Coupled Processes CIE4365 2017: The Lotka-Volterra Problem

Timo Heimovaara

30th April 2018

Introduction

The goal of the first two assignments was to get you up to speed on using Matlab for solving differential equations. In my opinion it is best to solve a differential equation by using a standard approach. As mentioned before, Matlab has several built in solvers for solving ordinary differential equations and the first assignment was to implement the Lotka Volterra approach using these built in solvers. The Lotka Volterra equations are a so-called predator prey model and describe the following rates:

$$\begin{aligned} \frac{\mathrm{d}R}{\mathrm{d}t} &= \alpha R - \beta RF \\ \frac{\mathrm{d}F}{\mathrm{d}t} &= -\gamma RF + \delta F \end{aligned}$$

in which R and F are the two state variables (number of Rabbits and Foxes). And α, β, γ and δ are the growth and predation parameters. The Matlab solvers solve the following equation:

$$M(t,y)y' = f(t,y)$$

where $y' = \frac{\mathrm{d}y}{\mathrm{d}t}$ and t is time. $M(t,y)$ is the mass matrix. For the Lotka-Volterra problem it is the identity matrix. It is important to realise that Matlab is a package that is matrix based. This means that the user may assume that any variable used can be a matrix, a vector or a scalar. It is up to the user to program a the function $f(t,y)$ in such a way that it will perform the correct type of calculations based on the type of variable passed in t and y. Given the fact that y' and y are related, the structure of y' should be the same as that of y. We achieve this by using the $y(1,:)$ syntax, where the colon (:) is used to indicate that the code should use all elements in this index.

Another very important thing to realise is that Matlab uses the order in which the parameters are passed to the function, not the name. So if you call a function with the following command:

```
NewVal = SomeFunction(a,b)
```

and the function is defined as:

```
function NV = SomeFunction(b,a)
```

the value a passed to the function during calling will be stored in the variable b when executing *SomeFunction*. This is because of the fact that all variables are local in Matlab, i.e. the scope in which a variable is defined is only within the scope of the function (or the main script).

Programming the rate function

The following code is my implementation for the rate function ($f(t,y)$)LotkaVolterra problem:

```
function dy = LotkaVolterraTHe(t,y,Par)
% Function for calculating rates for a Lotka Volterra equation system
% The function returns a matrix dy which contain the rates for the prey
% (dy(1)) and predator (dy(2));
% t contains the time vector (which is a dummy variable in this code as it
% is not used in the solution).
% y is a matrix containing the the states of the system. y is ordered as
% follows: the first row contains the values for y(1), the second row
% contains values for y(2). This function calculates rates for each column
% in the the 2xn matrix...
% par contains the four growth rate parameters


%Copy parameter to local variable
a = Par.a;
b = Par.b;
c = Par.c;
d = Par.d;

dy(1,:)=  a.*y(1,:) - b.*y(1,:).*y(2,:);
dy(2,:) = -c.*y(2,:) + d.*y(1,:).*y(2,:);
```

Solving with Explicit Euler

My implementation of the explicit first order forward difference Euler method is:

```
%% Coupled Processes CIE4365
% LOTKA - VOLTERRA
%
% Script introducing Numerical Approximation techniques for solving
% Ordinary Differential Equations

%clear memory and close all open figure windows
clear
close all

difftime = eps; % this is assumed to be zero (equality for time)
convcrit = 1e-8; % test value for assessing convergence

%% Discretization
%Initialize Solver (discretization of space and time)
%No spatial discretization required...

% Time Discretization
deltmax = 0.005;  %max time step;
outtime = (0:.25:250); %times for storing model output
t = 0; %initial time
tend = outtime(end);


%% Initial Model States and model parameters
%Define Parameters for the Lotka Volterra model (see Wikipedia
%and other internet sources

Par.a = 1;
Par.b = 0.2;
Par.c = 0.1;
Par.d = 0.01;

% initial values can be identified from equilibrium state
% equilibrium (y=a/b; x = c/d)
% yini = [Par.c/Par.d Par.a/Par.b]
yini = [11;3];

%% Explicit EULER
%Using explicit Euler

disp('explicit Euler');
tic
%Initialize output matrices
%store initial values in output matricese
T1(1) = 0;

%Note that yini is a column vector, output is stored rows (see ode help)
Y1(1,1:2) = yini';
y = yini;
nout = 1;

%Run model over time period
while abs(t-tend) > difftime %this approach is best to test if doubles are equal
   %Calc Rates
   dy = LotkaVolterraTHe(t,y,Par);

   %Time step should be small enough to prevent negative states from
   %occuring
   %Check delt
   dttest = abs(0.1*y./dy.*(dy<0))+(dy>=0); %we do not allow negative values
   dtout = outtime(nout+1)-t; %we do not want to miss an output time
   delt = min([dttest(:)' deltmax dtout]);

   %Update states
   y = y + dy.*delt;
   t = t + delt;
```

```
    %Update output matrix
    if abs(t-outtime(nout+1)) < difftime
       nout = nout+1;
       T1(nout) = t;
       Y1(nout,1) = y(1); Y1(nout,2) = y(2);
    end
end
toc
```

When you analyse this piece of code, you will see that I only pass the latest value of the states to the function LotkaVolterraTHe. The variable y is continuously updated and in principle I do not store all intermediate values of y for all time steps. I only store the results of y in my output vector when the time is equal to one of the values in *outtime*, then I copy the current value of y to $Y1(nout, 1)$ and I copy the current value of t to $T1(nout)$. After copying I increase the value of *nout* with 1 and I continue solving the equation. This approach is a common approach when solving so-called Initial Value Problems. We start with a known initial value and use the rate equation to update the values as time proceeds.

In order to be certain that the model will never calculate a negative value by taking a too large a time step, I limit the time step based on the current rate so that the maximum change in states will be 10%:

```
dttest = abs(0.1*y./dy.*(dy<0))+(dy>=0); %we do not allow negative values
```

Solving with Predictor Corrector and Runge Kutta

The same approach can be used for the predictor corrector method and the fourth order Runge Kutta Method. Please note because both methods are more stable than the Explicit Euler method, I chose to use a bigger time step, $deltmax = 0.1$ instead of 0.005. This significantly speeds up the solution.

```
%% Predictor Corrector
% Approach based on Predictor corrector from Wikipedia
disp ('Euler Predictor Corrector');
deltmax = 0.1;
tic
y = yini; %initial states;
t = 0;
%store initial values in output matricese
T2(1) = 0;
Y2(1,1:2) = yini';
nout = 1;

while abs(t-tend) > difftime
   %Calc Rates
   dy = LotkaVolterraTHe(t,y,Par);

   %Check delt
   dttest = abs(0.1*y./dy.*(dy<0))+(dy>=0);
   dtout = outtime(nout+1)-t;
   delt = min([dttest(:)' deltmax dtout]);

   %iteration with predictor corrector
   %start with Euler step
   converged = false;
   yn = y + dy.*delt;
   while ~converged
       dyn = LotkaVolterraTHe(t+delt,yn,Par);
       ynn = y+delt./2.*(dy+dyn);
       if abs(yn-ynn)<convcrit
           converged=true;
       else
           yn=ynn;
       end
   end

   %Update states
   y = ynn;
   t = t + delt;
   %Update output matrix
   if abs(t-outtime(nout+1)) < difftime
```

```
      nout = nout+1;
      T2(nout) = t;
      Y2(nout,1) = y(1); Y2(nout,2) = y(2);
   end
end
toc

%% Runge Kutta method
% Approach based on common fourth-order Runge-Kutta method from Wikipedia
disp ('Runge Kutta');

tic
y = yini; %initial states;
t = 0;
%store initial values in output matricese
T3(1) = 0;
Y3(1,1:2) = yini';
nout = 1;

while abs(t-tend) > difftime
   %Calc Rate in order to estimate max timestep
   dy = LotkaVolterraTHe(t,y,Par);

   %Check delt
   dttest = abs(0.1*y./dy.*(dy<0))+(dy>=0);
   dtout = outtime(nout+1)-t;
   delt = min([dttest(:)' deltmax dtout]);

   %Calc Rates (k1 to k4 for RK4)
   %dy1 = LotkaVolterraTHe(t, y, Par);
   k1 = delt.*dy;
   k2 = delt.*LotkaVolterraTHe(t+delt/2, y+k1/2, Par);
   k3 = delt.*LotkaVolterraTHe(t+delt/2, y+k2/2, Par);
   k4 = delt.*LotkaVolterraTHe(t+delt, y+k3, Par);

   %Update states
   y = y + (k1+2*k2+2*k3+k4)/6;
   t = t + delt;
   %Update output matrix
   if abs(t-outtime(nout+1)) < difftime
      nout = nout+1;
      T3(nout) = t;
      Y3(nout,1) = y(1); Y3(nout,2) = y(2);
   end
end
toc
```

Solving with built-in solvers (ode45 and ode15i)

As mentioned before, Matlab has a built in set of ode solvers which are very powerful implementations. Although many of you preferred the Runge Kutta approach above the ODE45 solvers, you should realise that by being able to pass certain properties of your problem to the solver, the ODE solvers can do a very good job. I agree that the benefit is not very clear for the Lotka Volterra Problem.

Using my implementation of the Lotka Volterra problem which was written in the form of $y' = f(t, y)$ I implemented the ode45 solver as follows:

```
%% Built in ODE Solver
%Using built in ODE solver
options = odeset('RelTol',1e-6,'AbsTol',1e-6);%,'OutputFcn','odeplot');

disp ('Built in ODE solver (ode45)');

tic
[T4,Y4] = ode45(@(t,y) LotkaVolterraTHe(t,y,Par),outtime,yini,options);
toc
```

First we the options for the ode solver by calling the function *odeset*. In this case we choose to set the accuracy of the solver to be 4 orders of magnitude more precise than the default values ($1e-4$) by setting the parameters

$RelTol$ and $AbsTol$. The ode solvers expect that t and y be passed to the rate function as the first two parameters and because I also want to be able to pass my own parameters (in this case a structure Par), I choose to use the anonymous function syntax:

```
@(t,y) LotkaVolterraTHe(t,y,Par)
```

which Matlab interprets as a function is passed with the basic parameter form $@(t, y)$ and the handle to this function points to $LotkaVolterraTHe(t, y, Par)$. The user must tell the Matlab where in the call to $LotkaVolterraTHe$, the parameters t and y can be found. Some of you implemented the rate function without the parameter t because it was not needed and correctly used the following syntax: $@(t, y)$ $RateFun(y, Par)$.

By passing *outtime* to the solver, the solver will use all times in the output matrix to save output. If only the start and the end time are given, Matlab will generate output times. I prefer to control the output times. The solver then requires the initial value for the problem, in my case I pass *yini*. You have to realise that *yini* is a different value than y, in this case *yini* needs to be initialised, y does not have to be, because y is used to identify the position of y in the calling sequence for $LotkaVolterraTHe$.

Fully implicit ODE solver: ode15i

The ode15i is a fully implicit solver, all the other solvers are explicit. The function solved by the ode15i solvers is $f(t, y, y') = 0$ and my implementation for this function is:

```
%% Built in ODE Solver Implicit!
%Using built in ODE solver ode15i
disp('Built in ODE solver implicit (ode15i)')
options = odeset('RelTol',1e-4,'AbsTol',1e-4);%,'OutputFcn','odeplot');
tic
t0 = 0;
y0 = yini;
yp0 = LotkaVolterraTHe(t0,y0,Par);
%[y0,yp0] = decic(@(t,y,dy) LotkaVolterraImpTHe(t,y,dy,Par),...
%    t0,y0,0,yp0,0);

[T5,Y5] = ode15i(@(t,y,dy) LotkaVolterraImpTHe(t,y,dy,Par),outtime,y0,...
    yp0,options);
toc
```

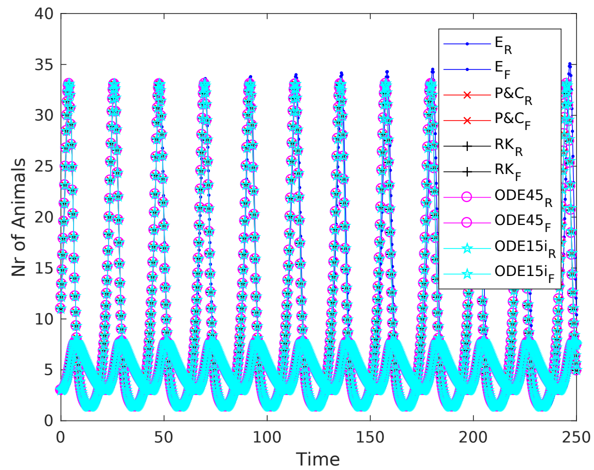
The Matlab function ode15i requires initial values both the states as well as the rates. Sometimes the analytical function for a correct set of initial values is not known and then you can use the built-in matlab function *decic* to estimate correct values. However, in our case we can use $LotkaVolterraTHe$ to calculate the value of the rates for the values of *yini*, which is what I have done. Then I call ode15i to solve my function $LotkaVolterraImpTHe$, which solves $res = f(t, y, y')$ so that $res = 0$.

```
function res = LotkaVolterraImpTHe(t,y,dy,Par)
%Function for calculating rates for a Lotka Volterra equation system
%dx are the rates for the prey (dx(1)) and predator (dx(2);
%t is time
%x are the states of the system (x(1): prey, x(2), predator)
%par contains the four parameters


rates = LotkaVolterraTHe(t,y,Par);

res = dy - rates;
```

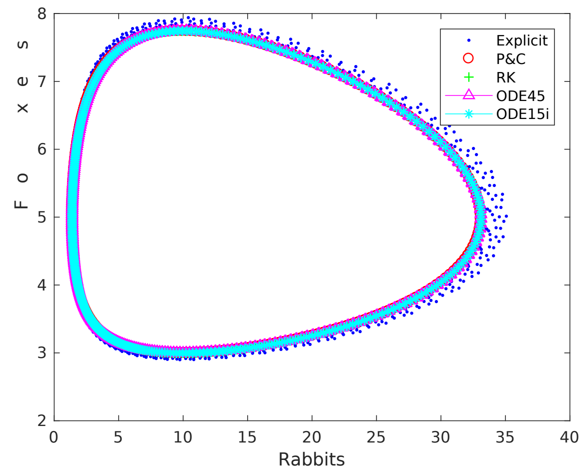
The main benefit of using the built-in ode solvers of matlab is that the programming of the time stepping, checking for negative results etc. is not required, using the odeset command we can control these issues to a great detail. As a result the effort for solving an ordinary differential equation is reduced to defining your rate function.

Plotting the results

Matlab comes with a very large range of plotting functions, so there are many possible options to present your results. I have chosen to plot the evolution in time of the rabits and the foxes for the different solutions. In addition I choose to plot the characteristic functions where we plot the rabbits against the foxes. The results are stored in a time vector (T1 to T5) and output matrices where the rows indicate time, and the columns the different states.



(a) Rabbits and Foxes as a function of time



(b) Foxes against Rabbits

I have annotated the graphs with different symbols for the different lines and latex style text for the legends with a subscript R for the rabbits and F for the foxes.

```
figure(1)
clf
plot(T1,Y1,'b.-')
hold on;
plot(T2,Y2,'rx--');
plot(T3,Y3,'k+:');
plot(T4,Y4,'mo-');
plot(T5,Y5,'cp-.');
xlabel('Time')
ylabel('Nr of Animals')
legend({'E_R','E_F', 'P&C_R','P&C_F','RK_R','RK_F','ODE45_R','ODE45_F','ODE15i_R','ODE15i_F'})

figure(2)
clf
plot(Y1(:,1),Y1(:,2),'b.')
hold on
plot(Y2(:,1),Y2(:,2),'ro')
plot(Y3(:,1),Y3(:,2),'g+')
plot(Y4(:,1),Y4(:,2),'m^-')
plot(Y5(:,1),Y5(:,2),'c*:')
xlabel('Rabbits');
ylabel('Foxes');
legend({'Explicit', 'P&C', 'RK','ODE45','ODE15i'})
```

The plots for this script are shown.