EPFL

Numerical Analysis and Computational Mathematics

Fall Semester 2025 – CSE Section

Prof. Laura Grigori

Assistant: Israa Fakih

Session 8 – November 5, 2025

Solutions – Numerical integration & Linear systems: direct methods

Exercise I (MATLAB)

a) For the implementation, we must choose the proper quadrature nodes and weights for $n = 0, 1,$ and 2. For other values of n the MATLAB function should return an error, since these cases are not implemented. Once the proper quadrature formula is defined, we rescale the quadrature nodes and weights from the reference interval $[-1, 1]$ to the generic interval $[a, b]$.

```
function [ Ih ] = gauss_legendre_simple_quadrature( fun, a, b, n )
% GAUSS_LEGENDRE_SIMPLE_QUADRATURE approximate the integral of a function in
% the interval [a,b] by means of the simple Gauss-Legendre quadrature formula
%  [ Ih ] = gauss_legendre_simple_quadrature( fun, a, b, n )
%  Inputs: fun = function handle,
%          a,b = extrema of the interval [a,b]
%          n + 1 = number of quadrature nodes and weights
%  Output: Ih = approximate value of the integral
%

% reference nodes and weights
switch n

    case 0
        y_ref = 0;
        alpha_ref = 2;

    case 1
        y_ref = [ -1/sqrt(3), 1/sqrt(3) ];
        alpha_ref = [ 1, 1 ];

    case 2
        y_ref = [ -sqrt(15)/5, 0, sqrt(15)/5 ];
        alpha_ref = [ 5/9, 8/9, 5/9 ];

    otherwise
```

```
        error('n must be 0, 1, or 2');

end

% nodes and weigths rescaled in the interval [a,b]
y_rescaled = ( a + b ) / 2 + ( b - a ) / 2 * y_ref;
alpha_rescaled = ( b - a ) / 2 * alpha_ref;

% Integral
Ih = sum( alpha_rescaled .* fun( y_rescaled ) );

return
```

b) We compute first the exact value of the integral:

```
a = 0; b = 1;
f = @(x) sin( 7/2 * x ) + exp( x ) - 1;
I_ex = exp( 1 ) - 2 + 2 / 7 * ( 1 - cos( 7 / 2 ) )
%  I_ex =
%    1.2716
```

Then, we move to the approximated values of the integral computed by means the simple Gauss-Legendre quadrature formulas:

```
[ Igl_n0 ] = gauss_legendre_simple_quadrature( f, a, b, 0 )
%  Igl_n0 =
%    1.6327
[ Igl_n1 ] = gauss_legendre_simple_quadrature( f, a, b, 1 )
%  Igl_n1 =
%    1.2409
[ Igl_n2 ] = gauss_legendre_simple_quadrature( f, a, b, 2 )
%  Igl_n2 =
%    1.2724
```

The simple Gauss-Legendre quadrature formula with $n = 0$ just coincides with the simple midpoint quadrature formula and the value of the approximated integral is the same obtained in Exercise II, point b) of Series 7. In contrast, the Gauss-Legendre quadrature formula with $n = 1$ already gets the first decimal figure correct, and $n = 2$ gets the first two decimal figures correct. We observe that the results are much more accurate than the simple Simpson quadrature formula, even though this latter approach uses the same number of quadrature nodes.

```
Is = simpson_composite_quadrature( f, a, b, 1 )
%  Is =
%    1.3164
```

In fact, the simple Gauss-Legendre quadrature formula with $n = 2$ is even more accurate than the composite trapezoidal quadrature formula with $M = 10$ sub-intervals:

```
It_c = trapezoidal_composite_quadrature( f, a, b, 10 )
%  It_c =
```

```
%    1.2673
```

Based on these result, we can easily deduce why the Gauss-Legendre quadrature formula is sometimes considered the "gold standard" of quadratures formulas and is implemented in most software libraries for numerical integration.

c) The theoretical justification for the higher accuracy of the Gauss-Legendre quadrature formulas with $n+1$ quadrature nodes is that they have degree of exactness $2n+1$. In contrast, the midpoint, trapezoidal, and Simpson quadrature formulas have degrees of exactness equal to $1, 1$, and 3 by using 1, 2, and 3 quadrature nodes, respectively. To verify this, we proceed as in Exercise II, point e) of Series 7:

```
f = @(x,d) x.^d;    a = 0;    b = 1;
Table = [ ];  % for visualization of the results
for d = [ 0 : 6 ]
    I_ex = 1 / ( d + 1 );
    % Gauss-Legendre n=0,1,2
    Igl0 = gauss_legendre_simple_quadrature( @(x)f(x,d), a, b, 0 );
    Igl1 = gauss_legendre_simple_quadrature( @(x)f(x,d), a, b, 1 );
    Igl2 = gauss_legendre_simple_quadrature( @(x)f(x,d), a, b, 2 );
    % Midpoint
    Imp = ( b - a ) * f( ( a + b ) / 2, d );
    % Trapezoidal
    It = ( b - a ) / 2 * ( f( a, d ) + f( b, d ) );
    % Simpson
    Is = ( b - a ) / 6 * ( f( a, d ) + 4 * f( ( a + b ) / 2, d ) + f( b, d ) );
    Table = [ Table;
              [ d, abs( I_ex - Igl0 ), abs( I_ex - Igl1 ), abs( I_ex - Igl2 ), ...
                  abs( I_ex - Imp ), abs( I_ex - It ), abs( I_ex - Is ) ] ];
end
disp('d,     GL n0,     GL n1,     GL n2,      MidP,      Trap,      Simp')
disp(num2str(Table))

% d,      GL n0,      GL n1,      GL n2,       MidP,       Trap,       Simp
% 0           0           0           0          0          0          0
% 1           0           0           0          0          0          0
% 2    0.083333           0  5.5511e-17   0.083333    0.16667          0
% 3       0.125  2.7756e-17           0      0.125       0.25          0
% 4      0.1375   0.0055556           0     0.1375        0.3  0.0083333
% 5     0.13542    0.013889  2.7756e-17    0.13542    0.33333   0.020833
% 6     0.12723    0.022487  0.00035714    0.12723    0.35714   0.034226
```

(Errors $\lesssim 10^{-15}$ should be considered as zero due to round-off.)

Again, we notice that the Gauss-Legendre quadrature formula with $n = 0$ and the midpoint quadrature formula are equivalent. For both these formulas and the trapezoidal quadrature formula, we verify the degree of exactness 1, since the corresponding errors are zero for $d \leq 1$. The Gauss-Legendre quadrature formula with $n = 1$ has degree of exactness equal to 3, just as the Simpson quadrature formula (but the former only uses two quadrature nodes instead of three). The Gauss-Legendre quadrature formula with $n = 2$ has degree of exactness equal to $2n + 1 = 5$ (the error is zero for $d \leq 5$). The results confirm the predictions given by the theory (see also Exercise II, point e) of Series 7).

Exercise II (Theoretical)

By taking $f(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3$ for some c_0, c_1, c_2 and $c_3 \in \mathbb{R}$. We have

$$I(f) = \int_{-1}^{1} f(x)dx = 2c_0 + \frac{2}{3}c_2$$

By setting $n = 1$, we have

$$\begin{aligned} I_{GL,1}(f) = &\left(\bar{\alpha}_0^{GL} + \bar{\alpha}_1^{GL}\right)c_0 + \left(\bar{\alpha}_0^{GL}\bar{y}_0^{GL} + \bar{\alpha}_1^{GL}\bar{y}_1^{GL}\right)c_1 \\ &+ \left(\bar{\alpha}_0^{GL}(\bar{y}_0^{GL})^2 + \bar{\alpha}_1^{GL}(\bar{y}_1^{GL})^2\right)c_2 + \left(\bar{\alpha}_0^{GL}(\bar{y}_0^{GL})^3 + \bar{\alpha}_1^{GL}(\bar{y}_1^{GL})^3\right)c_3. \end{aligned}$$

Plugging the values of $\bar{y}_0^{GL}$, $\bar{y}_1^{GL}$, $\bar{\alpha}_0^{GL}$, and $\bar{\alpha}_1^{GL}$ in the equation above, we verify that $I_{GL,1}(f) = I(f)$ for all c_0, c_1, c_2 and $c_3 \in \mathbb{R}$. We deduce that the Gauss-Legendre formula $I_{GL,1}(f)$ integrates exactly polynomials of degree (up to) 3 regardless of their coefficients c_0, c_1, c_2 and $c_3 \in \mathbb{R}$.

Exercise III (MATLAB)

a) The forward substitution algorithm for system $L\mathbf{y} = \mathbf{b}$, where L has components l_{ij} and $\mathbf{y}, \mathbf{b} \in \mathbb{R}^n$ have components y_i and b_i, reads:

$$\begin{cases} y_1 = \dfrac{b_1}{l_{11}}, \\ y_i = \dfrac{1}{l_{ii}}\left(b_i - \displaystyle\sum_{j=1}^{i-1} l_{ij}y_j\right) \qquad i = 2, \ldots, n. \end{cases}$$

Correspondingly, the backward substitution algorithm for system $U\mathbf{x} = \mathbf{y}$ reads:

$$\begin{cases} x_n = \dfrac{y_n}{u_{nn}}, \\ x_i = \dfrac{1}{u_{ii}}\left(y_i - \displaystyle\sum_{j=i+1}^{n} u_{ij}x_j\right) \qquad i = n-1, \ldots, 1. \end{cases}$$

We implement the algorithms in MATLAB as:

```
function [ y ] = forward_substitutions( L, b )
% FORWARD_SUBSTITUTIONS solve the linear system L y = b by means of the
% forward subsitutions algorithm; L must be a lower triangular matrix
%  [ y ] = forward_substitutions( L, b )
%  Inputs: L = lower triangular matrix (square matrix)
%          b = vector (right hand side of the linear system)
%  Output: y = solution vector (column vector)
%

n = size( L, 1 );
y = zeros( n, 1 );

y( 1 ) = b( 1 ) / L( 1, 1 );
for i = 2 : n
    j_v = 1 : i - 1;
```

```
    y( i ) = 1 / L( i, i ) * ( b( i ) - L( i, j_v ) * y( j_v ) );
end

return
```

```
function [ x ] = backward_substitutions( U, y )
% BACKWARD_SUBSTITUTIONS solve the linear system U x = y by means of the
% backward subsitutions algorithm; U must be an upper triangular matrix
%  [ x ] = backward_substitutions( U, y )
%  Inputs: U = upper triangular matrix (square matrix)
%          y = vector (right hand side of the linear system)
%  Output: x = solution vector (column vector)
%

n = size( U, 1 );
x = zeros( n, 1 );

x( n ) = y( n ) / U( n, n );
for i = n - 1 : -1 : 1
    j_v = i + 1 : n;
    x( i ) = 1 / U( i, i ) * ( y( i ) - U( i, j_v ) * x( j_v ) );
end

return
```

b) The first two outputs of the MATLAB command `[L,U,P] = lu(A)` are the matrices L and U, respectively. The third one is the permutation matrix `P` associated to the pivoting technique. When pivoting is performed, the LU factorization corresponds to:

$$LU = PA,$$

where P is the permutation matrix[1] corresponding to the reordering of the rows performed during pivoting. When pivoting is applied (i.e. $P \neq I$), we must take the permutation matrix P into account when performing the forward substitution, since now:

$$LU\mathbf{x} = P\mathbf{b} \qquad \Longleftrightarrow \qquad \begin{cases} L\mathbf{y} = P\mathbf{b}, \\ U\mathbf{x} = \mathbf{y}. \end{cases}$$

Note that if we had given only the command `[L,U] = lu(A)`, then MATLAB will return two matrices $\widetilde{L}$ and U such that $\widetilde{L}U = A$. In this case $\widetilde{L} = P^T L$, where L is a lower triangular matrix, while $\widetilde{L}$ is not, in general, lower triangular.

The MATLAB code to solve the linear system $A\mathbf{x} = \mathbf{b}$ via LU factorization reads:

```
A = [ 4 -2 -1; -1 3 -1; -1 -3 5 ];
x_ex = ones( 3, 1 );
b = A * x_ex;
% Note: MATLAB may perfom pivoting even when not strictly necessary
[ L, U, P ] = lu( A );
[ y ] = forward_substitutions( L, P * b );
```

[1] A permutation matrix P has elements that are either 0 or 1 and it is such that $P^T P = I$, i.e. P^T represents the inverse permutation of P, since $P^{-1} = P^T$ in this case.

```
[ x ] = backward_substitutions( U, y );
err = norm( x - x_ex )
%  err =
%    3.8459e-16
```

The obtained solution matches the exact one up to machine precision. If we inspect the permutation matrix P, we find:

```
P
% P =
%     1     0     0
%     0     0     1
%     0     1     0
```

This confirms that the MATLAB function `lu` has performed pivoting, opting to swap the second and third rows during the LU factorization.

c) Banded matrices such as A can be built in MATLAB in few lines of code, by using the `diag` command to construct each (sub)diagonal separately and then summing together all the contributions. The command `diag(vect,k)` builds a matrix where the values contained in the vector `vec` are placed on the subdiagonal `k`. The value $k = 0$ refers to the main diagonal of the matrix, $k \geq 1$ corresponds to the k-th upper subdiagonal and $k \leq -1$ to the k-th lower subdiagonal. Note that, in order to obtain matrices of proper dimension, we need to make sure that the size of `vec` is $n - k$.

```
n = 20;
A = diag( 4 * ones( n, 1 ), 0 ) + ...
    diag( -1 * ones( n - 1, 1 ), 1 ) + ...
    diag( -2 * ones( n - 1, 1 ), - 1 ) + ...
    diag( -1 * ones( n - 2, 1 ), - 2 );
A( n, 1 ) = - 1;
A( 1, n ) = 1;
figure; spy( A )
[ L, U, P ] = lu( A );
figure; spy( L )
figure; spy( U )
```

From the pattern of the matrix A (Figure 1) and its LU factorization matrices L and U (Figure 2), we observe that the two off-diagonal elements cause a "fill-in" in the two matrices. In this case the fill-in only affects one row (resp. column) of the matrix L (resp. U). The solution of the linear system can be obtained using the code from point b).

d) We build the matrix A in the *sparse* format. We notice that there are different ways to use the MATLAB function `sparse` to build a matrix in the sparse format. We choose the following way: `A = sparse(i,j,v,n,n)`. Here `i` and `j` are two vectors of equal length that contain the (i, j) indexes of all the nonzero elements of A. The vector `v` is a vector of the same length that contains the values of the nonzero elements A_{ij}. Finally, the last two arguments indicate the size of the matrix, here $n \times n$. The complete matrix A is built diagonal by diagonal as before, but by using the `sparse` commands:

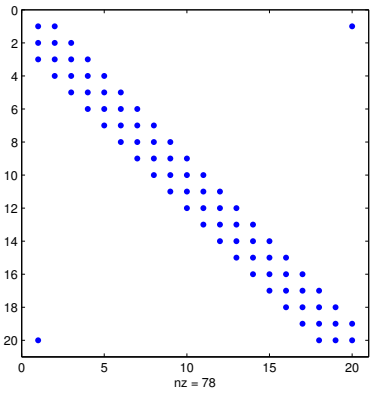

Figure 1: Pattern of the banded matrix A with $n = 20$.

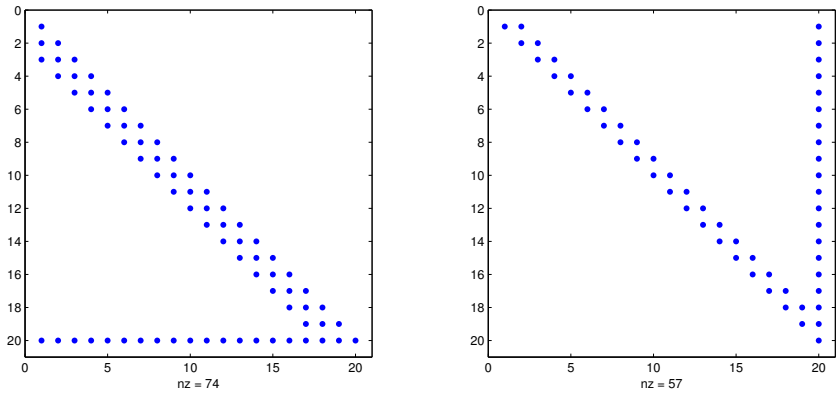

Figure 2: Patterns of the matrices L (left) and U (right) for the matrix $A = LU$ with $n = 20$.

```
n = 1000;
As = sparse( 1 : n, 1 : n, 4 * ones( n, 1 ), n, n ) + ...
     sparse( 1 : n - 1, 2 : n, -1 * ones( n - 1, 1 ), n, n ) + ...
     sparse( 2 : n, 1 : n - 1, -2 * ones( n - 1, 1 ), n, n ) + ...
     sparse( 3 : n, 1 : n - 2, -1 * ones( n - 2, 1 ), n, n );
As( n, 1 ) = - 1;
As( 1, n ) = 1;
A = diag( 4 * ones( n, 1 ), 0 ) + ...
    diag( -1 * ones( n - 1, 1 ), 1 ) + ...
    diag( -2 * ones( n - 1, 1 ), - 1 ) + ...
    diag( -1 * ones( n - 2, 1 ), - 2 );
A( n, 1 ) = - 1;
A( 1, n ) = 1;
whos A As
%  Name          Size                Bytes  Class     Attributes
%
%  A          1000x1000            8000000  double
%  As         1000x1000              72104  double    sparse
```

The *full* matrix `A` with all the zeros memorized takes about 8 MB of memory, while the sparse matrix `As` takes only about 72 kB. The lower memory requirements are achieved by storing the matrix as a list of nonzero elements, where each element is stored as its index (i, j) and the

corresponding nonzero value A_{ij}. All the other elements are assumed to be zero by default.

e) We recall that, for a symmetric and positive definite matrix $A \in \mathbb{R}^{n\times n}$, the Cholesky factorization returns an upper triangular matrix $R \in \mathbb{R}^{n\times n}$ such that $R^T R = A$. We solve the linear system by using the Cholesky factorization method as follows:

```
A = [ 4 -2 -1; -2 7 -4; -1 -4 6 ];
x_ex = ones( 3, 1 ); b = A * x_ex;
R = chol( A );
y = forward_substitutions( R', b );
x = backward_substitutions( R, y );
err = norm( x - x_ex )
% err =
%   9.1551e-16
```

The obtained solution matches the exact one up to machine precision.

Exercise IV (Theoretical)

a) We have $\det(A) = -3\alpha^2 - 2\alpha + \frac{35}{3}$. The values of α for which $\det(A) = 0$ are $\alpha_1 = -\frac{7}{3}$ and $\alpha_2 = \frac{5}{3}$.

b) We obtain:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ \alpha & 1 & 0 \\ -1 & \frac{6\alpha}{35-3\alpha^2} & 1 \end{bmatrix}, \qquad U = \begin{bmatrix} 1 & \alpha & -1 \\ 0 & \frac{35}{3} - \alpha^2 & 1+\alpha \\ 0 & 0 & \frac{-9\alpha^2-6\alpha+35}{35-3\alpha^2} \end{bmatrix}.$$

If the matrix A is non-singular (for $\alpha \neq -\frac{7}{3}$ and $\frac{5}{3}$ as deduced at point a)), the factorization exists and is unique if the $n-1$ pivot elements are non-zero. The values of the pivot elements are $a^{(1)}_{1,1} = 1$ and $a^{(2)}_{2,2} = \frac{35}{3} - \alpha^2$. So, the LU factorization of the matrix A exists and is unique for $\alpha \in \mathbb{R} \setminus \left\{-\frac{7}{3}, -\sqrt{\frac{35}{3}}, \frac{5}{3}, \sqrt{\frac{35}{3}}\right\}$.

c) For $\alpha = \sqrt{\frac{35}{3}}$, the matrix A reads $A = \begin{bmatrix} 1 & \sqrt{\frac{35}{3}} & -1 \\ \sqrt{\frac{35}{3}} & \frac{35}{3} & 1 \\ -1 & \sqrt{\frac{35}{3}} & 2 \end{bmatrix}$. We observe that the first submatrix of the matrix A, i.e. $A_1 = \begin{bmatrix} 1 & \sqrt{\frac{35}{3}} \\ \sqrt{\frac{35}{3}} & \frac{35}{3} \end{bmatrix}$, is singular, so that there does not exist a unique LU factorization of the matrix A. Since the matrix A is non-singular, we can still perform the LU factorization by introducing the permutation matrix $P \in \mathbb{R}^{3\times 3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$, which permutes the second and third rows of the matrix A. We obtain: $\widetilde{A} :=$

$PA = \begin{bmatrix} 1 & \sqrt{\frac{35}{3}} & -1 \\ -1 & \sqrt{\frac{35}{3}} & 2 \\ \sqrt{\frac{35}{3}} & \frac{35}{3} & 1 \end{bmatrix}$. We perform the Gauss factorization $\widetilde{A} = LU$ of the matrix PA by repeating the procedure of point b); we obtain:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ \sqrt{\frac{35}{3}} & 0 & 1 \end{bmatrix}, \qquad U = \begin{bmatrix} 1 & \sqrt{\frac{35}{3}} & -1 \\ 0 & 2\sqrt{\frac{35}{3}} & 1 \\ 0 & 0 & 1+\sqrt{\frac{35}{3}} \end{bmatrix},$$

where the pivot elements associated to the matrix $\widetilde{A} = PA$ are $\widetilde{a}^{(1)}_{1,1} = 1$ and $\widetilde{a}^{(2)}_{2,2} = 2\sqrt{\frac{35}{3}}$.

d) We obtain $R = \begin{bmatrix} 1 & 1 & -1 \\ 0 & \sqrt{\frac{32}{3}} & 2\sqrt{\frac{3}{32}} \\ 0 & 0 & \sqrt{\frac{5}{8}} \end{bmatrix}$.