Comparison of Second Order Numerical Techniques for Linear and Non-Linear ODEs

David Keffer
Department of Chemical Engineering
University of Tennessee, Knoxville
date begun: September 1999
updated: October 10, 2005
updated again: February 18, 2014

Table of Contents

I. Purpose

The purpose of this lecture package is not to introduce a new second order numerical solution technique for solving ODEs when we already have a very excellent fourth-second order numerical solution technique for solving ODEs. Rather, the purpose is to introduce these second order techniques because they are used to solve PDEs. By first applying the techniques to ODEs, we shall see how they fit into the family of methods that include the Euler method and the Runge-Kutta method. When we then see the technique used for the solution of PDEs, it won't seem so foreign.

Additionally, in this document we compare implicit and explicit second order algorithms. The implicit algorithms work only for linear ODEs. The explicit algorithms are applied to any (nonlinear or linear) ODEs. Again, these implicit techniques are not typically used to solve ODEs because if we have a linear ODE we will solve it analytically. However, the linear techniques are used to solve linear PDEs. We introduce them here, where the application to ODEs is familiar and we can clearly compare the advantages and disadvantages of implicit and explicit techniques.

II. Single Ordinary Differential Equation

Consider the single first-order ODE (either linear or nonlinear)

$$\frac{dy}{dt} = f(y,t) \tag{1}$$

with the initial condition

$$y(t_o) = y_o \tag{2}$$

We can again solve this numerically, using an Euler-like formula

$$y_i = y_{i-1} + (t_i - t_{i-1})\tilde{f} \tag{3}$$

where $\tilde{f}$ is an approximation to the derivative over the interval from t_{i-1} to t_i. For a second order method, the approximation to the derivative is simply an average of the values at t_{i-1} and t_i.

$$\tilde{f} = \frac{1}{2}\left[f(t_{i-1}, y_{i-1}) + f(t_i, y_i)\right] \tag{4}$$

The problem is that we don't know the value of y_i that appears on the right hand side of equation (4).

II.A. Second Order Method for Generic ODE

In the general case (meaning that equation (1) is either linear or nonlinear in *y*), we can use the Euler method to approximate y_i for the purposes of equation (4).

$$y_i = y_{i-1} + (t_i - t_{i-1}) f(t_{i-1}, y_{i-1}) \tag{5}$$

We then substitute this value into equation (4) and substitute the resulting value of $\tilde{f}$ into equation (3), yielding a second order method:

$$y_i = y_{i-1} + (t_i - t_{i-1}) \tilde{f} = y_{i-1} + (t_i - t_{i-1}) \frac{1}{2} \left[f(t_{i-1}, y_{i-1}) + f(t_i, y_{i-1} + (t_i - t_{i-1}) f(t_{i-1}, y_{i-1})) \right] \tag{6}$$

This implementation of the second order method is a member of a class of methods known as predictor-corrector methods, because you use Euler's method to predict y_i and you use equation (6) to correct the value. Specifically, equation (6) is called Heun's method. A code that implements equation (6) is provided at the end of these notes in appendix A.1.

II.B. Second Order Implicit Method for Linear ODE

In obtaining equation (6), we were forced to make an additional approximation, namely we had to use the Euler method for a preliminary estimate of y_i. If the ODE (eqn. (1)) is linear, then we do not have to make this approximation in order to solve the problem. Therefore, an implicit method that does not make this approximation will be more accurate than Heun's method. When we apply this technique to PDEs we will be seeking to gain advantages in accuracy wherever we can find them.

We now derive the implicit method. Consider a linear ODE of the form

$$\frac{dy}{dt} = f(y,t) = a(t)y(t) + b(t) \tag{7}$$

with the initial condition of equation (2). We substitute this linear ODE into equation (4) obtaining

$$\tilde{f} = \frac{1}{2} \left[a_{i-1} y_{i-1} + b_{i-1} + a_i y_i + b_i \right] \tag{8}$$

where we have used the shorthand notation that $a_i = a(t_i)$, as we have done for *y* and *b* as well. We substitute equation (8) into equation (3)

$$y_i = y_{i-1} + (t_i - t_{i-1})\frac{1}{2}\left[a_{i-1}y_{i-1} + b_{i-1} + a_i y_i + b_i\right] \tag{9}$$

We solve for the unknown, y_i, obtaining

$$A_{i,i}y_i = -A_{i,i-1}y_{i-1} + B_i \tag{10}$$

where $A_{i,j}$ is the coefficient (in general a function of the independent variable *t*) in front of the j^{th} variable at the i^{th} time, t_i, and has the form

$$A_{i,j} = \begin{cases} 1 - \frac{(t_i - t_{i-1})}{2}a_j & \text{if i = j} \\ -1 - \frac{(t_i - t_{i-1})}{2}a_j & \text{if i} \neq \text{j} \end{cases} \tag{11}$$

$$B_i = \frac{(t_i - t_{i-1})}{2}(b_i + b_{i-1}) \tag{12}$$

This method is classified as an implicit method because the value of the unknown y_i appears on both the left hand side and right hand side of equation (9).

Option 1. Solve for *y* sequentially

We could proceed as we did in the Euler method, where we evaluate *y* at t_1, then use that result to generate *y* at t_2, etc. through repeated applications of equation (10). This is totally legitimate and will lead to the correct approximation of the solution.

A code that implements the sequential solution of this implicit method is given in appendix A.2.

Option 2. Solve for all *y* simultaneously

Consider that we divide the independent variable, *t*, into *n* intervals, each of size

$$\Delta t = \frac{(t_f - t_o)}{n} \tag{13}$$

where is the initial time from the initial condition in equation (2) and t_f is the final time, beyond which we are no longer interested in the solution of the ODE. Our approximate solution will be evaluated at *n*+1 points, the initial condition and the *n* subsequent values of *t*. If we designate

the solution of the ODE at each of these points as y_i for i = 1 to n+1, then we can write the following set equations:

$$y_1 = y_o \qquad \text{for } i = 1$$

$$A_{i,i} y_i = -A_{i,i-1} y_{i-1} + B_i \qquad \text{for } i = 2 \text{ to } n+1 \tag{14}$$

This is a system of linear algebraic equations. It can be written in matrix form as:

$$\underline{\underline{A}}\,\underline{y} = \underline{B} \tag{15}$$

where the vector $\underline{B}$ is principally defined by equation (12), except for the first entry which is the initial condition,

$$\underline{B} = \begin{bmatrix} y_o \\ B_2 \\ \vdots \\ B_{n+1} \end{bmatrix} \tag{16}$$

and where the matrix $\underline{\underline{A}}$ is principally defined by equation (11), except for the first row which is the initial condition,

$$\underline{\underline{A}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ A_{2,1} & A_{2,2} & 0 & 0 \\ 0 & \ddots & \ddots & 0 \\ 0 & 0 & A_{n+1,n} & A_{n+1,n+1} \end{bmatrix} \tag{17}$$

Thus, we have transformed the numerical solution of a linear ODE into the solution of a system of linear algebraic equations, which we know how to solve. One inversion of this matrix effectively solve the ODE problem.

The transformation of ODEs to linear algebraic equations through the discretization of the independent and dependent variables is a commonly encountered transformation. It is one that is used ubiquitously through-out the solution of PDEs and Integral Equations. Therefore, it was educational to introduce the concept here, even though we would likely never use this methodology to solve a single linear ODE as we did here.

Note that the sequential and simultaneous solutions of the implicit method yield (to within machine precision) the same result. The implicit method will yield a more accurate solution than Heun's method.

A code that implements the simultaneous solution of this implicit method is given in appendix A.3.

II.C. Comparison of Second Order Explicit and Implicit Methods

Consider the initial value problem:

$$\frac{dy}{dx} + a(x)y = b(x)$$

where we have an initial condition of the form:

$$y(x = x_o) = y_o$$

with the specific values given by:

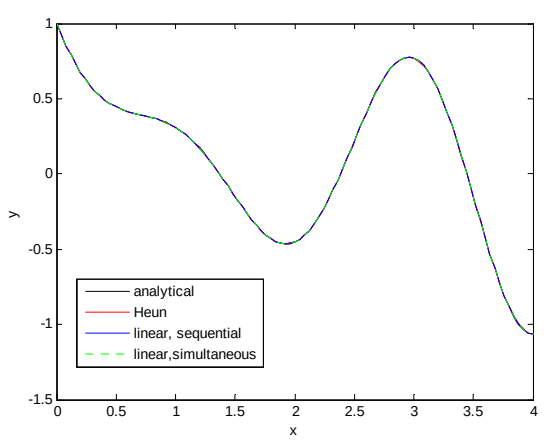
$$a(x) = 2\,, \qquad b(x) = x\sin(3x)\,,\ y(x=0) = 1$$

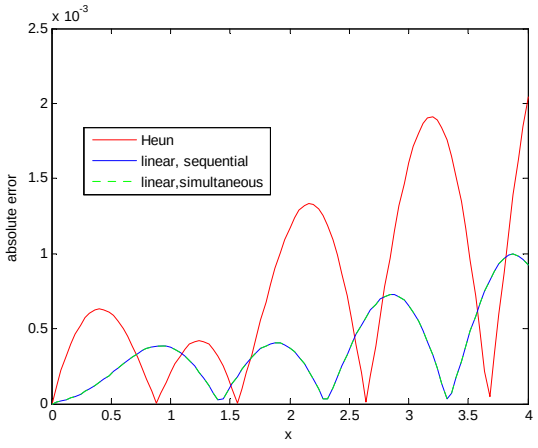
This IVP has the analytical solution

$$y_{nh}(x) = \left[\frac{157e^{-2x}}{169} + \frac{1}{169}\left[(26x+5)\sin(3x) + (-39x+12)\cos(3x)\right]\right]$$

We now this ODE using the three second-order methods described above, Heun's method, the implicit sequential method and the implicit simultaneous method. For all methods, we set $x_f = 4$ and use $n = 100$ intervals.

Plots of the solution and absolute error are shown below.





From the plot of the solutions, it is difficult to see the difference between the methods for this time step. From the plot of the error, one sees that the errors of the two implicit methods are

exactly the same and both are better than Heun's method. The waviness of the absolute error is due to the non-monotonic nature of the solution.

III. Systems of Ordinary Differential Equations

A general system of first order ODEs can be expressed as

$$\frac{d\underline{y}}{dt} = \underline{f}(\underline{y}, t) \tag{18}$$

with the initial conditions

$$\underline{y}(t_o) = \underline{y}_o \tag{19}$$

III.A. Second Order Method for a System of Generic ODEs

In the general case (the equations are either linear or nonlinear in $\underline{y}$, we can write the straightforward multi-equation analog of Heun's method. We again use Euler's method to predict the value of the function,

$$\underline{y}_i^p = \underline{y}_{i-1} + (t_i - t_{i-1})\underline{\tilde{f}} = \underline{y}_{i-1} + (t_i - t_{i-1})\underline{f}(\underline{y}_{i-1}, t_{i-1}) \tag{20}$$

We super-scripted this with a p to identify this as the prediction step. The prediction step is followed by the correction step

$$\underline{y}_i = \underline{y}_{i-1} + (t_i - t_{i-1})\underline{\tilde{f}} = \underline{y}_{i-1} + (t_i - t_{i-1})\frac{1}{2}\left[\underline{f}(\underline{y}_{i-1}, t_{i-1}) + \underline{f}(\underline{y}_i^p, t_i)\right] \tag{21}$$

Therefore, these techniques are sometimes called predictor-corrector methods. As we did in the single equation case, we sequentially solve for each $\underline{y}_i$ based on $\underline{y}_{i-1}$.

A code that implements equations (20) and (21) is provided at the end of these notes in appendix A.4. This is essentially Heun's method extended to a system of ODEs.

III.B. Second Order Implicit Method for a System of Linear ODE

We can also repeat the derivation for a more accurate second order method if every equation in the system of ODEs is linear. In fact, if the equations are linear, we don't even need to make the approximation that we can isolate the derivatives on the left hand side of the equation, as we did above in the general solution. If the equations are linear, the general system can be written as

$$\underline{\underline{C}}(t)\frac{d\underline{y}}{dt} = \underline{\underline{A}}(t)\underline{y}(t) + \underline{B}(t) \tag{22}$$

Equation (22) would be identical to equation (19) if $\underline{\underline{C}}(t)$ were the identity matrix. In solving a system of mass and energy balances, this is frequently the case.

The derivation of the method, now becomes more complicated. We need three indices on the elements of $\underline{\underline{A}}(t)$. The first index indicates the equation. The second index indicates the variable. The third index indicates the time step, once we have performed the discretization of time necessary to obtain the numerical solution. We will use the notation $A_{i,j}^{(k)}$ to designate the coefficient (the time functionality is now implicit) of the j^{th} variable in the i^{th} equation at discretized time t_k. Similarly, $\underline{\underline{A}}^{(k)}$ represents the entire matrix of coefficients at discretized time t_k.

If $\underline{\underline{C}}$ is a constant matrix, we can discretize equation (22) as

$$\underline{\underline{C}}\frac{\underline{y}^{(k)}-\underline{y}^{(k-1)}}{\left(t_k-t_{k-1}\right)}=\frac{1}{2}\left[\underline{\underline{A}}^{(k-1)}\,\underline{y}^{(k-1)}+\underline{B}^{(k-1)}+\underline{\underline{A}}^{(k)}\,\underline{y}^{(k)}+\underline{B}^{(k)}\right] \tag{23}$$

Once again, you can proceed to solve this problem sequentially or simultaneously.

Option 1. Solve for $\underline{y}$ sequentially

We can rearrange equation (23) to isolate our vector of unknowns $\underline{y}^{(k)}$

$$\left[\underline{\underline{C}}-\frac{\Delta t}{2}\underline{\underline{A}}^{(k)}\right]\underline{y}^{(k)}=\left[\underline{\underline{C}}+\frac{\Delta t}{2}\underline{\underline{A}}^{(k-1)}\right]\underline{y}^{(k-1)}+\frac{\Delta t}{2}\left(\underline{B}^{(k)}+\underline{B}^{(k-1)}\right) \tag{24}$$

Everything on the right-hand side of equation (24) is known. You invert and solve for $\underline{y}^{(k)}$.

A code that implements the sequential solution of this implicit method is given in appendix A.5.

Option 2. Solve for all $\underline{y}^{(k)}$ simultaneously

We could write equation (24) for all values of *k* from 1 to *n*+1 (for a discretization of *t* involving *n* intervals). We then have a system of *m* ODEs, we now have a system of *m*(*n*+1) linear algebraic equations. We could invert this larger matrix if we so chose. In solving all of the equations simultaneously, we would have to create a single matrix of unknowns. We have some freedom as to how we choose to order our unknowns. Two possible arrangements for a system with *m* equations and n time intervals are shown below.

$$\underline{Y} = \begin{bmatrix} y_1^{(1)} \\ y_1^{(2)} \\ \vdots \\ y_1^{(n+1)} \\ y_2^{(1)} \\ y_2^{(2)} \\ \vdots \\ y_2^{(n+1)} \\ \vdots \\ y_m^{(1)} \\ y_m^{(2)} \\ \vdots \\ y_m^{(n+1)} \end{bmatrix} \qquad \text{or} \qquad \underline{Y} = \begin{bmatrix} \underline{y}^{(1)} \\ \underline{y}^{(2)} \\ \vdots \\ \underline{y}^{(n)} \\ \underline{y}^{(n+1)} \end{bmatrix} \tag{25}$$

In the first arrangement, the unknowns are grouped by variable. In the second arrangement, the unknowns are grouped by time. The latter form is preferable for two reasons. First, it reduces the bandwidth of the resulting matrix, which equates to a quicker computation. Second, it also facilitates the mathematical description of the system. Here we assume that the vector of unknowns takes the form of the second arrangement.

We then can write a system of $m(n+1)$ linear algebraic equations of the form

$$\underline{\underline{A}}^{*}\,\underline{Y} = \underline{B}^{*} \tag{26}$$

where, in order to present a compact description of the matrix and vector in equation (26), we first rewrite equation (24) as

$$\underline{\underline{M}}_{k,k}\,\underline{y}^{(k)} = -\underline{\underline{M}}_{k,k-1}\,\underline{y}^{(k-1)} + \underline{B}_k^{*} \tag{27}$$

where the matrix, $\underline{\underline{M}}_{k,k}$, and the vector, $\underline{B}_k^{*}$, are defined by comparison to equation (24).

The matrix and vector used in equation (26) are related to the smaller matrices and vectors used in equation (27) by the following equations:

$$\underline{\underline{A}}^{*} = \begin{bmatrix} \underline{\underline{I}} & \underline{\underline{0}} & \underline{\underline{0}} & \underline{\underline{0}} & \underline{\underline{0}} \\ \underline{\underline{M}}_{2,1} & \underline{\underline{M}}_{2,2} & \underline{\underline{0}} & \underline{\underline{0}} & \underline{\underline{0}} \\ \underline{\underline{0}} & \underline{\underline{M}}_{3,2} & \underline{\underline{M}}_{3,3} & \underline{\underline{0}} & \underline{\underline{0}} \\ \underline{\underline{0}} & \underline{\underline{0}} & \ddots & \ddots & \underline{\underline{0}} \\ \underline{\underline{0}} & \underline{\underline{0}} & \underline{\underline{0}} & \underline{\underline{M}}_{n+1,n} & \underline{\underline{M}}_{n+1,n+1} \end{bmatrix} \tag{28}$$

and

$$\underline{B}^{*} = \begin{bmatrix} \underline{y}_o \\ \underline{B}_2^{*} \\ \underline{B}_3^{*} \\ \vdots \\ \underline{B}_{n+1}^{*} \end{bmatrix} \tag{29}$$

Thus, we have shown how we can transform the numerical solution of a system of linear first-order ODEs into the solution of a system of linear algebraic equations.

A code that implements the simultaneous solution of this implicit method is left as an exercise for the curious reader.

Appendix. Matlab Subroutines

In this section, we provide routines for implementing the various optimization methods described above that are not translated from "Numerical Recipes". Note that these codes correspond to the theory and notation exactly as laid out in this book. These codes do not contain extensive error checking, which would complicate the coding and defeat their purpose as learning tools. That said, these codes work and can be used to solve problems.
As before, on the course website, two entirely equivalent versions of this code are provided and are titled *code.m* and *code_short.m*. The short version is presented here. The longer version, containing instructions and serving more as a learning tool, is not presented here. The numerical mechanics of the two versions of the code are identical.

Code A.1. Heun's Method – 1 ODE (heun1_short)

```
function [x,y]=heun1(n,xo,xf,yo);
dx = (xf-xo)/n;
x = zeros(n+1,1);
for i = 1:1:n+1
   x(i) = xo + (i-1)*dx;
end
y = zeros(n+1,1);
y(1) = yo;
for i =  1:1:n
   x1 = x(i);
   y1 = y(i);
   k1 = funkeval(x1,y1);
   x2 = x(i) + dx;
   y2 = y(i) + dx*k1;
   k2 = funkeval(x2,y2);
   dydx = 1.0/2.0*(k1 + k2);
   y(i+1) = y(i) + dx*dydx;
end
close all;
iplot = 1;
if (iplot == 1)
   plot (x,y,'k-o'), xlabel( 'x' ), ylabel ( 'y' );
end
fid = fopen('heun1_out.txt','w');
fprintf(fid,'x                 y \n');
fprintf(fid,'%23.15e %23.15e   \n', [x,y]');
fclose(fid);

function dydx = funkeval(x,y);
dydx = -1.0*y^2;
```

An example of using heun1_short is given below.

```
» [x,y]=heun1_short(10,0,2,1);
```

This program generates outputs in three forms. First, the x and y vectors are stored in memory and can be directly accessed. Second, the program generates a plot of y vs. x. Third, the program generates an output file, *heun1_out.txt*, that contains x and y vectors in tabulated form.

Code A.2. Implicit Method, Sequential Solution – 1 ODE (linode_2o_seq_1_short)

```
function [x,y]=linode_2o_seq_1_short(n,xo,xf,yo);
dx = (xf-xo)/n;
x = zeros(n+1,1);
for i = 1:1:n+1
   x(i) = xo + (i-1)*dx;
end
y = zeros(n+1,1);
y(1) = yo;
for i =  2:1:n+1
   Aii = 1.0 - 0.50*dx*funkeval_a(x(i));
   Aij = -1.0 - 0.50*dx*funkeval_a(x(i-1));
   Bi = 0.50*dx*(funkeval_b(x(i)) + funkeval_b(x(i-1)));
   y(i) = 1.0/Aii*(Bi - Aij*y(i-1));
end
close all;
iplot = 1;
if (iplot == 1)
   plot (x,y,'k-o'), xlabel( 'x' ), ylabel ( 'y' );
end
%
%  write result to file 'linode_2o_seq_1_out.txt'
%
fid = fopen('linode_2o_seq_1_out.txt','w');
fprintf(fid,'x                 y \n');
fprintf(fid,'%23.15e %23.15e   \n', [x,y]');
fclose(fid);

%  dydx = a(x)*y(x) + b(x);
function a = funkeval_a(x);
a = -x;

function b = funkeval_b(x);
b = sin(x);
```

An example of using linode_2o_seq_1_short is given below.

```
>> [x,y]=linode_2o_seq_1_short(100,0,10,1);
```

Code A.3. Implicit Method, Simultaneous Solution – 1 ODE (linode_2o_sim_1_short)

```
function [x,y]=linode_2o_sim_1_short(n,xo,xf,yo);
dx = (xf-xo)/n;
x = zeros(n+1,1);
for i = 1:1:n+1
   x(i) = xo + (i-1)*dx;
end
y = zeros(n+1,1);
A = zeros(n+1,n+1);
B = zeros(n+1,1);
```

```
A(1,1) = 1.0;
B(1) = yo;
for i =  2:1:n+1
   B(i) = 0.50*dx*(funkeval_b(x(i)) + funkeval_b(x(i-1)));
   A(i,i) = 1.0 - 0.50*dx*funkeval_a(x(i));
   A(i,i-1) = -1.0 - 0.50*dx*funkeval_a(x(i-1));
end
invA = inv(A);
y = invA*B;
close all;
iplot = 1;
if (iplot == 1)
   plot (x,y,'k-o'), xlabel( 'x' ), ylabel ( 'y' );
end

fid = fopen('linode_2o_sim_1_out.txt','w');
fprintf(fid,'x                  y \n');
fprintf(fid,'%23.15e %23.15e   \n', [x,y]');
fclose(fid);

%  dydx = a(x)*y(x) + b(x);

function a = funkeval_a(x);
a = -x;

function b = funkeval_b(x);
b = sin(x);
```

An example of using linode_2o_sim_1_short is given below.

```
>> [x,y]=linode_2o_sim_1_short(100,0,10,1);
```

Code A.4. Heun's Method – n ODEs (heunn_short)

```
function [x,y]=heunn_short(n,xo,xf,yo);
dx = (xf-xo)/n;
x = zeros(n+1,1);
for i = 1:1:n+1
   x(i) = xo + (i-1)*dx;
end
m=max(size(yo));
y = zeros(n+1,m);
y(1,1:m) = yo(1:m);
dydx = zeros(1,m);
ytemp = zeros(1,m);
k1 = zeros(1,m);
k2 = zeros(1,m);
for i =  1:1:n
   x1 = x(i);
   ytemp(1:m) = y(i,1:m);
   k1(1:m) = funkeval(x1,ytemp);
   x2 = x(i) + dx;
   ytemp(1:m) = y(i,1:m) + dx*k1(1:m);
   k2(1:m) = funkeval(x2,ytemp);
   dydx(1:m) = 1.0/2.0*(k1(1:m) + k2(1:m));
   y(i+1,1:m) = y(i,1:m) + dx*dydx(1:m);
```

```
end
close all;
iplot = 1;
if (iplot == 1)
   for i = 1:1:m
      color_index = get_plot_color(i);
      plot (x(:),y(:,i),color_index);
      hold on;
   end
   hold off;
   xlabel( 'x' );
   ylabel ( 'y' );
   legend (int2str([1:m]'));
end
fid = fopen('heunn_out.txt','w');
fprintf(fid,'x  y(1) ... y(m) \n');
for i = 1:1:n+1
   fprintf(fid,'%23.15e ', x(i));
   for j = 1:1:m
      fprintf(fid,'%23.15e ', y(i,j));
   end
   fprintf(fid,' \n');
end
fclose(fid);

function dydx = funkeval(x,y);
dydx(1) = -1.0*y(1) - 2.0*y(2) - 0.5*y(3)^2;
dydx(2) = -0.1*y(1) - 4.0*y(2) - 0.5*y(3);
dydx(3) = -0.5*y(1) - 0.4*y(2) - 0.2*y(3);
```

An example of using heunn_short is given below.

```
>> [x,y1]=heunn_short(100,0,10,[1,1,1]);
```

Code A.5. Implicit Method, Sequential Solution – n ODEs (linode_2o_seq_n_short)

```
function [x,y]=linode_2o_seq_n(n,xo,xf,yo);
dx = (xf-xo)/n;
x = zeros(n+1,1);
for i = 1:1:n+1
   x(i) = xo + (i-1)*dx;
end
m=max(size(yo));
y = zeros(n+1,m);
y(1,1:m) = yo(1:m);
for k =  2:1:n+1
   AA = zeros(m,m);
   BB = zeros(m,1);
   for i = 1:1:m
      BB(i) = 0.5*dx*(funkeval_b(i,x(k)) + funkeval_b(i,x(k-1)));
      for j = 1:1:m
         AA(i,j) = funkeval_c(i,j,x(k))   - 0.50*dx*funkeval_a(i,j,x(k));
         term =    funkeval_c(i,j,x(k-1)) + 0.50*dx*funkeval_a(i,j,x(k-1));
         BB(i) = BB(i) + term*y(k-1,j);
      end
   end
   invAA = inv(AA);
```

```
   y(k,1:m) = invAA*BB;
end
close all;
iplot = 1;
if (iplot == 1)
   for i = 1:1:m
      color_index = get_plot_color(i);
      plot (x(:),y(:,i),color_index);
      hold on;
   end
   hold off;
   xlabel( 'x' );
   ylabel ( 'y' );
   legend (int2str([1:m]'));
end
fid = fopen('linode_2o_seq_n_out.txt','w');
fprintf(fid,'x  y(1) ... y(m) \n');
for i = 1:1:n+1
   fprintf(fid,'%23.15e ', x(i));
   for j = 1:1:m
      fprintf(fid,'%23.15e ', y(i,j));
   end
   fprintf(fid,' \n');
end
fclose(fid);

%  c(x)*dy_/dx = A(x)*y_ + b(x);
function aout = funkeval_a(i,j,x);
amat = [0 1; -1 -1];
aout = amat(i,j);

function bout = funkeval_b(i,x);
bvec = [1/(1+x); 0];
bout = bvec(i);

function cout = funkeval_c(i,j,x);
cmat = [1 0; 0 1];
cout = cmat(i,j);
```

An example of using `linode_2o_seq_n_short` is given below.

```
>> [x,y1]=linode_2o_seq_n_short(100,0,10,[1,1]);
```