7 Solving systems of linear algebraic equations. Introductory discussion

I now start the second part of the course, which will be devoted to different methods to solve numerically systems of linear algebraic equations. I will only look into the case when the systems I am dealing with have exactly n equations and n unknowns. In this case, these systems take the following general form

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2, \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n, \end{aligned} \tag{7.1}$$

which can be written much more succinctly in the vector form

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{7.2}$$

where I introduced matrix $\boldsymbol{A}$

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{bmatrix} \tag{7.3}$$

and column-vectors $\boldsymbol{x}$ and $\boldsymbol{b}$

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}. \tag{7.4}$$

It should be clear that the data we get is $\boldsymbol{A}, \boldsymbol{b}$ and our task is to determine $\hat{\boldsymbol{x}}$ such that, if I plug this vector into the equation, the equation will tern into identity.

In the following I assume that the student knows how to add and multiply matrices, knows basic facts about determinants, inverse matrices, elementary row operations, row reduced echelon form, etc. If you feel that you need some light refresher on these topics, I would recommend reading through the first four sections of my Intermediate Linear Algebra lecture notes.

A very basic fact from Linear Algebra states that

Theorem 7.1. *System* (7.1) *(or* (7.2)*) has a unique solution if and only if*

$$\det \boldsymbol{A} \neq 0.$$

There are a great number of different methods to solve (7.1), in this section I will talk about two most basic ones: Cramer's rule and Gaussian elimination and back substitution. Before I discuss these methods I would like to remark that if a student encounters a system of linear algebraic equations in

Math 488/688: Numerical Analysis
e-mail: artem.novozhilov@ndsu.edu. Fall 2021

their regular activities (in other classes, in some simple computations) and if the number of unknowns if not exceeding 3 (or 4), just use the common sense and eliminate the variables one by one using the simplest arithmetically path. Do not try to reduce the matrix $[\boldsymbol{A} \mid \boldsymbol{b}]$ to the echelon form, this is just waste of your time.

7.1 Cramer's method

To formulate Cramer's method let me recall first the formula to compute a determinant of a square matrix $\boldsymbol{A}$. I will use the formula that uses the expansion with respect to the i-th row. The formula reads

$$\det \boldsymbol{A} = \sum_{j=1}^{n} (-1)^{i+j} a_{ij} \det \boldsymbol{A}_{ij}, \tag{7.5}$$

where $\boldsymbol{A}_{ij}$ is the submatrix of matrix $\boldsymbol{A}$ obtained by removing the i-th row and j-th column from $\boldsymbol{A}$. Clearly, the size of $\boldsymbol{A}_{ij}$ is $(n-1)\times(n-1)$. Note the formula (7.5) is recursive, i.e., it computes the determinant of $n \times n$ matrix through determinants of several $(n-1)\times(n-1)$ matrices. For this kind of definitions one must specify the boundary case. For us it may be

$$\det[a_{ij}] = a_{ij},$$

i.e., the determinant of 1×1 matrix is equal to its only element. Or, for a 2×2 matrix,

$$\det \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = a_{11}a_{22} - a_{12}a_{21}.$$

Expression (7.5) will give the same answer for any $i = 1, \ldots, n$ (if done by hand pick a row with largest numbers of zeros). A similar formula is valid for the expansion of the determinant with respect to the j-th column.

Now I can state Cramer's method:

$$x_i = \frac{\det \boldsymbol{A}_j}{\det \boldsymbol{A}}, \quad i = 1, \ldots, n,$$

where $\boldsymbol{A}_j$ is an $n \times n$ matrix obtained from $\boldsymbol{A}$ by replacing the j-th column of $\boldsymbol{A}$ with the vector $\boldsymbol{b}$. You can see that this method can serve as a proof of the stated above theorem, if you know how to justify Cramer's formulas for the solution of (7.1).

Example 7.2. `To add an explicit example`

7.2 Gaussian elimination with backsubstitutions

There are a few modifications of the algorithm, which is usually called *Gaussian elimination* in introductory Linear Algebra classes, so let me be very precise with what I call *Gaussian elimination with backsubstitution.* This particular method consists of two steps. In the first step I sequentially eliminate the variables $x_1, x_2, \ldots$ form the equations by using only one elementary matrix operation, namely, multiplication of a row of a matrix with a scalar and adding the result to another row. First I eliminate x_1 from the first equation by multiplying the first equation by $-a_{21}/a_{11}$ and adding it to the second equation; then I eliminate x_1 from the third equation by multiplying the first equation with $-a_{31}/a_{11}$ and adding to the third equation, etc. When I am done with all x_1 such that only the first equation contains this variable, I move to the second equation to eliminate x_2 from all the equations other than 1 and 2, etc. This process of course can be nicely represented with the help of the augmented matrix, and here are my first and final states of the first step (which I call the forward step):

$$\left[\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \vdots & & & & \\ a_{n1} & a_{n2} & \dots & a_{nn} & b_n \end{array}\right] \longrightarrow \left[\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ 0 & a'_{22} & \dots & a'_{2n} & b'_2 \\ \vdots & & & & \\ 0 & 0 & \dots & a'_{nn} & b'_n \end{array}\right],$$

where the primes indicate that these coefficients are in general different from the original values (note that I do not require that the entries on the main diagonal to be equal to 1s as it is sometimes done in Linear Algebra class).

Thus the forward step brings matrix $\boldsymbol{A}$ to the upper triangular form, and in the second, or backward, stet I need to solve the upper triangular system, which can certainly be done sequentially starting from the last equation. For instance, I have

$$x_n = \frac{b'_n}{a'_{nn}}, \quad x_{n-1} = \frac{b'_{n-1}}{a'_{n-1,n-1}} - \frac{a'_{n-1,n}}{a'_{n-1,n-1}} x_n, \quad \dots$$

Example 7.3. `To add an explicit example`

Looking at these two method, one can see immediately that Cramer's method has an advantage of checking in advance whether the given system has a unique solution: it requires calculating $\det \boldsymbol{A}$. If this determinant 0 then one should stop any calculations if the goal to find the unique solution. Even worse, Gauss method with backsubstitution uses divisions by $a_{11}, a'_{22}, \dots$. Each of these numbers can be zero, and hence it is totally possible not to be able to use this method even in the case when $\det \boldsymbol{A} \neq 0$. Are these reasonings enough to choose Cramer's method over the Gauss method? Let me implement both in Python and see something, which may not be obvious at this point: namely, Cramer's method has one significant disadvantage that outweighs everything else.

7.3 Python implementation

I will start with the listing for Cramer's method. Note that I use recursion to compute the determinant as suggested by the formula. Other than that, the implementation is straightforward, and does not introduce any new programming concepts.

```python
def submatrix(A: list, col: int):
    '''takes matrix A and integer col and
    returns the submatrix obtained from A be removing
    the first row and col-th column.
    '''

    if len(A) < 2: raise Exception('Matrix is too small')

    return [[A[i][j] for j in range(len(A)) if j != col] for i in range(1, len(A))] #list
        comprehention
```

```python
def det(A: list):
    '''computes the determininat of matrix A by using the
    expansion with respect to the first row. The function is recursive.
    '''

    if len(A) == 2: #determinant of 2 by 2 matrix
        return A[0][0]*A[1][1]-A[0][1]*A[1][0]

    tmp = 0
    for i in range(len(A)):
        tmp += (-1) ** i * A[0][i] * det(submatrix(A, i)) #this is a recursive function

    return tmp

def replace_column(A: list, b: list, index: int):
    '''takes matrix A, vector b, and integer index, and returns a matrix
    obtained from A by replacing column with index index with vector b.
    Matrix A remains unchanged.
    '''

    B = [x[:] for x in A] #create a new copy of A without changing A
    for j in range(len(A)):
        B[j][index] = b[j]

    return B

def cramer_method(A: list, b: list):
    '''Takes as an input matrix A and vector b and uses Cramer's method
    to solve the system Ax=b. In particular, if det(A) is not zero, then
    x[i] = det(Aj)/det(A), where Aj is the matrix obtained from A by
    replacing the j-th coilumn with vector b.
    '''

    x = [0] * len(A)
    detA = det(A)

    if detA == 0:
        raise Exception('The matrix is singular')

    for i in range(len(A)):
        tmp = replace_column(A, b, i)
        x[i] = det(tmp)/detA

    return x
```

Probably the only line that requires an additional explanation is the line

```python
B = [x[:] for x in A] #create a new copy of A without changing A
```

that is used to create a new copy of given matrix `A`. Recall that in Python I represent matrices as lists of lists (at least for this moment, we will see a better way soon). Lists in Python are mutable objects, and it is nice to think about lists as just collections of references to the actual objects in physical memory. If `A` and `B` are two lists, the assignment $A = B$ does not create a new copy of list `B`, it creates just another reference (name) to the same object. Now if I change one of the objects, e.g., `A[0]='New element'` then `B[0]` will be pointing to this new element. So I mentioned before that if you need to create a new copy of a list, you need to do something like `A = list(B)` or `A = B.copy()`. Great. But it will not work for lists of lists! The reason is that in this case, say, `A[0]` is a list again, i.e., a collection of references to actual physical objects, and doing `A = list(B)` will create a new physical copy of this list of references that will still point to the same objects. So I need a deeper level of copying. There are various ways doing this, one that is above works fine.

Next I will write the code for the Gauss method. Here it is advisable first to carefully write down the general formulas for the computations of new elements, which can be easily translated into code (do not forget that the first element of Python list has index 0). Assume that I already worked through the first $i-1$ variables, and my matrix has the form.

$$
\left[\begin{array}{cccccc|c}
a_{11} & a_{12} & \dots & \dots & \dots & a_{1n} & b_1 \\
0 & a_{22} & \dots & \dots & \dots & a_{2n} & b_2 \\
0 & 0 & \ddots & & & & \vdots \\
\vdots & & & a_{ii} & \dots & a_{in} & b_i \\
\vdots & & & a_{i+1,i} & \dots & a_{i+1,n} & b_{i+1} \\
 & & & \vdots & & \vdots & \vdots \\
0 & & \dots & a_{ni} & \dots & a_{nn} & b_n
\end{array}\right]
$$

I need to compute for $j, k = i+1, \dots, n$

$$
\begin{aligned}
a'_{j,k} &= a_{jk} - \frac{a_{ji}}{a_{ii}} a_{ik}, \\
b'_j &= b_j - \frac{a_{ji}}{a_{ii}} b_i.
\end{aligned}
$$

After this calculation is done I change i to $i+1$ until I reach the last row.

For the backsubstitution (assuming that $\boldsymbol{A}$ already has the upper triangular form), I have

$$
x_i = \frac{1}{a'_{ii}} \left(b'_i - \sum_{j=i+1}^{n-1} a'_{ij} x_j \right), \quad i = n, n-1, n-2, \dots, 1.
$$

```python
def gauss_method(A: list, b: list):
    '''takes function A and vector b and attempts
    to solve the system Ax=b. The solution proceeds
    in two steps. In the first, forward step, matrix A
    is put into upper an triangular form by elementary
    row operations (vector b is changed accordingly).
    In the second, backward, step, this upper triangular
    system is solved interatively starting with the
    last equation.
    '''

    def forward_gauss(A: list, b: list):#puts A in upper triangular form
        #note that A will be changed after function call.

        m = len(A)
        for i in range(m - 1):#go through all rows except the last one
            if A[i][i] == 0: raise Exception('Division by zero!')
            for j in range(i + 1, m):
                k = - A[j][i]/A[i][i]
                for l in range(i + 1,m):
                    A[j][l] += k * A[i][l]
                A[j][i] = 0 #fill with zeros, optional, needed only for display
                b[j] += k * b[i]

    def backward_gauss(A: list, b: list):#solves upper triangular system

        x = [0]*len(b)
        for i in range(1, len(A)+1):
            x[-i] = b[-i]
            for j in range(i-1, 0, -1):
                x[-i] -= A[-i][-j] * x[-j]
            x[-i] /= A[-i][-i]
        return x

    #main body of gauss_method
    copyA = [x[:] for x in A] #make a copy of A
    copyb = list(b) #make a copy of b

    forward_gauss(copyA, copyb) #brings the augmented matrix [A|b] to the triangular form
    return backward_gauss(copyA, copyb) #solves the triangular system
```

In the last listing, note that I define auxiliary functions `forward_gauss` and `backward_gauss` inside the function `gauss_method` such that they cannot be called outside of this function. Also check how I implement the backsubstitution by using Python's negative indexes.

Now everything is ready for the test. To make a somewhat more automatized test, I first create a random matrix $\boldsymbol{A}$ and a solution $\hat{\boldsymbol{x}}$, after it I compute $\boldsymbol{b} = \boldsymbol{A}\hat{\boldsymbol{x}}$ and call my functions with the data $\boldsymbol{A}$ and $\boldsymbol{b}$. To see how my found solution is far from the original solution I compute the Euclidian norm

$$\|\boldsymbol{x} - \boldsymbol{y}\| = \sqrt{(x_1 - y_1)^2 + \ldots + (x_n - y_n)^2}$$

for two vectors $\boldsymbol{x}, \boldsymbol{y}$, which measures the distance between them.

```
import random

def prod(A: list, b: list):
    '''computes the product of matrix A and
    vector b and returns it as a result.
    '''
    m = len(A)
    x = [0]*m #the result will be stored in x

    for i in range(m):
        for j in range(m):
            x[i] += A[i][j] * b[j]

    return x

def distance(x: list, y: list):
    '''computes the Euclidian distance
    between vectors x and y.
    '''

    tmp = 0
    for i in range(len(x)):
        tmp += (x[i] - y[i]) ** 2

    return tmp**(0.5)
```

```
matrix_size = 8
int_bounds = 500
sol_bounds = 100000

A = [[float(random.randint(-int_bounds, int_bounds)) for j in range(matrix_size)]
     for i in range(matrix_size)]

sol = [random.randint(-sol_bounds, sol_bounds) for j in range(matrix_size)]

b = prod(A, sol)

sol1 = cramer_method(A,b)
sol2 = gauss_method(A,b)

print('For Cramer method the accuracy is', distance(sol, sol1))
print('For Gauss method the accuracy is', distance(sol, sol2))
#the result is
For Cramer method the accuracy is 9.929853824092895e-11
For Gauss method the accuracy is 1.555700240237909e-08
```

We can see that even for 8×8 matrix there is a loss of accuracy for both methods (I will very soon come to this point later), but the overall performance is comparable. And here is finally an example that will explain why no one will ever use Cramer's method to solve linear systems:

```
import time
matrix_size = 9 #I want to solve 9 by 9 system
... #rest of the commands to create A and b
t0 = time.time()
sol1 = cramer_method(A,b)
t1 = time.time()
```

```
print(f"Time required for computation is {t1-t0}")
#the result is
Time required for computation is 14.23080587387085
```

On my (fairly outdated I have to admit) desktop solving a 9×9 system takes whooping 14 seconds! If you decided to see what happens on your computer, do not try to increase the matrix size beyond 10.

Gauss method, however, can easily deal with systems 300×300:

```
import time
matrix_size = 300 #I want to solve 9 by 9 system
... #rest of the commands to create A and b
t0 = time.time()
sol1 = gauss_method(A,b)
t1 = time.time()
print(f"Time required for computation is {t1-t0}")
#the result is
Time required for computation is 2.6961541175842285
```

In the next section I will get to the bottom of this.