Goethe-Center for Scientific Computing (G-CSC)
Goethe-Universität Frankfurt am Main

Modeling and Simulation I

(Practical SIM1, WS 2018/19)
and
NeuroBioInformatik
(Übung NBI, WS 2018/19)

M. Huymayer, J. Wang, Dr. A. Nägel, Dr. M. Hoffer

Exercise sheet 3 (Due: Mo., 27.11.2017, 10h)

In Sheet 2 we introduced the explicit Euler method for the solution of systems of ODEs, i.e., we computed vector-valued solutions $\mathbf{u} : [t_0, t_e] \to \mathbb{R}^d$. For explicit methods, the step from a single ODE to systems of ODEs does not require much structural change of the algorithm. However, for implicit methods, a much broader framework has to be developed in order to implement even a simple solver. The implicit methods presented in the lecture employ the so called Newton method for estimating $\mathbf{u}^{\text{new}} \approx \mathbf{u}(t_{k+1})$ from a known approximation $\mathbf{u}^{\text{old}} \approx \mathbf{u}(t_k)$.
For $\mathbb{R}^d$, the Newton method requires a solver for systems of linear equations comprising the Jacobian matrix $\mathbf{J} \in \mathbb{R}^{d \times d}$. In task 1 we will develop a matrix solver which will then be used within the Newton method required for the implicit ODE solver.

Aufgabe 1 (8P + 2P)
The task of this exercise is to implement a solver for matrix equations, i.e., given a vector $\mathbf{b} \in \mathbb{R}^N$ and a non-singular matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$, the routine must return a vector $\mathbf{x} \in \mathbb{R}^N$ such that the matrix equation

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

holds. In order to do so, we employ the so called LU decomposition of a matrix. Its pseudo code is provided in Listing 1. The LU decomposition receives a matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ and replaces the entries $\mathbf{A}_{ij}$ $(1 \leq i, j \leq N)$ by an upper-diagonal matrix $\mathbf{U} \in \mathbb{R}^{N \times N}$ (i.e., matrix entries of $\mathbf{U}$ below the diagonal are zero) and a lower-diagonal matrix $\mathbf{L} \in \mathbb{R}^{N \times N}$ (i.e., matrix entries of $\mathbf{L}$ above the diagonal are zero) such that

$$\mathbf{A}_{ij} := \begin{cases} \mathbf{L}_{ij}, & i < j, \\ \mathbf{U}_{ij}, & i \geq j. \end{cases}$$

Algorithm 1 LU-Decomposition

Require: $\mathbf{A} \in \mathbb{R}^{N \times N}$
  for $k = 1, \dots, N-1$ **do**
    for $j = k+1, \dots, N$ **do**
      $\mathbf{A}_{jk} := \frac{\mathbf{A}_{jk}}{\mathbf{A}_{kk}}$
      for $i = k+1 \dots, N$ **do**
        $\mathbf{A}_{ji} := \mathbf{A}_{ji} - \mathbf{A}_{ki} \cdot \mathbf{A}_{jk}$
      end for
    end for
  end for

Result: Modified matrix $\mathbf{A}$ storing the two triangular matrices $\mathbf{L}$ (lower-diag.) and $\mathbf{U}$ (upper-diag.).

The diagonal entries of $\mathbf{L}$ are all equal to 1 and won't be stored. That makes the memory consumption optimal since both matrices $\mathbf{L}, \mathbf{U}$ are stored in the (no longer needed) memory of $\mathbf{A}$ and zero entries of the empty triangles are not stored as well.
The decomposition provides matrices $\mathbf{L}, \mathbf{U}$, such that $\mathbf{A} = \mathbf{LU}$ holds. Therefore, in order to solve the linear equation system $\mathbf{Ax} = \mathbf{b}$, instead the system $\mathbf{LUx} = \mathbf{b}$ can be solved with two triangular matrices. Thus, we first solve $\mathbf{Ly} = \mathbf{b}$ with an auxiliary variable $\mathbf{y}$ and then solve for the final result $\mathbf{Ux} = \mathbf{y}$ in a second step.
The solution $\mathbf{Ly} = \mathbf{b}$ is computed by forward substitution, i.e., the elements of vector $\mathbf{y}$ are computed via

$$y_i = \frac{1}{\mathbf{L}_{ii}} \left(b_i - \sum_{k=1}^{i-1} \mathbf{L}_{ik} \cdot y_k \right), \quad i = 1, 2, \dots, N-1, N. \tag{1}$$

The system $\mathbf{Ux} = \mathbf{y}$ is then solved using backward substitution, i.e., the elements of vector $\mathbf{x}$ are computed via

$$x_i = \frac{1}{\mathbf{U}_{ii}} \left(y_i - \sum_{k=i+1}^{N} \mathbf{U}_{ik} \cdot x_k \right), \quad i = N, N-1, N-2, \dots, 2, 1. \tag{2}$$

If required, also the inverse $\mathbf{A}^{-1} \in \mathbb{R}^{N \times N}$ can be computed by noting $\mathbf{AA}^{-1} = \mathbb{1}$ with the identity matrix $\mathbb{1} \in \mathbb{R}^{N \times N}$. Thus, choosing the vector $\mathbf{b}^i$ as the i-th column of the identity matrix and solving $\mathbf{Ax}^i = \mathbf{b}^i$, this solution $\mathbf{x}^i$ is the i-th column of the inverse matrix $\mathbf{A}^{-1}$.

Hints:

- Caution: the pseudo-code does NOT use zero based numbering.
- To specify a matrix $\mathbf{A}$ in Groovy, use `double[][] A`.
- For testing purposes a matrix input is provided on the GitHub page (http://bit.ly/2g4IRSh). It works similar to the `VectorRhsODE` component class introduced in Sheet 2.
- Use the `Matrix2String` component which is provided on the GitHub page to print your matrices (http://bit.ly/2eQNC5Q).
- Detect and handle errors caused by matrix singularity as follows: introduce a check for matrix singularity in the outer loop (`for k`), e.g., `if(A[k][k] == 0) throw new RuntimeException(‘‘matrix singular’’)`
- To simplify debugging check your LU decomposition with an online service, e.g., http://bit.ly/2g5QDyK.
- Similar services exist for matrix inversion, e.g., http://bit.ly/2eQz8mw

Tasks/Questions:

(a) Implement a Groovy class that performs the inversion of a given non-singular matrix $\mathbf{A} \in \mathbb{R}^{N\times N}$ of type `double[][]`. Structure your code by the three step procedure described above: provide a method to compute the LU decomposition, a method that returns a solution $\mathbf{x}$ for a vector $\mathbf{b}$, and a method that returns the matrix inverse $\mathbf{A}^{-1}$.

(b) Verify your implementation with the two non-singular 3x3 matrices available on GitHub: http://bit.ly/2fssoYb.

Provide the output obtained with the `Matrix2String` component. To verify your results, compute the product $\mathbf{A}\mathbf{A}^{-1}$ which is equal to the identity matrix.

Aufgabe 2 (7 P)
Implement the Crank-Nicolson scheme in order to solve the system of ordinary differential equations (ODE)

$$\begin{cases} \text{Find } \mathbf{u} : [t_0, t_n] \to \mathbb{R}^d, \text{ such that} \\ \frac{\partial}{\partial t}\mathbf{u}(t) = \mathbf{f}(t, \mathbf{u}) \qquad \text{on } [t_0, t_n], \\ \mathbf{u}(t_0) = \mathbf{u}_0, \end{cases} \tag{3}$$

where $\mathbf{u}_0 \in \mathbb{R}^d$ is the start value and $t_0, t_n \in \mathbb{R}$ are the start- and endpoints of the interesting time interval.

The Crank-Nicolson scheme is based on the iteration

$$t^{\text{new}} = t^{\text{old}} + h, \tag{4}$$

$$\mathbf{u}^{\text{new}} = \mathbf{u}^{\text{old}} + h \cdot \frac{1}{2} \left\{ \mathbf{f}(t^{\text{new}}, \mathbf{u}^{\text{new}}) + \mathbf{f}(t^{\text{old}}, \mathbf{u}^{\text{old}}) \right\}, \tag{5}$$

where h is a given step size. Please note, that the computation of the new solution value $\mathbf{u}^{\text{new}}$ in equation (5) is in general a nonlinear problem. That is why we reformulate the nonlinear problem as an equation of the form

$$\mathbf{g}(\mathbf{u}^{\text{new}}) = \mathbf{0}. \tag{6}$$

We use the Newton method to solve this equation. The Newton iteration is performed by successively updating

$$\mathbf{u}^{\text{new}} \leftarrow \quad \mathbf{u}^{\text{new}} - (\mathbf{J}_g(\mathbf{u}^{\text{new}}))^{-1} \mathbf{g}(\mathbf{u}^{\text{new}}) \tag{7}$$

until a tolerance threshold $\|\mathbf{g}(\mathbf{u}^{\text{new}})\| \leq \epsilon$ (with a small ϵ, e.g. 10^{-5}) for the Euclidean norm has been reached. Assume that the exact derivative of $\mathbf{f}$ with respect to $\mathbf{u}$, namely the Jacobian $\mathbf{J}(t, \mathbf{u})$, is known and provided by the user. Further assume that the iteration parameter ϵ and `maxIter` are used as shown in the practical session to control the Newton iteration.

Aufgabe 3 (3 P)
Use the Crank-Nicolson scheme in order to solve the Lotka-Volterra model from Sheet 2, Exercise 2a. Produce plots with the VectorTrajectoryPlotter with step-size $h = 0.01$ and $h = 0.001$. Compare your results with the solution that has been obtained with the explicit Euler method.

Hints:

- Use the `JacobianInput` component from the github page to provide the derivative for the Crank-Nicolson scheme: `http://bit.ly/2g81Cpk`.
- To prevent automatic project reloading or classloader problems, use the new interface `JacobianInputInterface` as paramater type instead of `JacobianInput`. The new type is part of the plugin `vectoroderhsinterface.jar`.

- Use $\mathbf{u}^{\text{old}}$ as a start value for the Newton method (just like we did in the last Practical session).

Remark: Send your implemented source code as VRL-Studio project (.vrlp file) and the answers to the questions as plain text in an email. Append the pdfs produced with the TrajectoryPlotter to the email.
Send your solution to `practical.sim1@gcsc.uni-frankfurt.de` until Monday, 27.11.2017, 10h.