Numerical Solution of Ordinary Differential Equations

James Keesling

1 Picard Iteration

The Picard method is a way of approximating solutions of ordinary differential equations. Originally it was a way of proving the existence of solutions. It is only through the use of advanced symbolic computing that it has become a practical way of approximating solutions.

In this chapter we outline some of the numerical methods used to approximate solutions of ordinary differential equations. Here is a reminder of the form of a differential equation.

$$\begin{aligned} \frac{dx}{dt} &= f(t, x0) \\ x(t_0) &= x_0 \end{aligned}$$

The first step is to transform the differential equation and its initial condition into an integral equation.

$$x(t) = x_0 + \int_0^t f(\tau, x(\tau))\, d\tau$$

We think of the right hand side of this equation as an operator $\mathcal{G}(x(t)) = \int_0^t f(\tau, x(\tau))\, d\tau$. The problem then is to find a fixed point, actually a fixed function, for $\mathcal{G}(x(t))$. Picard iteration gives a sequence of functions that converges to such a fixed function. We define $x_0(t) \equiv x_0$ and define the sequence inductively by

$$\begin{aligned} x_{n+1} &= \mathcal{G}(x_n(t)) \\ &= x_0 + \int_0^t f(\tau, x_n(\tau))\, d\tau. \end{aligned}$$

In another chapter we describe this approach in more detail and show the convergence to a solution of the initial differential equation.

2 Numerical Methods

Numerical solutions of differential equations calculate estimates of the solution at a sequence of node points $\{t_0, t_1, t_2, \ldots, t_n\}$. The initial value is given in the initial conditions x_0 at t_0. The estimates for the other values $\{x_1, x_2, \ldots, x_n\}$ are based on estimating the integral

$$\int_{t_n}^{t_{n+1}} f(\tau, x(\tau))\, d\tau.$$

We now give a few methods without explaining the derivation. In the following the n^{th}–step size is given by $h_n = t_{n+1} - t_n$.

Euler

$$x_{n+1} = x_n + h_n \cdot f(t_n, x_n)$$

Modified Euler

$$x_{n+1} = x_n + h_n \cdot f\left(t_n + \frac{h_n}{2}, x_n + \frac{h_n}{2} f(t_n, x_n)\right)$$

Heun

$$x_{n+1} = x_n + \frac{h_n}{2} \cdot (f(t_n, x_n) + f(t_{n+1}, x_n + h_n \cdot f(x_n, t_n)))$$

Runge-Kutta

$$x_{n+1} = x_n + \frac{h_n}{6}\left[k_1 + 2k_2 + 2k_3 + k_4\right]$$

where

$$\begin{aligned}
k_1 &= f(t_n, x_n) \\
k_2 &= f\left(t_n + \frac{h_n}{2}, x_n + \frac{h_n}{2} k_1\right) \\
k_3 &= f\left(t_n + \frac{h_n}{2}, x_n + \frac{h_n}{2} k_2\right) \\
k_4 &= f(t_n + h_n, x_n + h_n k_3)
\end{aligned}$$

These methods are based on numerical estimates of the integral $\int_{t_n}^{t_n+h_n} f(\tau, x(\tau))\, d\tau$ using various Newton-Cotes formulas. The *Euler* method just takes the left value of the function times the length of the interval as a crude estimate. Euler originally thought of this as following the tangent line to estimate the next x_{n+1}. The *Modified Euler* method is based on the Midpoint Rule for estimating the integral. It is the simplest of the Newton-Cotes open formulas. The *Heun* method is based on the Trapezoidal Rule of estimating the integral. It uses the two end points and is the simplest of the Newton-Cotes closed formulas. *Runge-Kutta* is based on Simpson's Rule, another Newton-Cotes closed formula. Of course, estimates have to be made of the points where the function $f(t,x)$ is unknown. These estimates have to be accurate enough that the accuracy of the estimate of the integral is not adversely affected. All of these methods and more like them are generally termed *Runge-Kutta Methods*. The one that we have designated Runge-Kutta is the standard one by that name. It is also sometimes called *Fourth-Order Runge-Kutta*.

3 Taylor Method

3.1 The Basic Theory

In this method, the Taylor series is used to estimate solutions. The Euler Method can be considered a Taylor Method of order one. One uses the Taylor series up to the given order. In the implementation, x_{n+1} is the left hand side of the equation. The right hand side will be a function of t_n, x_n, and h_n. We do an example below to illustrate.

$$x(t+h) = x(t) + \frac{dx}{dt}h + \frac{d^2x}{dt^2}\frac{h^2}{2} + \frac{d^3x}{dt^3}\frac{h^3}{3!} + \cdots$$

One uses the fact that $\frac{dx}{dt} = f(t,x)$ to obtain a formula for the various derivatives. The Chain Rule for functions of several variables is used. By this means one gets the following formula.

$$x(t+h) = x(t) + h \cdot f(t,x) + \frac{h^2}{2}\left[\frac{\partial f(t,x)}{\partial t} + \frac{\partial f(t,x)}{\partial x} \cdot f(t,x)\right] + \cdots$$

Of course, one would terminate this series at some n. That n would be the degree of the method. The local error would be $O(h^{n+1})$ and the global error would be $O(h^n)$ assuming a uniform step size of h.

Example 3.1. *Find the formula for the Taylor Method of third order for the following differential equation.*

$$\frac{dx}{dt} = xt^2 \qquad x(0) = 1$$

The second and third derivatives become the following.

$$\begin{aligned}\frac{d^2x}{dt^2} &= xt^4 + 2xt \\ \frac{d^3x}{dt^3} &= (t^4 + 2t)xt^2 + 4xt^3 + 2x\end{aligned}$$

So, we get the following inductive formula for computing the successive x_n's.

$$x_{n+1} = x_n + h_n x_n t_n^2 + \frac{h_n^2}{2}(x_n t_n^4 + 2x_n t_n) + \frac{h^3}{3!}((t_n^4 + 2t_n)x_n t_n^2 + 4x_n t_n^3 + 2x_n)$$

3.2 TI-89 Program for the Taylor Method

Here is a program for the Taylor Method. The variable f is the function assumed to be a function of t and x. The variables a and b are the initial conditions, t_0 and x_0, respectively. The variable k is the degree of the Taylor Method. The variable h is the step size and n is the number of steps. The output variables are $coef$, the Taylor formula for x_{n+1} and $soln$, the numerical solution of the differential equation.

```
:taymeth(f,a,b,k,h,n)
:Prgm
:newMat(1,k+1) → coef
:newMat(n+1,2) → soln
:approx(a) → a
:approx(b) → b
:approx(h) → h
:x → coef[1,1]
:a → soln[1,1]
:b → soln[1,2]
:f → coef[1,2]
:For i,2,k
:coef[1,i]|x=y(z) → g
:g|y(z)=x(t) → g
:d(g,t) → g
:g|d(x(t),t)=f → g
:g|x(t)=x → coef[1,i+1]
:EndFor
:For i,1,n
:b → temp
:For j,1,k
:temp + h∧j/(j!)*coef[1,j+1]|x=b and t=a → temp
:EndFor
:temp → b
:a+h → a
:a → soln[i+1,1]
:b → soln[i+1,2]
:EndFor
```

```
:EndPrgm
```

For the example done below with the Euler Method, we get the following output for the Taylor Method of order $k = 3$, $h = .1$ and $n = 10$.

$$\begin{array}{ll}
t_0 = 0.0 & x_0 = 1.0000000000000 \\
t_1 = 0.1 & x_1 = 1.0050000000000 \\
t_2 = 0.2 & x_2 = 1.0201756675000 \\
t_3 = 0.3 & x_3 = 1.0459874721220 \\
t_4 = 0.4 & x_4 = 1.0832293330732 \\
t_5 = 0.5 & x_5 = 1.1330694368407 \\
t_6 = 0.6 & x_6 = 1.1971114656355 \\
t_7 = 0.7 & x_7 = 1.2774807409924 \\
t_8 = 0.8 & x_8 = 1.3769417719573 \\
t_9 = 0.9 & x_9 = 1.4990563119840 \\
t_{10} = 1.0 & x_{10} = 1.6483945503684
\end{array}$$

The formula for each step can be determined from the *coef* output.

$$x_{n+1} = x_n + h \cdot t_n \cdot x_n + \frac{h^2}{2} \cdot (t_n^2 \cdot x_n + x_n) + \frac{h^3}{3!} \cdot (t_n \cdot (t_n^2 + 3) \cdot x_n)$$

4 Euler, Modified Euler, Heun, and Runge-Kutta

In this section all programs have the same input and the same output variables. This description applies to all the programs. The variable f is the right hand side of the original differential equation assuming the variables t and x. The variables a and b are the initial values t_0 and x_0, respectively. The variable h is the step size for the node points. The variable n is the number of iterations. The numerical solution is over the interval $[t_0, t_0 + n \cdot h]$. The output is the variable *soln* which is an $n + 1 \times 2$ matrix with the t_i's in the first column and the x_i's in the second column.

The data for the Euler Method is given. However, data for the other methods is not given. Compare the output for those against the Taylor Method given in the previous section. The x_{10} value should be close to $\sqrt{e} = 1.6487212707$.

4.1 Euler

Here is the Euler program.

```
:euler(f,a,b,h,n)
:Prgm
:NewMat(n+1,2) → soln
:approx(a) → a
```

```
:approx(b) → b
:approx(h) → h
:a → soln[1,1]
:b → soln[1,2]
:For i,1,n
:b+h*f|t=a and x=b → b
:a + h → a
:a → soln[i+1,1]
:b → soln[i+1,2]
:EndFor
:EndPrgm
```

Example 4.1. *Use the Euler Method to solve the following differential equation over the interval* $[0, 1]$ *using* $h = .1$.

$$\frac{dx}{dt} = t \cdot x$$
$$x(0) = 1$$

The output of the program for this example is below.

$$\begin{aligned}
t_0 &= 0.0 & x_0 &= 1.0000000000000\\
t_1 &= 0.1 & x_1 &= 1.0000000000000\\
t_2 &= 0.2 & x_2 &= 1.0100000000000\\
t_3 &= 0.3 & x_3 &= 1.0302000000000\\
t_4 &= 0.4 & x_4 &= 1.0611060000000\\
t_5 &= 0.5 & x_5 &= 1.1035502400000\\
t_6 &= 0.6 & x_6 &= 1.1587277520000\\
t_7 &= 0.7 & x_7 &= 1.2282514171200\\
t_8 &= 0.8 & x_8 &= 1.3142290163184\\
t_9 &= 0.9 & x_9 &= 1.4193673376290\\
t_{10} &= 1.0 & x_{10} &= 1.5471103980101
\end{aligned}$$

4.2 Modified Euler

Here is the Modified Euler program.

```
:meuler(f,a,b,h,n)
:Prgm
:approx(a) → a
:approx(b) → b
:approx(h) → h
:newMat(n+1,2) → soln
```

```
:a → soln[1,1]
:b → soln[1,2]
:For i,1,n
:f|x=b and t=a → temp
:b+h*f|t=a+h/2 and x=b+h/2*temp → b
:a+h → a
:a → soln[i+1,1]
:b → soln[i+1,2]
:EndFor
:EndPrgm
```

4.3 Heun

Here is the Heun program.

```
:heun(f,a,b,h,n)
:Prgm
:approx(a) → a
:approx(b) → b
:approx(h) → h
:newMat(n+1,2) → soln
:a → soln[1,1]
:b → soln[1,2]
:For i,1,n
:f|x=b and t=a → temp
:b+h/2*(temp + f|t=a+h and x=b+h*temp) → b
:a+h → a
:a → soln[i+1,1]
:b → soln[i+1,2]
:EndFor
:EndPrgm
```

4.4 Runge-Kutta

Here is the Runge-Kutta program.

```
:rungekut(f,a,b,h,n)
:Prgm
:approx(a) → a
:approx(b) → b
:approx(h) → h
:newMat(n+1,2) → soln
```

```
:a → soln[1,1]
:b → soln[1,2]
:For i,1,n
:f|x=b and t=a → k1
:f|t=a+h/2 and x=b+h/2*k1 → k2
:f|t=a+h/2 and x=b+h/2*k2 → k3
:f|t=a+h and x=b+h*k3 → k4
:b+h/6*(k1 + 2*k2 + 2*k3 + k4) → b
:a+h → a
:a → soln[i+1,1]
:b → soln[i+1,2]
:EndFor
:EndPrgm
```