MAT 275 Laboratory 3
Numerical Solutions by Euler and Improved Euler Methods

In this session we look at basic numerical methods to help us understand the fundamentals of numerical approximations. Our objective is as follows.

1. Implement Euler's method as well as an improved version to numerically solve an IVP.
2. Compare the accuracy and efficiency of the methods with methods readily available in MATLAB.
3. Apply the methods to specific problems and investigate potential pitfalls of the methods.

Instructions: For your lab write-up, follow the instructions of LAB 1.

Euler's Method

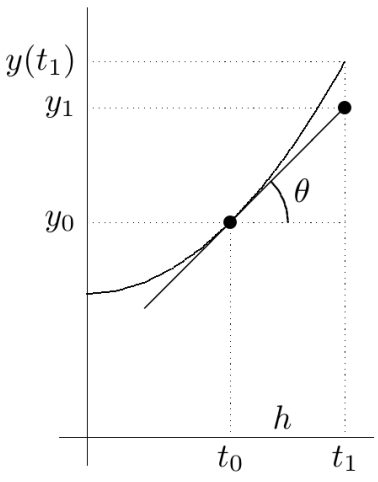
Consider the initial value problem (IVP): $y' = f(t, y),\ y(t_0) = y_0$, where t_0, y_0 are given numbers and f is a given function of two variables. We are looking for the solution (i.e. function) $y(t)$ on an interval $[t_0, t_{\text{final}}]$. In many practical applications, the function f can be so complicated that it is not possible to solve the IVP analytically. In such a situation we want to find a numerical solution. That is, for a given number of points $t_0 < t_1 < t_2 < ... < t_N = t_{\text{final}}$ we would like to find y_n's, the approximate values of $y(t_n)$ $(n = 1, 2, ..., N)$, the solution $y(t)$ at $t = t_n$. The following picture illustrates the first step, i.e. finding y_1, using Euler's method. The curve in the figure represents the graph of the unknown solution $y(t)$. It is known that (t_0, y_0) belongs to the graph and that $\tan(\theta) = y'(t_0) = f(t_0, y(t_0)) = f(t_0, y_0)$. So, we know the tangent line to the unknown curve $y(t)$ at the point (t_0, y_0) (see the figure).

To approximate $y(t_1)$, we use the definition of the derivative, by which $\frac{y(t_1)-y(t_0)}{h} \approx y'(t_0) = f(t_0, y_0)$ for h small. Solving for $y(t_1)$, we get $y(t_1) \approx y(t_0) + hf(t_0, y_0) = y_0 + hf(t_0, y_0)$. So, we set

$$y_1 = y_0 + hf(t_0, y_0) \tag{1}$$

and thus, $y_1 \approx y(t_1)$. Similarly, to approximate $y(t_2)$, we use $\frac{y(t_2)-y(t_1)}{h} \approx y'(t_1) = f(t_1, y(t_1))$, and therefore, $y(t_2) \approx y(t_1) + hf(t_1, y(t_1)) \approx y_1 + hf(t_1, y_1)$. So, we set $y_2 = y_1 + hf(t_1, y_1)$ as our approximation of $y(t_2)$. Continuing this process, in general, for $n = 0, 1, ..., N-1$, we have

$$\boxed{\begin{array}{rcl} y_{n+1} & = & y_n + hf(t_n, y_n) \\ t_{n+1} & = & t_n + h \end{array}}$$



As an example, consider the following initial value problem:

$$y' = y;\ y(0) = 2. \tag{2}$$

Here, the function $f(t, y) = y$ does not depend on t explicitly (*autonomous* equation), but does depend implicitly, through $y = y(t)$.

Remark: This IVP, in fact, does not require numerical solution, since we can easily solve it analytically and find that the solution is $y = 2\,e^t$ (do it!). There are two main reasons why we still want to solve

it numerically. First, we want to compare the numerical (i.e. approximate) solution with the analytical (i.e. exact) solution. This will help us understand the behavior and features of Euler's method. The other reason is to test the validity of our code. It is a common and useful practice, when developing a code, to test it in situations where the output is known.
To apply Euler's method to the IVP (2), we start with the initial condition and select a step size h; say, $h = 0.2$. Since we are constructing arrays t and y without dimensionalizing them first, it is best to clear these names in case they have been used already in the same MATLAB work session.

```
>> clear t y  % no comma b/w t and y! type "help clear" for more info
>> y(1)=2; t(1)=0; h=0.2;
```

Here, the initialization `y(1)=2` should **not** be interpreted as "the value of y at 1 is 2", but rather as "the first value in the array `y` is 2". In other words, the `1` in `y(1)` is an index, not a time value! Unfortunately, MATLAB indices in arrays must be positive (a legacy from FORTRAN...).
Since f is simple enough we may use the anonymous function syntax:

```
>> f=@(t,y) y;
```

The successive approximations at increasing values of t are then obtained as follows:

```
>> y(2)=y(1)+h*f(t(1),y(1)), t(2)=t(1)+h % Euler approximation at t = 0.2
y =
    2.0000    2.4000
t =
         0    0.2000

>> y(3)=y(2)+h*f(t(2),y(2)), t(3)=t(2)+h % Euler approximation at t = 0.4
y =
    2.0000    2.4000    2.8800
t =
         0    0.2000    0.4000
```

The arrays `y` and `t` are now 1×3 row arrays. The dimension increases as new values of y and t are computed.
It is obviously better to implement these steps in a `for` loop. The following loop implements the first 5 steps with the stepsize $h = 0.2$.

```
y(1)=2; t(1)=0; h=0.2;
for n = 1:5
   y(n+1)= y(n)+h*f(t(n),y(n));
   t(n+1) = t(n)+h;
end
```

Note that the output in each command has been suppressed (with a ;). The list of computed y values vs t values can be output at the end by typing:

```
>> [t(:),y(:)]    % same as [t',y']
ans =
         0    2.0000
    0.2000    2.4000
    0.4000    2.8800
    0.6000    3.4560
    0.8000    4.1472
    1.0000    4.9766
```

The next step is to write a *function* file (i.e. m-file) that takes in the following inputs: the name of the function (f) defining the ODE, the time span (i.e. the interval $[t_0, t_{\text{final}}]$), the initial condition (y_0), and

the number of steps (N) used in the approximation. Note that, if we are given the number of steps N, then the step size h can be easily computed as $h = \dfrac{t_{\text{final}} - t_0}{N}$.
As an output, we want the array of (t, y) points, where for each time step t, the corresponding y value is the numerical approximation of the solution of the IVP: $y' = f(t, y),\ y(t_0) = y_0$. The following Matlab function (given in the file `euler.m`) implements these ideas.

```
function [t,y] = euler(f,tspan,y0,N)

  % Solves the IVP y' = f(t,y), y(t0)=y0 in the interval tspan=[t0,tf]
  % using Euler's method with N time steps.
  % Input:
  % f = name of the anonymous function or  M-file that  evaluates
  % the ODE (if not an anonymous function,  use: euler(@f,tspan,y0,N))
  % For a system, the f must be given as column vector.
  % tspan = [t0, tf] where t0 = initial time and tf = final time
  % y0 = initial value of the dependent variable. If solving a system,
  %      initial conditions must be given as a vector.
  % N = number of steps used.
  % Output:
  % t = vector of time values where the solution was computed
  % y = vector of computed solution values.

  m = length(y0);
  t0 = tspan(1);
  tf = tspan(2);
  h = (tf-t0)/N;            % evaluate the time step size
  t = linspace(t0,tf,N+1); % create the vector of t-values
  y = zeros(m,N+1);         % allocate memory for the output y
  y(:,1) = y0';             % set initial condition
  for n=1:N
      y(:,n+1) = y(:,n) + h*f(t(n),y(:,n)); % implement Euler's method
  end
  t = t'; y = y';     % change t and y from row to column vectors
end  % end of the function euler
```

Remark: You should notice that the code above is slightly different from the first one we wrote (in particular, note the use of ":" when creating the output `y`). Although the two codes are equivalent in the scalar case, only the second one will work also for systems of Differential Equations (to be discussed in Lab 4, when we will use this code again).

We can implement the function with, say, $N = 50$ steps in the interval $[0, 1]$, by typing the following:

```
>> [t,y] = euler(f,[0,1],2,50) % use @f if defined in separate m-file
>> [t(1:5),y(1:5)]   % print first five rows of the matrix [t,y]
ans =
         0    2.0000
    0.0200    2.0400
    0.0400    2.0808
    0.0600    2.1224
    0.0800    2.1649
```

```
>> [t(end-4:end),y(end-4:end)]  % print last 5 rows of the matrix [t,y]
ans =
    0.9200    4.9732
    0.9400    5.0727
    0.9600    5.1741
    0.9800    5.2776
    1.0000    5.3832
```

We printed only the first five and last five points since the matrix `[t,y]` is long ($N+1 = 51$ rows). Let us now compare two approximations in the interval $[0, 1]$: one with $N = 5$ steps and the other with $N = 50$ steps. In the same figure, we plot both the approximate (i.e. numerical) solution, and the exact (i.e. analytical) solution, which is $y(t) = 2\,e^t$. Note that we can no longer call the output simply `[t,y]` because every time we call the function we will lose the output from the previous computations. Thus we will call `[t5,y5]` the output from Euler with $N = 5$ steps and `[t50,y50]` the output with $N = 50$ steps. The exact solution of the IVP is $y(t) = 2\,e^t$. We will store the exact solution in the vector `y`.

```
>> [t5,y5] = euler(f,[0,1],2,5); %use @f if defined in separate m-file
>> [t50,y50] = euler(f,[0,1],2,50);
>> t = linspace(0,1,100); y = 2*exp(t); %evaluate the exact solution
>> plot(t5,y5,'ro-',t50,y50,'bx-',t,y,'k-'); axis tight;
>> legend('Euler N = 5','Euler N = 50','Exact','location','northwest')
```

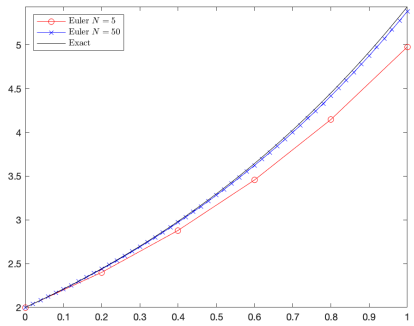


Figure 1: Euler's method applied to $y' = y$, $y(0) = 2$ with step sizes $h = 0.2$ and $h = 0.02$, compared to the exact solution.

IMPORTANT REMARK

When using 5 steps of size $h = 0.2$ the approximation of the exact value $y(1) = 2\,e^1 \approx 5.4366$ is stored in `y5(6)`. On the other hand, when using 50 intervals of size $h = 0.02$, it is stored in `y50(51)`. To avoid confusion we can denote these values by `y5(end)` and `y50(end)`, respectively. The exact value of the function at $t = 1$ can be retrieved using `y(end)`.

We can find the error (exact - approximate value) at $t = 1$ for both values of N by entering:

```
>> e5 = y(end) - y5(end)      % error when N = 5
e5 =
   0.4599
>> e50 = y(end) - y50(end)    % error when N = 50
e50 =
   0.0534
```

and also calculate the ratio of the errors to see how the approximation improves when the number of steps increases by a factor of 10:

```
>> ratio = e5/e50
ratio =
   8.6148
```

EXERCISES

1. (a) If you haven't already done so, enter the following commands:

```
f=@(t,y) y;
t=linspace(0,1,100); y=2*exp(t); %define exact solution of the ODE
[t50,y50]=euler(f,[0,1],2,50); %solve the ODE using Euler w/ 50 steps
```

Determine the Euler's approximation for $N = 500$ and $N = 5000$ and fill in the following table with the values of the approximations (`yN(end)`), errors (`eN=y(end)-yN(end)`) and ratios (`eNprevious/eN`) of consecutive errors at $t = 1$. Some of the values have already been entered based on the computations we did above.
Include the table in your report, as well as the MATLAB commands used to find the entries.

N	**approximation** `yN(end)`	**error** `eN`	**ratio** `eNprev/eN`
5	4.9766	0.4599	N/A
50		0.0534	8.6148
500			
5000			

(b) Examine the last column. How does the ratio of consecutive errors relate to the number of steps used? Your answer to this question should confirm the fact that Euler's method is a "first-order" method. That is, every time the step size is decreased by a factor of k, the error is also reduced (approximately) by the same factor, $k^1 = k$.

(c) Recall the geometrical interpretation of Euler's method based on the tangent line. Using this geometrical interpretation, can you explain why the Euler approximations y_N underestimate the solution y (that is, $y_N < y$, or equivalently, $e_N = y - y_N > 0$) in this particular example?

2. Consider the IVP

$$y' = -4.5\,y,\ y(0) = 1$$

for $0 \le t \le 5$.
The exact solution of this IVP is $y = e^{-4.5\,t}$.
The goal of this exercise is to visualize how Euler's method is related to the slope field of the differential equation. In order to do this we will plot the direction field together with the approximations and the exact solution.

(a) To plot the slope field we will use the MATLAB commands `meshgrid` and `quiver`. Enter the following commands:

```
t = 0:0.3:5; y = -9:2:11; %define a grid in t & y directions
[T,Y] = meshgrid(t,y); %create 2d matrices of points in ty-plane
dT = ones(size(T));    %dt=1 for all points
dY = -4.5*Y;           %dy = -4.5*y; this is the ODE
quiver(T,Y,dT,dY)      %draw arrows (t,y)->(t+dt, t+dy)
axis tight             %adjust look
hold on
```

After running these commands you should get the graph of the slope field.

(b) Use `linspace` to generate a vector of 100 t-values between 0 and 5. Evaluate the (exact) solution y at these t-values and plot it in black together with the direction field (use `'linewidth',2`).

(c) Enter the function defining the ODE as anonymous.
Use `euler.m` with $N = 10$ to determine the approximation to the solution. Plot the approximated points in red (use `'ro-','linewidth',2` as line-style in your plot), together with the exact solution and the direction field.
You should get Figure 2. The legend is not required, but to produce it, you can use
`legend('Slope Field','Exact Sltn','Approx. Sltn (N=10)','location','northwest');`
Based on the slope field and the geometrical meaning of Euler's method explain why the approximations are so inaccurate for this particular value of the stepsize.

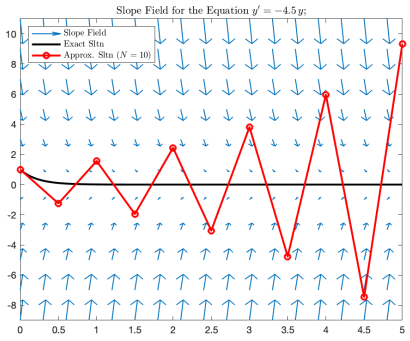


Figure 2: Euler's method applied to $y' = -4.5\,y$, $y(0) = 1$, $N = 10$ compared to the exact solution.

(d) Open a new figure by typing `figure`. Plot again the direction field but in a different window: `t = 0:0.3:5; y = -0.6:0.2:1.3;` Repeat part (b) and repeat part (c) but this time with $N = 20$. You should get Figure 3. Because of the different scaling of the y-axis, the exact solution (black curve) looks a bit different, but it is the same curve as in the previous figure. Comment on the result from a geometrical point of view.

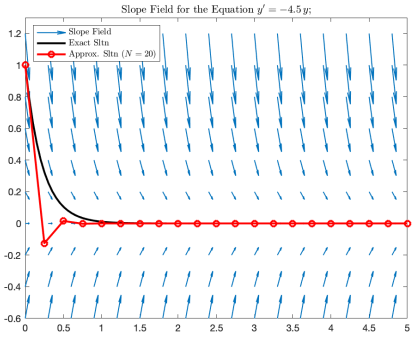


Figure 3: Euler's method applied to $y' = -4.5\,y$, $y(0) = 1$, $N = 20$, compared to the exact solution.

Note: Brief explanations of the commands `quiver` and `meshgrid` are included in Appendix A of this lab. In Appendix B we describe the Graphical User Interface `dfield8` for plotting slope fields.

Improved Euler's Method

The improved Euler's method is obtained by using an average of f values, i.e., a slope average:

$$y(t_1) \approx y(t_0) + h\,\frac{f(t_0, y(t_0)) + f(t_1, y(t_1))}{2} \tag{3}$$

Then substitute $y(t_0) = y_0$ and define y_1 as this approximate value of $y(t_1)$, i.e.

$$y_1 = y_0 + \frac{h}{2}\left(f(t_0, y_0) + f(t_1, y_1)\right). \tag{4}$$

The equation (4) defines the *trapezoidal* method. Unfortunately, this formula defines y_1 only implicitly, since y_1 appears on both sides of the equality so that an equation must be solved to obtain y_1. To avoid this problem, and since we already have made an approximation to get (4), we replace y_1 on the right-hand side by the approximation one would obtain by simply applying Euler's method from (t_0, y_0). The resulting quantity

$$y_1 = y_0 + \frac{h}{2}\left(f(t_0, y_0) + f(t_1, \underbrace{y_0 + hf(t_0, y_0)}_{\text{Euler } y_1 \text{ from (1)}})\right) \tag{5}$$

with $t_1 = t_0 + h$ is the *improved Euler* approximation. This approximation can be thought of as a correction to the Euler approximation. The iteration (5) is then repeated to obtain $y_2 \approx y(t_2), \ldots$, i.e.,

$$\begin{array}{rcl} f_1 & = & f(t_n, y_n) \\ f_2 & = & f(t_n + h, y_n + hf_1) \\ y_{n+1} & = & y_n + \frac{h}{2}(f_1 + f_2) \\ t_{n+1} & = & t_n + h \end{array}$$

Minor modifications should be made to the function `euler.m` to implement the improved Euler method.

EXERCISES

3. Modify the M-file `euler.m` to implement the algorithm for Improved Euler. Call the new file `impeuler.m` (include the file in your report).
   Test your code for the IVP $y' = y$, $y(0) = 2$, in the interval $[0, 1]$, using $N = 5$ steps.
   First enter the function $f(t, y) = y$ as anonymous function and then enter the following:

```
>> [t5,y5] = impeuler(f,[0,1],2,5); % use @f if defined in separate m-file
>> [t5,y5]
ans =
      0    2.0000
0.2000    2.4400
0.4000    2.9768
0.6000    3.6317
0.8000    4.4307
1.0000    5.4054
```

   Compare your output with the one above. You should obtain the same values.

4. Consider the IVP: $y' = y$, $y(0) = 2$, $\quad 0 \le t \le 1$

   (a) Determine the Improved Euler's approximation for $N = 50$, $N = 500$ and $N = 5000$. Fill in the following table with the values of the approximations, errors and ratios of consecutive errors at $t = 1$. Two values have already been entered to help you check your results. Recall that the exact solution to the ODE is $y(t) = 2\,e^t$. Include the table in your report, as well as the MATLAB commands used to find the entries.

N	**approximation** `yN(end)`	**error** `eN`	**ratio** `eN/eNprev`
5	5.4054		N/A
50			87.2394
500			
5000			

(b) Examine the last column. How does the ratio of the errors relate to the number of steps used? Your answer to this question should confirm the fact that Improved Euler's method is a "second-order" method. That is, every time the step size is decreased by a factor k, the error is reduced (approximately) by a factor of k^2.
Note: Since Euler's method is only of the 1st order, the Improved Euler's method is more efficient (hence the "improved").

5. Repeat Problem 2 for Improved Euler. Compare the results with the ones obtained with Euler's method.

APPENDIX A: The commands `meshgrid` and `quiver`

The function `meshgrid`: This command is especially important because it is used also for plotting 3D surfaces. Suppose we want to plot a 3D function $z = x^2 - y^2$ over the domain $0 \le x \le 4$ and $0 \le y \le 4$. To do so, we first take several points on the domain, say 25 points. We can create two matrices X and Y, each of size 5×5, and write the xy- coordinates of each point in these matrices. We can then evaluate z at these xy values and plot the resulting surface. Creating the two matrices X and Y is much easier with the `meshgrid` command:

```
x = 0:4;            % create a vector x = [0, 1, 2, 3, 4]
y = -4:2:4;         % create a vector y = [-4, -2, 0, 2, 4]
[X, Y] = meshgrid(x,y); % create a grid of 25 points and store
                        % - their coordinates in X and Y
```

Try entering `plot(X,Y,'o','linewidth',2)` to see a picture of the 25 points.
Then, just for fun try entering

```
 Z = X.^2 - Y.^2;
surf(X,Y,Z);
```

to get a picture of the surface representing $z = x^2 - y^2$. (The graph will be quite rough because of the limited number of points, but you should nonetheless recognize a "saddle").
Thus, in our code for the direction field, the `meshgrid` command is used to generate the points in the xy plane at which we want to draw the arrows representing the slope or tangent line to the solution at the point. Type `help meshgrid` for more information.

The function quiver: The command `quiver(X,Y,U,V)` draws vectors (arrows) specified by U and V at the points specified by X and Y. In our examples the vectors drawn are `(dT, dY)` where `dT =1` and `dY = -3*y`, thus these vectors are in the direction of the slope at the given point. The arrows are automatically scaled so as not to overlap. The matrices X and Y are built using the meshgrid command as explained above. Type `help quiver` for more explanation.

APPENDIX B: Plotting direction fields with `dfield8`

The Graphical User Interface `dfield8` originally from J. Polking can also be used to plot the direction field and selected solution curves. After downloading the file, enter `dfield8` in the command window. A new window will pop up, one that looks like Figure 4

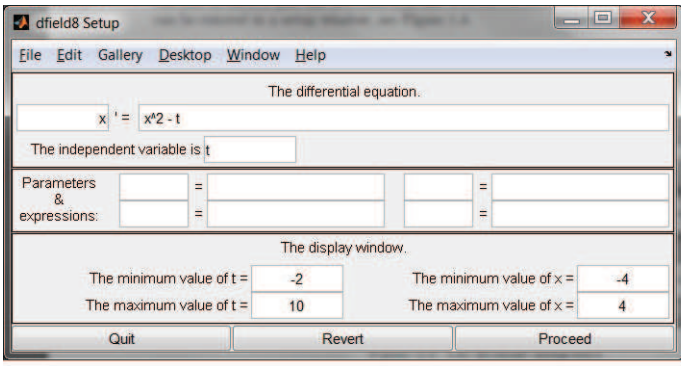
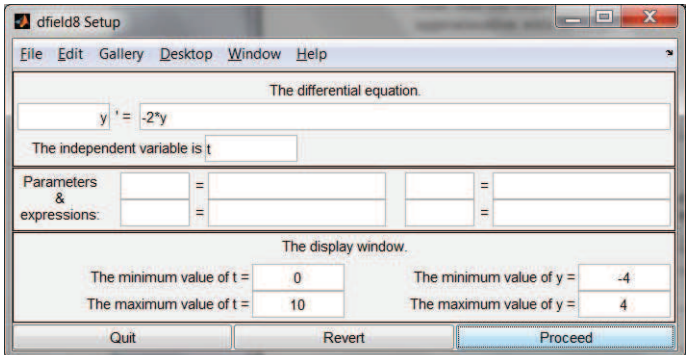


Figure 4: The `dfield8` setup GUI

ODEs with up to two parameters can be entered in the Setup window. For instance, we can enter

the ODE $y' = -2y$, identify the independent variable as t and choose the display window's dimension ($0 \le t \le 10$ and $-4 \le y \le 4$).



Click on *Proceed* to see the slope field. A sample display window is shown in Figure 5.

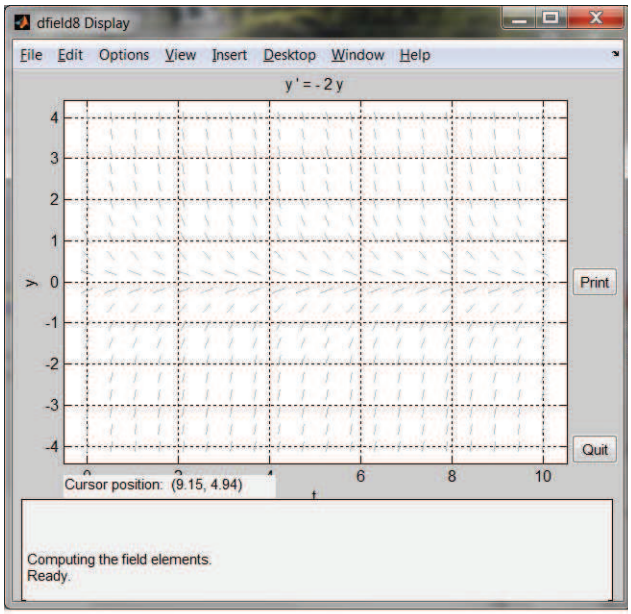


Figure 5: The `dfield8` direction field plot

Solutions curves can be drawn by clicking on the display window. Specific initial conditions can be entered by clicking on **Options ->Keyboard Input**. A new window will arise prompting the user to input initial values for the independent and dependent variables as an initial condition for the IVP.

More information about various options are available at `http://math.rice.edu/~dfield`.