Keck Cluster
Differential Equations

June 30, 2001

In practice researchers are generally interested in solving systems of differential equations. A system of \( m \) equations in \( m \) unknowns has the form

\[
y'_1 = f_1(t, y_1, y_2, \ldots, y_m) \\
y'_2 = f_2(t, y_1, y_2, \ldots, y_m) \\
\vdots \\
y'_m = f_m(t, y_1, y_2, \ldots, y_m).
\]

Initial value problems consist of a system together with initial conditions:

\[
y_1(t_0) = y_1^{(0)} \\
y_2(t_0) = y_2^{(0)} \\
\vdots \\
y_m(t_0) = y_m^{(0)}.
\]

We’ll use the convention that when a variable has both superscripts in parentheses and subscripts, the subscripts refer to components, and superscripts refer to timesteps. So, for example, \( y_i^{(0)} \) indicates the value of \( y_1 \) at the 0th timestep.

As an example, here’s a two-dimensional initial value problem:

\[
y'_1 = y_1 - 2y_2 + 4 \cos t - 2 \sin t, \quad y_1(0) = 1 \\
y'_2 = 3y_1 - 4y_2 + 5 \cos t - 5 \sin t, \quad y_2(0) = 2
\]
You should check that this has the solution
\[
\begin{align*}
\phi_1(t) & = \cos t + \sin t \\
\phi_2(t) & = 2 \cos t.
\end{align*}
\]

All of the methods we’ve discussed for the numerical solution of differential equations can be modified to solve systems of equations. For example, Euler’s method becomes
\[
\begin{align*}
y_1^{(i+1)} & = y_1^{(i)} + hf_1(t_i, y_1^{(i)}, y_2^{(i)}, \ldots, y_m^{(i)}) \\
y_2^{(i+1)} & = y_2^{(i)} + hf_2(t_i, y_1^{(i)}, y_2^{(i)}, \ldots, y_m^{(i)}) \\
& \quad \vdots \\
y_m^{(i+1)} & = y_m^{(i)} + hf_m(t_i, y_1^{(i)}, y_2^{(i)}, \ldots, y_m^{(i)}).
\end{align*}
\]

Similarly, if we define
\[
z_j^{(i)} = y_j^{(i)} + hf_j(t_i, y_1^{(i)}, y_2^{(i)}, \ldots, y_m^{(i)}),
\]
for \( j = 1, 2, \ldots, m \), our second-order Runge-Kutta method becomes
\[
\begin{align*}
y_1^{(i+1)} & = y_1^{(i)} + \frac{h}{2}[f_1(t_i, y_1^{(i)}, y_2^{(i)}, \ldots, y_m^{(i)}) + f_1(t_{i+1}, z_1^{(i)}, z_2^{(i)}, \ldots, z_m^{(i)})] \\
y_2^{(i+1)} & = y_2^{(i)} + \frac{h}{2}[f_2(t_i, y_1^{(i)}, y_2^{(i)}, \ldots, y_m^{(i)}) + f_2(t_{i+1}, z_1^{(i)}, z_2^{(i)}, \ldots, z_m^{(i)})] \\
& \quad \vdots \\
y_m^{(i+1)} & = y_m^{(i)} + \frac{h}{2}[f_m(t_i, y_1^{(i)}, y_2^{(i)}, \ldots, y_m^{(i)}) + f_2(t_{i+1}, z_1^{(i)}, z_2^{(i)}, \ldots, z_m^{(i)})].
\end{align*}
\]

For this week, you should write programs that implement Euler’s method and the second-order Runge-Kutta method for systems of equations. The programs should accept the following data as input: the dimension of the system \( m \), the limits of integration \( (a \text{ and } b) \), the initial value of \( y_i \) \( (y_1^{(0)}, y_2^{(0)}, \ldots, y_m^{(0)}) \), the number of timesteps \( n \), and the frequency at which data should be printed.

Output should include values of \( t \), values of the exact solution, values of \( y_1^{(i)}, y_2^{(i)}, \ldots, y_m^{(i)} \), the error and the relative error.

The error and relative error are most easily defined using vector subtraction and norms. If
\[
x = (x_1, x_2, \ldots, x_m)
\]
and
\[ \mathbf{y} = (y_1, y_2, \ldots, y_m), \]
then define
\[ \mathbf{x} - \mathbf{y} = (x_1 - y_1, x_2 - y_2, \ldots, x_m - y_m). \]
Also define
\[ \| \mathbf{x} \| = \sqrt{x_1^2 + x_2^2 + \cdots + x_m^2}. \]
Then if the exact solution is
\[ \phi(t_i) = (\phi_1(t_i), \phi_2(t_i), \ldots, \phi_m(t_i)), \]
and the numerical solution is
\[ \mathbf{y}^{(i)} = (y_1^{(i)}, y_2^{(i)}, \ldots, y_m^{(i)}), \]
the error is defined to be
\[ \| \phi(t_i) - \mathbf{y}^{(i)} \|, \]
and the relative error is defined to be
\[ \frac{\| \phi(t_i) - \mathbf{y}^{(i)} \|}{\| \phi(t_i) \|}. \]

Although \((f_1, f_2, \ldots, f_m)\) and the exact solution should be “hardwired,”
the rest of your programs should be general in the sense that they could
be used for any system of any dimension with changes needed only in the
definitions of \((f_1, f_2, \ldots, f_m)\) and the exact solution. In order to do this \(\mathbf{y}^{(i)}\)
and \(\phi(t_i)\) should be stored in dynamically allocated arrays.

C doesn’t have a `new` operator. So in order to dynamically allocate an
array, you should use the `malloc` function. For example, to dynamically
allocate an array that can store \(m\) floats, you might use the following code

```c
float *x;
int m;

scanf("%d", &m); /* Initialize m */
x = (float *) malloc(m*sizeof(float));
```

Now \(x\) can be used as an ordinary array. For example,
for (i = 0; i < m; i++)
x[i] = 0.0;

C, like C++, doesn’t use garbage collection. So when you’re through using an array, you should free the storage. In C, this is done with the free function:

    free(x);

Both malloc and free are defined in the stdlib.h header file. So your program should contain the line

    #include <stdlib.h>

Test your programs with the above initial value problem. Also test them with the following initial value problem.

\[
\begin{align*}
y_1' &= y_1 - 2y_2 - 2e^{-t} + 2, \quad y_1(0) = 1 \\
y_2' &= 2y_1 - y_2 - 2e^{-t} + 1, \quad y_2(0) = 1.
\end{align*}
\]

The exact solution to this problem is

\[
\phi_1(t) = e^{-t}, \quad \phi_2(t) = 1.
\]