Computational physics

Jona Ackerschott, Julian Mayr

Problem set 4

Problem 1

By using the gauss-jordan method (with and without pivoting) implemented in `linsolve.py` and, for comparison, LU-decomposition implemented in `scipy.linalg.lu_factor` and `scipy.linalg.lu_solve`, on the equation system

$$\begin{pmatrix} \varepsilon & 1/2 \\ 1/2 & 1/2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1/2 \\ 1/4 \end{pmatrix} \tag{1}$$

one obtains the following results for the solution $(x, y)^T$ with $\varepsilon = 10^{-6}$

	x	y	b_1	b_2
gauss-jordan	-0.4687500	1.0000010	0.50000000	0.26562548
gauss-jordan with pivoting	-0.5000011	1.0000011	0.50000006	0.25000000
LU-decomposition	-0.5000011	1.0000011	0.50000006	0.25000000

Table 1: Solutions of (1) with $\varepsilon = 10^{-6}$ using the corresponding algorithms. b_1 and b_2 are the results obtained by backsubstituting the solution into (1)

.

As one can see, the gauss-jordan method without pivoting loses some accuracy compared to the LU-decomposition while the gauss-jordan method with pivoting does not. By decreasing ε even further, one cannot observe any loss of accuracy between, or a significant deviation from the exact solution of the latter two methods. However, one gets the following deviation between the gauss-jordan method (without pivoting) and the LU-decomposition method, measured by the norm of the difference between the two solutions relative to the LU-decomposition solution.

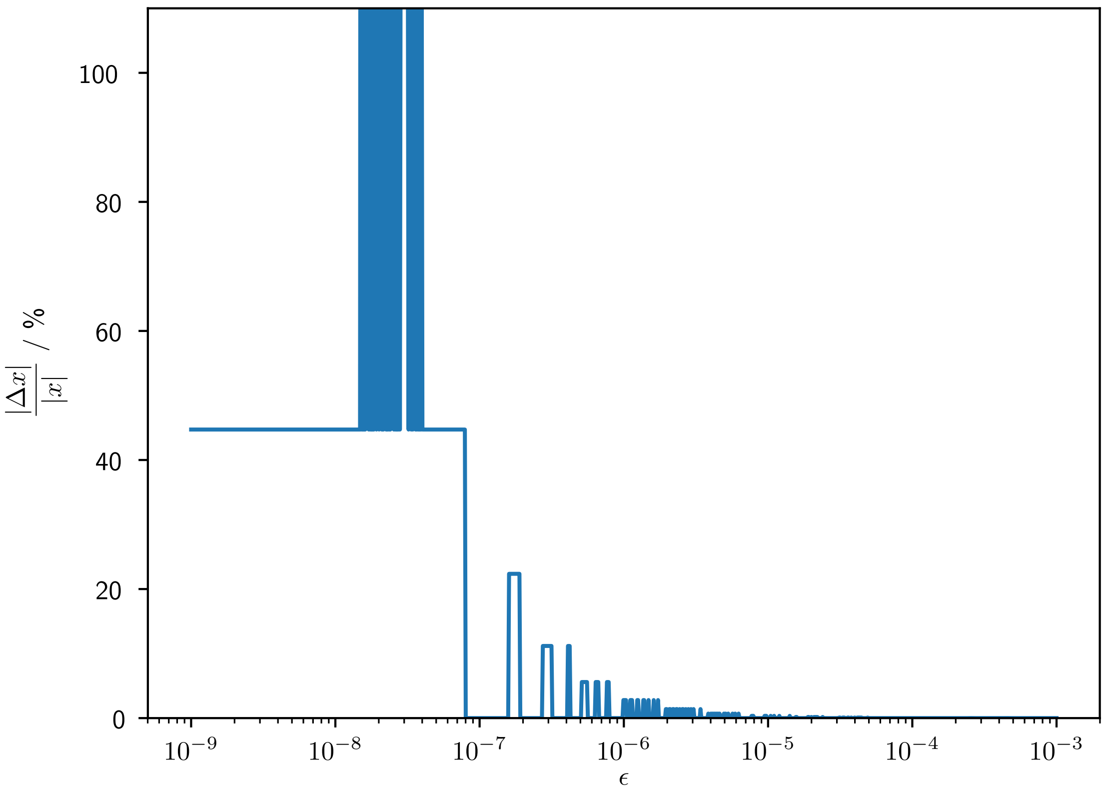

Figure 1: Deviation of the solutions of (1) obtained by the gauss-jordan method (without pivoting) and LU-decomposition, where Δx is the difference of the solution vectors and x is the LU-decomposition solution

So one gets a pretty reasonable solution with a deviation staying at less than 0.5%, if $\varepsilon > 10^{-5}$.

Problem 2

For a tridiagonal Matrix, the Gauss-Jordan-Algorithm simplifies by only having to consider the previous row of the matrix for each row to get to upper triangle form. For matrix rows y_n and a_n, b_n, c_n as given in the sheet, the iteration step for each line reads:

$$y_n = \frac{y_n - a_n * y_{n-1}}{b_n - c_{n-1}} \tag{2}$$

After Gaussian elimination, the Matrix equation can be solved using backward substitution. For tridiagonal matrices, this also simplifies to

$$y_n = y_n - y_{n+1} * y_{n,n+1} \tag{3}$$

since every row only has two elements. The subroutine "solution_tridiag" in the program implements these two functions and uses them to calculate the solution vector x_i. For the given parameters, it yields:

$$x = \begin{pmatrix} 0.49999999999999994 \\ 0.8999999999999999 \\ 1.2 \\ 1.4000000000000004 \\ 1.5000000000000004 \\ 1.5000000000000004 \\ 1.4000000000000004 \\ 1.2000000000000002 \\ 0.9 \\ 0.5 \end{pmatrix} \tag{4}$$

Testing the equation with this solution and calculating the error gives

$$\Delta x = \begin{pmatrix} -2.7755575615628914e-17 \\ -1.3877787807814457e-16 \\ -3.608224830031759e-16 \\ 3.0531133177191805e-16 \\ 8.326672684688674e-17 \\ 8.326672684688674e-17 \\ 8.326672684688674e-17 \\ -2.7755575615628914e-17 \\ -1.3877787807814457e-16 \\ -2.7755575615628914e-17 \end{pmatrix} \tag{5}$$

Since 64 bit floating point has an accuracy of roughly 16 digits, this lies in floating point error range and is a totally acceptable error.

Source Code

```
import numpy as np

def gaussJordan(A, b, pivoting=False, dtype=None):
  """ Solves mutiple systems of equations A X = B using the gauss-jordan algorithm"""
  B = np.transpose(b)
  n = np.size(A, axis=0)
  m = np.size(A, axis=1)
  if n != m or n != np.size(B, axis=0):
    raise ValueError

  M = np.zeros((n, n + np.size(B, axis=1)), dtype=dtype)
  M[:, :m] = A
  M[:, m:] = B

  for i in range(n):
    if pivoting:
      iMax = np.argmax(M[i:, i]) + i
      M[[i, iMax]] = M[[iMax, i]]

    M[i] /= M[i][i]
    for j in range(n):
```

```python
      if i != j:
        M[j] -= M[j, i] * M[i]

  return np.transpose(M[:, n:])
```

Source code 1: linsolve.py

```python
import matplotlib.pyplot as plt
import numpy as np
import os
from linsolve import gaussJordan
from numpy.linalg import norm
from scipy.linalg import lu_factor, lu_solve

# Solve the equation system for eps=1e-6 using the gauss-jordan algorithm
#   and LU-Decomposition
eps0 = 1e-6
A = np.array([[eps0, 0.5], [0.5, 0.5]], dtype=np.float32)
b = np.array([[0.5, 0.25]], dtype=np.float32)
X1 = gaussJordan(A, b, pivoting=False, dtype=np.float32)
X2 = gaussJordan(A, b, pivoting=True, dtype=np.float32)
x = lu_solve(lu_factor(A), b[0])

# Print the results
print(X1[0])
print(X2[0])
print(x)
print()
print(np.matmul(A, X1[0]))
print(np.matmul(A, X2[0]))
print(np.matmul(A, x))

# Compute relative deviations of the first solution element
#   for different values of eps
Eps = np.logspace(-3, -9, num=1000)
RDev = np.zeros_like(Eps)
for i, eps in enumerate(Eps):
  A = np.array([[eps, 0.5], [0.5, 0.5]], dtype=np.float32)
  b = np.array([[0.5, 0.25]], dtype=np.float32)
  S = gaussJordan(A, b, pivoting=False, dtype=np.float32)
  x = lu_solve(lu_factor(A), b[0])

  RDev[i] = abs(norm(S[0] - x) / norm(x))

# Plot the deviation in dependence of eps
plt.xlabel(r'$\epsilon$')
plt.ylabel(r'$\frac{|\Delta x|}{|x|}$ / \%')
plt.ylim(0, 110)
plt.semilogx(Eps, 100 * RDev)
```

```
# Save the figure
if not os.path.exists('figures'):
  os.mkdir('figures')
plt.savefig('figures/fig1_1.pgf')
plt.show()
```

Source code 2: problem1.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import numpy as np

def gauss_upper_diag_tridiag(M):
    """Take a tridiagonal LGS (Matrix N+1xN) M and returns the upper triangle matrix""
    M[0]/=M[0][0]
    for i in range(1,len(M[:,0])):
        M[i]-=M[i-1]*M[i,i-1]
        M[i]/=M[i][i]
    return M

def unit_from_upper_diag_backwards_sub(M):
    for i in range(0,len(M[:,0])):
        for j in range(1, i+1):##for compatibility with non-tridiag matrices
            M[-i-1]-=M[-i-1,-j-1]*M[-j]
    return M

M=np.identity(10)*2+np.pad(np.identity(9),((1,0),(0,1)), "constant")*-1+np.pad(np.identi


def make_tridiag(a,b,c, y=None, dim=None):
    """makes tridiagonal matrix M from either numbers or vectors a,b,c,. If numbers ar
    if not dim:
        dim=len(b)
    M=np.identity(dim)*b+np.pad(np.identity(dim-1)*a,((1,0),(0,1)), "constant")+np.pad(r
    if y:
        return np.hstack((M,np.array(10*[[1]])*y))
    else:
        return M

def solution_tridiag(a,b,c,y, dim):
    return unit_from_upper_diag_backwards_sub(gauss_upper_diag_tridiag(make_tridiag(a,b
s=solution_tridiag(-1,2,-1, 0.1, 10)
print("Solution for x:",s)
print("Error for M dot s:", make_tridiag(-1,2,-1, dim=10).dot(s)-0.1)
```

Source code 3: problem2.py