Numerical Methods
Bachelor in Physics

Practicum 2:
Systems of linear equations

Contact: jeroen.mulkers@uantwerpen.be U.305

1. **'Sparse' matrices in MATLAB**

   In order to reduce computation time, one can take into account the special structure of some matrices. E.g. this is the case for large *sparse* matrices wich have many elements equal to zero. Matrices used in the modelling and analysis of physical systems are often sparse. Only if one takes into account the sparsity of the matrix, it becomes possible to solve systems with thousands of variables. In this practicum session, we will use some of the powerful MATLAB operations with sparse matrices. Not only can algorithms be designed to run much faster for matrices with many zeros, the sparsity of matrices will also reduce the memory needed.

   MATLAB does **not** consider a matrix to be sparse by default. The user has to declare whether a matrix belongs to the *sparse class* or to the *full class*. For example

   $\gg$ b=sparse(a)

   converts a matrix a into a sparse matrix b. From then on, MATLAB will consider b to be sparse and will use dedicated algorithms for sparse matrices. In order to convert a sparse matrix to a full matrix, we can simply type

   $\gg$ c=full(b)

   The binary operands $*, +, -, /$ and $\backslash$ will return a sparse matrix if *both* of the arguments are sparse. Only if a is sparse, the functions chol(a) and lu(a) will return a sparse matrix. If one argument is sparse and another argument is full, than the result will be a full matrix. Use the command speye(n) instead of eye(n) to define a sparse unity matrix. In order to check whether or not a variable is sparse, use the command issparse.

   Let us construct a sparse matrix. This can be done with the command sparse in many different ways (see help sparse). Try to execute the following example

```
≫ colpos = [1 2 1 2 5 3 4 3 4 5];
≫ rowpos = [1 1 2 2 2 4 4 5 5 5];
≫ val=[12 -4 7 3 -8 -13 11 2 7 -4];
≫ a=sparse(rowpos,colpos,val,5,5)
```

Have a look at the result. Check also the output of

```
≫ b=full(a)
```

Now, execute the following

```
≫ [issparse(a) issparse(b) nnz(a) nnz(b)]
```

and inspect the meaning of the command nnz.

Construct, in an efficient manner, a sparse tridiagonal 3000×3000 matrix A with fours on the diagonal and twos on the off-diagonals. Cunstruct also a vector b of length 3000, with $b_i = i$. Now, solve the system $A\mathbf{x} = \mathbf{b}$ with augmented matrix:

$$\left[\begin{array}{cccccc|c} 4 & 2 & 0 & & & & 1 \\ 2 & 4 & 2 & & & & 2 \\ & \ddots & \ddots & \ddots & & & \vdots \\ & & & 2 & 4 & 2 & 2999 \\ & & & 0 & 2 & 4 & 3000 \end{array}\right]$$

Do this for A initialized as a sparse matrix and A initialized as a full matrix. Compare the computation time for both cases (use the command tic...toc).

Solve the same system, but make this time use of the LU decomposition (command lu).

Other ways to construct sparse matrices are with the commands sprand and sprandsym. These commands generate random sparse matrices and symmetrical random sparse matrices. E.g. try to generate some $m \times n$ random matrices with a nonzero element density d by using

```
≫ A=sprand(m,n,d)
```

Next, use the command

```
≫ A=sprandsym(6,.4,[1 2.5 6 9 2 4.3])
```

to construct a symmetrical random sparse matrix with a nonzero element density equal to 40 % and eigenvalues [1 2.5 6 9 2 4.3].

2. The method of least squares is a method to construct a polynomial fit for experimental data points. The 3rd order polynomial $y = a_0 + a_1 x + a_2 x^2 + a_3 x^3$ which fits best n data point, can be found by solving the following system of linear equations:

$$\begin{array}{lllllllll} a_0 n & + & a_1 \sum x_i & + & a_2 \sum x_i^2 & + & a_3 \sum x_i^3 & = & \sum y_i \\ a_0 \sum x_i & + & a_1 \sum x_i^2 & + & a_2 \sum x_i^3 & + & a_3 \sum x_i^4 & = & \sum y_i x_i \\ a_0 \sum x_i^2 & + & a_1 \sum x_i^3 & + & a_2 \sum x_i^4 & + & a_3 \sum x_i^5 & = & \sum y_i x_i^2 \\ a_0 \sum x_i^3 & + & a_1 \sum x_i^4 & + & a_2 \sum x_i^5 & + & a_3 \sum x_i^6 & = & \sum y_i x_i^3 \end{array}$$

Find the 3rd order least square-polynomial which fits the following data:

x	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5
y	4.3	6.2	10.1	13.5	19.8	22.6	24.7	29.2

Make a figure of your results (data points + the fit). Make use of the MATLAB command polyval and make sure to evaluate the fit on a smaller grid.

3. We have seen that it is important to interpret formulas with an inverse matrix in terms of LU decomposition (this is what MATLAB actually does internally when you use the \ operator). Write a MATLAB function which calculates $\alpha = \mathbf{c}^T A^{-1} \mathbf{d}$ with $\mathbf{c}, A$ and $\mathbf{d}$ as input arguments, without calculating A^{-1} explicitly.

4. During the lecture, we have derived Jacobi's method to solve a system of equations iteratively. The following example demonstrates Jacobi's method:

$$\begin{array}{rcl} -7x + 2y + 0z & = & 5 \\ 0x + 16y + 8z & = & 7 \\ 6x - 11y + 26z & = & -3 \end{array} \quad \rightarrow \quad \begin{array}{rcl} \hat{x}_{i+1} & = & -\frac{1}{7}(5 + 0\hat{x}_i - 2\hat{y}_i - 0\hat{z}_i) \\ \hat{y}_{i+1} & = & \frac{1}{16}(7 + 0\hat{x}_i + 0\hat{y}_i - 8\hat{z}_i) \\ \hat{z}_{i+1} & = & \frac{1}{26}(-3 - 6\hat{x}_i + 11\hat{y}_i + 0\hat{z}_i) \end{array}$$

We can also use a similar method, called the Gauss-Seidel method. After updating x (calculating $\hat{x}_{i+1}$), we can already use this result for the next calculation. For the given example, the Gauss-Seidel method looks like:

$$\begin{array}{rcl} \hat{x}_{i+1} & = & -\frac{1}{7}(5 + 0\hat{x}_i - 2\hat{y}_i - 0\hat{z}_i) \\ \hat{y}_{i+1} & = & \frac{1}{16}(7 + 0\hat{x}_{i+1} + 0\hat{y}_i - 8\hat{z}_i) \\ \hat{z}_{i+1} & = & \frac{1}{26}(-3 - 6\hat{x}_{i+1} + 11\hat{y}_{i+1} + 0\hat{z}_i) \end{array}$$

We can expect intuitively that the Gauss-Seidel method will converge faster than Jacobi's method.

Formally, we can derive the Gauss-Seidel method as follows:

$$A\mathbf{x} = \mathbf{b} \Rightarrow (L_A + M)\mathbf{x} = \mathbf{b} \Rightarrow \mathbf{x} = L_A^{-1}(\mathbf{b} - M\mathbf{x}),$$

with L_A as the lower triangular matrix of A ($L_{Aij} = A_{ij}$ if $i \leq j$ and 0 elsewhere) and M as the upper triangular matrix of A, but with zero on the diagonal elements ($M_{ij} = A_{ij}$ if $i > j$ and 0 elsewhere). The algorithm can be written down as

$$\hat{\mathbf{x}}_{i+1} = L_A^{-1}(\mathbf{b} - M\hat{\mathbf{x}}_i), \text{ for } i = 0, 1, 2, \ldots \text{ for a given } \hat{\mathbf{x}}_0.$$

Implement the Gauss-Seidel method by modifying the function jacobi.m appropriately. Make use of the MATLAB commands tril and triu which give the lower and upper triangular matrix of the input argument (in help triu is mentioned how to get zeros on the diagonal of M). Take into account with what has been said in exercise 3.

Test your implementation with an example and compare the rate of convergence with Jacobi's method.

5. At the end of chapter 2, we have seen how one can solve a system of **non-lineaire** by an extended Newton method. Consider a system of n non-linear equations and n variables. The algorithm can be written down as

$\mathbf{x}_0$ = Initial value
for i=0:N
Solve for $\Delta\mathbf{x}^{(i)}$: $\begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} \begin{bmatrix} \Delta x_1^{(i)} \\ \vdots \\ \Delta x_n^{(i)} \end{bmatrix} = - \begin{bmatrix} f_1 \\ \vdots \\ f_n \end{bmatrix}$
Update the solution: $\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)} + \Delta\mathbf{x}^{(i)}$
end

Implement this algorithm. Assume that the Jacobian matrix J(x), which calculates the first derivatives, is known.

Solve the following non-linear system of equations (there are two solutions):

$$\begin{cases} x_1 + 2x_2 - 2 = 0 \\ x_1^2 + 4x_2^2 - 4 = 0 \end{cases}$$

Start by implementing the Jacobian J(x) for this system.