{R}^n$.

Questions

1. **Helper functions**

   Before implementing the solvers, write helper functions for:

   (a) Checking that A is square and that b has a compatible shape.

   (b) Computing the relative residual and relative error (separate functions)

   $$r_{rel} = \frac{\|Ax - b\|_2}{\|b\|_2}, \quad e_{rel} = \frac{\|x - x_{ref}\|_2}{\|x_{ref}\|_2}.$$

   (c) Generating a reproducible random system (A, b) of size n using a random seed. Random matrices can occasionally be nearly singular. To ensure reliable results, you may generate a random matrix using

   ```
   A = rng.normal(size=(n, n)) + n*np.eye(n)
   b = rng.normal(size=n)
   ```

   Test your helper functions on

   $$A = \begin{bmatrix} 3 & 2 \\ 1 & 2 \end{bmatrix}, \qquad b = \begin{bmatrix} 5 \\ 5 \end{bmatrix}.$$

$$A = \begin{bmatrix} 3 & 2 & 2 \\ 1 & 2 & 2 \end{bmatrix}, \qquad b = \begin{bmatrix} 5 \\ 5 \end{bmatrix}.$$

$$A = \begin{bmatrix} 3 & 2 & 2 \\ 1 & 2 & 2 \\ 1 & 0 & 2 \end{bmatrix}, \qquad b = \begin{bmatrix} 5 \\ 5 \end{bmatrix}.$$

2. **Cramer's rule**

   Cramer's rule says that if $\det(A) \neq 0$, then

   $$x_i = \frac{\det(A_i)}{\det(A)},$$

   where A_i is obtained from A by replacing column i of A with b.

   (a) Write a function implementing Cramer's rule using `np.linalg.det`

   ```
   def cramer_solve(A, b, tol=1e-12):
       """Return x solving Ax=b using Cramer's rule."""
   ```

   Your function should raise an error if A is singular or nearly singular.

   (b) Test the function on the warm-up 2×2 system and compare with `np.linalg.solve`. Report relative residual and relative error with `np.linalg.solve` as reference.

   (c) Test the function for random matrices with $n = 10, 20, 50, 100, 150, 200$. Record solution time and relative residual.

   (d) Does the function work robustly for moderately large matrices, e.g., $n = 150$ or 200? Explain why?

   (e) Create a new function that uses `np.linalg.slogdet` instead. Does this improve numerical robustness for moderately large matrices, e.g., $n = 150$ or 200? Explain why?

   ```
   def cramer_solve_slogdet(A, b, tol=1e-12):
       """Return x solving Ax=b using Cramer's rule with np.linalg
          .slogdet"""
   ```

   (f) Explain why it is not advisable to use Cramer's rule, even with `np.linalg.slogdet`, for very large matrices, e.g., $n = 2000$

3. **Naive Gaussian elimination**

   Implement Gaussian elimination without pivoting. Your function should include a forward elimination phase and a back substitution phase.

   (a) Write a back substitution function for an upper-triangular system $Ux = y$:

   ```
   def back_substitution(U, y, tol=1e-12):
       """Solve Ux=y for upper-triangular U."""
   ```

   (b) Write a naive Gaussian elimination solver:

   ```
   def gaussian_elimination_naive(A, b, tol=1e-12):
       """Solve Ax=b using elimination without pivoting."""
   ```

   (c) Test your solver on the warm-up 2×2 system.

(d) Test it on at least three random systems. Compare with `np.linalg.solve`. Report relative residual and relative error with `np.linalg.solve` as reference.

(e) Identify one condition under which this naive implementation can fail.

4. **Timing and scaling**

Compare the following methods:

(i) your Cramer's rule solver: `cramer_solve_slogdet`,

(ii) your naive Gaussian elimination solver: `gaussian_elimination_naive`,

(iii) `np.linalg.solve` (Note: `np.linalg.solve` is essentially an optimized library implementation of Gaussian elimination/LU factorization with pivoting)

(a) Use $n = 50, 100, 500, 1000$. For Cramer's rule, you can stop at $n = 500$ if runtime is too long

(b) Plot solution time versus n on a log-log plot. What method is preferable for large matrix sizes (Cramer's rule or Gaussian elimination)?

5. **Gaussian elimination with partial pivoting**

Partial pivoting swaps rows so that the pivot is the largest available entry in magnitude in the current column. This improves numerical stability and avoids division by very small pivots.

(a) Modify your naive Gaussian elimination solver to include partial pivoting:

```
def gaussian_elimination_pp(A, b, tol=1e-12):
    """Solve Ax=b using Gaussian elimination with partial
       pivoting."""
```

Hint 1: At each elimination step, first find the pivot row, i.e., the row at or below the current row with the largest absolute entry in the current column. Then, if needed, swap the current row with the pivot row in both A and b before eliminating.

Hint 2: To swap rows i and j in matrix A, use:

```
A[[i, j], :]  = A[[j, i], :]
```

(b) Test your code on

$$A = \begin{bmatrix} 10^{-21} & 1 \\ 1 & 1 \end{bmatrix}, \qquad b = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

Compare the no-pivoting solution, the partial-pivoting solution, and `np.linalg.solve`. Report relative residual for all three cases.

(c) Repeat the previous experiment for $\epsilon = 10^{-1}, 10^{-3}, 10^{-5}, \ldots, 10^{-21}$ in

$$A_\epsilon = \begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix}.$$

Tabulate relative error (with `np.linalg.solve` as reference) versus ϵ for your Gaussian elimination solvers with and without partial pivoting.

(d) Based on your results, explain why pivoting is standard in production solvers.

Additional Questions (Optional)

1. **LU decomposition without pivoting**

   During Gaussian elimination, the multipliers used to eliminate entries below the diagonal form the lower-triangular matrix L, while the final upper-triangular matrix is U.

   (a) Modify your Gaussian elimination code to return L and U such that

   $$A = LU.$$

   Use the function signature

   ```
   def lu_nopivot(A, tol=1e-12):
       """Return L, U such that A=L@U, without pivoting."""
   ```

   (b) Write forward substitution for $Ly = b$.

   ```
   def forward_substitution(L, b, tol=1e-12):
       """Solve Ly=b for lower-triangular L"""
   ```

   (c) Use your L and U to solve $Ax = b$ by first solving $Ly = b$ and then solving $Ux = y$. Report the relative residual.

   (d) Compare the solution with `gaussian_elimination_naive` and `np.linalg.solve`.

2. **Timing and scaling: comparison with theoretical complexity**

   For Question 4 from the main tutorial:

   (a) Estimate the empirical scaling exponent p in $T(n) \approx Cn^p$ using a linear fit of $\log(T)$ versus $\log(n)$ for your Naive Gaussian Elimination and `np.linalg.solve`. Use large values of $n = 500, 1000, 2000, 3000$

   (b) Compare the observed scaling with the expected theoretical complexity for Gaussian-Elimination ($p = 3$).

   (c) Read about how `np.linalg.det` computes the matrix determinant. What is the expected theoretical complexity for Cramer's rule for (a) determinant computation using cofactor approach, (b) using `np.linalg.det` or `np.linalg.slogdet`?

   (d) Estimate the empirical scaling exponent p in $T(n) \approx Cn^p$ using a linear fit of $\log(T)$ versus $\log(n)$ for your Cramer's rule implementation `cramer_solve_slogdet`. Use $n = 100, 500, 1000$. How does it compare with the theoretical complexity?