BASIC TECHNIQUES FOR COMPUTER SIMULATIONS WPO – 2ND SESSION

LINEAR SYSTEMS

VRIJE UNIVERSITEIT BRUSSEL

INTRODUCTION

- In the previous WPO session we determined the value x that solves a single equation: $f(x)=0$
- Now, we want to determine the values $x_1,x_2,\ldots,x_n$ that simultaneously satisfy a **set of equations**
- We will deal with **linear algebraic equations** of this general form:

$$
\begin{gathered}
a_{11}x_1 + a_{11}x_2 + \ldots + a_{1n}x_n = b_1 \\
a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\
\vdots \\
a_{n1}x_1 + a_{n1}x_2 + \ldots + a_{nn}x_n = b_n
\end{gathered}
$$

a: constant coefficients, b: constants, x: unknowns

INTRODUCTION

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2$$
$$\vdots$$
$$a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n$$

$$[A] = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \quad (nxn)$$

$$[x] = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \quad (nx1)$$

$$[b] = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} \quad (nx1)$$

$$Ax = b$$

solve for **x**

The quick PYTHON solution: x = numpy.linalg.solve(A,b)

or: x = numpy.dot(numpy.linalg.inv(A),b) time consuming

GAUSS ELIMINATION
WITHOUT PIVOTING (NAÏVE)

- Example: 3 equations – 3 unknowns

$$
\left.\begin{array}{c}
\left[\begin{array}{ccc:c} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{array}\right] \\
\downarrow \\
\left[\begin{array}{ccc:c} a_{11} & a_{12} & a_{13} & b_1 \\ & a'_{22} & a'_{23} & b'_2 \\ & & a''_{33} & b''_3 \end{array}\right]
\end{array}\right\}
$$

(*a*) Forward elimination

Step 1: Perform **forward elimination** to reduce the set of equations to an upper triangluar system

a. Eliminate a_{21} from the 2^{nd} row:
- factor = a_{21}/a_{11}
- Substract factor*(1st_row) from 2^{nd} row

b. Eliminate a_{31} from the 3rd row:
- factor = a_{31}/a_{11}
- Substract factor*(1st_row) from 3^{rd} row

$$
\begin{array}{llllllll}
(1) & a_{11}x_1 & + & a_{12}x_2 & + & a_{13}x_3 & = & b_1 \\
(2) & & & a'_{22}x_2 & + & a'_{23}x_3 & = & b'_2 \\
(3) & & & a'_{32}x_3 & + & a'_{33}x_3 & = & b'_3
\end{array}
$$

c. Eliminate a'_{32} from the 3rd row:
- factor = a'_{32}/a'_{22}
- Substract factor*(2nd_row) from 3^{rd} row

$$
\begin{array}{lllllll}
a_{11}x_1 & + & a_{12}x_2 & + & a_{13}x_3 & = & b_1 \\
& & a'_{22}x_2 & + & a'_{23}x_3 & = & b'_2 \\
& & & & a''_{33}x_3 & = & b''_3
\end{array}
$$

source: Chapra SC. Applied numerical methods with MATLAB for engineers and scientistic, 2008.

GAUSS ELIMINATION
WITHOUT PIVOTING (NAÏVE)

$$\left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{array}\right]$$

$$\downarrow$$

$$\left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ & a'_{22} & a'_{23} & b'_2 \\ & & a''_{33} & b''_3 \end{array}\right]$$

(*a*) Forward elimination

PYTHON
FORWARD ELIMINATION

def main():

1) Define arrays A and b, using the numpy module
2) Call def forward_elimination(…)

def forward_elimination(…):

With 'for' loops eliminate the desired element from nth row, kth column by:

1) Computing the appropriate factor, using two of the elements of the kth column
2) Multiplying the computed factor with all the elements of the kth row and substracting the resulting elements from the nth row.

! Python uses **zero-based indexing**, meaning that the first element of a list has index 0 !

GAUSS ELIMINATION
WITHOUT PIVOTING (NAÏVE)

$$\left[\begin{array}{ccc:c} a_{11} & a_{12} & a_{13} & b_1 \\ & a'_{22} & a'_{23} & b'_2 \\ & & a''_{33} & b''_3 \end{array}\right]$$

$$\downarrow$$

$$\left.\begin{aligned} x_3 &= b''_3/a''_{33} \\ x_2 &= (b'_2 - a'_{23}x_3)/a'_{22} \\ x_1 &= (b_1 - a_{13}x_3 - a_{12}x_2)/a_{11} \end{aligned}\right\} \text{(\textit{b}) Back substitution}$$

Step 2: Perform **back subtitution** to solve for the unkowns

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a'_{22}x_2 + a'_{23}x_3 &= b'_2 \\ a''_{33}x_3 &= b''_3 \end{aligned}$$

The third equation is solved for x_3:

$$x_3 = b''_3/a''_{33}$$

Back-substitute the result to equation 2 to solve for x_2

Back-subsitute both results to equation 1 to solve for x_1

source: Chapra SC. Applied numerical methods with MATLAB for engineers and scientistic, 2008.

GAUSS ELIMINATION
WITHOUT PIVOTING (NAÏVE)

$$\left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ & a'_{22} & a'_{23} & b'_2 \\ & & a''_{33} & b''_3 \end{array}\right]$$

$$\downarrow$$

$$\left.\begin{aligned} x_3 &= b''_3/a''_{33} \\ x_2 &= (b'_2 - a'_{23}x_3)/a'_{22} \\ x_1 &= (b_1 - a_{13}x_3 - a_{12}x_2)/a_{11} \end{aligned}\right\} \text{(b) Back substitution}$$

PYTHON
BACK SUBSTITUTION

def main():

1) Use A (upper diagonal) and b arrays after forward elimination
2) Call def back_substitution(…)

def back_substitution(…):

For a system of N equations:

1) Start from the last row to compute x_N
2) Iterate though the equations in reverse order
3) At every iteration solve for the corresponding unkown.

! Python uses **zero-based indexing**, meaning that the first element of a list has index 0 !

LU FACTORIZATION

WITHOUT PIVOTING

LESS TIME CONSUMING FOR A COMPUTER

- Example: 3 equations – 3 unknowns

$$[A] = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \qquad [x] = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad [b] = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

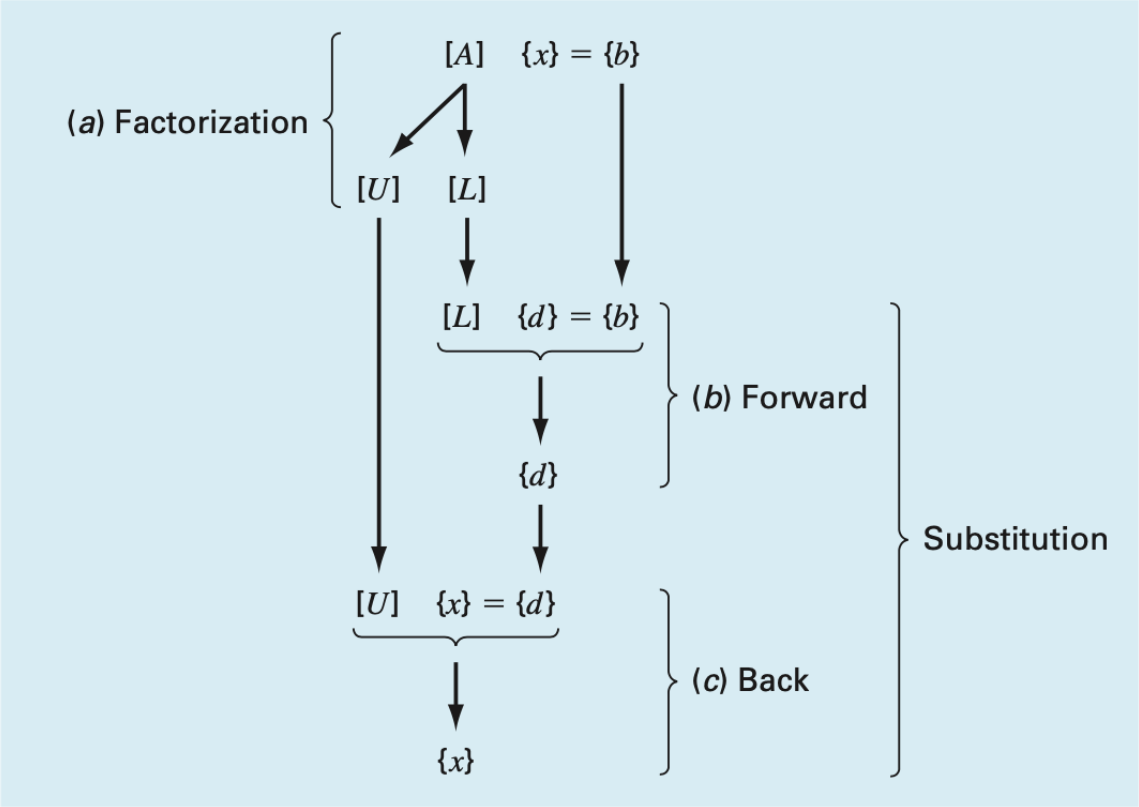
- In order to separate the time-consuming elimination of matrix A from the manipulations of the right-hand side vector b:

A is “factored” or “decomposed” into:

$$[U] = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \qquad [L] = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix}$$

Upper triangular matrix   Lower triangular matrix

LU FACTORIZATION
WITHOUT PIVOTING



$$\mathbf{A} = \mathbf{LU},$$

$$\text{So, } \mathbf{Ax} = \mathbf{b} \Leftrightarrow \mathbf{LUx} = \mathbf{b}$$

- **Ld** = **b**, solve by forward substitution for intermediate vector **d**
- **Ux** = **d**, solve by back substitution for unknown vector **x**

source: Chapra SC. Applied numerical methods with MATLAB for engineers and scientistic, 2008.

LU FACTORIZATION
WITH PIVOTING

- Not all square matrices have a ”pure” LU decomposition

- For more reliable results, partial <u>pivoting</u> is employed, so that:

$$\mathbf{PA} = \mathbf{LU} \ (1),$$

the permutation matrix **P** is employed, to keep track of the row switches

- So, $\mathbf{Ax} = \mathbf{b} \Leftrightarrow \mathbf{PAx} = \mathbf{Pb}$ (2) which is solved in a similar way as in LU without pivoting:

$$\begin{aligned} (2) \text{ using } (1): \quad & \mathbf{LUx} = \mathbf{Pb} \\ \Leftrightarrow \quad & \mathbf{Ld} = \mathbf{Pb} \ (3), \ \mathbf{d} = \mathbf{Ux} \ (4) \end{aligned}$$

- (3) Is solved for **d** and (4) is solved for **x**

LU FACTORIZATION
WITH PIVOTING

PYTHON

1) **A** is decomposed to **L** and **U** using **P**, with scipy module:

scipy.linalg.lu(A, permute_l=False)

2) Once **L** is known, **Ld** = **Pb** is solved for **d** with:

forward substitution

numpy.x = numpy.linalg.solve(A,b)
for forward and back substitution`

3) Once **d** is known, **Ux** = **d** is solved for **x** with:

back substitution

ITERATIVE METHODS

- Iterative methods provide an alternative to the two elimination methods (Gauss, LU factorization) described so far

- Useful for large, sparse matrices

- Similar logic to the root-finding methods discussed in the 1st WPO:
  - An initial guess for all roots of the system
  - Repetitive method to obtain a good approximation of the roots

GAUSS-SEIDEL

- In our 3x3 example, if the diagonal elements are all non-zero, every equation can be solved for the corresponding unknown:

$$\begin{aligned} a_{11}\mathbf{x_1} + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}\mathbf{x_2} + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}\mathbf{x_3} &= b_3 \end{aligned}$$

$$\begin{cases} \mathbf{x_{1,new}} = \dfrac{b_1 - a_{12}x_2 - a_{13}x_3}{a_{11}} & \text{(A)} \\ \big\downarrow & \\ \mathbf{x_{2,new}} = \dfrac{b2 - a_{21}\mathbf{x_{1,new}} - a_{23}x_3}{a_{22}} & \text{(B)} \\ \big\downarrow & \\ \mathbf{x_{3,new}} = \dfrac{b_3 - a_{31}x_{1,new} - a_{32}\mathbf{x_{2,new}}}{a_{33}} & \text{(C)} \end{cases}$$

- Initial guess: vector **x** (x_1,x_2,x_3)
- Substitute x_2,x_3 guesses to Eq. (A) and solve for new x_1
- Substitute the new x_1 along with the initial guess for x_3 to Eq. (B) and get new x_2…
- Continue the process till convergence criterion is met for every element of vector **x**:

$$\left|\frac{x_{i,new} - x_{i,old}}{x_{i,new}}\right| \leq \text{chosen\_tolerance},\ i = 1,2,3$$

GAUSS-SEIDEL

$$\mathbf{x_{1,new}} = \frac{b_1 - a_{12}x_2 - a_{13}x_3}{a_{11}}$$

$\downarrow$

$$\mathbf{x_{2,new}} = \frac{b2 - a_{21}\mathbf{x_{1,new}} - a_{23}x_3}{a_{22}}$$

$\downarrow$

$$\mathbf{x_{3,new}} = \frac{b_3 - a_{31}x_{1,new} - a_{32}\mathbf{x_{2,new}}}{a_{33}}$$

PYTHON

def main():

1) Define arrays **A** and **b**, using the numpy module
2) Make initial guess for solution vector **x**
3) Call def gauss_seidel(…)

def gauss_seidel(…):

1) For every element i of vector **x**, update values **x**[i] one at a time, using equations on the left
2) Compare **updated x** vector with **x** of previous iteration

Repeat until convergence criterion is met

JACOBI

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2$$
$$a_{31}x_1 + a_{23}x_2 + a_{33}x_3 = b_3$$

$$\rightarrow$$

$$\boxed{\begin{array}{l} \mathbf{x_{1,new}} = \dfrac{b_1 - a_{12}x_2 - a_{13}x_3}{a_{11}} \quad (1) \\[2em] \mathbf{x_{2,new}} = \dfrac{b2 - a_{21}x_1 - a_{23}x_3}{a_{22}} \quad (2) \\[2em] \mathbf{x_{3,new}} = \dfrac{b_3 - a_{31}x_1 - a_{32}x_2}{a_{33}} \quad (3) \end{array}}$$

- While Gauss-Seidel method always applies the latest updated values of the elements during the iterations, Jacobi method uses the set of values($x_{1,new}$, $x_{2,new}$, $x_{3,new}$) obtained from the previous step
- Solution vector **x** is updated as a whole in every iteration and not element by element
- Continue iterations till convergence criterion is met for all elements of solution vector **x**:

$$\left|\frac{x_{i,new} - x_{i,old}}{x_{i,new}}\right| \leq \text{chosen\_tolerance},\ i = 1,2,3$$

JACOBI

- Eq. (1-3) in the form of matrices:

$$x_1^{\text{new}} = \frac{b_1}{a_{11}} \qquad\qquad - \frac{a_{12}}{a_{11}}x_2 - \frac{a_{13}}{a_{11}}x_3$$

$$x_2^{\text{new}} = \frac{b_2}{a_{22}} - \frac{a_{21}}{a_{22}}x_1 \qquad\qquad - \frac{a_{23}}{a_{22}}x_3$$

$$x_3^{\text{new}} = \frac{b_3}{a_{33}} - \frac{a_{31}}{a_{33}}x_1 - \frac{a_{32}}{a_{33}}x_2$$

$$[d] = \begin{bmatrix} \frac{b_1}{a_{11}} \\ \frac{b_2}{a_{22}} \\ \frac{b_3}{a_{33}} \end{bmatrix} \qquad [C] = \begin{bmatrix} 0 & \frac{a_{12}}{a_{11}} & \frac{a_{13}}{a_{11}} \\ \frac{a_{21}}{a_{22}} & 0 & \frac{a_{23}}{a_{22}} \\ \frac{a_{31}}{a_{33}} & \frac{a_{32}}{a_{33}} & 0 \end{bmatrix} \qquad [x] = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

PYTHON

def main():

1) Define arrays **A** and **b**, using the numpy module
2) Make initial guess for solution vector **x**
3) Call def jacobi(…)

def jacobi(…):

1) Compute **d** and **C**
2) **x_new** = **d** – np.dot(**C**,**x**)
3) Compare **x_new** with **x** of previous iteration

Repeat until convergence criterion is met

! Make sure that all vectors are vertical during computations !