Paul Beasly, Nov 2022

Comparison of Gaussian Elimination and LU Decomposition Methods for Solving PDEs

Introduction

Gaussian Elimination and LU Decomposition methods for solving the matrix problem Ax = b were studied. The computational speed of these methods were compared in solving coupled Partial Differential Equations (PDE). The mathematical model in the study was the advection-diffusion behavior for the vorticity $\omega(x, y, t)$ which is coupled to the stream function $\psi(x, y, t)$as shown in the equations below (source: J.N. Kutz AMATH 581 lecture notes).

$$\frac{\partial \omega}{\partial t} + [\psi, \omega] = \nu \bigtriangledown^2 \omega$$

$$\bigtriangledown^2 \psi = \omega$$

where,
$$[\psi, \omega] = \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} - \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x}$$

and the two dimensional Laplacian function is represented by $\bigtriangledown^2 = \partial_x^2 + \partial_y^2$.

Problem Definition and Solution Methods

The problem was defined with periodic spatial boundary conditions for x = [-10, 10] and y = [-10, 10]. Both x and y were evaluated with 64 values and the model was solved for t = [0, 4] with steps of 0.5.

The approach to solving the problem is to discretize the Laplacian and the other spatial partial derivatives using the second order difference scheme and convert these partial derivatives into matrix operators. With discretization of the spatial derivatives, the problem becomes an ODE with respect to time which can be solved using Initial Value Problem (IVP) solvers. An initial condition of

$$\omega(x, y, 0) = exp\left(-2x^2 - \frac{y^2}{20}\right)$$

was used for the IVP solver which implemented the RK45 method in Python.

The steps of the algorithm are outlined as follows:

1. Solve $\nabla^2\psi = \omega$ for ψ beginning with $\omega(x, y, 0)$, the initial condition.
2. Use the built-in RK45 solver to calculate $\omega(x, y, t)$ from the ODE and create an array of solutions for each time value.
3. The $\nabla^2\psi = \omega$ term needs to be recomputed for each time value as the solver steps through the time span.

A key step in the algorithm requires solving $\nabla^2\psi = \omega$, an elliptical problem. Once the Laplacian has been discretized into matrix form, A, the problem is of the form Ax=b. The matrix A has dimensions of $64^2 \times 64^2$, or (4096, 4096). Computing the solution of this matrix equation is the time-consuming step in the process due to the number of operations. The next step in the process uses an explicit IVP solver which is very fast.

Two approaches were used to solve the matrix equation, Ax=b, Gaussian Elimination (GE) and LU Decomposition (LU). The GE method performs operations on the matrix equation to convert it into upper triangular form. The number of steps in the computation are $O(N^3)$ for an (NxN) matrix using the GE method. Back substitution is then used to find the values for the array, ψ, which is faster with $O(N^2)$ computational steps.

The LU method decomposes the matrix A into upper and lower triangular matrices. This factorization process also requires $O(N^3)$ computational steps; however this step is only required once and subsequent steps to solve the matrix equation are $O(N^2)$.

Code was included in the script to measure the time taken for the GE and LU methods. A segment of the code for each of these methods is shown below. This segment of code captures the start and end times of each solution method and the key steps outlined above. The entire script is provided in an appendix.

Code to determine computation time for the GE method

```
t0 = time.time()
def vortPDE(t, w):
   phi = scipy.sparse.linalg.spsolve(A, w)
   w_t = (C@phi)*(B@w)-(B@phi)*(C@w)+mu*(A@w)
   return w_t

sol3 = scipy.integrate.solve_ivp(lambda t, w: vortPDE(t, w), [0, tend], y0,
t_eval=tspan)

GE_time = (time.time()-t0)
print('GE_time', GE_time)
```

Code to determine computation time for the LU method

```
t0_PLU = time.time()
PLU = scipy.sparse.linalg.splu(A)
def vortPDE(t, w):
```

```
    phi = PLU.solve(w)
    w_t = (C@phi)*(B@w)-(B@phi)*(C@w)+mu*(A@w)
    return w_t

sol4 = scipy.integrate.solve_ivp(lambda t, w: vortPDE(t, w), [0, tend], y0,
t_eval=tspan)

PLU_time = (time.time()-t0_PLU)

print('PLU time', PLU_time)
```

Results

Both methods calculated a solution to the mathematical model as defined by the problem. Figure 1 shows the contour plot of the vorticity parameter at t = 4. Both the GE and LU methods had the same solution. The mathematical model, initiated as a Gaussian bump, rotates counterclockwise with time. The center of the model rotates more quickly than the outer part of the function.

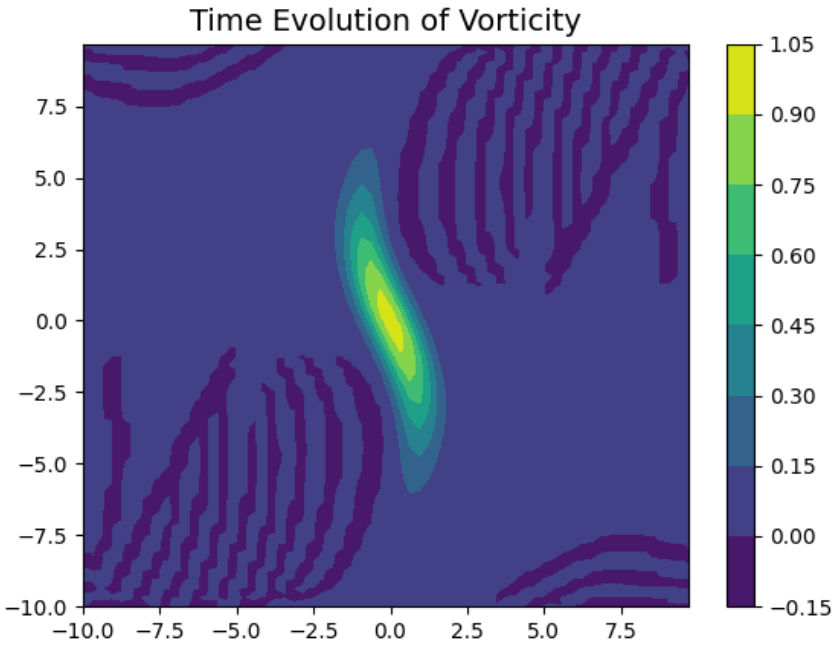

Figure 1: The evolution of the vorticity parameter in the mathematical model at t = 4. The initial Gaussian bump rotates counterclockwise.

The results for the computation speeds for GE and LU methods are summarized in Table 1. The LU method was significantly faster than the GE method. For the solution with x, y = 64 (a 4096x4096 matrix) and 9 time values, the speed was 22 times faster using the LU method. For the much larger

matrix with x = 128 and y = 128, the relative speed improvement was 29 times faster. The GE method took over 25 seconds to compute this matrix due to the size of the problem. The matrix was $(128^2, 128^2)$ in size which would require on the order of 128^6 (1e12) operations for each time step using the GE method. The relative speed improvement of LU compared to GE did not change significantly even though the matrix became significantly larger. Tthe computation time for each method was approximately 10 times longer for the larger matrix.

Table 1: Computation speeds for the GE and LU methods

Conditions	GE computation time (seconds)	LU computation time (seconds)	Relative Speed improvement (LU/GE)
9 time values x & y with 64 values each	2.65	0.123	22
9 time values x & y with 128 values each	25.76	1.142	29

Analysis of Results

The results show a significant improvement in speed with the LU method compared to the GE method. The LU method was approximately 22 times faster for the condition with x, y = 64 values and 9 time values. This improvement in computation speed matches with the relevant theory of these two methods in terms of the number of computational steps required.

The GE method requires $O(N^3)$ computational steps. For a (4096, 4096) matrix, this would be of the order of 1e10 computational steps.

The LU method also requires $O(N^3)$ computational steps for the initial factorization step. Subsequent iterations require $O(N^2)$ computational steps. We can compare these computational steps and determine the expected speed increase by examining the ratio of computation time for GE/LU methods. An equation for the ratio of GE computation time to LU computation time is shown below. The number of time values, or iterations, is represented by *iter*. This equation also represents the relative speed improvement of the LU method compared to the GE method.

GE/LU computation time (speed of LU/speed of GE) = $\dfrac{iter \times O(N^3)}{O(N^3) + iter \times O(N^2)}$

Factoring $O(N^2)$ and simplifying, we then have $\dfrac{iter \times O(N)}{O(N) + iter}$.

In the simulation, the number of steps *iter* = 9 and N = 4096.

For *iter* << N, the ratio can be approximated as $\frac{iter \times O(N)}{O(N) + iter} \approx iter$

This approximation shows that the speed improvement remains approximately proportional to the number of iterations and does not change significantly with the size of the matrix. The results in Table 1 agree with the approximation and the theory of the computational speeds of each method. The LU method was 22 times faster than the GE method for $N = 64^2$ and 29 times faster for $N = 128^2$. These results are within an order of magnitude of each other.

Further Work

The relative speed improvement of the LU method compared to the GE method is approximately proportional to the number of iterations, where the number of iterations is small compared to the size of the matrix. A short study was undertaken to measure the relative speed improvement of LU compared to the GE method. The number of time values was increased from 9 to 41 iterations. A simulation with also 161 iterations was performed. Table 2 summarizes the results. The results show that there is an improvement in relative speed of the LU method compared to the GE method as the number of iterations increases. This is expected since the benefit of the LU method's $O(N^2)$ computations for subsequent time steps is weighted more heavily with more iterations compared to the initial factorization of the LU matrix which requires $O(N^3)$ computation steps. The speed increase, however, doesn't follow the approximation of increasing linearly with the number of iterations. More work could be done to investigate the computational time for each iterative step. Furthermore, faster iterative schemes could also be investigated for this problem such as the Jacobi method.

Table 2. Computation time with increasing number of time values for GE and LU methods

Conditions	GE computation time (seconds)	LU computation time (seconds)	Relative speed improvement (LU/GE)
9 time values, x = 64	2.42	0.119	20
41 time values, x = 64	3.43	0.149	23
161 time values, x = 64	15.9	0.412	39

Appendix - Python code for analyzing computational time for GE and LU methods

```
import numpy as np
```

```
import scipy.sparse
import scipy.integrate
import matplotlib.pyplot as plt
import time

# Problem 2
# Define Matrix Functions____
# Define Laplacian
def LAP(m):
   n = m*m # total size of matrix

   e1 = np.ones(n)  # vector of ones
   Low1 = np.tile(np.concatenate((np.ones(m-1), [0])), (m,)) # Lower diagonal
1
   Low2 = np.tile(np.concatenate(([1], np.zeros(m-1))), (m,)) #Lower diagonal
2
                                       # Low2 is NOT on the second lower
diagonal,
                                       # it is just the next lower diagonal we
see
                                       # in the matrix.

   Up1 = np.roll(Low1, 1)  # Shift the array for spdiags
   Up2 = np.roll(Low2, m-1)  # Shift the other array

   LAP = scipy.sparse.spdiags([e1, e1, Low2, Low1, -4*e1, Up1, Up2, e1, e1],
                              [-(n-m), -m, -m+1, -1, 0, 1, m-1, m, (n-m)], n, n,
format='csc')
   LAP[0,0]=2
   return LAP

#plt.spy(LAP_func(6), markersize='8', marker='o') # view the matrix structure
#plt.show()

# Need to function for B matrix
def B(m):
   n = m*m  # total size of matrix
   e0 = np.ones(n)  # vector of ones

   # Create matrix from left-bottom to right-top
   B = scipy.sparse.spdiags([e0, -1*e0, e0, -1*e0],
                              [-1*(m**2-m), -m, m, m**2-m], n, n, format='csc')
   return B

# Create function for C matrix
# First create a function to generate required diagonals for C matrix
# Build from bottom-left to top-right; e1, e2, e3, e4
def C_Diags(m):
```

```
    # Algorithm to create m sets of [1,0,0,0...] diag
    e0 = [1]
    ex = np.zeros(m - 1)
    e0 = np.append(e0, ex)
    e1 = []
    for i in range(m):
        e1 = np.append(e1, e0)

    # Algorithm to create m sets of [-1,-1,-1...,0] diag
    e0 = -1*np.ones(m-1)
    e0 = np.append(e0, 0)
    e2 = []
    for i in range(m):
        e2 = np.append(e2, e0)

    # Algorithm to create m sets of [1,1,1...,0] diag
    e0 = np.ones(m-1)
    e0 = np.append(e0, 0)
    e3 = []
    for i in range(m):
        e3 = np.append(e3, e0)

    # Algorithm to create m sets of [-1,0,0,0...] diag
    e0 = [-1]
    ex = np.zeros(m-1)
    e0 = np.append(e0, ex)
    e4 = []
    for i in range(m):
        e4 = np.append(e4, e0)

    return e1, e2, e3, e4

# Create the C matrix
def C(m):
    n = m*m
    e1, e2, e3, e4 = C_Diags(m)
    e3 = np.roll(e3, 1)
    e4 = np.roll(e4, -1)
    # Create matrix from left-bottom to right-top
    C = scipy.sparse.spdiags([e1, e2, e3, e4],
                             [-(m-1), -1, 1, m-1], n, n, format='csc')
    return C

# Check matrix structure
#plt.spy(LAP(4), markersize='8', marker='o') # view the matrix structure
#plt.show()
# print(LAP(4).todense())
# print(np.shape((LAP(4))))
```

```
# Define Constants_____
L = 10
tend = 4
dt = 0.5
msize = 65
tspan = np.arange(0, tend+dt, dt)
x = np.linspace(-L, L, msize)
x = x[:-1]
y = np.linspace(-L, L, msize)
y = y[:-1]
h = 2*L/64 # step size in x & y
mu = 0.001
n = len(x) # matrix size for x & y

# initial condition for w
f3 = lambda x, y: np.exp(-2*x**2-(y**2/20))

w0 = np.zeros([msize-1, msize-1]) # initialize initial column vector

for i in range(len(x)):
   for j in range(len(y)):
       w0[i,j] = f3(x[i], y[j])

w0 = w0.reshape([-1,1]) # reshape to a column vector
y0 = np.squeeze(w0) # make one dimenstional for the solver


# Generate Matrices from functions above
A = (1/(h**2))*LAP(n)
B = (1/(2*h))*B(n)
C = (1/(2*h))*C(n)


t0 = time.time()

def vortPDE(t, w):
   phi = scipy.sparse.linalg.spsolve(A, w) # this needs to be the GE method
and PLU
   w_t = (C@phi)*(B@w)-(B@phi)*(C@w)+mu*(A@w)
   return w_t

sol3 = scipy.integrate.solve_ivp(lambda t, w: vortPDE(t, w), [0, tend], y0,
t_eval=tspan)

GE_time = (time.time()-t0)
print('GE_time', GE_time)

A7 = sol3.y.T  # assign to deliverable
```

```
# for PLU solution change A to PLU
t0_PLU = time.time()
PLU = scipy.sparse.linalg.splu(A)

def vortPDE(t, w):
   phi = PLU.solve(w)
   w_t = (C@phi)*(B@w)-(B@phi)*(C@w)+mu*(A@w)
   return w_t

sol4 = scipy.integrate.solve_ivp(lambda t, w: vortPDE(t, w), [0, tend], y0,
t_eval=tspan)

PLU_time = (time.time()-t0_PLU)
print('PLU time', PLU_time)

A8 = sol4.y.T  # assign to deliverable

# A9 =A8.reshape([9,64,64])
# Plotting the contour at t=4
sol_pl = sol4.y[:,8].reshape(64,64)
sol_pl = sol_pl.T

fig5, ax5 = plt.subplots()

lvls = np.linspace(0, 1.0, 15)
X, Y = np.meshgrid(x, y)
surf = ax5.contourf(X, Y, sol_pl) #, levels=lvls)
fig5.colorbar(surf)
plt.title('Time Evolution of Vorticity', fontsize=14, y=1.0)
plt.show()
```