Assignment 2-Q3

February 3, 2026

1 MECH 309: Assignment 2, Question 3

Cagri Arslan

January 30, 2025

All work can be found on https://github.com/imported-canuck/MECH-309

[107]:
```python
# Imports
import numpy as np
from scipy import linalg
from solver import *
```

Note: I wrote all scripts by hand. Therefore, code might not be as concise/clean as a script produced by ChatGPT or taken from StackOverflow. Regardless, scripts should still be fully functional if all **assumptions** stated in docstrings are respected. I trust that this won't be penalized during grading.

a: Forward Substitution

Below is the function, `forward_sub(A,b)` that solves a linear system via forward substitution. It requires two inputs: an (n x n) matrix A, assumed to be lower triangular (but not checked), and an (n x 1) column vector b. It ensures that the matrix A is nonsingular and square, and that vector b is of compatible shape with A. If all of these checks pass, it builds the solution vector x: an (n x 1) column vector.

[108]:
```python
def forward_sub(A, b):
    """
    Solve the equation Ax = b for x, where A is a lower triangular matrix.
    Assumes a lower triangular matrix A. Expects a that A is non-singular,
    and that the dimensions of A and b are compatible (but checks these and
    throws an exception if not).

    Parameters:
    A (ndarray): A lower triangular matrix of shape (n, n).
    b (ndarray): A vector of shape (n, 1).

    Returns:
    x (ndarray): The solution vector of shape (n, 1).
    """
```

```python
    n = A.shape[0]            # Size of matrix A
    if A.shape[1] != n:       # Check if A is square
        raise ValueError("Matrix A must be square.")
    if b.shape[0] != n:       # Check if dimensions of b are compatible
        raise ValueError("Vector b must have compatible dimensions with matrix␣
↪A.")

    x = np.zeros((n, 1))      # Initialize solution vector x as a column vector

    x[0, 0] = b[0, 0] / A[0, 0]     # First element of x (actually redundant,
                                    # kept for clarity)
    for i in range(n):        # Loop over each row
        if A[i, i] == 0:      # Check for singularity (zero diagonal element)
            raise ValueError("Matrix is singular.")

        # For each column of row i before the diagonal subtract the product of
        # the entry A(i, j) and the corresponding entry of vector x from b[i]
        for j in range(i):
            b[i, 0] -= A[i, j] * x[j, 0]

        # Finaly divide b[i] by the diagonal element A(i, i) to get x[i]
        x[i, 0] = b[i, 0] / A[i, i]

    return x
```

Here we validate `forward_sub(A,b)` on the following matrix:

$$\begin{bmatrix} 2 & 0 & 0 \\ 3 & 1 & 0 \\ -2 & -1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 6 \\ 1 \end{bmatrix} \tag{1}$$

[109]:
```python
A = np.array([[2.0, 0.0, 0.0],
              [3.0, 1.0, 0.0],
              [-2.0, -1.0, 3.0]])

b = np.array([[2.0],
              [6.0],
              [1.0]])

forward_sub(A, b)
```

[109]:
```
array([[1.],
       [3.],
       [2.]])
```

b: Backward Substitution

Below is the function, `backward_sub(A,b)` that solves a linear system via forward substitution. It requires two inputs: an (n x n) matrix A, assumed to be upper triangular (but not checked), and an (n x 1) column vector b. It ensures that the matrix A is nonsingular and square, and that vector b is of compatible shape with A. If all of these checks pass, it builds the solution vector x: an (n x 1) column vector.

[ ]:
```
def backward_sub(A, b):
    """
    Solve the equation Ax = b for x, where A is an upper triangular matrix.
    Assumes an upper triangular matrix A. Expects a that A is non-singular,
    and that the dimensions of A and b are compatible (but checks these and
    throws an exception if not).

    Parameters:
    A (ndarray): An upper triangular matrix of shape (n, n).
    b (ndarray): A vector of shape (n, 1).

    Returns:
    x (ndarray): The solution vector of shape (n, 1).
    """
    n = A.shape[0]                # Size of matrix A
    if A.shape[1] != n:           # Check if A is square
        raise ValueError("Matrix A must be square.")
    if b.shape[0] != n:           # Check if dimensions of b are compatible
        raise ValueError("Vector b must have compatible dimensions with matrix␣
↪A.")

    x = np.zeros((n, 1))                # Initialize solution vector x

    x[-1, 0] = b[-1, 0] / A[-1, -1]     # First element of x (bottom-right;
                                        # actually redundant, kept for clarity)
    for i in range(n - 1, -1 , -1):     # Loop over each row from bottom to top
        # Check for singularity (zero diagonal element)
        if A[i, i] == 0:
            raise ValueError("Matrix is singular.")

        # For each column of row i after the diagonal subtract the product of
        # the entry A(i, j) and the corresponding entry of vector x from b[i]
        for j in range(i + 1, n):
            b[i, 0] -= A[i, j] * x[j, 0]
        # Finaly divide b[i] by the diagonal element A(i, i) to get x[i]
        x[i, 0] = b[i, 0] / A[i, i]

    return x
```

Here we validate `backward_sub(A,b)` on the following matrix:

$$\begin{bmatrix} 1 & -3 & 1 \\ 0 & 2 & -2 \\ 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ -2 \\ 6 \end{bmatrix} \tag{2}$$

[111]:
```
A = np.array([[1.0, -3.0, 1.0],
              [0.0, 2.0, -2.0],
              [0.0, 0.0, 3.0]])

b = np.array([[2.0],
              [-2.0],
              [6.0]])

backward_sub(A, b)
```

[111]:
```
array([[3.],
       [1.],
       [2.]])
```

c: Gaussian Elimination

Below is the function, `gaussian_elimination(A,b)` that solves a linear system via gaussian elimination with partial pivoting. It requires two inputs: an (n x n) matrix A, and an (n x 1) column vector b. It ensures that the matrix A is nonsingular and square, and that vector b is of compatible shape with A. If all of these checks pass, it builds the solution vector x: an (n x 1) column vector. It continually applies partial pivoting to address non-singular zero-pivots and floating point issues that can occur from dividing by very small numbers.

[ ]:
```
def gaussian_elimination(A, b):
    """
    Solve the equation Ax = b for x using Gaussian elimination with partial
    pivoting. Expects a that A is non-singular, and that the dimensions of A
    and b are compatible (but checks these and throws an exception if not).

    Parameters:
    A (ndarray): An matrix of shape (n, n).
    b (ndarray): A vector of shape (n, 1).

    Returns:
    x (ndarray): The solution vector of shape (n, 1).
    """
    tol = 1e-12                     # Tolerance to avoid floating point issues
    n = A.shape[0]                  # Size of matrix A

    if A.shape[1] != n:             # Check if A is square
        raise ValueError("Matrix A must be square.")
    if b.shape[0] != n:             # Check if dimensions of b are compatible
```

```
        raise ValueError("Vector b must have compatible dimensions with matrix␣
↪A.")

    for i in range(n - 1):       # Loop over all rows of A (apart from last,
                                 # since A is already upper-triangular by then)

    # Start partial pivoting: find the row with the largest element
    # at the column position of the pivot (don't want small numbers on pivot)
        p = i + np.argmax(np.abs(A[i:, i]))

    # If no row with a nonzero entry on the pivot exists, matrix is singular
        if np.abs(A[p, i]) <= tol:
                raise ValueError("Matrix is singular.")

    # If the current row is not the one with the greatest pivot element,
    # do partial pivoting by swapping current "row i" with row with greatest
    # pivot element "row p". And apply the same operation on vector b
        if p != i:
                A[[i, p], :] = A[[p, i], :]  # Row swap "i" and "p" on A
                b[[i, p], :] = b[[p, i], :]  # Swap corresponding entries in b

    # End partial pivoting, now eliminate entries below the pivot:
    # For each subsequent row (starting at i + 1) compute the factor that
    # would eliminate the element of row j that is below the pivot
        for j in range(i + 1, n):
            factor = A[j, i] / A[i, i]   # Compute elimination factor
            A[j] = A[j] - factor * A[i]  # Rj <- Rj - factor*Ri
            b[j] = b[j] - factor * b[i]  # Apply the same operation on vector b

    # The matrix is now upper triangular, so x can be solved for in O(n^2)
    # time with backward substituiton (from previous part b)
    return backward_sub(A, b)
```

Here we validate `gaussian_elimination(A,b)` on the $n = 2$, $n = 3$, and $n = 5$ square matrices provided in `solver.py`

[113]:
```
for CASE in ["2x2", "3x3", "5x5"]:
    print(f"\n=== Test case: {CASE} ===")

    A, b = load_test_case(CASE)

    # Make deep copies of A and b for reference solution
    A_ref = np.copy(A)
    b_ref = np.copy(b)

    x = gaussian_elimination(A, b)
    print_solution_report(A_ref, b_ref, x, label="My solver")
```

```python
# Reference solution (allowed for checking)
x_ref = linalg.solve(A_ref, b_ref)
print_solution_report(A_ref, b_ref, x_ref, label="Reference (scipy.linalg.
↪solve)")
```

```
=== Test case: 2x2 ===
--- My solver ---
x =
 [[1.]
 [2.]]
||r||_2 = 0.0

--- Reference (scipy.linalg.solve) ---
x =
 [[1.]
 [2.]]
||r||_2 = 1.7763568394002505e-15


=== Test case: 3x3 ===
--- My solver ---
x =
 [[1.]
 [1.]
 [1.]]
||r||_2 = 0.0

--- Reference (scipy.linalg.solve) ---
x =
 [[1.]
 [1.]
 [1.]]
||r||_2 = 0.0


=== Test case: 5x5 ===
--- My solver ---
x =
 [[ 1.]
 [-2.]
 [ 3.]
 [-4.]
 [ 1.]]
||r||_2 = 5.0242958677880805e-15

--- Reference (scipy.linalg.solve) ---
```

```
x =
 [[ 1.]
 [-2.]
 [ 3.]
 [-4.]
 [ 1.]]
||r||_2 = 8.331852114593072e-15
```

d: LU Factorization

Below is the function, `LU_factorization(A,b)` that solves a linear system via LU factorization with partial pivoting. It requires two inputs: an (n x n) matrix A, and an (n x 1) column vector b. It ensures that the matrix A is nonsingular and square, and that vector b is of compatible shape with A. If all of these checks pass, it builds the (n x n) lower triangular matrix L and (n x n) upper triangular matrix U. It then calls performs forward substitution to compute y in $Ly = b$, and backward substitution to compute x in $Ux = y$. It returns matrices L, U and the (n x 1) solution vector x. It continually applies partial pivoting to address non-singular zero-pivots and floating point issues that can occur from dividing by very small numbers.

[114]:
```python
def LU_factorization(A, b):
    """
    Solve the equation Ax = b for x using LU factorization with partial
    pivoting. Expects a that A is non-singular, and that the dimensions of A
    and b are compatible (but checks these and throws an exception if not).

    Parameters:
    A (ndarray): An matrix of shape (n, n).
    b (ndarray): A vector of shape (n, 1).

    Returns:
    x (ndarray): The solution vector of shape (n, 1).
    L (ndarray): The lower triangular matrix of shape (n, n).
    U (ndarray): The upper triangular matrix of shape (n, n).
    """
    tol = 1e-12                    # Tolerance to avoid floating point issues
    n = A.shape[0]                 # Size of matrix A

    if A.shape[1] != n:            # Check if A is square
        raise ValueError("Matrix A must be square.")
    if b.shape[0] != n:            # Check if dimensions of b are compatible
        raise ValueError("Vector b must have compatible dimensions with matrix␣
 ↪A.")

    L = np.eye(n)                  # Initialize L as identity matrix

    # Essentially apply gaussian elimination with partial pivoting to A, with
    # the added step of building the lower triangular matrix L along the way
```

```
    for i in range(n - 1):

        p = i + np.argmax(np.abs(A[i:, i]))

        if np.abs(A[p, i]) <= tol:
                raise ValueError("Matrix is singular.")

        if p != i:
                A[[i, p], :] = A[[p, i], :]
                b[[i, p], :] = b[[p, i], :]
            # Apply same row swap to L up to column i to maintain consistency
                L[[i, p], :i] = L[[p, i], :i]

        for j in range(i + 1, n):
            factor = A[j, i] / A[i, i]

            A[j] = A[j] - factor * A[i]
            # for Rj <- Rj - factor * Ri, insert L[j, i] = factor
            L[j, i] = factor # Store the factor in L

    U = A.copy() # Upper triangular matrix is the modified A after elimination

    # Solve Ly = b and Ux = y using O(n^2) forward and backward substitution
    y = forward_sub(L, b)
    x = backward_sub(U, y)

    return L, U, x
```

Here we validate `LU_factorization(A,b)` on the $n = 2$, $n = 3$, and $n = 5$ square matrices provided in `solver.py`

[115]:
```
for CASE in ["2x2", "3x3", "5x5"]:
    print(f"\n=== Test case: {CASE} ===")

    A, b = load_test_case(CASE)

    # Make deep copies of A and b for reference solution
    A_ref = np.copy(A)
    b_ref = np.copy(b)

    L, U, x = LU_factorization(A, b)
    print_solution_report(A_ref, b_ref, x, label="My solver")

    # Reference solution (allowed for checking)
    x_ref = linalg.solve(A_ref, b_ref)
    print_solution_report(A_ref, b_ref, x_ref, label="Reference (scipy.linalg.
 ↪solve)")
```

```
=== Test case: 2x2 ===
--- My solver ---
x =
 [[1.]
 [2.]]
||r||_2 = 0.0

--- Reference (scipy.linalg.solve) ---
x =
 [[1.]
 [2.]]
||r||_2 = 1.7763568394002505e-15


=== Test case: 3x3 ===
--- My solver ---
x =
 [[1.]
 [1.]
 [1.]]
||r||_2 = 0.0

--- Reference (scipy.linalg.solve) ---
x =
 [[1.]
 [1.]
 [1.]]
||r||_2 = 0.0


=== Test case: 5x5 ===
--- My solver ---
x =
 [[ 1.]
 [-2.]
 [ 3.]
 [-4.]
 [ 1.]]
||r||_2 = 5.0242958677880805e-15

--- Reference (scipy.linalg.solve) ---
x =
 [[ 1.]
 [-2.]
 [ 3.]
 [-4.]
 [ 1.]]
```

```
||r||_2 = 8.331852114593072e-15
```