Practical Numerical and Scientific Computing with MATLAB® and Python

Eihab B. M. Bashier

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

Practical Numerical and Scientific Computing with MATLAB® and Python

Practical Numerical and Scientific Computing with MATLAB® and Python

Eihab B. M. Bashier

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

MATLAB® is a trademark of The MathWorks, Inc. and is used with permission. The Mathworks does not warrant the accuracy of the text or exercises in this book. This book's use or discussion of MATLAB® software or related products does not constitute endorsement or sponsorship by The MathWorks of a particular pedagogical approach or particular use of the MATLAB® software.

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper

International Standard Book Number-13: 978-0-367-07669-6 (Hardback)

Library of Congress Cataloging-in-Publication Data

Names: Bashier, Eihab Bashier Mohammed, author.
Title: Practical Numerical and Scientific Computing with MATLAB®
and Python / Eihab B.M. Bashier.
Description: Boca Raton : CRC Press, 2020. | Includes bibliographical
references and index.
Identifiers: LCCN 2019052363 | ISBN 9780367076696 (hardback) | ISBN
9780429021985 (ebook)
Subjects: LCSH: Science--Data processing. | MATLAB. | Python (Computer
program language)
Classification: LCC Q183.9 B375 2020 | DDC 502.85/53--dc23
LC record available at https://lccn.loc.gov/2019052363

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

To my parents, family and friends.

Contents

Preface

The past few decades have witnessed tremendous development in the manufacture of computers and software, and scientific computing has become an important tool for finding solutions to scientific problems that come from various branches of science and engineering. Nowadays, scientific computing has become one of the most important means of research and learning in the fields of science and engineering, which are indispensable to any researcher, teacher, or student in the fields of science and engineering.

One of the most important branches of scientific computing is a numerical analysis which deals with the issues of finding approximate numerical solutions to such problems and analyzing errors related to such approximate methods. Both the MATLAB® and Python programming languages provide many libraries that can be used to find solutions of scientific problems visualizing them. The ease of use of these two languages became the most languages that most scientists who use computers to solve scientific problems care about.

The idea of this book came after I taught courses of scientific computing for physics students, introductory and advanced courses in mathematical software and mathematical computer applications in many Universities in Africa and the gulf area. I also conducted some workshops for mathematics and science students who are interested in computational mathematics in some Sudanese Universities. In these courses and workshops, MATLAB and Python were used for the implementation of the numerical approximation algorithms. Hence, the purpose of introducing this book is to provide the student with a practical guide to solve mathematical problems using MATLAB and Python software without the need for third-party assistance. Since numerical analysis is concerned with the problems of approximation and analysis of errors of numerical methods associated with approximation methods, this book is more concerned with how these two aspects are applied in practice by software, where illustrations and tables are used to clarify approximate solutions, errors and speed of convergence, and its relations to some of the numerical method parameters, such as step size and tolerance. MATLAB and Python are the most popular programming languages for mathematicians, scientists, and engineers. Both the two programming languages possess various libraries for numerical and symbolic computations and data representation and visualization. Proficiency with the computer programs contained in this book requires that the student have prior knowledge of the basics of the programming languages MATLAB and Python, such as branching, Loops, symbolic packages, and the graphical libraries. The MATLAB version used for this book is 2017b and the Python version is 3.7.4.

The book consists of 11 chapters divided into three parts: the first part is concerned with discussing numerical solutions for linear and nonlinear systems and numerical difficulties facing these types of problems with how to overcome these numerical difficulties. The second part deals with methods of completing functions, differential and numerical integration, and solutions of differential equations. The last part of the book discusses methods to solve linear and nonlinear programming and optimal control problems. It also contains some specialized software in Python language to solve some problems numerically. These software packages must be downloaded from a third party, such as Gekko which is used for the solutions of differential equations and linear and nonlinear programming in addition to the optimal control problems. Also, the Pulp package is used to solve linear programming problems and finally Pyomo a package is used for solving linear and nonlinear programming problems. How to install and run such a package is also presented in the book.

What distinguishes this book from many other numerical analysis books is that it contains some topics that are not usually found in other books, such as nonstandard finite difference methods for solving differential equations and solutions of optimal control problems. In addition, the book discusses implementations of methods with high convergence rates, such as Gauss integration methods discussed in the numerical differentiation and integration, exact finite difference schemes for solving differential equations discussed in the nonstandard finite differences Chapter. It also uses efficient python-based software for solving some kinds of mathematical problems numerically.

The parts of the book are separate from each other so that the student can study any part of it without having to read the previous parts of that part. The exception to this is the optimal control chapter in the third part, which requires studying numerical methods to solve the differential equations discussed in the second part.

After reading this book and implementing the programs contained on it, a student will be able to deal with and solve many kinds of mathematical problems such as differential equations, static, and dynamical optimization problems and apply the methods to real-life problems.

Acknowledgment

I am very grateful to the African Institute of Mathematical Sciences (AIMS), Cape Town, South Africa, which hosted me on a research visit during which some parts of this book have been written. I would also like to thank the editorial team of this book under the leadership of publisher, Randi Cohen, for their continuous assistance in formatting, coordinating, editing, and directing the book throughout all stages. Special thanks go to all professors who taught me the courses of numerical analysis in the various stages of my under- and post-graduate studies, and, in particular, I thank Dr. Mohsin Hashim University of

Khartoum, Professor Arieh Iserles African Institute of Mathematical Sciences, Professor Kailash Patidar University of the Western Cape in Cape Town, and the spirit of my great teacher Professor David Mackey (Cambridge University and the African Institute for Mathematical Sciences) who passed away four years ago. Finally, I am also grateful to my family for continuous encouragement and patience while writing this book.

MATLAB® is a registered trademark of The MathWorks, Inc. For product information, please contact:

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA, 01760-2098 USA
Tel: 508-647-7000
Fax: 508-647-7001
E-mail: info@mathworks.com
Web: www.mathworks.com

Author

Eihab B. M. Bashier obtained his PhD in 2009 from the University of the Western Cape in South Africa. His permanent job is in the Department of Applied Mathematics on the faculty of mathematical sciences and information technology, University of Khartoum. Currently, he is an Associate Professor of Applied Mathematics at the College of Arts and Applied Sciences at Dhofar University, Oman. His research interests are mainly in numerical methods for differential equations with applications to biology and in information and computer security with a focus in cryptography. In 2011, Dr. Bashier won the African Union and the Third World Academy of Science (AU-TWAS) Young Scientists National Award in Basic sciences, Technology and Innovation. Dr. Bashier is a reviewer for some international journals and a member of the IEEE and the EMS. (Email: eihabbashier@gmail.com, eihabbash@aims.ac.za).

Part I

Solving Linear and Nonlinear Systems of Equations

1

Solving Linear Systems Using Direct Methods

Abstract

Linear systems of equations have many applications in mathematics and science. Many of the numerical methods used for solving mathematics problems such as differential or integral equations, polynomial approximations of transcendental functions and solving systems of nonlinear equations arrive at a stage of solving linear systems of equations. Hence, solving a linear system of equations is a fundamental problem in numerical computing.

This chapter discusses the direct methods for solving linear systems of equations, using Gauss and Gauss-Jordan elimination techniques and the matrix factorization approach. MATLAB® and Python implementations of such algorithms are provided.

1.1 Testing the Existence of the Solution

A linear system consisting of m equations in n unknowns, can be written in the matrix form:

$$A\boldsymbol{x} = \boldsymbol{b} \tag{1.1}$$

where,

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{21} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}, \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \text{ and } \boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

Here, the coefficients a_{ij} of matrix $A \in \mathbb{R}^{m\times n}$ are assumed to be real, $\boldsymbol{x} \in \mathbb{R}^n$ is the vector of unknowns and $\boldsymbol{b} \in \mathbb{R}^m$ is a known vector. Depending on the relationship between m and n three kinds of linear systems are defined [30, 53]:

1. **overdetermined linear systems:** there are more equations than unknown ($m > n$).

2. **determined linear systems:** equal numbers of equations and unknowns ($m = n$).

3. **underdetermined linear systems:** there are more unknowns than equations ($m < n$).

Let $\tilde{A} = [A \mid \boldsymbol{b}]$ be the augmented matrix of the linear system $A\boldsymbol{x} = \boldsymbol{b}$. Then, the existence of a solution for the given linear system is subject to one of the two following cases:

1. $rank(\tilde{A}) = rank(A)$: in this case, there is at least one solution, and we have two possibilities:

   (a) $rank(\tilde{A}) = rank(A) = n$: in this case there is a unique solution.

   (b) $rank(\tilde{A}) = rank(A) < n$: in this case there is infinite number of solutions.

2. $rank(\tilde{A}) > rank(A)$: in this case there is no solution and we can look for a least squares solution.

If the linear system $A\boldsymbol{x} = \boldsymbol{b}$ has a solution, it is called a `consistent` linear system, otherwise, it is an `inconsistent` linear system [30].

In MATLAB, the command rank can be used to test the rank of a given matrix A.

```
>> A = [1 2 3; 4 5 6; 7 8 9]
A =
1     2     3
4     5     6
7     8     9
>> b = [1; 1; 1]
b =
1
1
1
>> r1 = rank(A)
r1 =
2
>> r2 = rank([A b])
r2 =
2
```

In python, the function `matrix_rank` (located in `numpy.linalg`) is used to compute the rank of matrix A and the augmented system $[Ab]$.

```
In [1]: import numpy as np
In [2]: A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
In [3]: b = np.array([1, 1, 1])
```

```
In [4]: r1, r2 = np.linalg.matrix_rank(A), np.linalg.matrix_rank
    (np.c_[A, b])
In [5]: r1
Out[5]: 2
In [6]: r2
Out[6]: 2
```

In the special case when $m = n$ (A is a squared matrix) and there is a unique solution ($rank(\tilde{A}) = rank(A) = n$), this unique solution is given by:

$$\boldsymbol{x} = A^{-1}\boldsymbol{b}.$$

Hence, finding the solution of the linear system requires the inversion of matrix A.

1.2 Methods for Solving Linear Systems

This section considers three special types of linear systems which are linear systems with diagonal, upper triangular and lower triangular matrices.

1.2.1 Special Linear Systems

We consider the linear system:

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

where $A \in \mathbb{R}^{n\times n}$, $\boldsymbol{x}$ and $\boldsymbol{b} \in \mathbb{R}^n$. We consider two cases.

1. **A is a diagonal matrix:**
   In this case, matrix A is of the form:

$$A = \begin{pmatrix} a_{11} & 0 & 0 & \dots & 0 \\ 0 & a_{22} & 0 & \dots & 0 \\ 0 & 0 & a_{33} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & a_{nn} \end{pmatrix}$$

   which leads to the linear system:

$$\begin{pmatrix} a_{11} & 0 & 0 & \dots & 0 \\ 0 & a_{22} & 0 & \dots & 0 \\ 0 & 0 & a_{33} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix} \tag{1.2}$$

The solution of the linear system (1.2) is given by:

$$x_i = \frac{b_i}{a_{ii}}$$

The MATLAB code to compute this solution is given by:

```
function x = SolveDiagonalLinearSystem(A, b)
    % This function solves the linear system Ax = b, where ...
        A is a diagonal matrix
    % b is a known vector and n is the dimension of the ...
        problem.
    n = length(b) ;
    x = zeros(n, 1) ;
    for j = 1: n
        x(j) = b(j)/A(j, j) ;
    end
```

We can apply this function to solve the diagonal system:

$$\begin{pmatrix} 2 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 1 \\ -3 \end{pmatrix}$$

by using the following MATLAB commands:

```
>> A = diag([2, -1, 3])
A =
2     0     0
0    -1     0
0     0     3
>> b = [4; 1; 3]
b =
4
1
3
>> x = SolveDiagonalLinearSystem(A, b)
x =
2
-1
1
```

The python code of the function `SolveDiagonalLinearSystem` is as follows.

```
import numpy as np
def SolveDiagonalLinearSystem(A, b):
    n = len(b)
```

```
    x = np.zeros((n, 1), 'float')
    for i in range(n):
        x[i] = b[i]/A[i, i]
    return x
```

```
In [7]: A = np.diag([2, -1, 3])
In [8]: b = np.array([4, -1, 3])
In [9]: x = SolveDiagonalLinearSystem(A, b)
In [10]: print('x = \n', x)
x =
[[ 2.]
[ 1.]
[ 1.]]
```

2. **A is an upper triangular matrix:**
   In this case, matrix A is of the form:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22} & a_{23} & \dots & a_{2n} \\ 0 & 0 & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & a_{nn} \end{pmatrix}$$

   Therefore, we have the linear system:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22} & a_{23} & \dots & a_{2n} \\ 0 & 0 & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix} \tag{1.3}$$

   In this case we use the back substitution method for finding the solution of system 1.3. The MATLAB function `SolveUpperSystem.m` solves the linear system 1.3 using the back-substitution method.

```
function x = SolveUpperLinearSystem(A, b)
    % This function uses the backward substitution method ...
        for solving
    % the linear system Ax = b, where A is an upper ...
        triangular matrix
    % b is a known vector and n is the dimension of the ...
        problem.
    n = length(b) ;
    x = zeros(n, 1) ;
    x(n) = b(n)/A(n, n) ;
    for j = n-1: -1 : 1
        x(j) = b(j) ;
```

```
        for k = j+1 : n
            x(j) = x(j) - A(j, k)*x(k) ;
        end
        x(j) = x(j)/A(j, j) ;
    end
```

The python code for the `SolveUpperSystem`, is as follows.

```
import numpy as np
def SolveUpperLinearSystem(A, b):
    n = len(b)
    x = np.zeros((n, 1), 'float')
    x[n-1] = b[n-1]/A[n-1, n-1]
    for i in range(n-2, -1, -1):
        x[i] = b[i]
        for j in range(i+1, n):
            x[i] -= A[i, j]*x[j]
        x[i] /= A[i, i]
    return x
```

3. **A is a lower triangular system:**
   In this case, matrix A is of the form:

$$A = \begin{pmatrix} a_{11} & 0 & 0 & \dots & 0 \\ a_{21} & a_{22} & 0 & \dots & 0 \\ a_{31} & a_{32} & a_{33} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{pmatrix}$$

   Therefore, we have the linear system:

$$\begin{pmatrix} a_{11} & 0 & 0 & \dots & 0 \\ a_{21} & a_{22} & 0 & \dots & 0 \\ a_{31} & a_{32} & a_{33} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix} \tag{1.4}$$

   The forward substitution method is used to find the solution of system 1.4. The MATLAB function `SolveLowerSystem.m` solves the linear system 1.4 using the forward-substitution method.

```
function x = SolveLowerLinearSystem(A, b)
    % This function uses the forward substitution method ...
        for solving
    % the linear system Ax = b, where A is an lower ...
        triangular matrix
    % b is a known vector and n is the dimension of the ...
        problem.
```

```
    n = length(b) ;
    x = zeros(n, 1) ;
    x(1) = b(1)/A(1, 1) ;
    for j = 2 : n
        x(j) = b(j) ;
        for k = 1 : j-1
            x(j) = x(j) - A(j, k)*x(k) ;
        end
        x(j) = x(j)/A(j, j) ;
    end
```

The python code of the function `SolveLowerSystem` is as follows.

```
def SolveLowerLinearSystem(A, b):
    import numpy as np
    n = len(b)
    x = np.zeros((n, 1), 'float')
    x[0] = b[0]/A[0, 0]
    for i in range(1, n):
        x[i] = b[i]
        for j in range(i):
            x[i] -= A[i, j]*x[j]
        x[i] /= A[i, i]
    return x
```

1.2.2 Gauss and Gauss-Jordan Elimination

Gauss and Gauss-Jordan elimination methods are related to each other. If given a matrix $A \in \mathbb{R}^{n\times n}$, then both Gauss and Gauss-Jordan apply elementary row operations through consequent steps over matrix A. The Gauss method stops after obtaining the row echelon form of matrix A (If A is nonsingular, then its row echelon form is an upper triangular matrix), whereas Gauss-Jordan continuous until reaching the `reduced row echelon form` (If A is nonsingular, then its reduced row echelon form is the identity matrix).

To illustrate the differences between the row echelon and the reduced row echelon forms, the two forms are computed for the matrix:

$$A = \begin{pmatrix} 4 & -1 & -1 \\ -1 & 4 & -1 \\ -1 & -1 & 4 \end{pmatrix}$$

Starting by finding the row echelon form for the given matrix.

$$A = \begin{pmatrix} 4 & -1 & -1 \\ -1 & 4 & -1 \\ -1 & -1 & 4 \end{pmatrix} \xrightarrow[R_3 \leftarrow 4R3+R_1]{R_2 \leftarrow 4R2+R_1} \begin{pmatrix} 4 & -1 & -1 \\ 0 & 15 & -5 \\ 0 & -5 & 15 \end{pmatrix} \xrightarrow{R_3 \leftarrow 3R3+R_2} \begin{pmatrix} 4 & -1 & -1 \\ 0 & 15 & -5 \\ 0 & 0 & 40 \end{pmatrix}$$

The upper triangular matrix

$$\begin{pmatrix} 4 & -1 & -1 \\ 0 & 15 & -5 \\ 0 & 0 & 40 \end{pmatrix}$$

is the row echelon form of matrix A.

Gauss-Jordan elimination continues above the pivot elements, to obtain the reduced row echelon form.

$$\begin{pmatrix} 4 & -1 & -1 \\ 0 & 15 & -5 \\ 0 & 0 & 40 \end{pmatrix} \xRightarrow{R_3 \leftarrow R3/40} \begin{pmatrix} 4 & -1 & -1 \\ 0 & 15 & -5 \\ 0 & 0 & 1 \end{pmatrix} \xRightarrow[R_2 \leftarrow R2+5R_3]{R_1 \leftarrow R_1+R_3} \begin{pmatrix} 4 & -1 & 0 \\ 0 & 15 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\xRightarrow{R_2 \leftarrow R2/15} \begin{pmatrix} 4 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \xRightarrow{R_1 \leftarrow R_1+R_2} \begin{pmatrix} 4 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \xRightarrow{R_1 \leftarrow R_1/4} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

1.2.3 Solving the System with the rref Function

The Gauss and Gauss-Jordan methods are two familiar approaches for solving linear systems. Both begin from the augmented matrix, obtain the row echelon form or the reduced row echelon form, respectively. Then, the Gauss method uses the back-substitution technique to obtain the solution of the linear system, whereas in Gauss-Jordan method the solution is located in the last column.

The MATLAB code below, reads a matrix A and a vector $\boldsymbol{b}$ from the user, then it applies the Gauss-Seidel elimination through applying the rref to the augmented system $[A \quad b]$

```
clear ; clc ;
A = input('Enter the matrix A: ') ; % Reading matrix A from ...
    the user
b = input('Enter the vector b: ') ; % Reading vector b from ...
    the user
[m, n] = size(A) ;                  % m and n are the matrix ...
    dimensions
r1 = rank(A) ;                      % the rank of matrix A is ...
    assigned to r1
r2 = rank([A b]) ;                  % the rank of the ...
    augmented system [A b] is assigned to r2
if r1 ≠ r2                         % testing whether rank(A) ...
    not equal rank([A b])
    disp(['Rank(A) = ' num2str(r1) ' ≠ ' num2str(r2) ' = ...
        Rank([A b]).']) ;
    fprintf('There is no solution.\n') ; % No solution in this ...
        case
end
if r1 == r2                    % testing whether rank(A) = ...
            rank([A b])
```

```
    if r1 == n                  % if yes, testing whether the ...
        rank equals n
        R = rref([A b]) ;       % the reduced row echelon form ...
            of [A b]
        x = R(:, end) ;         % the solution is at the last ...
            column of the reduced
        % row echelon form
        disp(['Rank(A) = Rank([A b]) = ' num2str(r1) ' = ...
            #Col(A).']) ;
        disp('There is a unique solution, given by: ') ; ...
            disp(x) ;
        %displaying the solution of the linear system
    else                        % rank(A) = rank([A b]) < n
        disp(['Rank(A) = Rank([A b]) = ' num2str(r1) ' < ' ...
            num2str(n) ' = #Col(A).']) ;
        fprintf('Infinite number of solutions.\n') ;
    end
end
```

The result of executing the above MATLAB script is:

```
Enter the matrix A: [1 2 3; 4 5 6; 7 8 9]
Enter the vector b: [1;3;5]
Rank(A) = Rank([A b]) = 2 < 3 = #Col(A).
Infinite number of solutions.

Enter the matrix A: [1 3 5; 2 4 6; 7 8 9]
Enter the vector b: [1;1;1]
Rank(A) = 2 ~= 3 = Rank([A b]).
There is no solution.

Enter the matrix A: [2 2 -1; 1 2 1; -1 -1 2]
Enter the vector b: [2;4;1]
Rank(A) = Rank([A b]) = 3 = #Col(A).
There is a unique solution, given by:
0.6667
1.0000
1.3333
```

In Python, the built-in function `sympy.Matrix` is used to construct a matrix. The Matrix class has a method `rref` to compute the reduced row echelon from of the matrix.

```
import sympy as smp
A = smp.Matrix([[2, 2, -1], [1, 2, 1], [-1, -1, 2]])
b = smp.Matrix([[2], [4], [1]])
m, n = A.rows, A.cols
r1 = A.rank()
C = A.copy()
r2 = (C.row_join(b)).rank()
```

```
if r1 != r2:                          # testing whether rank(A) ...
    not equal rank([A b])
    print('Rank(A) = ' +str(r1) +' != ' +str(r2) +' = Rank([A ...
        b]).')
    print('There is no solution.\n') ; # No solution in this case
if r1 == r2:                     # testing whether rank(A) = ...
        rank([A b])
    if r1 == n:                  # if yes, testing whether the ...
        rank equals n
        R = (A.row_join(b)).rref()      # the reduced row ...
            echelon form of [A b]
        x = R[0][:, -1]        # the solution is at the last ...
            column of the reduced
                               # row echelon form
        print('Rank(A) = Rank([A b]) = '+str(r1) +' = #Col(A).')
        print('There is a unique solution, given by: ') ; ...
            print(x) ;
        #displaying the solution of the linear system
    else:                        # rank(A) = rank([A b]) < n
        print('Rank(A) = Rank([A b]) = ' +str(r1) +' < ' ...
            +str(n) +' = #Col(A).')
        print('Infinite number of solutions.\n')
```

By executing the code, the following results are shown:

```
Rank(A) = Rank([A b]) = 3 = #Col(A).
There is a unique solution, given by:
Matrix([[2/3], [1], [4/3]])
```

1.3 Matrix Factorization Techniques

Matrix factorization means to express a matrix A as a multiplication of two or more matrices, each is called a factor [34, 21]. That is, to write:

$$A = A_1 \cdot A_2 \cdot \ldots \cdot A_n$$

In this section, three important matrix factorization techniques will be discussed; namely, the `LU factorization`, the `QR factorization` and the `singular value decomposition (SVD)`. Then, the use of those factorization methods in solving linear systems of equations will be discussed.

Because cases of solving linear systems with upper or lower triangular matrices will be encountered, this section will start by writing MATLAB and Python codes for solving such a linear system.

1.3.1 The LU Factorization

In this factorization, the matrix A is expressed as a multiplication of two matrices L and U, where L is an lower triangular matrix and U is an upper triangular matrix. That is:

$$A = L \cdot U = \begin{pmatrix} l_{11} & 0 & 0 & \dots & 0 \\ l_{21} & l_{22} & 0 & \dots & 0 \\ l_{31} & l_{32} & l_{33} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \dots & l_{nn} \end{pmatrix} \cdot \begin{pmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ 0 & u_{22} & u_{23} & \dots & u_{2n} \\ 0 & 0 & u_{33} & \dots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & u_{nn} \end{pmatrix} \quad (1.5)$$

where $l_{jj} = 1$ for $j = 1, 2, \dots, n$.

The function `lu` can be used for finding the L and U factors of matrix S. In MATLAB, this can be done as follows:

```
>> A = [4 -1 -1; -1 4 -1; -1 -1 4]
A =
 4    -1    -1
-1     4    -1
-1    -1     4

>> [L, U] = lu(A)
L =
 1.0000         0         0
-0.2500    1.0000         0
-0.2500   -0.3333    1.0000

U =
4.0000   -1.0000   -1.0000
0         3.7500   -1.2500
0         0         3.3333
```

In Python, the function `lu` is located in the `scipy.linalg` sub-package and can be used to find the LU factors of matrix A.

```
In [1]: import numpy as np, scipy.linalg as lg
In [2]: A = np.array([[4, -1, -1], [-1, 4, -1], [-1, -1, 4]])
In [3]: P, L, U = lg.lu(A)
In [4]: print('L = \n', L, '\nU = \n', U)
L =
[[ 1.          0.          0.        ]
[-0.25        1.          0.        ]
[-0.25       -0.33333333  1.        ]]
U =
[[ 4.         -1.         -1.        ]
[ 0.          3.75       -1.25      ]
[ 0.          0.          3.33333333]]
```

However, python can compact both the L and U factors of matrix A using the function `lu_factor`.

```
In [5]: LU = lg.lu_factor(A)
In [6]: print('LU = \n', LU)
LU =
(array([[ 4.          , -1.          , -1.          ],
[-0.25        ,  3.75        , -1.25        ],
[-0.25        , -0.33333333,  3.33333333]]), array([0, 1, 2],
    dtype=int32))
```

Now, the linear system 1.1 becomes:

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ l_{21} & 1 & 0 & \dots & 0 \\ l_{31} & l_{32} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \dots & 1 \end{pmatrix} \cdot \begin{pmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ 0 & u_{22} & u_{23} & \dots & u_{2n} \\ 0 & 0 & u_{33} & \dots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & u_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix} \tag{1.6}$$

The solution of the linear system 1.6 is found in three stages:

1. **First:** we let $\boldsymbol{y} = U\boldsymbol{x}$, that is

$$\boldsymbol{y} = \begin{pmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ 0 & u_{22} & u_{23} & \dots & u_{2n} \\ 0 & 0 & u_{33} & \dots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & u_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix}$$

   Then, solving system 1.6 is equivalent to solving the linear system

$$L\boldsymbol{y} = \boldsymbol{b}$$

2. **Second:** we solve the system $L\boldsymbol{y} = \boldsymbol{b}$ using the function SolveLower System.m to find $\boldsymbol{y}$.

3. **Finally:** we solve the linear system $U\boldsymbol{x} = \boldsymbol{y}$ using the back-substitution method, implemented by the MATLAB function SolveUpperSystem.

Example 1.1 In this example, the LU-factors will be used to solve the linear system:

$$\begin{pmatrix} 4 & -1 & -1 \\ -1 & 4 & -1 \\ -1 & -1 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \\ 2 \end{pmatrix}$$

In MATLAB, the following commands can be used:

```
>> A = [4 -1 -1; -1 4 -1; -1 -1 4] ;
>> b = [2; 2; 2] ;
>> [L, U] = lu(A)
```

```
L =
1.0000         0         0
-0.2500    1.0000         0
-0.2500   -0.3333    1.0000

U =
4.0000   -1.0000   -1.0000
0    3.7500   -1.2500
0         0    3.3333
>> y = SolveLowerLinearSystem(L, b, 3)
y =
2.0000
2.5000
3.3333
>> x = SolveUpperLinearSystem(U, y, 3)
x =
1.0000
1.0000
1.0000
```

In Python, similar steps can be followed to solve the linear system $A\boldsymbol{x} = \boldsymbol{b}$ using the LU factors of matrix A.

```
In [7]: y = lg.solve(L, b)
In [8]: x = lg.solve(U, y)
In [9]: print('x = \n', x)
x =
[[ 0.5]
[ 0.5]
[ 0.5]]
```

Python has the LU solver `lu_solve` located in `scipy.linalg` sub-package. It receives the matrix LU obtained by applying the `lu_solve` function, to return the solution of the given linear system.

```
In [10]: x = lg.lu_solve(LU, b)
In [11]: print(x)
[[ 0.5]
[ 0.5]
[ 0.5]]
```

The Python's symbolic package `sympy` can also be used to find the LU factors of a matrix A. This can be done as follows:

```
In [10]: import sympy as smp
In [11]: A = smp.Matrix([[4., -1., -1.], [-1., 4., -1.],
    [-1., -1., 4.]])
In [12]: LU = B.LUdecomposition()
```

```
In [13]: LU
Out[13]:
(Matrix([
[    1,                  0, 0],
[-0.25,                  1, 0],
[-0.25, -0.333333333333333, 1]]), Matrix([
[4.0, -1.0,             -1.0],
[  0, 3.75,            -1.25],
[  0,    0, 3.33333333333333]]), [])
In [14]: LU[0]
Out[14]:
Matrix([
[    1,                  0, 0],
[-0.25,                  1, 0],
[-0.25, -0.333333333333333, 1]])

In [15]: LU[1]
Out[15]:
Matrix([
[4.0, -1.0,             -1.0],
[  0, 3.75,            -1.25],
[  0,    0, 3.33333333333333]])
```

The symbolic package `sympy` *can be also used to solve a linear system, using the LU factors.*

```
In [16]: b = [[2.0], [2.0], [2.0]]
In [17]: A.LUSolve(b)
Out[17]:
Matrix([
[1.0],
[1.0],
[1.0]])
```

1.3.2 The QR Factorization

In this type of factorization, the matrix A is expressed as a multiplication of two matrices Q and R. The matrix Q is orthogonal (its columns constitute an orthonormal set) and the matrix R is an upper triangular.

From the elementary linear algebra, an orthogonal matrix satisfies the following two conditions:

1. $Q^{-1} = Q^T$, and
2. if $Q = [\boldsymbol{q}_1 \boldsymbol{q}_2 \dots \boldsymbol{q}_n]$, then,

$$(\boldsymbol{q}_i, \boldsymbol{q}_j) = \boldsymbol{q}_i^T \cdot \boldsymbol{q}_j = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$