From Differential Equations to Algorithms

We have given brief treatment to the problem of how to solve ordinary differential equations in Matlab. We now return to that problem, discussing the underlying algorithms. We hope to gain insight into success or failure, numerical efficiency, and appropriateness of various algorithms for solving ODEs.

The focus for numerical treatment of ODEs can be reduced to the following initial value problem with the from

\[
\frac{dx}{dt} = f(t, x), \quad x(t_0) = x_0. \tag{1} \]

The problem in numerics is to get from \( x(t) \) to \( x(t + \Delta t) \). We assume that \( f(t, x) \) is well known, but not necessarily easily differentiated. If it is easy to differentiate, then why not just use a Taylor’s series? We will develop methods that allow for estimating \( x(t + \Delta t) \) without explicit differentiation.

An Intuitive Approach

The Euler Algorithm

- **Strategy:** convert the differential equation to a difference equation work forward algebraically to find \( x(t + \Delta t) \).

- Recall from calculus that \( dx \) and \( dt \) are infinitesimal quantities. This is pure abstraction.

- Convert to the tangible, and computable \( \Delta x \), a small number of known value.

- To demonstrate, return to problem we focused on

\[
\frac{dx}{dt} = f(t, x). \tag{3} \]

- Suppose that we are at the initial point \((t_0, x_0)\).
Now, we would like to know where a second point \((t_1, x_1)\) lies.

- Assume that \(t_1\) is given by some small distance, \(\Delta t\), from \(t_0\):
  \[
  t_1 = t_0 + \Delta t
  \]

- Likewise, \(x_1\) is given by some small distance, \(\Delta x\), from \(x_0\):
  \[
  x_1 = x_0 + \Delta x
  \]

- Assume that \(\Delta t\) is specified by the algorithm, we need to develop an expression for \(\Delta x\)

- Use 3 to find \(\Delta x\)
  \[
  \frac{\Delta x}{\Delta t} = f(t, x) \quad (6)
  \]
  \[
  \Delta x = f(t, x)\Delta t \quad (7)
  \]

- And, including the initial values gives
  \[
  x_1 = x_0 + f(t_0, x_0)\Delta t \quad (8)
  \]

- Likewise, if we are interested in \(x_2\),
  \[
  x_2 = x_1 + f(t_1, x_1)\Delta t \quad (9)
  \]

- This is possible because \(x_1\) is now known.

- Generalizing to the recursion relation:
  \[
  x_{n+1} = x_n + f(t_n, x_n)\Delta t \quad (10)
  \]

- This is the basis of the Euler method.
  - It is critical that \(\Delta t\) be sufficiently small.
  - Later, formalism will be introduced for determining how small.
  - Notice that the rate of change is evaluated at the beginning of the interval.
Improvement – Euler-Richardson

The results of the previous analysis were unsatisfying. The method tended to over shoot or undershoot, because it is evaluating the slope at the beginning of the interval and racing forward.

**Big Idea:** why not try to evaluate the slope in the middle of the interval!

An algorithm is:

1. Find the point half way forward from the Euler (forward shooting) method:

   \[ t_{n+1/2} = t_n + \frac{\Delta t}{2} \]
   \[ x_{n+1/2} = f(t_n, x_n)\frac{\Delta t}{2} + x_n \]

2. Evaluate and save the slope at that midpoint

   \[ f_{\text{midpoint}} = f(t_{n+1/2}, x_{n+1/2}) \]
3. Use the slope at the midpoint to move forward an entire interval (time step) from the initial value

\[ x_{n+1} = f_{\text{midpoint}} \Delta t + x_n \]  

(14)

4. Repeat steps 1–3 for desired time period

---

Figure 2: Graphical representation of Euler-Richardson method. Derivatives at the starting point of each interval (solid circles) are used to extrapolate that value at the halfway point (open circles). The midpoint slope is used in order to interpolate to the next point (dotted line, note that it is parallel to points 2, 4 and 6).

---

Runge Kutta Order 2 and Error Analysis

Introduce a pair of function evaluations specified as

\[ F_1 = hf(t, x) \]  

(15)

\[ F_2 = hf(t + \alpha h, x + \beta F_1) \]  

(16)
Now, accomplish the basic task of getting from \( x(t) \) \( x(t+h) \) with

\[
x(t + h) = x(t) + w_1 F_1 + w_2 F_2 \quad (17)
\]

Writing out the \( F_1 \) and \( F_2 \) gives

\[
x(t + h) = x(t) + w_1 hf(t, x) + w_2 hf(t + \alpha h, x + \beta hf) \quad (18)
\]

Now, goal is to find \( \alpha, \beta, w_1, w_2 \) such that the equation is as accurate as possible.

State another way, we want the Taylor series expansion for \( x(t+h) \) to contain as high order terms as possible.

\[
x(t + h) = x(t) + hx'(t) + \frac{1}{2!} h^2 x''(t) + \frac{1}{3!} h^3 x'''(t) + ... \quad (19)
\]

Examination of the previous two equations reveals that the choice of \( w_1 = 1 \) and \( w_2 = 0 \) yield an approximation good through order 2 (remember \( x' = f' \)). As a matter of fact, this is our Euler method! A better choice of \( w_2 \) will improve the order of approximation. Let’s start by expanding the last term of equation 18 with the Taylor’s expansion of two variables.

\[
f(t + \alpha h, x + \beta hf) = f + \alpha hf_t + \beta hf f_x + \frac{1}{2!} \left( \alpha h \frac{\partial}{\partial t} + \beta hf \frac{\partial}{\partial x} \right)^2 f(\bar{x}, \bar{y}) \quad (20)
\]

where \( f_x \) is introduced to mean \( \frac{\partial f}{\partial x} \).

Inserting into equation 18 and collecting in terms of \( h \) yields

\[
x(t + h) = x(t) + (w_1 + w_2) hf + \alpha w_2 h^2 f_t + \beta w_2 h^2 f f_x + O(h^3) \quad (21)
\]

Return to equation 19 and observe that because of the chain rule

\[
x'' = \frac{dx'}{dt} = \frac{df}{dt} = \frac{\partial f}{\partial t} \frac{dt}{dt} + \frac{\partial f}{\partial x} \frac{dx}{dt} = f_t + f_x f \quad (22)
\]

which means that, based on equation 19

\[
x(t + h) = x + hf + \frac{1}{2} h^2 f_t + \frac{1}{2} h^2 f f_x + O(h^3) \quad (23)
\]

by comparing this equation with equation 21 we can place conditions on the unknown \( \alpha, \beta, w_1, w_2. \)

\[
w_1 + w_2 = 1 \quad (24)
\]

\[
\alpha w_2 = \frac{1}{2} \quad (25)
\]

\[
\beta w_2 = \frac{1}{2} \quad (26)
\]
This is solved with the choices (among others, the system is under-determined) \( w_1 = \frac{1}{2}, \ w_2 = \frac{1}{2}, \ \alpha = 1, \ \beta = 1 \). This choice allows us to finally write the Runge Kutta 2nd order equations as:

\[
x(t + h) = x(t) + \frac{h}{2} f(t, x) + \frac{h}{2} f(t + h, x + hf(t, x))
\] (27)

or, equivalently

\[
x(t + h) = x(t) + \frac{1}{2} (F_1 + F_2)
\] (28)

with

\[
F_1 = hf(t, x) \quad (29)
\]

\[
F_2 = hf(t + h, x + F_1) \quad (30)
\]

**Variable Time Stepping**

So far, we have only considered the possibility of having methods that use a constant value of \( \Delta t \). Compare this idea to that of driving across the country at a constant speed. You drive across the plains at the same speed that you drive across the mountains, as you drive through the cities. Indeed, you must select one speed to get it right, and that is the lowest speed.

Much better if we could develop algorithms that use a very large time step to traverse the dull country side, and then tip-toes through jagged and complicated terrain.

**Embedded Time Stepping**

Fehlberg discovered that it is possible to find coefficients for a Runge-Kutta process, such that the 4\(^{th}\) and 5\(^{th}\) order evaluations share the same coefficients. Hence, to improve the order of the method by 1, a single extra function evaluation is required. Now, error estimates are not provided by the differences between two evaluations, but rather, the difference in 4\(^{th}\) and 5\(^{th}\) order evaluations.

Fehlberg order 4\(^{th}\) is:

\[
x(t + h) = x(t) + \frac{25}{216} F_1 + \frac{1408}{2565} F_3 + \frac{2197}{4104} - \frac{1}{5} F_5
\] (31)

where

\[
F_1 = hf(t, x)
\] (32)
\( F_2 = h f(t + \frac{1}{4}h, x + \frac{1}{4}F_1) \) \tag{33}

\( F_3 = h f(t + \frac{3}{8}h, x + \frac{3}{32}F_1 + \frac{9}{32}F_2) \) \tag{34}

\( F_4 = h f(t + \frac{12}{13}h, x + \frac{1932}{2197}F_1 - \frac{7200}{2197}F_2 + \frac{7296}{2197}F_3) \) \tag{35}

\( F_5 = h f(t + h, x + \frac{439}{216}F_1 - 8F_2 + \frac{3680}{513}F_3 - \frac{845}{4104}F_4) \) \tag{36}

This in itself is no better than the Runge-Kutta order 4 that we have already studied. However, an additional function evaluation can take us to order 5.

\( F_6 = h f(t + \frac{1}{2}h, x - \frac{8}{27}F_1 + 2F_2 - \frac{3544}{2565}F_3 + \frac{1859}{4104}F_4 - \frac{11}{40}F_5) \) \tag{38}

Now Runge-Kutta order 5 is

\[ x(t + h) = x(t) + \frac{16}{135}F_1 + \frac{6656}{12825}F_3 + \frac{28561}{56430}F_5 - \frac{9}{50}F_5 + \frac{2}{55}F_6 \] \tag{39}

Now, suppose we have \( x_4 \) and \( x_5 \), corresponding to the 4th and 5th order Runge-Kutta evaluations above. Describe the error, \( \epsilon \), as

\[ \epsilon = |x_4 - x_5| \] \tag{40}

and return to the time stepping algorithm outlined in the previous section. We can also improve the time stepping scheme as

\[ h_0 = \left( \frac{\epsilon_0}{\epsilon_1} \right)^{1/5} h_1 \] \tag{41}

where \( \epsilon_0 \) is a desired accuracy. In this scheme, if \( h_0 > h_1 \) the time step is increased to \( h_1 \) for the next step. If \( h_0 < h_1 \) the method fails and must repeat. Note that for systems of ODEs, \( \epsilon_0 \) is a vector of desired accuracies, and as such the weakest length controls the chain.

### Beyond Runge Kutta

There are two techniques that are commonly used. One is the class of methods referred to as *predictor-corrector* methods. The other are those derived from the Bulirsch-Stoer method and are sometimes called backwards differentiation schemes.
**Predictor Corrector Methods**

The basic premise of these methods is to extend the dependence. Prior to this point, we have seen methods such that \( x(t + h) = f(x(t)) \). In plain English, the future depends upon the present. We can make some headway if we allow methods like \( x(t + h) = f(x(t), x(t - h), x(t - 2h), ...) \). Which is to say, the future depends on both the present and the past.

These so called *multi-step methods* have an obvious draw back that if we have an initial value \( x(t_0) = a \), how do we find \( x(t_0 - h) \), etc.? That is what is the 'pre-history'. Well, we can not know that, but we can use a technique like Runge Kutta to find \( x(t_c) \), with \( t_c \) such that we now can know \( x(t_c - h), x(t_c - 2h), ... \) for our multi-step method. This, in conjunction with the method below is called the *Adams-Moulton* method.

Adams-Bashforth-Moulton is one such predictor corrector or multi step method.

\[
x(t + h) = x(t) + \frac{h}{24}[55f(x(t)) - 59f(x(t - h)) + 37f(x(t - 2h)) - 9f(x(t - 3h))] \tag{42}
\]

Notice that if a systematic storage method is used, the \( f(x - nh) \) need only be computed once for each step once initialization is done. This is a huge improvement over Runge Kutta (4 evaluations).

In practice, equation 42 is rarely used by itself. It is referred to as the 'predictor'. The equation for the corrector is

\[
x(t + h) = x(t) + \frac{h}{24}[9(\hat{x}(t + h) + 19f(x(t)) - 5f(x(t - h)) + f(x(t - 2h))] \tag{43}
\]

where \( \hat{x}(t + h) \) in the above formula is computed in the predictor step with equation 42. It would be possible to iterate the previous equation, and one might think that the accuracy would increase. However, research has demonstrated that a single application is as good as anything (the order of the error is the same, regardless of the coefficient).

Finally, the method can easily be made adaptive by comparing \( x(t + h) \) computed in each of the predictor and corrector equations. This difference would then be the basis for changing the time step.

**Stiff Equations**

It is possible to have a system of equations where there are very different time scales that the solution is operating on. For example, consider
\[ u' = 998u + 1998v \quad (44) \]
\[ v' = -999u - 1999v \quad (45) \]

subsection to initial conditions \(u(0) = 1\) and \(v(0) = 0\). This has solutions,
\[ u = 2e^{-t} - e^{-1000t} \quad (46) \]
\[ v = -e^{-t} + e^{-1000t}. \quad (47) \]

Now, \(e^{-1000t}\) is small. Super small. But, the ODE integrators discussed so far would adapt the time step to accommodate. This means that the time step becomes so small, the problem take a very long time (sometimes for ever) to solve.

For example, coding this exact system into Matlab, and integrating over a 10 unit interval (initial time =0, final time =10) requires \(\sim 20,000\) function evaluations.

The solution to this problem is to use an \textit{implicit} integration scheme. Recall that in all cases so far we have used some version of the old Euler standby
\[ x_{n+1} = f(t_n, x_n)h + x_n, \quad (48) \]

at least to get the first point, that is later averaged. Implicit schemes involve setting up an linear system and solving for \(x_{n+1}\) by using a form like
\[ x_{n+1} = f(t_n, x_{n+1})h + x_n. \quad (49) \]

Note that I have put what we don’t know \((x_{n+1})\) in the function.

Let us consider an example. Our old friend, exponential decay is
\[ x' = -cx. \quad (50) \]

An Euler scheme gives
\[ x_{n+1} = x_n + hx_n' = (1 - ch)x_n. \quad (51) \]

Note that in this scheme, as \(h\) grows, \(x_n\) goes to \(-\infty\), it is unstable.

In implicit terms, this becomes
\[ x_{n+1} = x_n + hx_{n+1}' = x_n - hcx_{n+1} \quad (52) \]
Which can be solved algebraically, giving
\[ x_{n+1} = \frac{x_n}{1 + ch}. \quad (53) \]
Which is stable for large $h$. Note also that this method is only possible if the ODEs are linear.

Using Matlab’s stiff solver on the example above reduces the function evaluations to $\sim 300$ and produces a speed up of over 10.