L_Bijman_S_Gijsbers_HW2

November 13, 2023

1 Homework set 2

Please **submit this Jupyter notebook through Canvas** no later than **Mon Nov. 13, 9:00**. **Submit the notebook file with your answers (as .ipynb file) and a pdf printout. The pdf version can be used by the teachers to provide feedback. A pdf version can be made using the save and export option in the Jupyter Lab file menu.**

Homework is in **groups of two**, and you are expected to hand in original work. Work that is copied from another group will not be accepted.

2 Exercise 0

Write down the names + student ID of the people in your group.

Loes Bijman 15211312

Sacha Gijsbers 12798525

2.1 Importing packages

Execute the following statement to import the packages `numpy`, `math` and `scipy.sparse`. If additional packages are needed, import them yourself.

[ ]:
```
import math
import numpy as np
import scipy.sparse as sp
import scipy
from fractions import Fraction
import time
import sys
```

3 Sparse matrices

A matrix is called sparse if only a small fraction of the entries is nonzero. For such matrices, special data formats exist. `scipy.sparse` is the scipy package that implements such data formats and provides functionality such as the LU decomposition (in the subpackage `scipy.sparse.linalg`).

As an example, we create the matrix

$$\begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 5 \\ 0 & 0 & 0 & 6 \end{bmatrix}$$

in the so called compressed sparse row (CSR) format. As you can see, the arrays `row`, `col`, `data` contain the row and column coordinate and the value of each nonzero element respectively.

[ ]:
```
# a sparse matrix with 6 nonzero entries
row = np.array([0, 0, 1, 2, 2, 3])
col = np.array([0, 2, 1, 2, 3, 3])
data = np.array([1.0, 2, 3, 4, 5, 6])
sparseA = sp.csr_array((data, (row, col)), shape=(4, 4))

# convert to a dense matrix. This allows us to print to screen in regular␣
  ↪formatting
denseA = sparseA.toarray()
print(denseA)
```

```
[[1. 0. 2. 0.]
 [0. 3. 0. 0.]
 [0. 0. 4. 5.]
 [0. 0. 0. 6.]]
```

For sparse matrices, a sparse data format is much more efficient in terms of storage than the standard array format. Because of this efficient storage, very large matrices of size $n \times n$ with $n = 10^7$ or more can be stored in RAM for performing computations on regular computers. Often the number of nonzero elements per row is quite small, such as 10's or 100's nonzero elements per row. In a regular, dense format, such matrices would require a supercomputer or could not be stored.

In the second exercise you have to use the package `scipy.sparse`, please look up the functions you need (or ask during class).

4 Heath computer exercise 2.1

4.1 (a)

Show that the matrix

$$A = \begin{bmatrix} 0.1 & 0.2 & 0.3 \\ 0.4 & 0.5 & 0.6 \\ 0.7 & 0.8 & 0.9 \end{bmatrix}.$$

is singular. Describe the set of solutions to the system $Ax = b$ if

$$b = \begin{bmatrix} 0.1 \\ 0.3 \\ 0.5 \end{bmatrix}.$$

(N.B. this is a pen-and-paper question.)

We can see that

$\det(\mathrm{A}) = 0.1 \cdot (0.5 \cdot 0.9 - 0.6 \cdot 0.8) - 0.2(0.4 \cdot 0.9 - 0.6 \cdot 0.7) + 0.3 \cdot (0.4 \cdot 0.8 - 0.5 \cdot 0.7) = 0$

Therefore matrix A is singular.

Since A is singular, we know there is a nonzero vector $\mathbf{z}$ such that $A\mathbf{z} = \mathbf{0}$. Thus, if $\mathbf{x}$ is a solution to $A\mathbf{x} = \mathbf{b}$, this means $\mathbf{x} + \gamma \cdot \mathbf{z}$ is also a solution, for any $\gamma \in \mathbb{R}$.

LU decomposition for matrix A gives $L = \begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & -2 & 1 \end{bmatrix}$ and $U = \begin{bmatrix} 0.1 & 0.2 & 0.3 \\ 0 & -0.3 & -0.6 \\ 0 & 0 & 0 \end{bmatrix}$. Using these matrices and performing forward and backward substitution, we find $\mathbf{z} = \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}$ and $\mathbf{x} = \begin{bmatrix} 1/3 \\ 1/3 \\ 0 \end{bmatrix}$.

Therefore the general solution is $x = \begin{bmatrix} 1/3 \\ 1/3 \\ 0 \end{bmatrix} + \gamma \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1/3 + \gamma \\ 1/3 - 2\gamma \\ \gamma \end{bmatrix}$ for all $\gamma \in \mathbb{R}$ (or in $\mathbb{C}$).

4.2 (b)

If we were to use Gaussian elimination with partial pivoting to solve this system using exact arithmetic, at what point would the process fail?

For Gaussian elimination, or LU decomposition, it is assumed that the original matrix A can be decomposed into two matrices, L and U, such that $A = LU$. L is a lower triangular matrix, and U is an upper triangular matrix. If matrix A is singular, it is still possible to find L and U such that $A = LU$. However, U can contain a row of zeros. Therefore, an exact arithmetic will fail during the back-substitution process because of a division by zero.

4.3 (c)

Because some of the entries of A are not exactly representable in a binary floating point system, the matrix is no longer exactly singular when entered into a computer; thus, solving the system by Gaussian elimination will not necessarily fail. Solve this system on a computer using a library routine for Gaussian elimination. Compare the computed solution with your description of the solution set in part (a). What is the estimated value for cond(A)? How many digits of accuracy in the solution would this lead you to expect?

[ ]:
```
# sparse matrix A
row = np.array([0, 0, 0, 1, 1, 1, 2, 2, 2])
col = np.array([0, 1, 2, 0, 1, 2, 0, 1, 2])
data = np.array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
sparseA = sp.csr_array((data, (row, col)), shape=(3, 3))

# convert to dense matrix
A = sparseA.toarray()
```

```
print(f"Matrix A:\n {A}")

b = np.array([0.1, 0.3, 0.5])
print(f"b: {b}")

# Perform Gaussian elimination with partial pivoting
lu, piv = scipy.linalg.lu_factor(A)
x = scipy.linalg.lu_solve((lu, piv), b)

print(f"Solution set x: {x}")
```

```
Matrix A:
 [[0.1 0.2 0.3]
 [0.4 0.5 0.6]
 [0.7 0.8 0.9]]
b: [0.1 0.3 0.5]
Solution set x: [ 0.16145833  0.67708333 -0.171875  ]
```

[ ]:
```
# Comparison of computed solution with calculation of 2.1(a)

gamma = x[2] # using vectors from 2.1(a)
x_1 = 1/3 + gamma
x_2 = 1/3 - 2*gamma
x_3 = gamma

solution = np.array([x_1,x_2,x_3])

print(f"Solution from 2.1(a) with gamma = {gamma}: {solution}")
print(f"Computed solution set: {x}")
```

```
Solution from 2.1(a) with gamma = -0.171875: [ 0.16145833  0.67708333 -0.171875
]
Computed solution set: [ 0.16145833  0.67708333 -0.171875  ]
```

We can see that the solution found by the computer fits the general solution as found in 2.1(a) for $\gamma = -0.171875$.

[ ]:
```
condA = np.linalg.cond(A, p = np.inf)
print("Condition number of A with infinity norm:", condA)
print(f"cond(A)*machine_epsilon = {condA * 2*10**(-16)}")
print(f"log(cond(A)) = {math.log(condA,10)}")
```

```
Condition number of A with infinity norm: 8.64691128455135e+16
cond(A)*machine_epsilon = 17.2938225691027
log(cond(A)) = 16.93686100323057
```

From Heath, we know that $\frac{||\hat{\mathbf{x}}-\mathbf{x}||}{||\mathbf{x}||} \lessapprox \text{cond(A)}_{\text{mach}}$. For double precision we have $\epsilon_{\text{mach}} \approx 2 \cdot 10^{-16}$. In our case, we obtain $\frac{||\hat{\mathbf{x}}-\mathbf{x}||}{||\mathbf{x}||} \lessapprox 8.6469 \cdot 10^{16} \cdot 2 \cdot 10^{-16} \approx 17.29$. Furthermore, we can see that

$\log_{10}(\text{cond(A)}) \approx \log_{10}(8.6469 \cdot 10^{16}) \approx 16.9$ decimal digits of accuracy are lost relative to the accuracy of the input. Therefore we expect no correct digits in the solution unless the input data is accurate to more than 17 digits. Since only 16 digits can be accurately stored, the solution has lost all accuracy.

5 Heath computer exercise 2.17

Consider a horizontal cantilevered beam that is clamped at one end but free along the remainder of its length. A discrete model of the forces on the beam yields a system of linear equations $Ax = b$, where the $n \times n$ matrix A has the banded form

$$\begin{bmatrix} 9 & -4 & 1 & 0 & \dots & \dots & 0 \\ -4 & 6 & -4 & 1 & \ddots & & \vdots \\ 1 & -4 & 6 & -4 & 1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & 1 & -4 & 6 & -4 & 1 \\ \vdots & & \ddots & 1 & -4 & 5 & -2 \\ 0 & \dots & \dots & 0 & 1 & -2 & 1 \end{bmatrix},$$

the n-vector b is the known load on the bar (including its own weight), and the n-vector x represents the resulting deflection of the bar that is to be determined. We will take the bar to be uniformly loaded, with $b_i = 1/n^4$ for each component of the load vector.

5.1 (a)

Make a python function that creates the matrix A given the size n.

[ ]:
```
n = 16

def create_beam_matrix_A(n):
    b = np.ones(n)
    A = scipy.sparse.diags([b, -4 * b, 6 * b, -4 * b, b], [-2, -1, 0, 1, 2],␣
 ↪(n, n), format='csc')
    A[0, 0] = 9
    A[n - 2, n - 2] = 5
    A[n - 1, n - 1] = 1
    A[n - 1, n - 2] = -2
    A[n - 2, n - 1] = -2

    A_dense = A.toarray()

    return A_dense

A_dense = create_beam_matrix_A(n)
print(A_dense)
```

```
[[ 9. -4.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [-4.  6. -4.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 1. -4.  6. -4.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

```
[ 0.  1. -4.  6. -4.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
[ 0.  0.  1. -4.  6. -4.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
[ 0.  0.  0.  1. -4.  6. -4.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
[ 0.  0.  0.  0.  1. -4.  6. -4.  1.  0.  0.  0.  0.  0.  0.  0.]
[ 0.  0.  0.  0.  0.  1. -4.  6. -4.  1.  0.  0.  0.  0.  0.  0.]
[ 0.  0.  0.  0.  0.  0.  1. -4.  6. -4.  1.  0.  0.  0.  0.  0.]
[ 0.  0.  0.  0.  0.  0.  0.  1. -4.  6. -4.  1.  0.  0.  0.  0.]
[ 0.  0.  0.  0.  0.  0.  0.  0.  1. -4.  6. -4.  1.  0.  0.  0.]
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1. -4.  6. -4.  1.  0.  0.]
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1. -4.  6. -4.  1.  0.]
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1. -4.  6. -4.  1.]
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1. -4.  5. -2.]
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1. -2.  1.]]
```

5.2 (b)

Solve this linear system using both a standard library routine for dense linear systems and a library routine designed for sparse linear systems. Take $n = 100$ and $n = 1000$. How do the two routines compare in the time required to compute the solution? And in the memory occupied by the LU decomposition? (Hint: as part of this assignment, look for the number of nonzero elements in the matrices L and U of the sparse LU decomposition.)

[ ]:
```python
def time_dense_sparse(n):
    b = np.full(n, 1/(n^4))
    A_dense = create_beam_matrix_A(n)
    A_sparse = scipy.sparse.csr_matrix(create_beam_matrix_A(n))

    start_dense = time.time()
    lu_dense, piv_dense = scipy.linalg.lu_factor(A_dense)
    x_dense = scipy.linalg.lu_solve((lu_dense, piv_dense), b)
    end_dense = time.time()
    time_dense = end_dense-start_dense

    start_sparse = time.time()
    lu_sparse = scipy.sparse.linalg.splu(A_sparse)
    x_sparse = lu_sparse.solve(b)
    end_sparse = time.time()
    time_sparse = end_sparse-start_sparse

    return time_dense, time_sparse, lu_dense, lu_sparse, x_dense, x_sparse
```

[ ]:
```python
def time_memory(n_list,runs):
    dense_time, dense_std, sparse_time, sparse_std, dense_mem, sparse_mem =␣
 ↪[],[],[],[],[],[]
    for n in n_list:
        all_time_dense, all_time_sparse = [],[]
        for _ in range(runs):
            time_dense, time_sparse, _, lu_sparse, _, _ = time_dense_sparse(n)
```

```
            all_time_dense.append(time_dense)
            all_time_sparse.append(time_sparse)

        dense_mem.append(n*n)
        sparse_mem.append((lu_sparse.L != 0).sum() + (lu_sparse.U != 0).sum())

        dense_time.append(np.mean(all_time_dense))
        dense_std.append(np.std(all_time_dense))

        sparse_time.append(np.mean(all_time_sparse))
        sparse_std.append(np.std(all_time_sparse))
    return dense_time, dense_std, sparse_time, sparse_std, dense_mem, sparse_mem

runs = 100
n_list = [100, 1000]
dense_time, dense_std, sparse_time, sparse_std, dense_mem, sparse_mem =␣
 ↪time_memory(n_list,runs)

for i in range(len(n_list)):
    print(f"n = {n_list[i]} ({runs} runs): dense average running time =␣
 ↪{dense_time[i]} seconds with std {dense_std[i]}")
    print(f"n = {n_list[i]} ({runs} runs): sparse average running time =␣
 ↪{sparse_time[i]} seconds with std {sparse_std[i]}")

    print(f"For n = {n_list[i]}, sparse is {dense_time[i] - sparse_time[i]}␣
 ↪seconds faster than dense")
    print('-----------------------------------------------------')

    print(f"n = {n_list[i]} ({runs} runs): dense # values stored:␣
 ↪{dense_mem[i]} elements")
    print(f"n = {n_list[i]} ({runs} runs): sparse # values stored:␣
 ↪{sparse_mem[i]} elements")

    print('-----------------------------------------------------')

print(f"Dense running time increase factor: {dense_time[1]}/{dense_time[0]} =␣
 ↪{dense_time[1]/dense_time[0]}")
print(f"Sparse running time increase factor: {sparse_time[1]}/{sparse_time[0]}␣
 ↪= {sparse_time[1]/sparse_time[0]}")

print(f"Dense memory increase factor: {dense_mem[1]}/{dense_mem[0]} =␣
 ↪{dense_mem[1]/dense_mem[0]}")
print(f"Sparse memory increase factor: {sparse_mem[1]}/{sparse_mem[0]} =␣
 ↪{sparse_mem[1]/sparse_mem[0]}")
```

```
n = 100 (100 runs): dense average running time = 0.0035157012939453125 seconds
with std 0.007749172022943856
```

```
n = 100 (100 runs): sparse average running time = 0.0006611752510070801 seconds
with std 0.0003330868330316168
For n = 100, sparse is 0.0028545260429382324 seconds faster than dense
--------------------------------------------------
n = 100 (100 runs): dense memory occupation: 10000 elements
n = 100 (100 runs): sparse memory occupation: 691 elements
--------------------------------------------------
n = 1000 (100 runs): dense average running time = 0.03292563438415527 seconds
with std 0.020021581623980735
n = 1000 (100 runs): sparse average running time = 0.001732015609741211 seconds
with std 0.001311362415144259
For n = 1000, sparse is 0.03119361877441406 seconds faster than dense
--------------------------------------------------
n = 1000 (100 runs): dense memory occupation: 1000000 elements
n = 1000 (100 runs): sparse memory occupation: 6990 elements
--------------------------------------------------
Dense running time increase factor: 0.03292563438415527/0.0035157012939453125 =
9.365310540135848
Sparse running time increase factor: 0.001732015609741211/0.0006611752510070801
= 2.619601394793684
Dense memory increase factor: 1000000/10000 = 100.0
Sparse memory increase factor: 6990/691 = 10.115774240231548
```

We can see that the method for sparse LU decomposition is faster than the method for dense LU decomposition for both values of n. For a larger n, LU decomposition is slower for both the dense and sparse method. The difference between the two methods is larger when n is larger. The running time for the dense method increases with a factor of approximately 9.4 whereas the running time of the sparse method only increases with a factor of approximately 2.6. The increased difference is thus due to the larger increase in running time for the dense method. The difference in running time can be explained by the difference in memory occupation between the two methods.

We can see that the number of values stored is much less for the sparse method, for both $n = 100$ and $n = 1000$. The sparse method does not explicitly store the zeros in the matrix, where the dense method does store those values. When the size of A is increased by a factor 100, the dense method needs to increase the number of stored elements with a factor 100. In this example, the sparse method only needs to increase the number of stored values by a factor of 10 (approximately). This is a big difference, leading to a much smaller running time.

5.3 (c)

For $n = 100$, what is the condition number? What accuracy do you expect based on the condition number?

[ ]:
```python
n = 100
A_dense = create_beam_matrix_A(n)

condA_dense = np.linalg.cond(A_dense, p = np.inf)
print("Condition number of A dense with infinity norm:", condA_dense)
print(f"cond(A)*machine_epsilon = {condA_dense * 2*10**(-16)}")
```

```
print(f"log(cond(A)) = {math.log(condA_dense,10)}")
```

```
Condition number of A dense with infinity norm: 200666800.074768
cond(A)*machine_epsilon = 4.01333600149536e-08
log(cond(A)) = 8.302475525267644
```

For $n = 100$ we obtain $\text{cond(A)} \approx 200666800.0746656$. The relative error in the approximate solution $\hat{\mathbf{x}}$ is therefore approximately bounded by $\text{cond(A)}_{\text{mach}} \approx 4.0133 \cdot 10^{-8}$. Therefore we expect to lose about $\log_{10}(\text{cond(A)}) \approx \log_{10}(200666800.0746656) \approx 8.3$ digits of accuracy relative to the accuracy of the input. Thus we expect no correct digits unless the input data and working memory are accurate to more than 8 decimal digits.

5.4 (d)

How well do the answers of (b) agree with each other (make an appropriate quantitative comparison)?

Should we be worried about the fact that the two answers are different?

[ ]:
```
def relative_residual(x_hat,A,b):
    r = b-A.dot(x_hat)
    relative_res = np.linalg.norm(r)/(np.linalg.norm(A)*np.linalg.norm(b))
    return relative_res

n = 100
_,_,_,_,x_dense, x_sparse = time_dense_sparse(n)
b = np.full(n, 1/(n^4))
A_dense = create_beam_matrix_A(n)
diff_x = x_sparse-x_dense
rel_diff_x = (x_sparse-x_dense)/x_sparse

print(f"Difference in solutions: {diff_x}")
print(f"Relative residual for dense method:␣
 ↪{relative_residual(x_dense,A_dense,b)}")
print(f"Relative residual for sparse method:␣
 ↪{relative_residual(x_sparse,A_dense,b)}")

print(f"maximum relative difference: {max(rel_diff_x)}")
```

```
Difference in solutions: [6.69851730e-10 4.00341094e-09 9.98497285e-09
1.85988256e-08
 2.98292662e-08 4.36610890e-08 6.00787189e-08 7.90628292e-08
 1.00593525e-07 1.24651706e-07 1.51217591e-07 1.80272309e-07
 2.11797214e-07 2.45775482e-07 2.82188921e-07 3.21017978e-07
 3.62245373e-07 4.05851097e-07 4.51819687e-07 5.00132955e-07
 5.50775439e-07 6.03729859e-07 6.58981662e-07 7.16510840e-07
 7.76301022e-07 8.38343112e-07 9.02613465e-07 9.69088433e-07
 1.03775528e-06 1.10859037e-06 1.18157186e-06 1.25666702e-06
 1.33384310e-06 1.41306009e-06 1.49427797e-06 1.57748218e-06
```

```
 1.66265454e-06 1.74975867e-06 1.83875454e-06 1.92960579e-06
 2.02227238e-06 2.11673614e-06 2.21296796e-06 2.31093145e-06
 2.41059024e-06 2.51190068e-06 2.61481182e-06 2.71928729e-06
 2.82529800e-06 2.93282210e-06 3.04184505e-06 3.15236684e-06
 3.26434383e-06 3.37774691e-06 3.49253241e-06 3.60864215e-06
 3.72603972e-06 3.84467421e-06 3.96451651e-06 4.08554479e-06
 4.20772267e-06 4.33102105e-06 4.45541809e-06 4.58089926e-06
 4.70741361e-06 4.83493204e-06 4.96344001e-06 5.09290840e-06
 5.22333721e-06 5.35466825e-06 5.48685784e-06 5.61987690e-06
 5.75368176e-06 5.88825787e-06 6.02356158e-06 6.15957833e-06
 6.29627903e-06 6.43357635e-06 6.57141209e-06 6.70974259e-06
 6.84855331e-06 6.98780059e-06 7.12748442e-06 7.26760481e-06
 7.40813266e-06 7.54903886e-06 7.69030885e-06 7.83195719e-06
 7.97395478e-06 8.11628706e-06 8.25892494e-06 8.40179564e-06
 8.54485552e-06 8.68807547e-06 8.83142638e-06 8.97490827e-06
 9.11852112e-06 9.26226494e-06 9.40609607e-06 9.54994175e-06]
Relative residual for dense method: 4.3991116791007416e-11
Relative residual for sparse method: 4.7239361960273103e-11
maximum relative difference: 7.309983775724946e-11

/Users/loesbijman/anaconda3/envs/my-env/lib/python3.12/site-
packages/scipy/sparse/linalg/_dsolve/linsolve.py:412: SparseEfficiencyWarning:
splu converted its input to CSC format
  warn('splu converted its input to CSC format', SparseEfficiencyWarning)
```

We can see that the solutions found by the two methods are different. Furthermore, their relative residual is different. The sparse method has a smaller residual, which implies a slightly smaller backward error in the method. We should not be worried that the solutions are not the same, as they are quite similar. The largest relative difference between the two solutions is approximately $7.31 \cdot 10^{-11}$.