7.1 First-order initial-value problems

A first-order *initial-value problem* (IVP) is a first-order ordinary differential equation with a specified value:

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}y(t) &= f\left(t, y(t)\right)\\ y(a) &= y_0\end{aligned}$$

That is, this says that the function $y(t)$ satisfies the ode while simultaneously having a value of y_0 at time a.

For example, you have already seen that

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}y(t) &= \alpha y(t)\\ y(0) &= y_0\end{aligned}$$

has a solution $y(t) = y_0 e^{\alpha t}$, for the function satisfies the ODE:

$$\frac{\mathrm{d}}{\mathrm{d}t}y(t) = \frac{\mathrm{d}}{\mathrm{d}t}\left(y_0 e^{\alpha t}\right) = y_0 \frac{\mathrm{d}}{\mathrm{d}t}e^{\alpha t} = y_0 \alpha e^{\alpha t} = \alpha\left(y_0 e^{\alpha t}\right) = \alpha y(t)$$

and

$$y(0) = y_0 e^{\alpha \cdot 0} = y_0 \cdot 1 = y_0 .$$

In your calculus course, you will see that you can always find explicit solutions to such initial-value problems under a few very specific set of conditions. While it is fortunate that many simplified models of reality can in some cases be described by such simple differential equations; in general, however, we cannot find an explicit solution; for example, a variation of the *Van der Pol* equation is

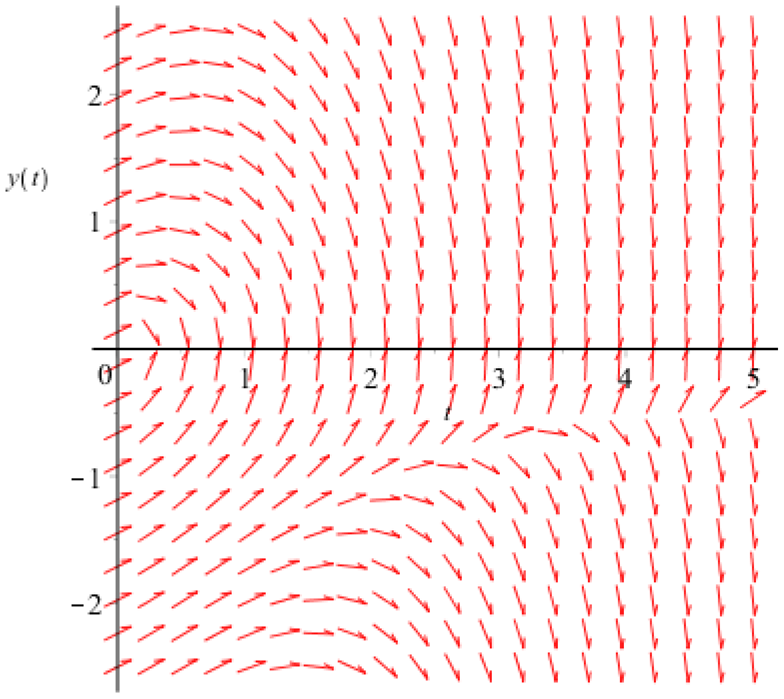
$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}y(t) &= -\frac{t}{y(t)} + \mu\left(1 - t^2\right)\\ y(0) &= y_0\end{aligned}$$

This function does not have an explicit solution that we can write down, so if we want to find such a solution, we must approximate it.

Interpreting IVPs

If we examine the ODE $\frac{\mathrm{d}}{\mathrm{d}t}y(t) = f\left(t, y(t)\right)$ we see that at every point (*t*, *y*), we can find a slope $f(t, y)$. Thus, if the solution passes through a point (*t*, *y*), it must do so with the slope given by the ODE. One thing that we can do is to evaluate the slope at a set of points.

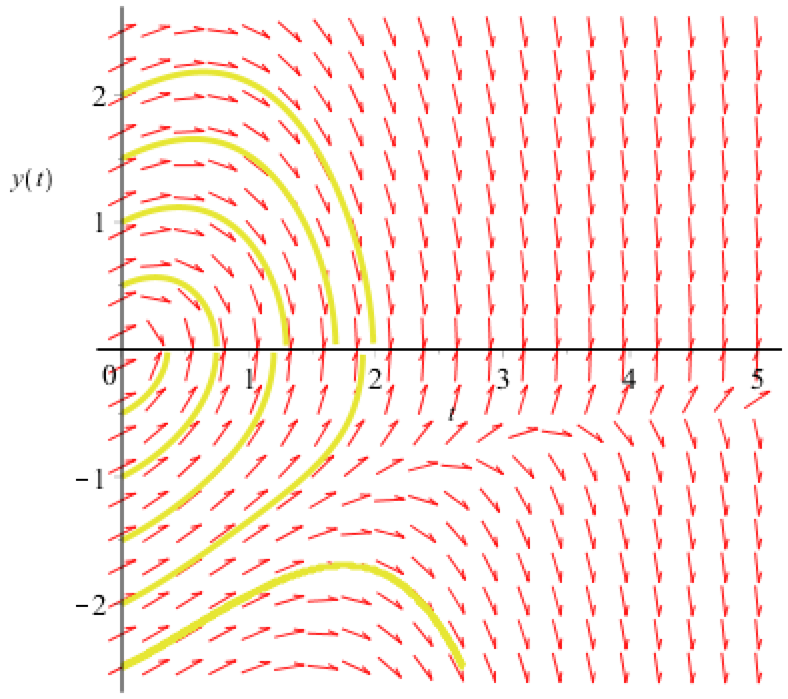
For example, if we consider the ode $\frac{\mathrm{d}}{\mathrm{d}t}y(t) = -\frac{t}{y(t)} + 0.5\left(1 - t^2\right)$ and plot a line equal to the slope $f(t, y) = -\frac{t}{y} + 0.5\left(1 - t^2\right)$ at a number of different points, we see



For example, if we consider those solutions the initial value of which at $t = 0$ is –2.5, –2, –1.5, –1, –0.5, 0.5, 1, 1.5 and 2, we see a few oddities:

1. most of the solutions appear to terminate when they reach a value of $y(t) = 0$, for some t, but
2. if the initial value is small enough, the solution appears to drop to $-\infty$.

Note that at any point where a solution is close to an arrow, the slope of that solution matches very close the slope of the arrow.



Euler's method

How can we approximate a solution? We have an initial value, let us say $y(a) = y_0$. Let us now consider the Tayler series at this point:

$$y(a+h) = y(a) + y^{(1)}(a)h + \frac{1}{2}y^{(2)}(\tau)h^2.$$

where τ is some value $a < \tau < a + h$.

From the differential equation, we have two of these:

$$y(a) = y_0$$
$$y^{(1)}(a) = f(a, y_0)$$

Thus, we have

$$y(a+h) = y_0 + f(a, y_0)h + \frac{1}{2}y^{(2)}(\tau)h^2 .$$

Therefore,

$$y(a+h) \approx y_0 + h \cdot f(a, y_0).$$

Thus, let us designate

$$y_1 = y_0 + h \cdot f(a, y_0)$$

and if we designate $t_k = a + kh$, so y_0 is the value at $t_0 = a$, and an approximation to the solution at $t_1 = a + h$ is $y_1 = y_0 + h \cdot f(t_0, y_0)$.

Now, if (t_1, y_1) is approximately on the solution to the initial value problem, then we can repeat this procedure:

$$y(t_1 + h) \approx y_1 + h \cdot f(t_1, y_1).$$

For example, if we consider the following time-independent non-linear first-order ode,

$$\frac{\mathrm{d}}{\mathrm{d}t} y(t) = y(t)\left(1 - y(t)\right)$$
$$y(0) = 0.1$$

it is possible to determine that the solution to this IVP is the function $y(t) = \frac{1}{1+9e^{-t}}$. You can confirm this by observing that $y(0) = \frac{1}{1+9e^{-0}} = \frac{1}{1+9\cdot 1} = \frac{1}{10} = 0.1$ and

$$\frac{\mathrm{d}}{\mathrm{d}t} y(t) = \frac{\mathrm{d}}{\mathrm{d}t}\frac{1}{1+9e^{-t}} = \frac{9e^{-t}}{\left(1+9e^{-t}\right)^2} = \frac{1}{1+9e^{-t}}\left(\frac{1+9e^{-t}-1}{1+9e^{-t}}\right) = \frac{1}{1+9e^{-t}}\left(1 - \frac{1}{1+9e^{-t}}\right) = y(t)\left(1 - y(t)\right).$$

Now, let us approximate $y(0.2)$, $y(0.4)$, …, $y(3.0)$ using our technique. First, define $t_k = 0 + k\,0.2$, and thus y_k will be our approximation of $y(t_k)$. In this case, the function f is defined as $f(t, y) = y(1 - y)$, so

$y_1 = y_0 + 0.2\ f(t_0, y_0) = 0.1 + 0.2\,(0.1(1 - 0.1)) = 0.118$
$y_2 = y_1 + 0.2\ f(t_1, y_1) = 0.118 + 0.2\,(0.118\,(1 - 0.118)) = 0.1388152$
$y_3 = y_2 + 0.2\ f(t_2, y_2) = 0.1388152 + 0.2\,(0.1388152\,(1 - 0.1388152)) = 0.162724308049792$
$y_4 = y_3 + 0.2\ f(t_3, y_3) = 0.1899733295736937$
$y_5 = y_4 + 0.2\ f(t_4, y_4) = 0.2207500222985694$
$y_6 = y_5 + 0.2\ f(t_5, y_5) = 0.2551539122893195$
$y_7 = y_6 + 0.2\ f(t_6, y_6) = 0.2931639909558742$
$y_8 = y_7 + 0.2\ f(t_7, y_7) = 0.3346077640284139$
$y_9 = y_8 + 0.2\ f(t_8, y_8) = 0.3791368456844777$
$y_{10} = y_9 + 0.2\ f(t_9, y_9) = 0.4262152652702582$
$y_{11} = y_{10} + 0.2\ f(t_{10}, y_{10}) = 0.4751264278544305$
$y_{12} = y_{11} + 0.2\ f(t_{11}, y_{11}) = 0.5250026889361743$
$y_{13} = y_{12} + 0.2\ f(t_{12}, y_{12}) = 0.5748776620453665$
$y_{14} = y_{13} + 0.2\ f(t_{13}, y_{13}) = 0.6237563291906905$
$y_{15} = y_{14} + 0.2\ f(t_{14}, y_{14}) = 0.6706932033877396$

To see how good our approximation is, $y(3) = \frac{1}{1+9e^{-3}} = 0.6905678577030156$, so we note the error is 0.01987.

Note that the error of one approximation is $\frac{1}{2}y^{(2)}(\tau)h^2$. Thus, reducing the step size should reduce the error by a factor of four:

$y_0 =$	0.1
$y_1 =$	0.109
$y_2 =$	0.1187119
	0.129173838479839
	0.140422634273061346
	0.152493046078748886
	0.165416937776386232
	0.179222355223693172
	0.193932525484869723
	0.209564795589262731
	0.226129534793152057
	0.243629031621890526
	0.262056424279177565
	0.281394709756496489
	0.301615882464251848
	0.322680256655208094
	0.344536027517221807
	0.367119122843209206
	0.390353390091813394
	0.414151152185377590
	0.438414149718267770
	0.463034868022775376
	0.487898225924566013
	0.512883580630988728
	0.537866981966001209
	0.562723591133679863
	0.587330166245209359
	0.611567510451567769
	0.635322779512731701
$y_{29} =$	0.658491554047226561
$y_{30} =$	0.680979596776796067

We observe that the error of y_{30} to $y(3)$ is 0.009588, so the error only drops by a factor of two, so actually it does: the previous y_1 = 0.118 as an approximation to $y(0.2)$ = 0.1194946317113934 has an error to the correct answer of 0.001495, while the current y_1 = 0.109 as an approximation to $y(0.1)$ = 0.1194946317113934 has an error of 0.0003669, or almost exactly one quarter the previous error. The issue is that the error is cumulative: when we are approximating y_2, we are using already the approximation of y_1, and so on and so forth. Consequently, the error accumulates:

$$\sum_{k=1}^{n}\frac{1}{2}y^{(2)}\left(\tau_{k-1}\right)h^2 = \frac{h^2}{2}\sum_{k=1}^{n}y^{(2)}\left(\tau_{k-1}\right).$$

.

You may already recognize the right-hand side, as we may now multiply by $1 = \frac{n}{n}$ to get

$$\sum_{k=1}^{n} \frac{1}{2} y^{(2)}\left(\tau_{k-1}\right) h^2 = \frac{nh^2}{2}\left(\frac{1}{n}\sum_{k=1}^{n} y^{(2)}\left(\tau_{k-1}\right)\right)$$

and the term in parentheses is the average of the second derivative sampled on the interval $0 < t < nh$, and we may also write $nh = t_n - t_0$, to get

$$\frac{h}{2}\left(t_n - t_0\right) y^{(2)}(\tau);$$

where $t_0 < \tau < t_n$. Incidentally, we may note that the second derivative on the interval [0, 3] spans the range [–0.09622, 0.08144] and thus the possible range of the error is [–0.02443, 0.02887] when h = 0.2, and [–0.01443, 0.01222], and both our errors, 0.001495 and 0.0003669 fall into these two intervals, respectively.

Integration, in disguise

When we are attempting to solve an IVP, we are actually attempting to integrate

$$y_0 + \int_{t_0}^{t_0+h} \frac{\mathrm{d}}{\mathrm{d}t} y(t)\,\mathrm{d}t .$$

Thus, one absolutely horrible approximation of an integral is to evaluate the integrand at one end-point and multiply by the width of the interval:

$$y_0 + \int_{t_0}^{t_0+h} \frac{\mathrm{d}}{\mathrm{d}t} y(t)\,\mathrm{d}t \approx y_0 + h\frac{\mathrm{d}}{\mathrm{d}t} y(t_0) = y_0 + hf(t_0, y_0).$$

We didn't even attempt this approximation of an integral in the previous chapters—we jumped immediately to the trapezoidal rule. If we were to attempt to approximate such an integral with such a poor approximation, we would have

$$\begin{aligned} f(x) &= f(a) + f^{(1)}(\alpha)(x-a) \\ \int_a^b f(x)\,\mathrm{d}x &= \int_a^b f(a)\,\mathrm{d}x + \int_a^b f^{(1)}(\alpha)(x-a)\,\mathrm{d}x \\ \int_a^b f(x)\,\mathrm{d}x &= f(a)(b-a) + f^{(1)}(\alpha)\frac{(b-a)^2}{2} \end{aligned}$$

Consequently, this is a horrible approximation of the integral, but it still works.

Implementation of Euler’s method

Here is a C++ implementation of Euler’s method. It returns a tuple of three vectors that contain

1. the time values $t_k = a + kh$,
2. the approximations y_k, and
3. the derivatives at the points (t_k, y_k).

As the user is creating the approximation, it is assumed that the first and third pieces of information are therefore also necessary.

```
#include <vector>
#include <tuple>
#include <cassert>
#include <iostream>
#include <cmath>

std::tuple< std::vector<double>, std::vector<double>, std::vector<double> >
  euler( double f(double, double),
         double t0, double y0, double tf,
         size_t n ) {
    assert( n > 0 );

    double h{(tf - t0)/n};

    std::vector<double>  t(n + 1);
    std::vector<double>  y(n + 1);
    std::vector<double> dy(n + 1);

    y[0] = y0;

    for ( size_t k{0}; k < n; ++k ) {
        t[k] = t0 + k*h;

        double s0{f( t[k], y[k] )};
        dy[k] = s0;

        y[k + 1] = y[k] + h*s0;
    }

     t[n] = tf;
    dy[n] = f( tf, y[n] );

    return std::make_tuple( t, y, dy );
}
```

As an example of how this code works:

```
#include <iostream>

// Approximate the solution to the differential equation D(y)(t) = -y(t)*t
double f( double t, double y ) {
    return -y*t;
}

int main() {
    std::size_t const N{20};
    double t0{0.3};
    double y0{2.7};

    auto result = euler( f, t0, y0, 0.9, N );

    // Print the t-values
    for ( std::size_t k{0}; k < N; ++k ) {
        std::cout << std::get<0>( result )[k] << ", ";
    }

    std::cout << std::get<0>( result )[N] << std::endl;

    // Print the approximations y[k] ~ y(t[k])
    for ( std::size_t k{0}; k < N; ++k ) {
        std::cout << std::get<1>( result )[k] << ", ";
    }

    std::cout << std::get<1>( result )[N] << std::endl;

    // Print the actual solution y( t[k] )
    for ( std::size_t k{0}; k < N; ++k ) {
        double t{std::get<0>( result )[k]};
        std::cout << y0*std::exp(0.5*t0*t0 - 0.5*t*t) << ", ";
    }

    {
        double t{std::get<0>( result )[N]};
        std::cout << y0*std::exp(0.5*t0*t0 - 0.5*t*t) << std::endl;
    }

    return 0;
}
```

Example of Euler's method

Suppose we want to approximate the solution to the initial-value problem

$$\begin{aligned} y^{(1)}(t) &= (t-1)y(t)+0.5 \\ y(0) &= 1.2 \end{aligned}$$

where we want to approximate $y(2)$ with h = 1.0, 0.5 and 0.25. Here,

$$f(t,y)=(t-1)y+0.5\,.$$

First, we note that this does have a solution:

$$y(t)=0.05\left(5\sqrt{2}\sqrt{\pi}\mathrm{e}^{0.5}\left(\operatorname{erf}\left(0.5\sqrt{2}\right)+\operatorname{erf}\left(0.5\sqrt{2}(t-1)\right)\right)+24\right)\mathrm{e}^{0.5t(t-2)}$$

This can be implemented as the C++ function

```
double y( double t ) {
    double const SQRT_PI{std::sqrt( std::acos( -1.0 ) )};
    double const ROOT_2_5{std::sqrt( 2.0 )*0.5};

    return 0.05*(10.0*ROOT_2_5*SQRT_PI*std::exp( 0.5 )*(
        std::erf( ROOT_2_5 ) + std::erf( ROOT_2_5*(t - 1) )
    ) + 24.0)*exp( 0.5*t*(t - 2.0) );
}
```

This solution evaluated at this point is $y(2.0)$ = 2.610686134642448.

Applying Euler's method with h = 1, we have

$$\begin{aligned} y_1 &= y_0 + h\cdot f(t_0,y_0) \\ &= 1.2+1\cdot f(0,1.2) \\ &= 1.2+1\cdot\left((0-1)\cdot 1.2+0.5\right) \\ &= 1.2-0.7 \\ &= 0.5 \end{aligned} \qquad \begin{aligned} y_2 &= y_1 + h\cdot f(t_1,y_1) \\ &= 0.5+1\cdot f(1.0,0.5) \\ &= 0.5+1\cdot\left((1.0-1)\cdot 0.5+0.5\right) \\ &= 0.5+0.5 \\ &= 1.0 \end{aligned}$$

Thus, we are not off to a very good start, with an error approximately equal to 1.611.

Applying Euler's method with h = 0.5, we have

$$\begin{aligned} y_1 &= y_0 + h \cdot f(t_0, y_0) \\ &= 1.2 + 0.5 \cdot f(0, 1.2) \\ &= 1.2 + 0.5 \cdot ((0-1) \cdot 1.2 + 0.5) \\ &= 1.2 - 0.35 \\ &= 0.85 \end{aligned} \qquad \begin{aligned} y_2 &= y_1 + h \cdot f(t_1, y_1) \\ &= 0.85 + 0.5 \cdot f(0.5, 0.85) \\ &= 0.85 + 0.5 \cdot ((0.5-1) \cdot 0.85 + 0.5) \\ &= 0.85 + 0.0375 \\ &= 0.8875 \end{aligned}$$

$$\begin{aligned} y_3 &= y_2 + h \cdot f(t_2, y_2) \\ &= 0.8875 + 0.5 \cdot f(1.0, 0.8875) \\ &= 0.8875 + 0.5 \cdot ((1.0-1) \cdot 0.8875 + 0.5) \\ &= 0.8875 + 0.25 \\ &= 1.1375 \end{aligned} \qquad \begin{aligned} y_4 &= y_3 + h \cdot f(t_3, y_3) \\ &= 1.1375 + 0.5 \cdot f(1.5, 1.1375) \\ &= 1.1375 + 0.5 \cdot ((1.5-1) \cdot 1.1375 + 0.5) \\ &= 1.1375 + 0.534375 \\ &= 1.671875 \end{aligned}$$

The error has now been reduced to 0.9388.

t	*h* = 0.5	*h* = 0.25	*h* = 0.125	*h* = 0.0625
Calls	4	8	16	32
0.0	1.2	1.2	1.2	1.2
0.0625				1.15625
0.125			1.1125	1.1197509765625
0.875				1.089764595031738
0.25		1.025	1.0533203125	1.065674986690283
0.3125				1.046971471689176
0.3750			1.017071533203125	1.033234416265032
0.4375				1.024123696879679
0.5	0.85	0.9578125	1.000112819671631	1.019369348161252
0.5625				1.018764056031213
0.625			1.000105768442154	1.022157226374110
0.6875				1.029450416380967
0.75		0.9630859374999999	1.015725810546428	1.040593962936026
0.8125				1.055584682265150
0.875			1.046484378966852	1.074464549269856
0.9375				1.097320294978685
1.0	0.8875	1.02789306640625	1.092633060545495	1.124283887576425
1.0625				1.155533887576425
1.125			1.155133060545495	1.191297691824770
1.875				1.231854705042151
1.25		1.15289306640625	1.235682014616518	1.277540502366864
1.3125				1.328752072716346
1.3750			1.336797077573284	1.385954261636587
1.4375				1.449687564643694
1.5	1.1375	1.349948883056641	1.461959440584532	1.520577458989421
1.5625				1.599345504582840
1.625			1.615831905621065	1.686822494978330
1.6875				1.783963998688421
1.75		1.643692493438721	1.804568773247711	1.891868701757065
1.8125				2.011800047151927
1.875			2.036247095739684	2.145211768296361
1.9375				2.293778036875068
2.0	1.671875	2.076884835958481	2.321461621836212	2.459429093723216

Recall that the exact solution is $y(2.0) = 2.610686134642448$, so the error when $h = 0.125$ is 0.2892, while the error when $h = 0.0625$ is 0.1513, which is approximately half the previous error.

n	*h*	Function calls	Approximation of *y*(2.0)	Absolute error
1	2.0	1	–0.2	2.811
2	1.0	2	1.0	1.611
4	0. 5	4	1.671875	0.9388
8	0. 25	8	2.076884835958481	0.5338
16	0.125	16	2.321461621836212	0.2892
32	0.0625	32	2.459429093723216	0.1513
64	0.03125	64	2.533236823913693	0.07745
128	0.015625	128	2.571484266405220	0.03920
256	0.0078125	256	2.590963000669263	0.01972
512	0.00390625	512	2.600793646021044	0.009892
1024	0.001953125	1024	2.605732112846550	0.004954

You will see that each time you double the number of steps, the error drops by approximately one half.

Estimating intermediate points

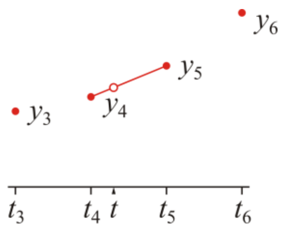
Suppose we have estimated values of the solution to an initial-value problem at the points t_0, …, t_n where $t_n = t_f$. Suppose now we want to estimate the value of the solution at a point between two of these approximations:

$$t_{k-1} < t < t_k .$$

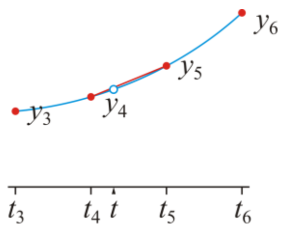
In this case, we could do a linear approximation, so for example, if we were to interpolate with a linear polynomial, by calculating

$$y_{k-1} + \frac{y_k - y_{k-1}}{h}\left(t - t_{k-1}\right),$$

so for example, if we were approximating a point between t_4 and t_5 in the graphic below, we would be returning the point shown here:



This, however, seems unsatisfying, as it seems the solution is clearly concave up at this point, so you would expect the solution to be somewhat lower than the value that appears on the interpolating line. Instead, we could find the interpolating cubic that passes through four points:



Finding the interpolating polynomial, however, is expensive, and there is actually more information available here than just the points: we have, after all, the fact that the solution is that of a differential equation. Thus, ideally, the interpolating polynomial should match both

$$\begin{gathered} p\left(t_{k-1}\right) = y_{k-1} \\ p\left(t_k\right) = y_k \\ p^{(1)}\left(t_{k-1}\right) = f\left(t_{k-1}, y_{k-1}\right) \\ p^{(1)}\left(t_k\right) = f\left(t_k, y_k\right) \end{gathered}$$

Now, if that polynomial is of the form $p(t) = at^3 + bt^2 + ct + d$, then this requires us to solve the following system of linear equations:

$$\begin{pmatrix} t_{k-1}^3 & t_{k-1}^2 & t_{k-1} & 1 \\ t_k^3 & t_k^2 & t_k & 1 \\ 3t_{k-1}^2 & 2t_{k-1} & 1 & 0 \\ 3t_{k-1}^2 & 2t_k & 1 & 0 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} y_{k-1} \\ y_k \\ f\left(t_{k-1}, y_{k-1}\right) \\ f\left(t_k, y_k\right) \end{pmatrix}$$

We could perform Gaussian elimination on this, but this leads to serious issues, as we must solve this system of linear equations for every new point. Instead, let us find the value at $y(t_{k-1} + \alpha(t_k - t_{k-1}))$ let us make the observation that we can find an interpolating polynomial $p(t) = a\alpha^3 + b\alpha^2 + c\alpha + d$ that passes through the points $(0, y_{k-1})$ and $(1, y_k)$ with slopes s_{k-1} and s_k:

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} y_{k-1} \\ y_k \\ s_{k-1} \\ s_k \end{pmatrix}$$

Performing Gaussian elimination, we have:

$$\left(\begin{array}{cccc:c} 0 & 0 & 0 & 1 & y_{k-1} \\ 1 & 1 & 1 & 1 & y_k \\ 0 & 0 & 1 & 0 & s_{k-1} \\ 3 & 2 & 1 & 0 & s_k \end{array}\right) \sim \left(\begin{array}{cccc:c} 1 & 1 & 1 & 1 & y_k \\ 3 & 2 & 1 & 0 & s_k \\ 0 & 0 & 1 & 0 & s_{k-1} \\ 0 & 0 & 0 & 1 & y_{k-1} \end{array}\right)$$

$$\sim \left(\begin{array}{cccc:c} 1 & 1 & 1 & 1 & y_k \\ 0 & -1 & -2 & -3 & s_k - 3y_k \\ 0 & 0 & 1 & 0 & s_{k-1} \\ 0 & 0 & 0 & 1 & y_{k-1} \end{array}\right)$$

$$\sim \left(\begin{array}{cccc:c} 1 & 1 & 1 & 1 & y_k \\ 0 & 1 & 2 & 3 & 3y_k - s_k \\ 0 & 0 & 1 & 0 & s_{k-1} \\ 0 & 0 & 0 & 1 & y_{k-1} \end{array}\right)$$

$$\sim \left(\begin{array}{cccc:c} 1 & 0 & 0 & 0 & -2y_k + 2y_{k-1} + s_k + s_{k-1} \\ 0 & 1 & 0 & 0 & 3y_k - 3y_{k-1} - s_k - 2s_{k-1} \\ 0 & 0 & 1 & 0 & s_{k-1} \\ 0 & 0 & 0 & 1 & y_{k-1} \end{array}\right)$$

Now, the slopes assume that the width between the points is 1 – 0 = 1; however, the points t_{k-1} and t_k may be closer or further apart. Therefore, we must adjust the slopes to the new width; specifically, we must multiply each of the slopes by $\Delta t_k = t_k - t_{k-1}$. Thus, $s_{k-1} = \left(t_k - t_{k-1}\right) f\left(t_{k-1}, y_{k-1}\right)$ and $s_k = \left(t_k - t_{k-1}\right) f\left(t_k, y_k\right)$. Therefore, the interpolating polynomial is

$$\left(-2y_k + 2y_{k-1} + s_k + s_{k-1}\right)\alpha^3 + \left(3y_k - 3y_{k-1} - s_k - 2s_{k-1}\right)\alpha^2 + s_{k-1}\alpha + y_{k-1},$$

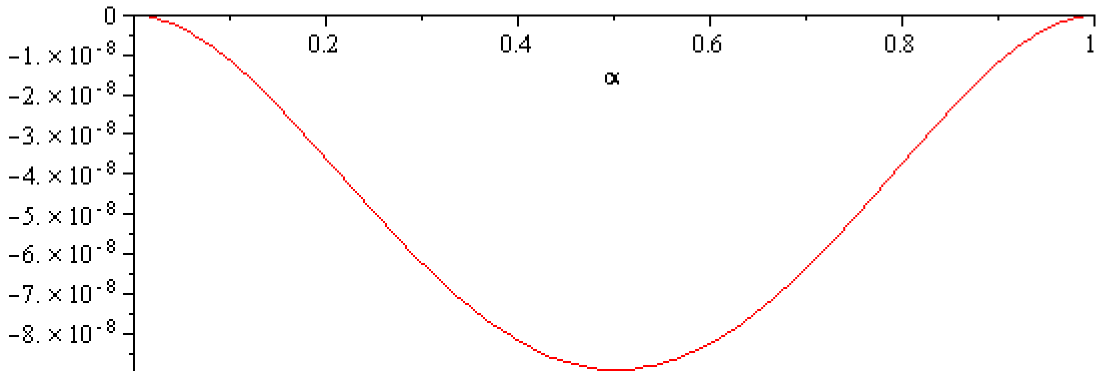
and you will note that evaluated at $\alpha = 0$, this gives y_{k-1} while evaluated at $\alpha = 1$, this gives y_k.

Example of estimating intermediate points

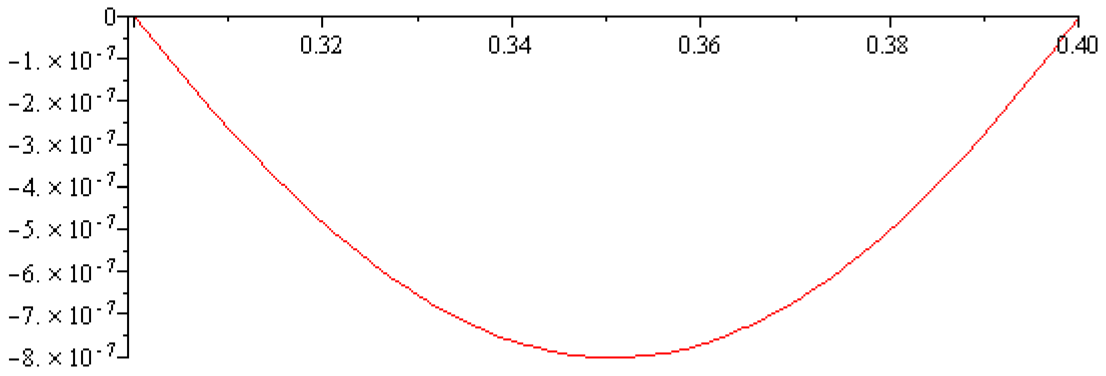
Let us take a real function for which we have the exact values and derivatives. Specifically, let us estimate the values of sin(x) on the interval [0.3, 0.4] using the techniques above. $y_{k-1} = \sin(0.3)$ while $y_k = \sin(0.4)$. Similarly, $s_{k-1} = 0.1\cos(0.3)$ and $s_k = 0.1\cos(0.4)$. Thus, the interpolating polynomial in α is to 10 decimal digits of precision,

$$p(\alpha) = -0.0001565229818\alpha^3 - 0.001478990283\alpha^2 + 0.09553364891\alpha + 0.2955202067$$

If we plot $p(\alpha) - \sin(0.3 + 0.1\alpha)$ on the interval [0, 1], we get that we have a very good approximation:



Note that the absolute error does not exceed 10^{-7}. If you were to find the interpolating polynomial that passes through the four points (0.2, sin(0.2)), (0.3, sin(0.3)), (0.4, sin(0.4)), (0.5, sin(0.5)) and plotted the error between this and the correct values of the function on [0.3, 0.4], we get a more significant error:



The error is larger by almost an order of magnitude. Thus, finding an interpolating polynomial that matches the values and derivatives of a function at two points is more effective at approximating an intermediate value than finding an interpolating polynomial that matches the function at four points. The problem is that seldom do we know the exact value of the derivatives, but for an IVP, the derivative is given as part of the problem.

Implementation of estimating intermediate points

Thus, we could implement the following function:

```
double ivp_interp( double t, double t0, double t1,
                             double y0, double y1,
                             double dy0, double dy1 ) {
    double delta_t{t1 - t0};
    double   alpha{(t - t0)/delta_t};
    double delta_y{y1 - y0};
    double  sum_dy{(dy0 + dy1)*delta_t};
    return (((-2*delta_y + sum_dy)*alpha - delta_t*dy0 - sum_dy + 3*delta_y)*alpha
                + dy0*delta_t)*alpha + y0;
}
```

However, more reasonable may be a function that takes as an argument the tuple returned by one of our interpolating functions and then finds the appropriate interval on which to interpolate:

```
#include <cassert>
#include <vector>
#include <tuple>

double ivp_interp( double t, std::tuple< std::vector<double>,
                                         std::vector<double>,
                                         std::vector<double> > const &approx ) {
    double t0{*std::get<0>( approx ).begin()};
    double tf{*std::get<0>( approx ).rbegin()};
    std::size_t n{std::get<0>( approx ).size()};

    assert( t >= t0 && t <= tf );

    std::size_t k{
        static_cast<std::size_t>( std::floor( (t - t0)/(tf - t0)*(n - 1) ) )};

    if ( k == (n - 1) ) {
        return *std::get<1>( approx ).rbegin();
    } else {
        double  t0{std::get<0>( approx )[k]};
        double  t1{std::get<0>( approx )[k + 1]};
        double delta_t{t1 - t0};

        double  y0{std::get<1>( approx )[k]};
        double  y1{std::get<1>( approx )[k + 1]};
        double dy0{delta_t*std::get<2>( approx )[k]};
        double dy1{delta_t*std::get<2>( approx )[k + 1]};

        double  offset{(t - t0)/delta_t};
        double delta_y{y1 - y0};
        double  sum_dy{dy0 + dy1};

        return (
            ((-2*delta_y + sum_dy)*offset - dy0 - sum_dy + 3*delta_y)*offset + dy0
        )*offset + y0;
    }
}
```

Heun’s method

To use the trapezoidal rule, we must be able to calculate

$$h\frac{f(a)+f(b)}{2}$$

or in this case,

$$h\frac{\frac{\mathrm{d}}{\mathrm{d}t}y(t_0)+\frac{\mathrm{d}}{\mathrm{d}t}y(t_0+h)}{2}.$$

The problem here is that if we knew $\frac{\mathrm{d}}{\mathrm{d}t}y(t_0+h)=f\left(t_0+h, y(t_0+h)\right)$, we wouldn’t have to approximate it! However, we do have an approximation: recall that $y_0+hf(t_0,y_0)$ is an approximation of $y(t_0+h)$, so could we not use $f(t_1,y_1)$ as an approximation to $\frac{\mathrm{d}}{\mathrm{d}t}y(t_0+h)$? In that case, we now have Heun’s method:

1. let $s_0=f(t_0,y_0)$,
2. let $s_1=f(t_1,y_0+hs_0)$,
3. let $y_1=y_0+h\frac{s_0+s_1}{2}$.

Proof that Heun’s method is O(h^3)

For Euler’s method, the proof that it is O(h^2) is simple enough:

$$\begin{aligned} y(t_0+h) &= y(t_0) + y^{(1)}(t_0)h + \frac{1}{2}y^{(2)}(\tau_0)h^2 \\ &= y_0 + f(t_0, y_0)h + \frac{1}{2}y^{(2)}(\tau_0)h^2 \end{aligned}$$

For Heun’s method, that it is a significantly better approximation requires more delicate care:

$$\begin{aligned} y(t_0+h) &= y(t_0) + y^{(1)}(t_0)h + \frac{1}{2}y^{(2)}(t_0)h^2 + \frac{1}{6}y^{(3)}(\tau_0)h^3 \\ &= y_0 + f(t_0, y_0)h + \frac{1}{2}y^{(2)}(t_0)h^2 + \frac{1}{6}y^{(3)}(\tau_0)h^3 \end{aligned}$$

Let us label $s_0 = f(t_0, y_0)$ as the initial slope, so we now have

$$y(t_0+h) = y_0 + s_0 h + \frac{1}{2}y^{(2)}(t_0)h^2 + \frac{1}{6}y^{(3)}(\tau_0)h^3$$

Next, let us use the forward divided-difference approximation of the second derivative:

$$y^{(2)}(t_0) = \frac{y^{(1)}(t_0+h) - y^{(1)}(t_0)}{h} - \frac{1}{2}y^{(3)}(\tau_1)h\,.$$

Recall that $s_0 = y^{(1)}(t_0) = f(t_0, y_0)$, so we have that

$$y^{(2)}(t_0) = \frac{y^{(1)}(t_0+h) - s_0}{h} - \frac{1}{2}y^{(3)}(\tau_1)h$$

We can substitute this into the previous equation to get

$$y(t_0+h) = y_0 + s_0 h + \frac{1}{2}\left(\frac{y^{(1)}(t_0+h) - s_0}{h} - \frac{1}{2}y^{(3)}(\tau_1)h\right)h^2 + \frac{1}{6}y^{(3)}(\tau_0)h^3$$

Expanding this and collecting on powers of *h*, we have

$$y(t_0+h) = y_0 + s_0 h - \frac{1}{2}s_0 h + \frac{1}{2}y^{(1)}(t_0+h)h + \left(-\frac{1}{4}y^{(3)}(\tau_1) + \frac{1}{6}y^{(3)}(\tau_0)\right)h^3\,.$$

Observing that $s_0 h - \frac{1}{2}s_0 h = \frac{1}{2}s_0 h$ and by continuity, we can simplify the error term to

$$y(t_0+h) = y_0 + \frac{1}{2}s_0 h + \frac{1}{2}y^{(1)}(t_0+h)h - \frac{1}{12}y^{(3)}(\tau)h^3\,. \qquad [1]$$

We must now examine the term $y^{(1)}(t_0+h)$. From our IVP, we have that this equals

$$y^{(1)}(t_0+h) = f(t_0+h, y(t_0+h)).$$

Now this becomes more subtle: for a function of two variables, we have that

$$g(x, y+h) = g(x,y) + \frac{\partial}{\partial y} g(x, y+h)h + \frac{1}{2}\frac{\partial^2}{\partial y^2} g(x,\upsilon)h^2.$$

In this case, we will be substituting $y(t_0+h) = y_0 + y^{(1)}(t_0)h + \frac{1}{2}y^{(2)}(\tau_2)h^2$, so instead of calculating

$$f(x, y+h) = f(x,y) + \frac{\partial}{\partial y} f(x, y+h)h + \frac{1}{2}\frac{\partial^2}{\partial y^2} f(x,\upsilon)h^2,$$

we will instead be calculating

$$\begin{aligned}
y^{(1)}(t_0+h) &= f(t_0+h, y(t_0+h)) \\
&= f\left(t_0+h, \underbrace{y_0 + y^{(1)}(t_0)h}_{\text{To remain here}} + \underbrace{\frac{1}{2}y^{(2)}(\tau_2)h^2}_{\text{To be expanded}}\right) \\
&= f\left(t_0+h, \underbrace{y_0 + y^{(1)}(t_0)h}_{\text{To remain here}}\right) + \frac{\partial}{\partial y} f(t_0+h, y(\tau_3))\left(\underbrace{\frac{1}{2}y^{(2)}(\tau_2)h^2}_{\text{To be expanded}}\right)
\end{aligned}$$

Now, recall that $y^{(1)}(t_0) = s_0$, so we now have that we can write this as

$$y^{(1)}(t_0+h) = f(t_0+h, y_0+s_0h) + \frac{1}{2}\left(\frac{\partial}{\partial y} f(t_0+h, y(\tau_3))y^{(2)}(\tau_2)\right)h^2.$$

Substituting this back into Equation [1], we have

$$\begin{aligned}
y(t_0+h) &= y_0 + \frac{1}{2}s_0h + \frac{1}{2}\left(f(t_0+h, y_0+s_0h) + \frac{1}{2}\left(\frac{\partial}{\partial y} f(t_0+h, y(\tau_3))y^{(2)}(\tau_2)\right)h^2\right)h - \frac{1}{12}y^{(3)}(\tau)h^3 \\
&= y_0 + \frac{1}{2}\left(s_0 + f(t_0+h, y_0+s_0h)\right)h + \left(\frac{1}{4}\left(\frac{\partial}{\partial y} f(t_0+h, y(\tau_3))y^{(2)}(\tau_2)\right) - \frac{1}{12}y^{(3)}(\tau)\right)h^3
\end{aligned}$$

If we allow $s_1 = f(t_0+h, y_0+s_0h)$, we may thus write this as

$$\begin{aligned}
y(t_0+h) &= y_0 + \frac{1}{2}s_0h + \frac{1}{2}\left(f(t_0+h, y_0+s_0h) + \frac{1}{4}\left(\frac{\partial}{\partial y} f(t_0+h, y(\tau_3))y^{(2)}(\tau_2)\right)h^2\right)h - \frac{1}{12}y^{(3)}(\tau)h^3 \\
&= y_0 + \frac{1}{2}(s_0+s_1)h + \left(\frac{1}{4}\left(\frac{\partial}{\partial y} f(t_0+h, y(\tau_3))y^{(2)}(\tau_2)\right) - \frac{1}{12}y^{(3)}(\tau)\right)h^3
\end{aligned}$$

which is the formula for Heun’s method together with the O(h^3) error.

To demonstrate that this is indeed, true, what we will do is take an initial-value problem and calculate what the error should be with the above formula. The example we will use is the initial-value problem

$$\begin{aligned} y^{(1)}(t) &= 4\left(y(t)-t\right)^2 - 1 \\ y(1) &= 1 \end{aligned}$$

This has an exact solution:

$$y(t) = \frac{\sqrt{3}}{6}\left(4\sqrt{3}t - 3\tanh\left(\frac{1}{3}\left(6t + \sqrt{3}\tanh^{-1}\left(\frac{2}{3}\sqrt{3}\right) - 6\right)\sqrt{3}\right)\right).$$

Next, let us evaluate the error term at $t = 1$:

$$\left(\frac{1}{4}\left(\frac{\partial}{\partial y} f\left(1, y(1)\right) y^{(2)}(1)\right) - \frac{1}{12} y^{(3)}(1)\right) h^3 = \left(-6+16\right) h^3 = 10h^3 .$$

Thus, the error should approach $10h^3$ as h goes to zero. We can therefore approximate $y(1.1)$, $y(1.01)$, $y(1.001)$, and so on and compare it to the actual solution by evaluating the above function at these points:

h	**Exact solution** ***y*****(1 +** ***h*****)**	**Approximation using Heun’s method**	**Error**	**Approximation of the Error**
0.1	1.26945970562049697770861211078	1.262	0.74597×10^{-2}	10^{-2}
0.01	1.02961168810797661702844033508	1.029602	0.968810×10^{-5}	10^{-5}
0.001	1.00299601196808298297084006823	1.002996002	0.9968083×10^{-8}	10^{-8}
0.0001	1.00029996001199680083178245722	1.000299960002	$0.999680083 \times 10^{-11}$	10^{-11}
0.00001	1.00002999960001199968000831978	1.000029999600002	$0.99996800083 \times 10^{-14}$	10^{-14}
0.000001	1.00000299999600001199996800008	1.000002999996000002	$0.999996800008 \times 10^{-17}$	10^{-17}

As you can see, the error is not precisely 10 h^3, but as h gets closer to zero, the error approaches the predicted error.

Implementation of Heun's method

Here is a C++ implementation of Heun's method. The differences between this implementation and that of Euler's method are shown in bold red.

```
#include <vector>
#include <tuple>
#include <cassert>

std::tuple< std::vector<double>, std::vector<double>, std::vector<double> >
  heun( double f(double, double),
        double t0, double y0, double tf,
        size_t n ) {
    assert( n > 0 );

    double h{(tf - t0)/n};

    std::vector<double>  t(n + 1);
    std::vector<double>  y(n + 1);
    std::vector<double> dy(n + 1);

    y[0] = y0;

    for ( size_t k{0}; k < n; ++k ) {
         t[k] = t0 + k*h;

        double s0{f( t[k],     y[k]  )};
        double s1{f( t[k] + h, y[k] + h*dy[k] )};
        dy[k] = s0;

        y[k + 1] = y[k] + h*(s0 + s1)/2.0;
    }

     t[n] = tf;
    dy[n] = f( tf, y[n] );

    return std::make_tuple( t, y, dy );
}
```

Example of Heun's method

Suppose we want to approximate the solution to the initial-value problem

$$\begin{aligned} y^{(1)}(t) &= (t-1)y(t)+0.5 \\ y(0) &= 1.2 \end{aligned}$$

where we want to approximate $y(2)$ with h = 1.0, 0.5 and 0.25. Here,

$$f(t,y) = (t-1)y+0.5\,.$$

First, we note that this does have a solution:

$$y(t) = 0.05\left(5\sqrt{2}\sqrt{\pi}\mathrm{e}^{0.5}\left(\operatorname{erf}\left(0.5\sqrt{2}\right)+\operatorname{erf}\left(0.5\sqrt{2}(t-1)\right)\right)+24\right)\mathrm{e}^{0.5t(t-2)}$$

This can be implemented as the C++ function

```
double y( double t ) {
    double const SQRT_PI{std::sqrt( std::acos( -1.0 ) )};
    double const ROOT_2_5{std::sqrt( 2.0 )*0.5};

    return 0.05*(10.0*ROOT_2_5*SQRT_PI*std::exp( 0.5 )*(
        std::erf( ROOT_2_5 ) + std::erf( ROOT_2_5*(t - 1) )
    ) + 24.0)*exp( 0.5*t*(t - 2.0) );
}
```

This solution evaluated at this point is $y(2.0)$ = 2.610686134642448.

Applying Heun's method with h = 2, we have

$$\begin{aligned} s_0 &= f(t_0, y_0) \\ &= f(0,1.2) \\ &= (0-1)\cdot 1.2+0.5 \\ &= -0.7 \\ s_1 &= f(t_0+h, y_0+hs_0) \\ &= f(0+2, 1.2+2\cdot(-0.7)) \\ &= (2-1)\cdot(-0.2)+0.5 \\ &= 0.3 \\ y_1 &= y_0 + h\frac{s_0+s_1}{2} \\ &= 1.2+2\cdot\frac{-0.7+0.3}{2} \\ &= 0.8 \end{aligned}$$

Thus, we are not off to a very good start, with an error approximately equal to 1.811, which is worse than Euler's method with two function evaluations.

Applying Heun's method with h = 1, we have

$$\begin{aligned} s_0 &= f(t_0, y_0) \\ &= f(0, 1.2) \\ &= (0-1)\cdot 1.2 + 0.5 \\ &= -0.7 \\ s_1 &= f(t_0 + h, y_0 + hs_0) \\ &= f(0+1, 1.2 + 1\cdot(-0.7)) \\ &= (1-1)\cdot(-0.2) + 0.5 \\ &= 0.5 \\ y_1 &= y_0 + h\frac{s_0 + s_1}{2} \\ &= 1.2 + 1\cdot\frac{-0.7+0.5}{2} \\ &= 1.1 \end{aligned}$$

$$\begin{aligned} s_0 &= f(t_1, y_1) \\ &= f(1, 1.1) \\ &= (1-1)\cdot 1.1 + 0.5 \\ &= 0.5 \\ s_1 &= f(t_1 + h, y_1 + hs_0) \\ &= f(1+1, 1.1 + 1\cdot 0.5) \\ &= (2-1)\cdot 1.6 + 0.5 \\ &= 2.5 \\ y_2 &= y_1 + h\frac{s_0 + s_1}{2} \\ &= 1.1 + 1\cdot\frac{0.5+2.1}{2} \\ &= 2.4 \end{aligned}$$

The error has now been reduced to 0.2107.

Step size	h = 1.0	h = 0.5	h = 0.25	h = 0.125
t	4 calls	8 calls	16 calls	32 calls
0.0	1.2	1.2	1.2	1.2
0.125				1.12666015625
0.25			1.07890625	1.048980627459241
0.3750				1.077580380439758
0.5		1.04375	1.040158081054688	1.038332039951021
0.625				1.044104406965346
0.75			1.067800128459931	1.065607240975708
0.875				1.102903948033156
1.0	1.1	1.16328125	1.159431374445558	1.156787510939147
1.125				1.228813194618359
1.25			1.324569854896982	1.321390069588570
1.3750				1.437939527924248
1.5		1.58994140625	1.586734879789265	1.583136327587963
1.625				1.763257085796267
1.75			1.989975504128868	1.986673919411313
1.875				2.264549042161006
2.0	2.4	2.598040771484375	2.612547696535078	2.611812448757147

Recall that the exact solution is $y(2.0) = 2.610686134642448$, so the error when $h = 0.\ 25$ is 0.001862, while the error when $h = 0.125$ is 0.001126, which is nowhere near one quarter the previous error.

n	h	Function calls	Approximation of $y(2.0)$	Absolute error
1	2.0	2	0.8	1.811
2	1.0	4	2.4	0.2107
4	0. 5	8	2.598040771484375	0.01265
8	0. 25	16	2.612547696535078	0.001862
16	0.125	32	2.611812448757147	0.001126
32	0.0625	64	2.611054569394839	0.0003684
64	0.03125	128	2.610789453526507	0.0001033
128	0.015625	256	2.610713390760282	0.00002726
256	0.0078125	512	2.610693128641439	0.000006994
512	0.00390625	1024	2.610687905745948	0.000001771
1024	0.001953125	2048	2.610686580250618	0.0000004456

You will see that after n = 36, each time you double the number of steps, the error drops by approximately one quarter.

4th-order Runge-Kutta method

To use Simpson's rule, we must be able to calculate

$$h\frac{f(a)+f\left(\frac{a+b}{2}\right)+f(b)}{6}$$

or in this case,

$$h\frac{\frac{\mathrm{d}}{\mathrm{d}t}y(t_0)+\frac{\mathrm{d}}{\mathrm{d}t}y\left(t_0+\frac{h}{2}\right)+\frac{\mathrm{d}}{\mathrm{d}t}y(t_0+h)}{6}.$$

Again, we don't know the exact values, but we can find approximations of them. This is a formula found by Runge and Kutta:

1. let $s_0 = f(t_0, y_0)$,
2. let $s_1 = f\left(t_0+\frac{h}{2}, y_0+\frac{h}{2}s_0\right)$, an estimate of the slope at $\left(t_0+\frac{h}{2}, y\left(t_0+\frac{h}{2}\right)\right)$,
3. let $s_2 = f\left(t_0+\frac{h}{2}, y_0+\frac{h}{2}s_1\right)$, a second estimate of the slope at $\left(t_0+\frac{h}{2}, y\left(t_0+\frac{h}{2}\right)\right)$,
4. let $s_3 = f(t_0+h, y_0+hs_2)$, an estimate of the slope at $(t_0+h, y(t_0+h))$, and
5. let $y_1 = y_0 + h\frac{s_0+2s_1+2s_2+s_3}{6}$.

Note that we have two estimators of the slope. The error of this method is O(h^5); however, if it is applied iteratively, the error drops to O(h^4).

Implementation of Runge-Kutta method

Here is a C++ implementation of the 4th-order Runge-Kutta method:

```
#include <vector>
#include <tuple>
#include <cassert>

std::tuple< std::vector<double>, std::vector<double>, std::vector<double> >
  heun( double f(double, double),
        double t0, double y0, double tf,
        size_t n ) {
    assert( n > 0 );

    double h{(tf - t0)/n};
    double h2{0.5*h};

    std::vector<double>  t(n + 1);
    std::vector<double>  y(n + 1);
    std::vector<double> dy(n + 1);

    y[0] = y0;

    for ( size_t k{0}; k < n; ++k ) {
         t[k] = t0 + k*h;

        double s0{f( t[k],      y[k] )};
        double s1{f( t[k] + h2, y[k] + h2*dy[k] )};
        double s2{f( t[k] + h2, y[k] + h2*s1    )};
        double s3{f( t[k] + h,  y[k] +  h*s2    )};
        dy[k] = s0;

        y[k + 1] = y[k] + h*(s0 + 2*(s1 + s2) + s3)/6.0;
    }

     t[n] = tf;
    dy[n] = f( tf, y[n] );

    return std::make_tuple( t, y, dy );
}
```

Example of 4th-order Runge-Kutta method

Suppose we want to approximate the solution to the initial-value problem

$$y^{(1)}(t) = (t-1)\,y(t) + 0.5$$
$$y(0) = 1.2$$

where we want to approximate $y(2)$ with h = 1.0, 0.5 and 0.25. Here,

$$f(t, y) = (t-1)\,y + 0.5\,.$$

First, we note that this does have a solution:

$$y(t) = 0.05\left(5\sqrt{2}\sqrt{\pi}\mathrm{e}^{0.5}\left(\operatorname{erf}\left(0.5\sqrt{2}\right) + \operatorname{erf}\left(0.5\sqrt{2}\,(t-1)\right)\right) + 24\right)\mathrm{e}^{0.5t(t-2)}$$

This can be implemented as the C++ function

```
double y( double t ) {
    double const SQRT_PI{std::sqrt( std::acos( -1.0 ) )};
    double const ROOT_2_5{std::sqrt( 2.0 )*0.5};

    return 0.05*(10.0*ROOT_2_5*SQRT_PI*std::exp( 0.5 )*(
        std::erf( ROOT_2_5 ) + std::erf( ROOT_2_5*(t - 1) )
    ) + 24.0)*exp( 0.5*t*(t - 2.0) );
}
```

This solution evaluated at this point is $y(2.0)$ = 2.610686134642448.

Applying 4th-order Runge-Kutta method with h = 2, we have

$$
\begin{aligned}
s_0 &= f\left(t_0, y_0\right) \\
&= f\left(0, 1.2\right) \\
&= \left(0-1\right)\cdot 1.2 + 0.5 \\
&= -0.7 \\
s_1 &= f\left(t_0 + \tfrac{h}{2}, y_0 + \tfrac{h}{2} s_0\right) \\
&= f\left(0+1, 1.2 + 1\cdot\left(-0.7\right)\right) \\
&= f\left(1, 0.5\right) \\
&= \left(1-1\right)\cdot\left(0.5\right) + 0.5 \\
&= 0.5 \\
s_2 &= f\left(t_0 + \tfrac{h}{2}, y_0 + \tfrac{h}{2} s_1\right) \\
&= f\left(0+1, 1.2 + 1\cdot\left(0.5\right)\right) \\
&= f\left(1, 1.7\right) \\
&= \left(1-1\right)\cdot\left(1.7\right) + 0.5 \\
&= 0.5 \\
s_3 &= f\left(t_0 + h, y_0 + h s_2\right) \\
&= f\left(0+2, 1.2 + 2\cdot\left(0.5\right)\right) \\
&= f\left(2, 2.2\right) \\
&= \left(2-1\right)\cdot\left(2.2\right) + 0.5 \\
&= 2.7 \\
y_1 &= y_0 + h\frac{s_0 + 2s_1 + 2s_2 + s_3}{6} \\
&= 1.2 + 2\cdot\frac{-0.7 + 2\cdot 0.5 + 2\cdot 0.5 + 2.7}{6} \\
&= 2.5\bar{3}
\end{aligned}
$$

Recall that with Euler's method, when h = 0.5, we used four function evaluations and found an approximation of $y(2)$ equal to 1.671875. With Heun's method, when h = 1, we also used four function evaluations, and the approximation of $y(2)$ was 2.4. With 4th-order Runge-Kutta, the approximation when h = 2 also uses four function evaluations, but now the approximation is $2.53333\cdots$, which is even closer to the 16-digit $y(2) = 2.610686134642448$.

Applying the 4th-order Runge Kutta method with $h = 1$, we have

$$\begin{aligned}
s_0 &= f\left(t_0, y_0\right) \\
&= f\left(0, 1.2\right) \\
&= \left(0-1\right)\cdot 1.2 + 0.5 \\
&= -0.7 \\
s_1 &= f\left(t_0 + \tfrac{h}{2}, y_0 + \tfrac{h}{2} s_0\right) \\
&= f\left(0+0.5, 1.2 + 0.5\cdot\left(-0.7\right)\right) \\
&= f\left(0.5, 0.85\right) \\
&= \left(0.5-1\right)\cdot\left(0.85\right) + 0.5 \\
&= 0.075 \\
s_2 &= f\left(t_0 + \tfrac{h}{2}, y_0 + \tfrac{h}{2} s_1\right) \\
&= f\left(0+0.5, 1.2 + 0.5\cdot 0.075\right) \\
&= f\left(0.5, 1.2375\right) \\
&= \left(0.5-1\right)\cdot\left(1.2375\right) + 0.5 \\
&= -0.11875 \\
s_3 &= f\left(t_0 + h, y_0 + h s_2\right) \\
&= f\left(0+1, 1.2 + 1\cdot\left(-0.11875\right)\right) \\
&= f\left(1, 1.08125\right) \\
&= \left(1-1\right)\cdot\left(1.08125\right) + 0.5 \\
&= 0.5 \\
y_1 &= y_0 + h\frac{s_0 + 2s_1 + 2s_2 + s_3}{6} \\
&= 1.2 + 1\cdot\frac{-0.7 + 2\cdot 0.075 + 2\cdot\left(-0.11875\right) + 0.5}{6} \\
&= 1.15208\overline{3}
\end{aligned}$$

$$\begin{aligned}
s_0 &= f\left(t_1, y_1\right) \\
&= f\left(1.0, 1.52083\overline{3}\right) \\
&= \left(1-1\right)\cdot 1.5208\overline{3} + 0.5 \\
&= 0.5 \\
s_1 &= f\left(t_1 + \tfrac{h}{2}, y_1 + \tfrac{h}{2} s_0\right) \\
&= f\left(1+0.5, 1.5208\overline{3} + 0.5\cdot 0.5\right) \\
&= f\left(1.5, 1.40208\overline{3}\right) \\
&= \left(1.5-1\right)\cdot\left(1.40208\overline{3}\right) + 0.5 \\
&= 1.201041\overline{6} \\
s_2 &= f\left(t_1 + \tfrac{h}{2}, y_1 + \tfrac{h}{2} s_1\right) \\
&= f\left(1+0.5, 1.5208\overline{3} + 0.5\cdot 0.075\right) \\
&= f\left(1.5, 1.2375\right) \\
&= \left(1.5-1\right)\cdot\left(1.2375\right) + 0.5 \\
&= 1.37630208\overline{3} \\
s_3 &= f\left(t_0 + h, y_0 + h s_2\right) \\
&= f\left(1+1, 1.5208\overline{3} + 1\cdot\left(-0.11875\right)\right) \\
&= f\left(2, 1.08125\right) \\
&= \left(2-1\right)\cdot\left(1.08125\right) + 0.5 \\
&= 3.02838541\overline{6} \\
y_1 &= y_0 + h\frac{s_0 + 2s_1 + 2s_2 + s_3}{6} \\
&= 1.5208\overline{3} + 1\cdot\tfrac{0.5 + 2\cdot 1.201041\overline{6} + 2\cdot 1.37630208\overline{3} + 3.02838541\overline{6}}{6} \\
&= 2.599262152\overline{7}
\end{aligned}$$

The error has now been reduced to 0.01142.

Step size:	h = 2.0	h = 1.0	h = 0.5	h = 0.25
t	4 calls	8 calls	16 calls	32 calls
0.0	1.2	1.2	1.2	1.2
0.25				1.077087720235189
0.5			1.037556966145833	1.037608979017124
0.75				1.064694810566422
1.0		1.152083333333333	1.155539366934035	1.155645642124058
1.25				1.319964728593053
1.5			1.581236177180135	1.581426855926445
1.75				1.984872474943279
2.0	2.533333333333333	2.599262152777778	2.610087820593202	2.610654583581414

Recall that the exact solution is $y(2.0) = 2.610686134642448$, so the error when $h = 0.\ 5$ is 0.0005983, while the error when $h = 0.025$ is 0.00003155, which is approximately one-sixteenth the previous error.

n	h	Function calls	Approximation of $y(2.0)$	Absolute error
1	2.0	2	2.533333333333333	0.07735
2	1.0	4	2.599262152777778	0.01142
4	0. 5	8	2.610087820593202	0.0005983
8	0. 25	16	2.610654583581414	0.00003155
16	0.125	32	2.610684379040146	0.000001756
32	0.0625	64	2.610686032624805	0.0000001020
64	0.03125	128	2.610686128526171	0.000000006116
128	0.015625	256	2.610686134268633	0.0000000003738
256	0.0078125	512	2.610686134619355	0.00000000002309
512	0.00390625	1024	2.610686134641016	0.000000000001433
1024	0.001953125	2048	2.610686134642358	0.00000000000009059

You will see that after $n = 16$, each time you double the number of steps, the error drops by approximately one quarter.

Iterative methods

One problem with Euler's method, Huen's method and 4th-order Runge-Kutta is that we do not know the error the overall approximation. If sufficiently high derivatives and partial derivatives are bounded, we know the error is bounded, but suppose we use $n = 10$ for an IVP. What is the actual error of $y(t_f)$? Is y_n a good approximation of $y(t_f)$ or not? We will look at how we can estimate the error of an approximation iteratively: by first using n steps, then $2n$ steps, etc. We will first do this with Euler's method, and then with 4th-order Runge-Kutta.

Iterative Euler's method: estimating the error of Euler's method

Suppose we now want to get a good approximation to a solution to a differential equation; that is, we would like to ensure that our approximation of $|y(t_f) - y_n| < \varepsilon_{\text{abs}}$. Now, how can we ensure this?

Recall that if we applied Euler's method, that if we double the number of points, then this reduces the error by approximately a factor of two. Thus, if h is small enough, the we could find two approximations:

$$y_0, y_1, y_2, y_3, \ldots, y_{n-2}, y_{n-1}, y_n$$

approximating the points

$$y(t_0), y(t_0+h), y(t_0+2h), y(t_0+3h), \ldots, y(t_0+(n-2)h), y(t_0+(n-1)h), y(t_0+nh) = y(t_f),$$

and

$$z_0, z_1, z_2, z_3, z_4, z_5, z_6, z_7, \ldots, z_{2n-4}, z_{2n-3}, z_{2n-2}, z_{2n-1}, z_{2n}$$

approximating the values

$$y(t_0), y\left(t_0+\tfrac{h}{2}\right), y\left(t_0+2\tfrac{h}{2}\right), y\left(t_0+3\tfrac{h}{2}\right), \ldots, y\left(t_0+(2n-2)\tfrac{h}{2}\right), y\left(t_0+(2n-1)\tfrac{h}{2}\right), y\left(t_0+2n\tfrac{h}{2}\right) = y(t_f).$$

Thus, from this, we have that in general y_k and z_{2k} both approximate $y(t_0+kh)$ and specifically,

1. y_n is an approximation of $y(t_f)$ when n steps, and
2. z_{2n} is an approximation of $y(t_f)$ when $2n$ steps.

Now, z_{2n} has half the error of y_n, so $\dfrac{z_{2n} - y(t_f)}{y_n - y(t_f)} \approx \dfrac{1}{2}$, then we can write

$$2z_{2n} - 2y(t_f) \approx y_n - y(t_f)$$

or

$$z_{2n} - y(t_f) \approx y_n - z_{2n}.$$

Thus, the error of z_{2n} is approximated by $y_n - z_{2n}$. Additionally, bringing the actual values to one side, we have

$$y(t_f) \approx 2z_{2n} - y_n$$

so this is an even better estimate of the actual value of $y(t_f)$. Looking at how we used multiple steps of Euler's method to approximate $y(2)$, we had the following entries in the table:

n	Approximation	Error
256	2.590963000669263	0.01972
512	2.600793646021044	0.009892
1024	2.605732112846550	0.004954

First, we see that the difference 2.600793646021044 – 2.590963000669263 = 0.009830645351781 is indeed a good approximation of the error of the approximation with 512 steps.

Now, let us calculate this approximation:

$$y(2) \approx 2 \cdot 2.600793646021044 - 2.590963000669263$$
$$= 2.610624291372825$$

You will now note that the error of *extrapolated* estimation of $y(2) = 2.610686134642448$ is now 0.00006184, which is significantly better than even the approximation using $n = 1024$ steps. Thus, we have a very good estimation of the error of z_{512} and we can use this estimate to get an even better approximation of $y(2)$.

Now, one important question we must ask is what do we return? Having calculated both $y_0, \ldots, y_n$ and $z_0, \ldots, z_{2n}$, surely the second is more accurate than the first. However, notice that if we have both y_k and z_{2k}, then the approximation $2z_{2k} - y_k$ is more accurate than either of the first two approximations, and thus, instead, we will only return a vector containing $n + 1$ entries with the kth entry (from 0 to n) being assigned $2z_{2k} - y_k$.

Thus, we may proceed as follows:

1. Let n begin with some reasonable number of intervals.
2. Approximate y_n creating a vector **y** of size $n + 1$ (indexed from 0 to n).
3. Approximate z_{2n} creating a vector **z** of size $n + 1$ (indexed from 0 to $2n$), and
    a. if $|y_n - z_{2n}| < \varepsilon_{\text{abs}}$, we are done, and let our approximation of $y(t_f)$ be $y(t_f) \approx 2z_{2n} - y_n$, and in general, we could find better approximations of each point: $y(t_k) \approx 2z_{2k} - y_k$,
    b. otherwise, let $\mathbf{y} \leftarrow \mathbf{z}$, $n \leftarrow 2n$ and return to Step 3.

Implementation of iterative Euler's method

Here is a C++ implementation of the iterative Euler's method:

```
#include <vector>
#include <tuple>
#include <cassert>
#include <stdexcept>

std::tuple< std::vector<double>, std::vector<double>, std::vector<double> >
  iterative_euler( double f(double, double),
                   double t0, double y0, double tf,
                   std::size_t n, std::size_t N, double eps_abs ) {
    assert( n > 0 );

    double h{(tf - t0)/n};

    std::vector<double>  t(n + 1);
    std::vector<double>  y(n + 1);
    std::vector<double> dy(n + 1);

    // Find the approximation for 'n' intervals
    y[0] = y0;

    for ( std::size_t k{0}; k < n; ++k ) {
        t[k] = t0 + k*h;

        double s0{ f( t[k], y[k] )};
        dy[k] = s0;

        y[k + 1] = y[k] + h*s0;
    }

     t[n] = tf;
    dy[n] = f( tf, y[n] );
    std::vector<double>  s{};
    std::vector<double>  z{};
    std::vector<double> dz{};

    for ( std::size_t m{2*n}; m <= N; m *= 2 ) {
        h /= 2.0;

         s = std::vector<double>(m + 1);
         z = std::vector<double>(m + 1);
        dz = std::vector<double>(m + 1);

        z[0] = y0;

        for ( std::size_t k{0}; k < m; ++k ) {
            s[k] = t0 + k*h;

            double s0{f( s[k], z[k] )};
            dz[k] = s0;

            z[k + 1] = z[k] + h*s0;
        }

         s[m] = tf;
        dz[m] = f( tf, z[m] );

        double error{std::abs( z[m] - y[n] )};

        if ( error < eps_abs ) {
            for ( std::size_t k{1}; k <= n; ++k ) {
                 y[k] = 2*z[2*k] - y[k];
                dy[k] = f( t[k], y[k] );
            }

            return std::make_tuple( t, y, dy );
        } else {
             n =  m;
             t =  s;
             y =  z;
            dy = dz;
        }
    }

    throw std::runtime_error( "Euler's method did not find a solution" );
}
```

Example of the iterative Euler's method

As an explicit example, consider the initial value problem

$$y^{(1)}(t) = ty(t) + 1$$
$$y(0) = 1$$

Approximating $y(1)$ with $n = 10$ yields the approximate points, and then again with $n = 20$ yields these approximations, where you will see that the error is indeed approximately halved, but also that the extrapolated approximation is significantly better than either approximation.

t	**Approximation with *n* = 10**	**Approximation with *n* = 20**	**Exact solution**	**Error with *n* = 10**	**Error with *n* = 20**	**Extrapolated approximation** $2z_{2k} - y_k$	**Error**
0	1	1	1	0	0	1	0
0.05		1.05	1.048709135083809		–0.001291		
0.1	1.1	1.097375	1.094679811574692	–0.005320	–0.002695	1.09475	–0.00007019
0.15		1.141888125	1.137693090879509		–0.004195		
0.2	1.189	1.1833239640625	1.177553218608495	–0.01145	–0.005771	1.177647928125	–0.00009471
0.25		1.221490724421875	1.214089428036366		–0.007401		
0.3	1.26522	1.256222090366602	1.247157419635286	–0.01806	–0.009065	1.247224180733203	–0.00006676
0.35		1.287378759011103	1.276640497461595		–0.01074		
0.4	1.3272634	1.314849630728408	1.302450349325715	–0.02481	–0.01240	1.302435861456816	0.00001449
0.45		1.338552638113840	1.324527464012505		–0.01403		
0.5	1.374172864	1.358435203756279	1.342841185204080	–0.03133	–0.01559	1.342697543512557	0.0001436
0.55		1.374474323662372	1.357389408045940		–0.01708		
0.6	1.4054642208	1.386676279761656	1.368197930352226	–0.03727	–0.01849	1.367888338723313	0.0003096
0.65		1.395075991368807	1.375319476137191		–0.01976		
0.7	1.421136367552	1.399736021649321	1.378832414369550	–0.04230	–0.02090	1.378335675746641	0.0004967
0.75		1.400745260891594	1.378839200470858		–0.02191		
0.8	1.42165682182336	1.398217313608160	1.375464572031941	–0.04619	–0.02275	1.374777805392959	0.0006868
0.85		1.392288621063833	1.368853533435354		–0.02344		
0.9	1.407924276077491	1.383116354668620	1.359169166499498	–0.04876	–0.02395	1.358308433259749	0.0008607
0.95		1.370876118708532	1.346590305874907		–0.02429		
1	1.381211091230517	1.355759503069877	1.331309118719710	–0.04990	–0.02445	1.330307914909237	0.001001

Solving the same problem we did previously using Euler's method, we will now look at the number of steps required to ensure that the error is less than a specified error. Because Euler's method is O(h), if we halve the allowable error, the number of steps should double.

m	Acceptable error ($\varepsilon_{\text{abs}} = 2^{-m}$)	Steps required
0	1	2
1	0.5	4
2	0.25	8
3	0.125	32
4	0.06252	64
5	0.03125	128
6	0.015625	256
7	7.812×10^{-3}	512
8	3.906×10^{-3}	1024
9	1.953×10^{-3}	2048
10	9.766×10^{-4}	4096
11	4.883×10^{-4}	8192
12	2.441×10^{-4}	16384
13	1.221×10^{-4}	32768
14	6.104×10^{-5}	65536
15	3.052×10^{-5}	131072
16	1.526×10^{-5}	262144
17	7.629×10^{-6}	524288
18	3.815×10^{-6}	1048576
19	1.907×10^{-6}	2097152
20	9.537×10^{-7}	4194304
21	4.768×10^{-7}	8388608

Because we ultimately use extrapolation to calculate the resulting approximations, the actual values returned will be more accurate than suggested by the acceptable error; however, because we have no assurance as to how large the error of the extrapolated values, we cannot risk reducing the number of steps.

Iterative 4th-order Runge-Kutta method: estimating the error

Suppose we now want to get a good approximation to a solution to a differential equation; that is, we would like to ensure that our approximation of $|y(t_f) - y_n| < \varepsilon_{\text{abs}}$. Now, how can we ensure this?

Recall that if we applied the Runge-Kutta method, that if we double the number of points, then this reduces the error by approximately a factor of 16. Thus, if h is small enough, the we could find two approximations, one with n approximations and one with $2n$ steps, just like we did with the iterative Euler's method. Thus, from this, we have that in general y_k and z_{2k} both approximate $y(t_0 + kh)$ and specifically,

1. y_n be an approximation of $y(t_f)$ when n steps, and
2. z_{2n} be an approximation of $y(t_f)$ when $2n$ steps.

Now, the error of the z_{2n} approximation is one-sixteenth that of y_n, so

Now, if $\frac{z_{2n} - y(t_f)}{y_n - y(t_f)} \approx \frac{1}{16}$, then we can write

$$16z_{2n} - 16y(t_f) \approx y_n - y(t_f).$$

Adding $y(t_f) - z_{2n}$ to both sides, we get that

$$15z_{2n} - 15y(t_f) \approx y_n - z_{2n}$$

or

$$z_{2n} - y(t_f) \approx \frac{y_n - z_{2n}}{15}$$

Thus, a good approximation of the error of z_{2n} is $\frac{y_n - z_{2n}}{15}$. We can now also use this to find a better approximation of $y(t_f)$, as we can isolate $y(t_f)$ in the equation $16z_{2n} - 16y(t_f) \approx y_n - y(t_f)$ to get:

$$15y(t_f) \approx 16z_{2n} - y_n$$

so

$$y(t_f) \approx \frac{16z_{2n} - y_n}{15}.$$

This is thus a better approximation than either previous approximation.

We can actually see this, as with our example,

n	**Approximation**	**Error**
128	2.610686128526171	0.000000006116
256	2.610686134268633	0.0000000003738

First, we note that $\frac{2.610686134268633 - 2.610686128526171}{15} = 0.0000000003828308$ is already a reasonable approximation of the approximation with 256 steps, but also, we can now find a better approximation by calculating:

$$\begin{aligned} y(t_f) &\approx \frac{16 \cdot 2.610686134268633 - 2.610686128526171}{15} \\ &= 2.610686134651464 \end{aligned}$$

You will now note that the error of *extrapolated* estimation of $y(2)$ = 2.610686134642448 is now 0.000000000009016, or smaller than the error of the last estimate by a factor of 41.46. Thus, we have a very good estimation of the error of $y(2)$.

Thus, we may proceed as follows:

1. Let n begin with some reasonable number of intervals.
2. Approximate y_n creating a vector **y** of size n + 1 (indexed from 0 to n).
3. Approximate z_{2n} creating a vector **z** of size n + 1 (indexed from 0 to $2n$), and
    a. if $\frac{y_n - z_{2n}}{15} < \varepsilon_{\text{abs}}$, we are done, and let our approximation of $y(t_f)$ be $y(t_f) \approx \frac{16z_{2n} - y_n}{15}$, and in general, we could find better approximations of each point: $y(t_k) \approx \frac{16z_{2k} - y_k}{15}$,
    b. otherwise, let **y** ← **z**, $n \leftarrow 2n$ and return to Step 3.

Now, this technique is, as we will see, significantly better than the approximation using an iterative Euler's method, but

1. it requires us to redo the entire calculation with twice the number of points at each step, thus requiring $n + 2n + 4n + 8n + \ldots$ calculations, and
2. all intervals are the same…

This last point is important, because if a solution is not changing rapidly on one time interval, then a larger step size should be reasonable; however, if the solution is changing rapidly, we need a smaller step size. Thus, we will next consider adaptive techniques.

Implementation of iterative 4th-order Runge-Kutta method

Here is a C++ implementation of the iterative 4th-order Runge-Kutta method:

```
#include <vector>
#include <tuple>
#include <cassert>
#include <stdexcept>

std::tuple< std::vector<double>, std::vector<double>, std::vector<double> >
  iterative_runge_kutta( double f(double, double),
                         double t0, double y0, double tf,
                         std::size_t n, std::size_t N, double eps_abs ) {
    assert( n > 0 );

    double h{(tf - t0)/n};
    double h2{h/2.0};

    std::vector<double>  t(n + 1);
    std::vector<double>  y(n + 1);
    std::vector<double> dy(n + 1);

    y[0] = y0;

    for ( std::size_t k{0}; k < n; ++k ) {
         t[k] = t0 + k*h;

        double s0{f( t[k],      y[k] )};
        double s1{f( t[k] + h2, y[k] + h2*dy[k] )};
        double s2{f( t[k] + h2, y[k] + h2*s1    )};
        double s3{f( t[k] + h,  y[k] +  h*s2    )};
        dy[k] = s0;

        y[k + 1] = y[k] + h*(s0 + 2.0*(s1 + s2) + s3)/6.0;
    }

     t[n] = tf;
    dy[n] = f( tf, y[n] );
    std::vector<double>  s{};
    std::vector<double>  z{};
    std::vector<double> dz{};

    for ( std::size_t m{2*n}; m <= N; m *= 2 ) {
        h = h2;
        h2 /= 2.0;

         s = std::vector<double>(m + 1);
         z = std::vector<double>(m + 1);
        dz = std::vector<double>(m + 1);

        z[0] = y0;

        for ( std::size_t k{0}; k < m; ++k ) {
            s[k] = t0 + k*h;

            double s0{f( s[k],      z[k] )};
            double s1{f( s[k] + h2, z[k] + h2*s0 )};
            double s2{f( s[k] + h2, z[k] + h2*s1 )};
            double s3{f( s[k] + h,  z[k] +  h*s2 )};
            dz[k] = s0;

            z[k + 1] = z[k] + h*(s0 + 2.0*(s1 + s2) + s3)/6.0;
        }

         s[m] = tf;
        dz[m] = f( tf, z[m] );

        double error{std::abs( z[m] - y[n] )/15.0};

        if ( error < eps_abs ) {
            for ( std::size_t k{1}; k <= n; ++k ) {
                 y[k] = (16.0*z[2*k] - y[k])/15.0;
                dy[k] = f(t[k], y[k]);
            }

            return std::make_tuple( t, y, dy );
        } else {
             n =  m;
             t =  s;
             y =  z;
            dy = dz;
        }
    }

    throw std::runtime_error(
        "The iterative Runge-Kutta method did not find a solution"
    );
}
```

Example of the iterative 4th-order Runge-Kutta method

Solving the same problem we did previously using the Runge-Kutta method, we will now look at the number of steps required to ensure that the error is less than a specified error. Because the 4th-order Runge-Kutta method is O(h^4), if we divide the error by $2^4 = 16$, the number of steps should double.

m	Acceptable error ($\varepsilon_{abs} = 2^{-m}$)	Steps required
0	1	1
1	0.5	1
2	0.25	1
3	0.125	1
4	0.06252	1
5	0.03125	1
6	0.015625	1
7	7.812×10^{-3}	1
8	3.906×10^{-3}	2
9	1.953×10^{-3}	2
10	9.766×10^{-4}	2
11	4.883×10^{-4}	4
12	2.441×10^{-4}	4
13	1.221×10^{-4}	4
14	6.104×10^{-5}	4
15	3.052×10^{-5}	8
16	1.526×10^{-5}	8
17	7.629×10^{-6}	8
18	3.815×10^{-6}	8
19	1.907×10^{-6}	16
20	9.537×10^{-7}	16
21	4.768×10^{-7}	16

Instead of showing the halving of the step size, let us divide the acceptable error by sixteen:

m	Acceptable error ($\varepsilon_{abs} = 2^{-m}$)	Steps required
0	1	1
4	6.250×10^{-2}	1
8	3.906×10^{-3}	2
12	2.441×10^{-4}	4
16	1.526×10^{-5}	8
20	9.537×10^{-7}	16
24	5.960×10^{-8}	32
28	3.725×10^{-9}	64
32	2.328×10^{-10}	128
36	1.455×10^{-11}	256
40	9.095×10^{-13}	512
44	5.684×10^{-14}	1024
48	3.553×10^{-15}	2048

Thus, using only 2048 steps or four times as many function evaluations, we can approximate $y(2)$ with extreme accuracy. The last approximation of $y(2)$ is 2.610686134642437 which matches most decimal digits of $y(2.0)$ = 2.610686134642448.

Adaptive methods

Now, you are applying a method like Heun's or Euler's, and you know that the error is bounded so long as the behavior of the higher derivatives are not too large. The problem is, how do you know how large the error is if you don't know the solution? Additionally, these techniques can be very expensive: if you were to use a very small value of *h*, you will have to iterate many times, so you will have an unnecessarily accurate answer that took far too long to calculate.

Adaptive Euler-Heun method

Here is an idea: suppose we estimate $y(t_k + h)$ with Euler's method, but then also estimate it with Heun's method:

$$\begin{aligned} s_0 &= f(t_k, y_k) \\ y_{k+1} &= y_k + h\, s_0 \\ s_1 &= f(t_0 + h, y_k + h\, s_0) \\ z_{k+1} &= y_0 + h\, 0.5(s_0 + s_1) \end{aligned}$$

If the actual answer is $y(t_k + h)$, then the error of each of these approximations is $|y_{k+1} - y(t_k + h)|$ and $|z_{k+1} - y(t_k + h)|$. If, however, z_{k+1} is much more accurate than y_{k+1}, then

$$\left|y_{k+1} - y(t_0 + h)\right| \approx \left|y_{k+1} - z_{k+1}\right| .$$

Thus, we will estimate how good the solution to Euler's method is by contrasting it with the solution to Heun's method.

For example, let us consider the rather simple IVP where $y^{(1)}(t) = y(t) - ty^2(t), y(0) = 1$. This has the exact solution $y(t) = \frac{1}{t - 1 + 2e^{-t}}$. If we use $h = 0.1$, we estimate the solution with Euler's method: $y_1 = 1.1$, or Heun's method $z_1 =$ 1.09895. The actual solution, to 16 decimal digits, is 1.099293901893687. We see that

$$\left|y_1 - y(0.1)\right| = \left|1.1 - y(0.1)\right| = 0.0007061 \text{ while } \left|z_1 - y(0.1)\right| = \left|1.09895 - y(0.1)\right| = 0.0003439 .$$

Observe now that $\left|y_1 - z_1\right| = \left|1.1 - 1.09895\right| = 0.00105$, which is in the ballpark of the actual error of Euler's method.

The next question is: what is the error we are willing to tolerate?

One issue with IVPs is that we may not know at which point we'd like to stop approximating our solution. Let us therefore assume we are willing to accept at most ε_{abs} per unit time. Thus, given an initial condition at $t = 0$ and we are willing to accept ε_{abs} per unit time, then in approximating $y(10)$, the error should be no more than $10\varepsilon_{\text{abs}}$. Alternatively, if we are quite certain we are estimating $y(20)$ and we want the error to be no greater than 10^{-5}, then we should use $\varepsilon_{\text{abs}} = 10^{-5} \div 20 = 5 \cdot 10^{-7}$.

Now, at any one point, we are given our current approximation (t_k, y_k), and we'd like to approximate the solution at time $t_k + h$, and we do so by using two techniques:

1. using Euler's method to calculate a first approximation $y_{k+1} = y_k + h\, f(t_k, y_k)$ and
2. using Heun's method to calculate a second approximation z_{k+1}.

We estimate the error of Euler's method by calculating $|y_{k+1} - z_{k+1}|$, but as we saw above, while it is *in the same ballpark* of the error, it is not necessarily a very precise approximation of the error. Thus, we will assume the actual error of Euler's method is within no larger that $2|y_{k+1} - z_{k+1}|$; that is, we are being rather cautious. Thus, we want to ensure that $2|y_{k+1} - z_{k+1}| \approx \varepsilon_{\text{abs}} h$, for

1. if $2|y_{k+1} - z_{k+1}| \ll \varepsilon_{\text{abs}} h$, our step size is likely too small—our approximation is too good, we can use a larger *h*, and
2. if $2|y_{k+1} - z_{k+1}| \gg \varepsilon_{\text{abs}} h$, our step size is likely too large—too much error, we should use a smaller *h*.

The question is, how much should we change the size of *h*? If we use one step of Euler's method, its error is O(h^2), so the error is approximately ch^2. Thus, $ch^2 \approx 2|y_{k+1} - z_{k+1}|$ under the assumption that $2|y_{k+1} - z_{k+1}|$ is either a reasonable or overestimation of the error. Now, we want to find an *h* so that

$$c(ah)^2 = \varepsilon_{\text{abs}}(ah),$$

however, we see that $c(ah)^2 = ca^2h^2 = (ch^2)a^2 \approx 2|y_{k+1} - z_{k+1}|a^2$. Thus, substituting this into the above equation, we have

$$2|y_{k+1} - z_{k+1}|a^2 = \varepsilon_{\text{abs}}(ah),$$

or, solving for *a*, we determine that the correct scaling factor is

$$a = \frac{\varepsilon_{\text{abs}} h}{2|y_{k+1} - z_{k+1}|}.$$

Thus, the ideal step size to achieve the required error is ah.

Now, if $a > 1$, this suggests our step size is too small, so we can use the calculated approximation of y_1 or z_1 and just increase our step size for the next iteration; however, if $a < 1$, this suggests that our step size is too large—the associated error is too large. Thus, our current approximation is too poor, so we should try again but this time using a smaller step size.

One small problem: The issue is, if we always use the optimal step size, then there is essentially a 50-50 chance that with the next step, the step size will be too large. Thus, instead, we always ensure that the step size is 90% of the optimal step size, so always let the new value of *h* be $0.9ah$.

Now, here is an interesting problem: We are using Heun's method to approximate the error of Euler's method. Should we be using y_{k+1} or z_{k+1} as our next approximation? As we are estimating the error of Euler's method, the default is that we should use y_{k+1}. Using z_{k+1} is somewhat questionable, as we don't know the error of that technique. If we were, this is a technique called *local extrapolation* and in this case, we will use this for Euler and Heun.

Yet anohter issue: because both y_{k+1} and z_{k+1} are approximations, it may by some fluke that $|y_{k+1} - z_{k+1}|$ may either seriously over estimate or underestimate the error in question. Thus, we should be very careful in adjusting h too much. Thus, we will adopt the following rule:

1. if $0.9a > 2$ then restrict the new value of h to be $2h$, and
2. if $0.9a < ½$ then restrict the new value of h to be $½h$.

One final issue: at a discontinuity in the ODE, no step size will ever be sufficiently small to ensure that the requisite error is satisfied. Consequently, it is necessary to ensure that there is some minimal $h_{\min}$ value that is to be used, and thus, if this minimal $h_{\min}$ value is used, the result will be accepted regardless of the calculation of a.

Our algorithm is as follows:

```
1. let $k \leftarrow 0$,
2. while $t_k \neq t_f$
    a. if $h \leq h_{\min}$, set $h = h_{\min}$.
    b. set $s_0 \leftarrow f(t_k, y_k)$,
    c. set $\tilde{y}_{k+1} \leftarrow y_k + hs_0$,
    d. set $s_1 \leftarrow f(t_k + h, y_k + hs_0)$,
    e. set $\tilde{z}_{k+1} \leftarrow y_k + h\frac{s_0 + s_1}{2}$,
    f. calculate $a \leftarrow 0.9\frac{h\varepsilon_{\text{abs}}}{2|y_1 - z_1|}$,
    g. if $a \geq 0.9$ or $h = h_{\min}$,
        i. set $y_{k+1} \leftarrow \tilde{y}_{k+1}$,
        ii. set $t_{k+1} \leftarrow t_k + h$,
        iii. set $k \leftarrow k + 1$.
    h. if $a \geq 2$, let $h \leftarrow 2h$,
       else if $a < ½$, let $h \leftarrow ½ ah$,
       else let $h \leftarrow ah$.
    i. if $t_k + h > t_f$, let $h \leftarrow t_f - t_k$ and let $t_k \leftarrow t_f$.
```

Note that we only move onto the next point if $a \geq 1$, otherwise, we try again with a smaller step size.

Now, we do have one problem here: recall that we are using floating-point numbers. In this case, it may happen that h is so small that when you actually calculate $t_{k+1} \leftarrow t_k + h$, it leaves t_k unchanged, even though $h \neq 0$.

Implementation issues

In all previous techniques, we were always aware of the number of steps, and thus we could always pre-allocate sufficient memory for the resulting vectors. For an adaptive technique, there could be arbitrarily many entries.

Now, you will recall from your algorithms and data structures course that if you are using a fixed-size array and you are required to increase the capacity of the array, you have two strategies:

1. increase the capacity of the array by a fixed amount and copy the old entries over, or
2. increase the capacity of the array by a scalar multiply (say, double the size) and copy the old entries over.

Neither of these is ideal, for in the first case, while the wasted memory is now $\Theta(1)$, the amortized run-time of inserting an entry into the array is $\Theta(n)$; while in the second case, the wasted memory is now $\Theta(n)$ while the amortized run-time is of inserting an entry into the array is $\Theta(1)$.

An alternative is to use a queue data structure where all operations are $\Theta(1)$ and where a reasonable implementation will ensure that there is only $\Theta(1)$ wasted memory. This could be achieved, for example, by having linked list of arrays storing the entries in the queue. Once one array is full, allocate another node in the linked list and start filling the new array. In our implementation, however, we will simply use the STL queue data structure.

Issues with queues, however, include that they are difficult to search, and therefore when the algorithm is finished, it will be necessary to copy them all back into a vector; however, this is reasonably straight-forward.

Implementation of the adaptive Euler-Heun method

Here is a C++ implementation of the adaptive Euler-Heun method.

```
#include <vector>
#include <tuple>
#include <cassert>

std::tuple< std::vector<double>, std::vector<double>, std::vector<double> >
  adaptive_euler_heun( double f(double, double),
         double t0, double y0, double tf,
         double h, double h_min, double eps_abs ) {
    assert( h > 0 );

    std::size_t k{0};

    std::queue<double>  q_t{};
    std::queue<double>  q_y{};
    std::queue<double> q_dy{};

    q_t.push( t0 );
    q_y.push( y0 );

    while ( t0 < tf ) {
        double s0{f( t0, q_y.back() )};
        q_dy.push( s0 );
        double t1, z1, a;
        bool using_h_min;

        do {
            using_h_min = (h <= h_min);

            if ( using_h_min ) {
                h = h_min;
            }

            t1 = t0 + h;

            if ( t1 > tf ) {
                t1 = tf;
                h = tf - t0;
            }

            // Approximation using Euler's method
            double y1{q_y.back() + h*s0};

            double s1{f( t0 + h, y1 )};
            // Approximation using Heun's method
            z1 = q_y.back() + h*(s0 + s1)/2.0;

            a = 0.9*eps_abs*h/(2.0*std::abs( z1 - y1 ));

            if ( a >= 2.0 ) {
                h = 2.0*h;
            } else if ( a <= 0.5 ) {
                h = 0.5*h;
            } else {
                h = a*h;
            }
        } while ( (a < 0.9) && !using_h_min );

        q_t.push( t1 );
        q_y.push( z1 );  // Store result from Heun's method

        t0 = t1;
    }

    q_dy.push( f( t0, q_y.back() ) );

    assert( (q_t.size() == q_y.size()) && (q_t.size() == q_dy.size()) );

    std::size_t n{q_t.size()};

    std::vector<double>  t(n);
    std::vector<double>  y(n);
    std::vector<double> dy(n);

    // Copy the approximate values from the queues into the vectors
    for ( std::size_t k{0}; k < n; ++k ) {
         t[k] = q_t.front();
         y[k] = q_y.front();
        dy[k] = q_dy.front();
        q_t.pop();
        q_y.pop();
        q_dy.pop();
    }

    return std::make_tuple( t, y, dy );
}
```

Examples of the adaptive Euler-Heun method

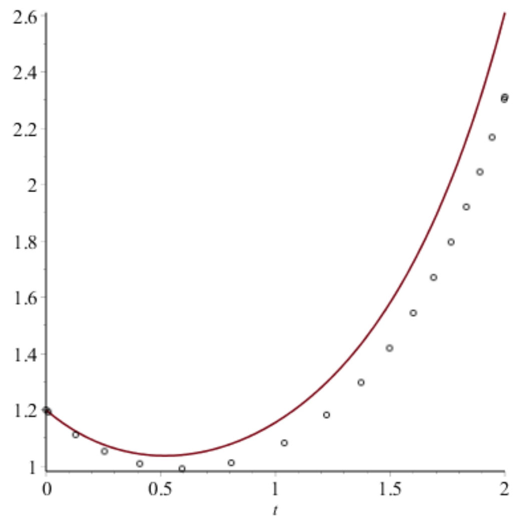
Suppose we want to approximate the solution to the initial-value problem

$$y^{(1)}(t) = (t-1)y(t) + 0.5$$
$$y(0) = 1.2$$

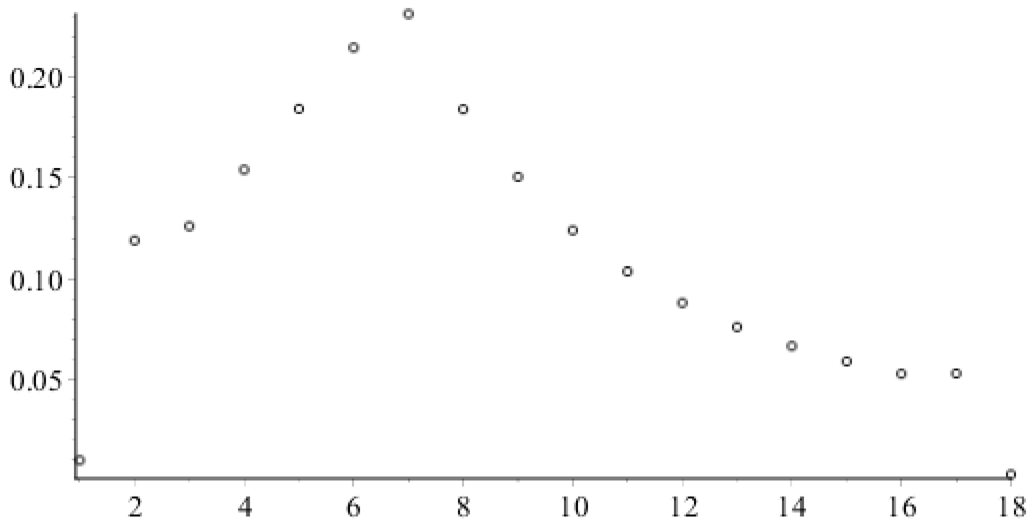
where we want to approximate $y(2)$ with an initial value of $h = 0.01$ using the Euler and Heun methods. The maximum error we will be willing to accept is $\varepsilon_{\text{abs}} = 0.5$.

k	t_k	y_k
0	0	1.2
1	0.01	1.193
2	0.1288589540413766	1.11204873217104
3	0.2548170732166789	1.053005700603626
4	0.4087838533621759	1.009174149730043
5	0.5930116274660248	0.9913703680442406
6	0.807602770909739	1.012083517937175
7	1.038894018000716	1.082691632357804
8	1.222883273641074	1.182434089652769
9	1.373348562334614	1.297321075521263
10	1.49726426362119	1.419297862678912
11	1.600916695266546	1.544278451617767
12	1.689074377936684	1.67016609767054
13	1.765225412359968	1.795881454192293
14	1.831914480642036	1.920873715601325
15	1.8910172390656	2.044871459957359
16	1.943937625153487	2.167753428493952
17	1.997044877792072	2.302976390969048
18	2	2.311239417439521

Compared to the correct solution $y(2.0) = 2.610686134642448$, the error is 0.2994, which is on the order of the maximum allowable error. Plotting the approximate points versus the exact solution, we see that the approximation shares the behavior of the actual solution:

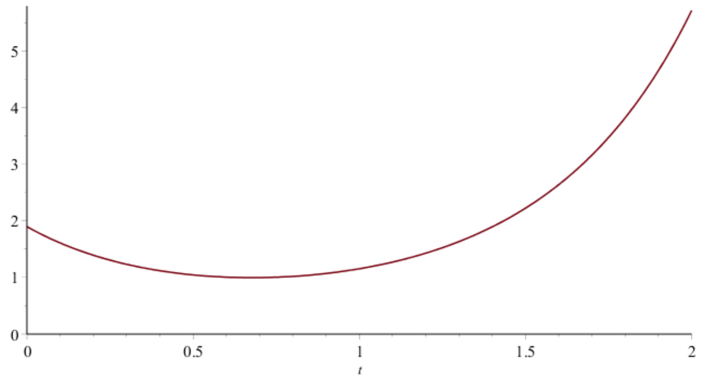


There are 19 points, so 18 intervals, and looking at the width of consecutive intervals, we see that the width of the intervals first increases, and then begins to decrease.



Recall that the error of Euler's method is proportional to the second derivative, so the larger the second derivative, the larger the error, and so we must use a correspondingly smaller step size:

If you plot the second derivative of the solution, you will see that the smallest step size corresponds with where the second derivative is at a minimum and as the second derivative increases, the step size increases:



We will now look at two variations:

1. First, we note that the allowed error was $\varepsilon_{\text{abs}} = 0.5$, but the actual error was significantly smaller.
2. Next, we will see what happens to the final error when we use $\varepsilon_{\text{abs}} = 0.25$.

Next, we now divide the requisite absolute error by half, to 0.25:

k	t_k	y_k
0	0	1.2
1	0.01	1.193
2	0.0694294770206883	1.15252436608552
3	0.131013123781153	1.117267407951197
4	0.1991939554783665	1.085161687381958
5	0.2744770430436135	1.057381924036714
6	0.3576328179438272	1.035166454621805
7	0.4491711422787923	1.02006657106115
8	0.5490797141597694	1.013884018411813
9	0.6564300675588123	1.018480667000217
10	0.7690232211341611	1.035378722481034
11	0.8833739280709005	1.065207282518167
12	0.9953190574906565	1.10727279844303
13	1.101106989932274	1.159618457313987
14	1.198315822126041	1.219620174626288
15	1.276885778999757	1.277908866773209
16	1.355831295868995	1.345315295753892
17	1.419636489931812	1.407761776396204
18	1.483701974675043	1.477641089235152
19	1.541830729763125	1.548252291860094
20	1.594503623505865	1.618775537809694
21	1.642493843305843	1.688954895856688
22	1.686467705751836	1.758659761469926
23	1.726977525130752	1.827820682565525
24	1.764480004198793	1.896404637104159
25	1.799354101956823	1.964400877390409
26	1.831916021433189	2.031812253028917
27	1.862431309911914	2.098649802898627
28	1.891124484209522	2.164929351195461
29	1.918186651087179	2.230669350203172
30	1.943781543780426	2.295889508913987
31	1.968050320464888	2.360609922934814
32	1.991115397938426	2.424850527435741
33	2	2.450645252087435

Compared to the correct solution $y(2.0) = 2.610686134642448$, the error is 0.1600, which approximately half the previous error when we required an error of 0.5.

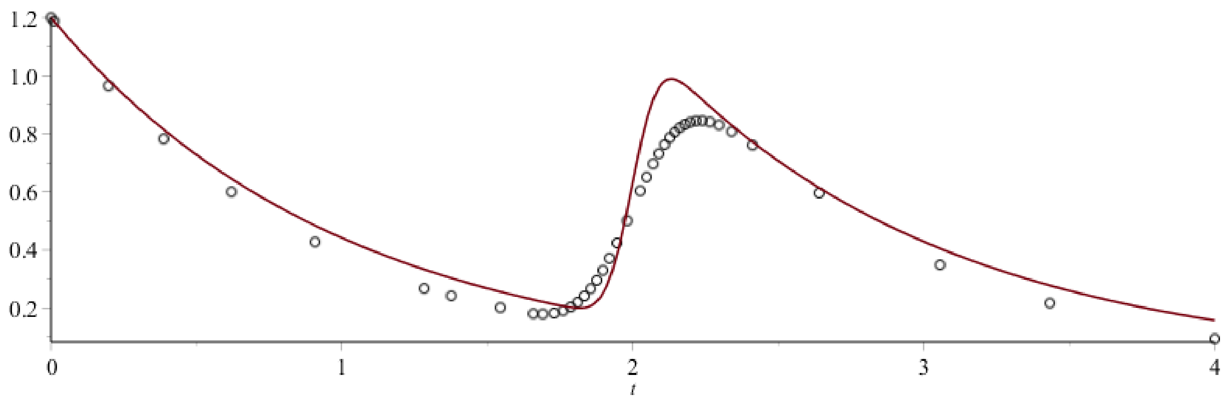
Let’s now look at a more demonstrative example. The differential equation

$$\frac{d}{dt}y(t)+y(t)=\frac{10}{\sqrt{\pi}}e^{-100(t-2)^2}$$
$$y(0)=1.2$$

describes a decaying exponential where a forcing function with unit impulse affects the system at approximately time t = 2. An approximation to this using the adaptive Euler-Heun technique with a maximum error of 1:

k	t_k	y_k
0	0	1.2
1	0.01	1.188
2	0.1975000000001595	0.9652499999998105
3	0.3868939393940987	0.7824374999998466
4	0.6199941724943774	0.6000511363634831
5	0.9075570768797683	0.4274986888110457
6	1.282525119375567	0.2672003422981639
7	1.375630045889661	0.2423233506472151
8	1.544393696206725	0.2014558427151085
9	1.657563922131868	0.1804369061874284
10	1.690359470036848	0.1794516723021727
11	1.729099717872922	0.1824441794448262
12	1.760141737412585	0.1907623223300844
13	1.786408228476261	0.2033370360692868
14	1.809708467988044	0.2196092758817238
15	1.833228214176173	0.2412773669474623
16	1.854840163838437	0.2664798007798025
17	1.875695954669577	0.2956631406054607
18	1.896824904877933	0.3299212630805546
19	1.919408773055457	0.3712923787380395
20	1.945470583961783	0.4241160660616655
21	1.980400687350985	0.5007784635455076
22	2.025237606953853	0.6035988747248763
23	2.04709159839882	0.6510829136580895
24	2.069564912104507	0.6964279168462464
25	2.089134996127231	0.7317142052884810
26	2.108968184982086	.7630716220221758
27	2.126981624957125	0.7870894432135407
28	2.144203041795319	0.8059982139043934
29	2.161300084545617	0.8208956055497342
30	2.178767674128301	0.8322690360325411
31	2.197088932385955	0.8402680963595561
32	2.216843828978687	0.844770799995231
33	2.23884808632352	0.8453413452841601
34	2.264417393850828	0.8410533703544706
35	2.296006559720823	0.8300027705026978
36	2.339180586605646	0.807791963700681
37	2.411474947594179	0.7608860930803515
38	2.640154441259184	0.5962485283721206
39	3.055765119212582	0.3484829371432204
40	3.43334076657958	0.216904266555858
41	4	0.0939934611436971

Plotting a solution, we see that when the concavity of the solution begins to change rapidly, the step size begins to quickly adjust, and yet when the concavity is closer to zero, the step size subsequently increases in size.



Please note, the acceptable error in this example was $\varepsilon_{\text{abs}} = 1$, so the approximation is not required to be very precise; however, we will now look at other examples of adaptive techniques that result in significantly more precise approximations with significantly fewer steps.

Adaptive Runge-Kutta-Fehlberg method

Fehlberg came up with a better adaptive technique where he found two approximations: one O(h^5) and the other O(h^6) for one step. The algorithm then used the higher-order approximation to estimate the error of the lower-order approximation. The choice of coefficients in the algorithm were such that the error of the 5th-order approximation is minimized, and thus it is necessary to use the 5th-order approximation in the result.

First, you calculate six samples of the slope throughout the interval $[t_k, t_k + h]$:

$$\begin{aligned}
s_0 &= f\left(t_k, y_k\right)\\
s_1 &= f\left(t_k + \tfrac{1}{4}h, y_k + \tfrac{1}{4}h\left(s_0\right)\right)\\
s_2 &= f\left(t_k + \tfrac{3}{8}h, y_k + \tfrac{3}{8}h\left(\tfrac{1}{4}s_0 + \tfrac{3}{4}s_1\right)\right)\\
s_3 &= f\left(t_k + \tfrac{12}{13}h, y_k + \tfrac{12}{13}h\left(\tfrac{161}{169}s_0 - \tfrac{600}{169}s_1 + \tfrac{608}{169}s_2\right)\right)\\
s_4 &= f\left(t_k + 1\cdot h, y_k + 1\cdot h\left(\tfrac{439}{216}s_0 - 8s_1 + \tfrac{3680}{513}s_2 - \tfrac{845}{4104}s_3\right)\right)\\
s_5 &= f\left(t_k + \tfrac{1}{2}h, y_k + \tfrac{1}{2}h\left(-\tfrac{16}{27}s_0 + 4s_1 - \tfrac{7088}{2565}s_2 + \tfrac{1859}{2052}s_3 - \tfrac{11}{20}s_4\right)\right)
\end{aligned}$$

You will note that each linear combination of the previously calculated slopes is weighted average of those slopes, meaning that the coefficients sum to one. Given these six slopes, we now calculate two approximations of the next point:

$$\begin{aligned}
y_{k+1} &= y_k + h\left(\tfrac{25}{216}s_0 + \tfrac{1408}{2565}s_2 + \tfrac{2197}{4104}s_3 - \tfrac{1}{5}s_4\right)\\
z_{k+1} &= y_k + h\left(\tfrac{16}{135}s_0 + \tfrac{6656}{12825}s_2 + \tfrac{28561}{56430}s_3 - \tfrac{9}{50}s_4 + \tfrac{2}{55}s_5\right)
\end{aligned}$$

As before, we estimate the error of the less-accurate estimator by calculating and we want to ensure that $2\left|y_{k+1} - z_{k+1}\right| \approx \varepsilon_{\text{abs}}h$. The less accurate approximation is O(h^5), so the error is approximately ch^5. Thus, $ch^5 \approx 2\left|y_{k+1} - z_{k+1}\right|$ under the assumption that $2\left|y_{k+1} - z_{k+1}\right|$ is either a reasonable or overestimation of the error. Now, we want to find an h so that

$$c\left(ah\right)^5 = \varepsilon_{\text{abs}}ah,$$

however, we see that $c\left(ah\right)^5 = ca^5h^5 = \left(ch^5\right)a^5 \approx 2\left|y_{k+1} - z_{k+1}\right|a^5$. Thus, substituting this into the above equation, we have

$$2\left|y_{k+1} - z_{k+1}\right|a^5 = \varepsilon_{\text{abs}}ah,$$

or, solving for a, we determine that the correct scaling factor is

$$a = \sqrt[4]{\frac{\varepsilon_{\text{abs}}h}{2\left|y_{k+1} - z_{k+1}\right|}}.$$

Thus, the ideal step size to achieve the required error is

$$ah = h\sqrt[4]{\frac{\varepsilon_{\text{abs}}h}{2\left|y_{k+1} - z_{k+1}\right|}}.$$

The coefficients in calculating the slopes and the two estimators ensured that error of y_{k+1} is minimized, and thus in this case we must use y_{k+1} even though z_{k+1} has a higher-order error.

Implementation of the adaptive Runge-Kutta-Fehlberg method

Here is a C++ implementation of the adaptive Runge-Kutta-Fehlberg method.

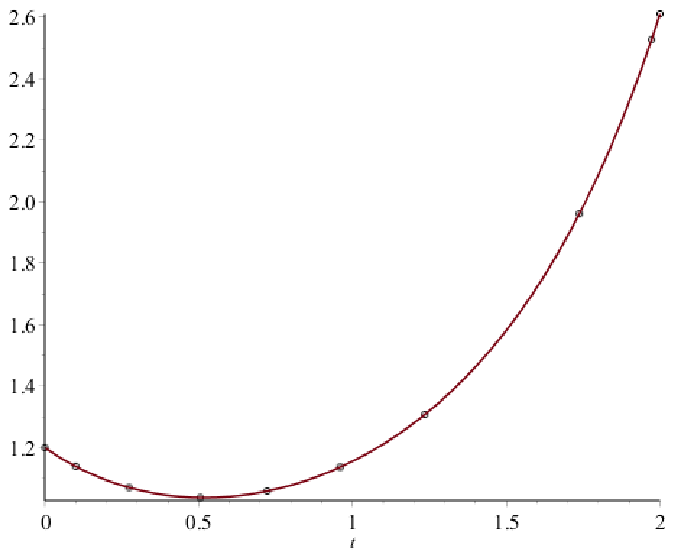
```
std::tuple< std::vector<double>, std::vector<double>, std::vector<double> >
  adaptive_fehlberg( double f(double, double),
                     double t0, double y0, double tf,
                     double h, double h_min, double eps_abs ) {
    std::size_t const DIM{6};
    double step[DIM - 1]{1.0/4.0, 3.0/8.0, 12.0/13.0, 1.0, 1.0/2.0};

    double tableau[DIM - 1][DIM - 1]{
        {  1.0},
        {  1.0/4.0,      3.0/4.0},
        {161.0/169.0, -600.0/169.0, 608.0/169.0},
        {439.0/216.0,   -8.0,      3680.0/513.0,  -845.0/4104.0},
        {-16.0/27.0,     4.0,     -7088.0/2565.0, 1859.0/2052.0, -11.0/20.0}
    };

    double y_coeff[DIM]{25.0/216.0, 0.0, 1408.0/2565.0,   2197.0/4104.0,  -1.0/5.0,  0.0};
    double z_coeff[DIM]{16.0/135.0, 0.0, 6656.0/12825.0, 28561.0/56430.0, -9.0/50.0, 2.0/55.0};

    assert( h > 0 );

    std::size_t k{0};

    std::queue<double>  q_t{};
    std::queue<double>  q_y{};
    std::queue<double> q_dy{};

    q_t.push( t0 );
    q_y.push( y0 );

    while ( t0 < tf ) {
        double s[DIM]{f( t0, q_y.back() )};
        q_dy.push( s[0] );
        double t1, y1, z1, a;
        bool using_h_min;

        do {
            using_h_min = (h <= h_min);

            if ( using_h_min ) {
                h = h_min;
            }

            t1 = t0 + h;

            if ( t1 > tf ) {
                t1 = tf;
                h = tf - t0;
            }

            for ( std::size_t i{0}; i < DIM - 1; ++i ) {
                double slope{0.0};

                for ( std::size_t j{0}; j <= i; ++j ) {
                    slope += tableau[i][j]*s[j];
                }

                s[i + 1] = f( t0 + h*step[i], q_y.back() + h*step[i]*slope );
            }

            double slope_y{0.0};
            double slope_z{0.0};
            for ( std::size_t i{0}; i < DIM; ++i ) {
                slope_y += y_coeff[i]*s[i];
                slope_z += z_coeff[i]*s[i];
            }

            y1 = q_y.back() + h*slope_y;
            z1 = q_y.back() + h*slope_z;
            a = 0.9*std::pow( eps_abs*h/( 2.0*std::abs( z1 - y1 )), 0.25 );

            if ( a >= 2.0 ) {
                h = 2.0*h;
            } else if ( a < 0.5 ) {
                h = 0.5*h;
            } else {
                h = a*h;
            }
        } while ( (a < 0.9) && !using_h_min );

        q_t.push( t1 );
        // We cannot use the higher-order approximation, as the higher order
        // approximation may be worse due to the magnitude of the coefficients
        //  - Use Dormand-Prince if you want to use
        //    local extrapolation using z1 instead of y1
        q_y.push( y1 );

        t0 = t1;
    }

    q_dy.push( f( t0, q_y.back() ) );

    assert( (q_t.size() == q_y.size()) && (q_t.size() == q_dy.size()) );

    std::size_t n{q_t.size()};

    std::vector<double>  t(n);
    std::vector<double>  y(n);
    std::vector<double> dy(n);

    // Remove the approximations from the queue and place them into the vector
    for ( std::size_t k{0}; k < n; ++k ) {
         t[k] = q_t.front();
         y[k] = q_y.front();
        dy[k] = q_dy.front();
        q_t.pop();
        q_y.pop();
        q_dy.pop();
    }

    return std::make_tuple( t, y, dy );
}
```

Examples of the Runge-Kutta-Fehlberg method

Solving the same problem we did previously using the adaptive Euler-Heun methods, we now look at the Runge-Kutta-Fehlberg technique for estimating the solution to the initial-value problem with a maximum error of 10^{-5}.

k	t_k	y_k
0	0	1.2
1	0.1	1.138985744767423
2	0.2736395192719883	1.070171640922223
3	0.5053336491504283	1.037524176584444
4	0.7228127565721129	1.058711631263671
5	0.9602290555275999	1.136668927595979
6	1.235119765873569	1.307789803793384
7	1.737754136156426	1.960835819016158
8	1.97235401364141	2.526888512670361
9	2	2.610742963367295

Compared to the correct solution $y(2.0) = 2.610686134642448$, the error is 0.00005683, which is in the same ballpark as the maximal allowed error; however, it is still greater by a factor of five. Plotting the approximate points versus the exact solution, we have a reasonable fit as is shown here:



Now, we know that the error drops by O(h^4) and so we will divide the required error by 16. Thus, it should require approximately twice as many points:

k	t_k	y_k
0	0	1.2
1	0.1	1.138985744767423
2	0.1921103310200981	1.097150992823859
3	0.3082170093714797	1.061332359840794
4	0.426410719079129	1.041868948634119
5	0.5426357255049833	1.037751773235398
6	0.6615262018211252	1.047929668852698
7	0.7857451834504847	1.073699133650115
8	0.919015775598463	1.118864175892402
9	1.069686657714534	1.19335833288605
10	1.239885737602207	1.311653362644173
11	1.369173481331456	1.430351038803992
12	1.496174181903581	1.576513441826428
13	1.611930056625299	1.740776585800352
14	1.721672898665665	1.929923549464155
15	1.831478642603781	2.159082627210289
16	1.954403153544082	2.474612855837278
17	2	2.610686344678339

The error of approximating $y(2.0) = 2.610686134642448$, after 17 steps, is 0.0000002100, which is significantly better than the previous approximation and below our required error of 0.000000625. Because the lower-order approximation is more precise, if the step size is not sufficiently small to sufficiently reduce the error of the higher-order approximation, our approximation of the error of the lower-order approximation will be seriously flawed. Thus, instead, we will look at the next algorithm: the Dormand-Prince algorithm where the coefficients are chosen to ensure the higher-order approximation has the lowest coefficients.

Next, we now look at an adaptive technique for estimating the solution to the initial-value problem with a maximum error of 0.25, but this time we will continue to use Heun's method to approximate y_k.

k	t_k	y_k
0	0	1.2
1	0.01	1.19309465
2	0.0694294770206883	1.155839743047291
3	0.1310069940291906	1.123535963443275
4	0.1989408015299992	1.094649098899352
5	0.2736858500275396	1.070301731956509
6	0.3558997676019153	1.05171856028585
7	0.4459747660330011	1.04040346305198
8	0.5438000551793166	1.038073667340005
9	0.6484257577345817	1.046469810012483
10	0.757778449354733	1.066979559813892
11	0.8686732243346846	1.100132312575068
12	0.9773418953102755	1.145214645177801
13	1.080356625382221	1.200327595317542
14	1.175429096725381	1.262931610756725
15	1.252923479753652	1.323188243732917
16	1.330755000870755	1.392995468582204
17	1.394015942531341	1.457317608655455
18	1.457519925297492	1.529463485478861
19	1.515248100620187	1.602301988658898
20	1.567631241175899	1.674970893700697
21	1.615407128302798	1.747213044389893
22	1.659219250018435	1.818901500512818
23	1.699604849147442	1.889971516970396
24	1.737010480529295	1.960395061836511
25	1.771808135137865	2.030166327622896
26	1.804309134776486	2.09929287772241
27	1.834775539000568	2.167790152337993
28	1.863429324445504	2.235678032233944
29	1.890459713771624	2.302978679898203
30	1.916029018468141	2.369715183081677
31	1.940277306369811	2.435910707704172
32	1.963326145352486	2.501587976973898
33	1.985281621506243	2.566768960767788
34	2	2.611956892741668

Compared to the correct solution $y(2.0) = 2.610686134642448$, the error is 0.001271, which approximately half the previous error when we required an error of 0.5 while still using Heun's method.

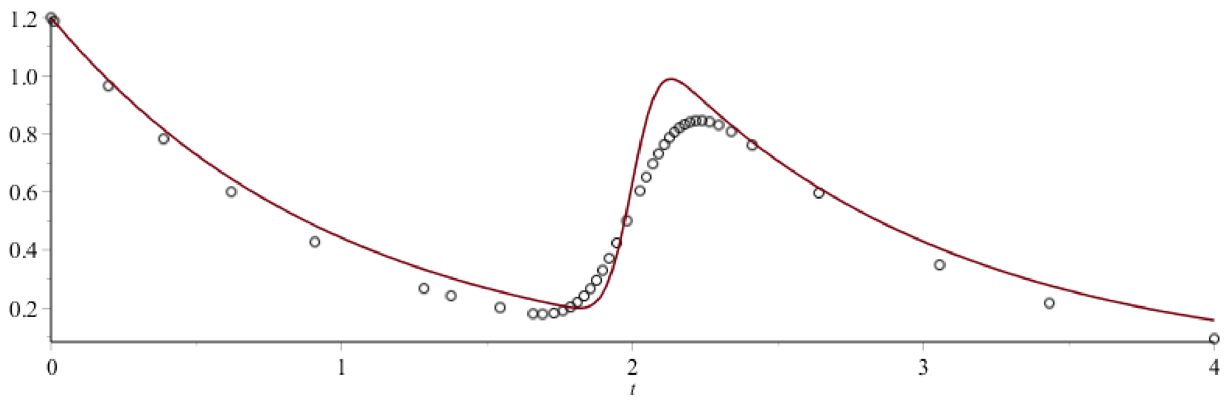
Let’s now look at a more demonstrative example. The differential equation

$$\frac{d}{dt}y(t)+y(t)=\frac{10}{\sqrt{\pi}}e^{-100(t-2)^2}$$
$$y(0)=1.2$$

describes a decaying exponential where a forcing function with unit impulse affects the system at approximately time t = 2. An approximation to this using the adaptive Euler-Heun technique where we use Heun’s approximation for y_k is:

k	t_k	y_k
0	0	1.2
1	0.01	1.188
2	0.1975000000001595	0.9652499999998105
3	0.3868939393940987	0.7824374999998466
4	0.6199941724943774	0.6000511363634831
5	0.9075570768797683	0.4274986888110457
6	1.282525119375567	0.2672003422981639
7	1.375630045889661	0.2423233506472151
8	1.544393696206725	0.2014558427151085
9	1.657563922131868	0.1804369061874284
10	1.690359470036848	0.1794516723021727
11	1.729099717872922	0.1824441794448262
12	1.760141737412585	0.1907623223300844
13	1.786408228476261	0.2033370360692868
14	1.809708467988044	0.2196092758817238
15	1.833228214176173	0.2412773669474623
16	1.854840163838437	0.2664798007798025
17	1.875695954669577	0.2956631406054607
18	1.896824904877933	0.3299212630805546
19	1.919408773055457	0.3712923787380395
20	1.945470583961783	0.4241160660616655
21	1.980400687350985	0.5007784635455076
22	2.025237606953853	0.6035988747248763
23	2.04709159839882	0.6510829136580895
24	2.069564912104507	0.6964279168462464
25	2.089134996127231	0.7317142052884810
26	2.108968184982086	.7630716220221758
27	2.126981624957125	0.7870894432135407
28	2.144203041795319	0.8059982139043934
29	2.161300084545617	0.8208956055497342
30	2.178767674128301	0.8322690360325411
31	2.197088932385955	0.8402680963595561
32	2.216843828978687	0.844770799995231
33	2.23884808632352	0.8453413452841601
34	2.264417393850828	0.8410533703544706
35	2.296006559720823	0.8300027705026978
36	2.339180586605646	0.807791963700681
37	2.411474947594179	0.7608860930803515
38	2.640154441259184	0.5962485283721206
39	3.055765119212582	0.3484829371432204
40	3.43334076657958	0.216904266555858
41	4	0.0939934611436971

Plotting a solution, we see that when the concavity of the solution begins to change rapidly, the step size begins to quickly adjust, and yet when the concavity is closer to zero, the step size subsequently increases in size.



Please note, the acceptable error in this example was $\varepsilon_{\text{abs}} = 1$, so the approximation is not required to be very precise; however, we will now look at other examples of adaptive techniques that result in significantly more precise approximations with significantly fewer steps.

Adaptive Dormand-Prince method

Dormand and Prince came up with a better approximation in 1980. Like the Fehlberg, it found one O(h^5) and the other O(h^6); however, the coefficients were chosen to ensure that the O(h^6) approximation had minimal error. In this case, while we use the 6th-order approximation to determine the error of the 5th-order approximation, once we have the appropriate step size, it is more appropriate to use the 6th-order approximation (unlike Felhberg).

Implementation of the adaptive Dormand-Prince method

Here is a C++ implementation of the adaptive Euler-Heun method.

```
#include <vector>
#include <tuple>
#include <cassert>

// Approximate a solution to y'(t) = f(t, y(t))
std::tuple< std::vector<double>, std::vector<double>, std::vector<double> >
  adaptive_dormand_prince( double f( double, double ),
         double t0, double y0, double tf,
         double h, double using_h_min, double eps_abs ) {
    std::size_t const DIM{7};
    double step[DIM - 1]{1.0/5.0, 3.0/10.0, 4.0/5.0, 8.0/9.0, 1.0, 1.0};

    double tableau[DIM - 1][DIM - 1]{
        {   1.0},
        {   1.0/4.0,        3.0/4},
        {  11.0/9.0,      -14.0/3.0,      40.0/9.0},
        {4843.0/1458.0, -3170.0/243.0,  8056.0/729.0,  -53.0/162.0},
        {9017.0/3168.0,  -355.0/33.0,  46732.0/5247.0,  49.0/176.0, -5103.0/18656.0},
        {  35.0/384.0,      0.0,         500.0/1113.0, 125.0/192.0, -2187.0/6784.0,  11.0/84.0}
    };

    double y_coeff[DIM]{
       5179.0/57600.0, 0.0, 7571.0/16695.0, 393.0/640.0, -92097.0/339200.0, 187.0/2100.0, 1.0/40.0
    };

    double z_coeff[DIM]{
         35.0/384.0,   0.0,  500.0/1113.0,  125.0/192.0,  -2187.0/6784.0,    11.0/84.0,    0.0
    };

    assert( h > 0 );

    std::size_t k{0};

    std::queue<double>  q_t{};
    std::queue<double>  q_y{};
    std::queue<double> q_dy{};

    q_t.push( t0 );
    q_y.push( y0 );

    while ( t0 < tf ) {
        double s[DIM]{f( t0, q_y.back() )};
        q_dy.push( s[0] );
        double t1, z1, a;
        bool using_h_min;

        do {
            using_h_min = (h <= h_min);

            if ( using_h_min ) {
                h = h_min;
            }

            t1 = t0 + h;

            if ( t1 > tf ) {
                t1 = tf;
                h = tf - t0;
            }
            for ( std::size_t i{0}; i < DIM - 1; ++i ) {
                double slope{0.0};

                for ( std::size_t j{0}; j <= i; ++j ) {
                    slope += tableau[i][j]*s[j];
                }

                s[i + 1] = f( t0 + h*step[i], q_y.back() + h*step[i]*slope );
            }


            double slope_y{0.0};
            double slope_z{0.0};

            for ( std::size_t i{0}; i < DIM; ++i ) {
                slope_y += y_coeff[i]*s[i];
                slope_z += z_coeff[i]*s[i];
            }

            double y1{q_y.back() + h*slope_y};
            z1 = q_y.back() + h*slope_z;
            a = 0.9*std::pow( eps_abs*h/(2.0*std::abs( z1 - y1 )), 0.25 );

            if ( a >= 2.0 ) {
                h = 2.0*h;
            } else if ( a <= 0.5 ) {
                h = 0.5*h;
            } else {
                h = a*h;
            }
        } while ( (a < 0.9) && !using_h_min );

        q_t.push( t1 );
        q_y.push( z1 );

        t0 = t1;
    }

    q_dy.push( f( t0, q_y.back() ) );

    assert( (q_t.size() == q_y.size()) && (q_t.size() == q_dy.size()) );

    std::size_t n{q_t.size()};

    std::vector<double>  t(n);
    std::vector<double>  y(n);
    std::vector<double> dy(n);


    for ( std::size_t k{0}; k < n; ++k ) {
         t[k] = q_t.front();
         y[k] = q_y.front();
        dy[k] = q_dy.front();
        q_t.pop();
        q_y.pop();
        q_dy.pop();
    }

    return std::make_tuple( t, y, dy );
}
```

Backward Euler's (implicit) method

How can we approximate a solution? We have an initial value, let us say $y(t_0) = y_0$. Let us now consider the Tayler series at the point $t_0 + h$:

$$y(t_0) = y(t_0 + h) - y^{(1)}(t_0 + h)h + \frac{1}{2}y^{(2)}(\tau)h^2 .$$

In this case, we have

$$y(t_0 + h) = y(t_0) + y^{(1)}(t_0 + h)h - \frac{1}{2}y^{(2)}(\tau)h^2 ,$$

but now, substituting what we know from the initial-value problem, we have

$$y(t_0 + h) = y_0 + f(t_0 + h, y(t_0 + h))h - \frac{1}{2}y^{(2)}(\tau)h^2 .$$

Thus, we want to find a point y_1 such that

$$y_1 = y_0 + f(t_0 + h, y_1)h .$$

This is no longer an explicit formula, for we must now solve for y_1. Notice that we have something rather beautiful here: this is of the form $x = g(x)$ where we are solving for x, in which case, this says we can use the fixed-point theorem and simply iterate:

.

For example, let us return to the IVP:

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}y(t) &= y(t)(1 - y(t))\\ y(0) &= 0.1\end{aligned}$$

Thus, if $h = 0.5$, we have the problem:

$$y_1 = 0.1 + f(0.5, y_1)0.5$$

which means we are searching for a root of

$$y_1 = 0.1 + y_1(1 - y_1)0.5 .$$

This is a quadratic, but if we use y_0 as an initial condition for fixed-point iteration, this quickly converges to the root 0.1708203932499369, and thus this becomes our approximation of $y(0.5)$.

Next, approximating $y(1)$ requires us to solve

$$0.1708203932499369 - y_2 + f(1.0, y_2)0.5 = 0$$

or

$$y_2 = 0.1708203932499369 + y_2\left(1 - y_2\right)0.5 = 0$$

Again, using fixed-point iteration with y_1 as an initial value, we have that $y_2 = 0.2691818942876086$.

Implementation of the backward Euler's method

Here is a C++ implementation of the backward Euler's method. The difference between this implementation and the implementation of Euler's method is highlighted in red.

```
std::tuple< std::vector<double>, std::vector<double>, std::vector<double> >
  backward_euler( double f(double, double),
         double t0, double y0, double tf,
         std::size_t n, double eps_abs, unsigned int max_iterations ) {
    assert( n > 0 );

    double h{(tf - t0)/n};

    std::vector<double>  t(n + 1);
    std::vector<double>  y(n + 1);
    std::vector<double> dy(n + 1);

    y[0] = y0;

    for ( std::size_t k{0}; k < n; ++k ) {
         t[k] = t0 + k*h;
        dy[k] = f( t[k], y[k] );

        unsigned int iterations{0};

        // The initial approximation for fixed-point iteration
        double y_curr{y[k]};
        double y_prev{};

        do {
            if ( iterations > max_iterations ) {
                throw std::runtime_error(
                    "The backward Euler's method did not find a solution" );
            }

            // The iteration process
            y_prev = y_curr;
            y_curr = y[k] + h*f(t[k] + h, y_curr);
            ++iterations;
        } while ( std::abs( y_prev - y_curr ) > eps_abs );

        y[k + 1] = y_curr;
    }

     t[n] = tf;
    dy[n] = f( tf, y[n] );

    return std::make_tuple( t, y, dy );
}
```

Example of the backward Euler's method

Suppose we want to approximate the solution to the initial-value problem

$$\begin{aligned} y^{(1)}(t) &= (t-1)y(t)+0.5 \\ y(0) &= 1.2 \end{aligned}$$

where we want to approximate $y(2)$ with h = 1.0, 0.5 and 0.25. Here,

$$f(t,y)=(t-1)y+0.5\,.$$

First, we note that this does have a solution:

$$y(t)=0.05\left(5\sqrt{2}\sqrt{\pi}\mathrm{e}^{0.5}\left(\operatorname{erf}\left(0.5\sqrt{2}\right)+\operatorname{erf}\left(0.5\sqrt{2}(t-1)\right)\right)+24\right)\mathrm{e}^{0.5t(t-2)}$$

This can be implemented as the C++ function

```
double y( double t ) {
    double const SQRT_PI{std::sqrt( std::acos( -1.0 ) )};
    double const ROOT_2_5{std::sqrt( 2.0 )*0.5};

    return 0.05*(10.0*ROOT_2_5*SQRT_PI*std::exp( 0.5 )*(
        std::erf( ROOT_2_5 ) + std::erf( ROOT_2_5*(t - 1) )
    ) + 24.0)*exp( 0.5*t*(t - 2.0) );
}
```

This solution evaluated at this point is $y(2.0)$ = 2.610686134642448.

Applying the backward Euler's method with h = 1, we have must solve

$$\begin{aligned} y_1 &= y_0+h\cdot f(t_1,y_1) \\ y_1 &= 1.2+1\cdot f(1,y_1) \\ &= 1.2+1\cdot\left((1-1)y_1+0.5\right) \end{aligned}$$

The solution for this is y_1 = 1.7. Next,

$$\begin{aligned} y_2 &= y_1+h\cdot f(t_2,y_2) \\ y_2 &= 1.7+1\cdot f(2,y_2) \\ &= 1.7+1\cdot\left((2-1)y_2+0.5\right) \end{aligned}$$

Now, this has no solution, so in essence, our value of h is too large.

Thus, let us use h = 0.5, so we find:

$$\begin{aligned} y_1 &= y_0 + h \cdot f\left(t_1, y_1\right) \\ y_1 &= 1.2 + 0.5 \cdot f\left(0.5, y_1\right) \\ &= 1.2 + 0.5 \cdot \left(\left(0.5 - 1\right) y_1 + 0.5\right) \end{aligned}$$

which has the exact solution y_1 = 1.16. Next, finding y_2:

$$\begin{aligned} y_2 &= y_1 + h \cdot f\left(t_2, y_2\right) \\ y_2 &= 1.16 + 0.5 \cdot f\left(1, y_2\right) \\ &= 1.16 + 0.5 \cdot \left(\left(1 - 1\right) y_2 + 0.5\right) \end{aligned}$$

which has the solution y_2 = 1.41. Next, we find y_3:

$$\begin{aligned} y_3 &= y_2 + h \cdot f\left(t_3, y_3\right) \\ y_3 &= 1.41 + 0.5 \cdot f\left(1.5, y_3\right) \\ &= 1.41 + 0.5 \cdot \left(\left(1.5 - 1\right) y_3 + 0.5\right) \end{aligned}$$

which has the solution $y_3 = 2.21\bar{3}$. Finally, we find y_4:

$$\begin{aligned} y_4 &= y_3 + h \cdot f\left(t_4, y_4\right) \\ y_4 &= 2.21\bar{3} + 0.5 \cdot f\left(2, y_4\right) \\ &= 2.21\bar{3} + 0.5 \cdot \left(\left(2 - 1\right) y_4 + 0.5\right) \end{aligned}$$

which has the solution $y_4 = 4.92\bar{6}$.

We will now use $\varepsilon_{\text{abs}} = 10^{-15}$ in our use of fixed-point iteration. Because we anticipate the number of function calls, this technique will no doubt require significantly more function calls than any previous method.

t	$h = 0.5$	$h = 0.25$	$h = 0.125$	$h = 0.0625$
0.0	1.2	1.2	1.2	1.2
0.0625				1.16309963099631
0.125			1.138028169014085	1.132420390870575
0.875				1.107433531832221
0.25		1.11578947368421	1.097625754527163	1.087697702048688
0.3125				1.072848733050428
0.3750			1.076058670865774	1.062591261883119
0.4375				1.056691936007843
0.5	1.16	1.102923976608187	1.071584631403082	1.05497399855306
0.5625				1.057313093648606
0.625			1.083304722534287	1.063634167839859
0.6875				1.073909375352506
0.75		1.155693154454764	1.111083367305975	1.088156923424005
0.8125				1.106440820063882
0.875			1.155528238578191	1.128871511381216
0.9375				1.155607419897242
1.0	1.41	1.280693154454764	1.218028238578191	1.186857419897242
1.0625				1.222884311739977
1.125			1.300854083634988	1.264009385060764
1.875				1.310618191998243
1.25		1.499406031418415	1.407333247623213	1.363167687109327
1.3125				1.422194931872460
1.3750			1.542120128653863	1.488327610237399
1.4375				1.562296659521182
1.5	2.213333333333333	1.85646403590676	1.711594803897454	1.644951390473478
1.5625				1.737277554498827
1.625			1.924441821176899	1.840418918502845
1.6875				1.955703033211136
1.75		2.438724967269859	2.192487526815889	2.084672034844470
1.8125				2.229119509959606
1.875			2.531915819582752	2.391134688221732
1.9375				2.573155519438852
2.0	4.926666666666666	3.418299956359811	2.965046650951717	2.778032554068109

Recall that the exact solution is $y(2.0) = 2.610686134642448$, so the error when $h = 0.125$ is 0.3544, while the error when $h = 0.0625$ is 0.1673, which is approximately half the previous error.

n	h	Approximation of $y(2.0)$	Absolute error
1	2.0	undefined	n/a
2	1.0	undefined	n/a
4	0. 5	4.926666666666666	2.316
8	0. 25	3.418299956359811	0.8076
16	0.125	2.965046650951717	0.3544
32	0.0625	2.778032554068109	0.1673
64	0.03125	2.692145811868931	0.08146
128	0.015625	2.650889848727405	0.04020
256	0.0078125	2.630659683510977	0.01997
512	0.00390625	2.620641224077971	0.009955
1024	0.001953125	2.615655806460025	0.004970

You will see that each time you double the number of steps, the error drops by approximately one half.

A linear-algebra vector class in C++

The `std::vector` that is included in the Standard Template Library the class equivalence of a C++ array; however, that does not make it equivalent to the vectors in linear algebra. We require a vector class that describes the finite-dimensional vectors seen in linear algebra: objects that include scalar multiplication and vector addition.

```
template <std::size_t N>
class vec {
        public:
                vec();
                vec( vec const &v );
                vec( std::initializer_list<double> init );

                vec operator =( vec const &v );
                vec operator =( double s );

                vec &operator *=( double s );
                vec operator *( double s ) const;
                vec &operator /=( double s );
                vec operator /( double s ) const;
                vec &operator +=( vec const &v );
                vec operator +( vec const &v ) const;
                vec operator +() const;
                vec &operator -=( vec const &v );
                vec operator -( vec const &v ) const;
                vec operator -() const;

                double operator *( vec const &v ) const;

                double &operator []( std::size_t k );
                double operator []( std::size_t k ) const;

        template <std::size_t M>
        friend vec<M> operator *( double s, vec<M> const &v );

        template <std::size_t M>
        friend double norm( vec<M> const &v );

        private:
                double entries[N];
};
```

The `std::vector` class is resizable; however, this is nonsensical for linear algebra, and thus rather than dealing with dynamic memory allocation, the size of the vector will be fixed and thus we may use a template parameter *N* describing the dimension.

```
// Default constructor: creates the zero vector
template <std::size_t N>
vec<N>::vec() {
    for ( std::size_t k{0}; k < N; ++k ) {
        entries[k] = 0.0;
    }
}

// Copy constructor: makes a copy of the vector 'v'
template <std::size_t N>
vec<N>::vec( vec<N> const &v ) {
    for ( std::size_t k{0}; k < N; ++k ) {
        entries[k] = v.entries[k];
    }
}

// Construct a vector with an initializer list
template <std::size_t N>
vec<N>::vec( std::initializer_list<double> init ) {
    std::size_t k{0};

    for ( std::initializer_list<double>::iterator itr{init.begin()};
          itr != init.end(); ++itr ) {
        entries[k] = *itr;
        ++k;
    }

    assert( k <= N );

    for ( ; k < N; ++k ) {
        entries[k] = 0.0;
    }
}
```

```
// Assignment operator: assign all the entries of this vector the entries of 'v'
template <std::size_t N>
vec<N> vec<N>::operator =( vec<N> const &v ) {
    for ( std::size_t k{0}; k < N; ++k ) {
        entries[k] = v.entries[k];
    }

    return *this;
}

// Assignment operator: assign all the entries of this vector the scalar 's'
template <std::size_t N>
vec<N> vec<N>::operator =( double s ) {
    for ( std::size_t k{0}; k < N; ++k ) {
        entries[k] = s;
    }

    return *this;
}

// Auto-scalar-multiplication: multiply each entry in this vector by the scalar 's'
template <std::size_t N>
vec<N> &vec<N>::operator *=( double s ) {
    for ( std::size_t k{0}; k < N; ++k ) {
        entries[k] *= s;
    }

    return *this;
}

// Return a vector that is this vector multiplied by the scalar 's'
template <std::size_t N>
vec<N> vec<N>::operator *( double s ) const {
    vec<N> ret;

    for ( std::size_t k{0}; k < N; ++k ) {
        ret.entries[k] = entries[k]*s;
    }

    return ret;
}
```

```
// Auto-scalar-division: divide each entry in this vector by the scalar 's'
template <std::size_t N>
vec<N> &vec<N>::operator /=( double s ) {
    for ( std::size_t k{0}; k < N; ++k ) {
        entries[k] /= s;
    }

    return *this;
}

// Return a vector that is this vector divided by the scalar 's'
template <std::size_t N>
vec<N> vec<N>::operator /( double s ) const {
    vec<N> ret;

    for ( std::size_t k{0}; k < N; ++k ) {
        ret.entries[k] = entries[k]/s;
    }

    return ret;
}

// Auto-vector-addition: add to this vector the vector 'v'
template <std::size_t N>
vec<N> &vec<N>::operator +=( vec const &v ) {
    for ( std::size_t k{0}; k < N; ++k ) {
        entries[k] += v.entries[k];
    }

    return *this;
}

// Return a vector that is the sum of this vector added to the vector 'v'
template <std::size_t N>
vec<N> vec<N>::operator +( vec const &v ) const {
    vec<N> ret;

    for ( std::size_t k{0}; k < N; ++k ) {
        ret.entries[k] = entries[k] + v.entries[k];
    }

    return ret;
}

// Return a copy of this vector
template <std::size_t N>
vec<N> vec<N>::operator +( vec const &v ) const {
    vec<N> ret{ v };
    return ret;
}
```

```cpp
// Auto-vector-subtraction: subtract from this vector the vector 'v'
template <std::size_t N>
vec<N> &vec<N>::operator -=( vec const &v ) {
    for ( std::size_t k{0}; k < N; ++k ) {
        entries[k] -= v.entries[k];
    }

    return *this;
}

// Return a vector that is the vector 'v' subtracted from this vector
template <std::size_t N>
vec<N> vec<N>::operator -( vec const &v ) const {
    vec<N> ret;

    for ( std::size_t k{0}; k < N; ++k ) {
        ret.entries[k] = entries[k] - v.entries[k];
    }

    return ret;
}

// Return this vector negated
template <std::size_t N>
vec<N> vec<N>::operator -() const {
    vec<N> ret;

    for ( std::size_t k{0}; k < N; ++k ) {
        ret.entries[k] = -entries[k];
    }

    return ret;
}

// Return the inner product of this vector and the vector 'v'
template <std::size_t N>
double vec<N>::operator *( vec const &v ) const {
    double result{0.0};

    for ( std::size_t k{0}; k < N; ++k ) {
        result += entries[k]*entries[k];
    }

    return result;
}
```

```
// Access the 'k'th entry of this vector by reference
template <std::size_t N>
double &vec<N>::operator []( std::size_t k ) {
    return entries[k];
}

// Access the 'k'th entry of this vector by value
template <std::size_t N>
double vec<N>::operator []( std::size_t k ) const {
    return entries[k];
}

// A friend function that is called if the user calculates s*v
// for a vector 'v' and a scalar 's'
template <std::size_t M>
vec<M> operator *( double s, vec<M> const &v ) {
   return v * s;
}

// A friend function that calculates the norm of the vector 'v'
template <std::size_t M>
double norm( vec<M> const &v ) {
    double result{0.0};

    for ( std::size_t k{0}; k < M; ++k ) {
        result += v.entries[k]*v.entries[k];
    }

    return std::sqrt( result );
}
```

Solving a system of initial-value problems

A system of initial-value problems is a collection of *n* initial-value problems of the form

$$\begin{aligned}
y_1^{(1)}(t) &= f_1\left(t, y_1(t), y_2(t), \ldots, y_n(t)\right)\\
y_2^{(1)}(t) &= f_2\left(t, y_1(t), y_2(t), \ldots, y_n(t)\right)\\
&\vdots\\
y_n^{(1)}(t) &= f_n\left(t, y_1(t), y_2(t), \ldots, y_n(t)\right)\\
y_1(t_0) &= y_{0,1}\\
y_2(t_0) &= y_{0,2}\\
&\vdots\\
y_n(t_0) &= y_{0,n}
\end{aligned}$$

Now, if we define the vector

$$\mathbf{y}(t) = \begin{pmatrix} y_1(t)\\ y_2(t)\\ \vdots\\ y_n(t) \end{pmatrix},$$

then

$$\mathbf{y}^{(1)}(t) = \begin{pmatrix} y_1^{(1)}(t)\\ y_2^{(1)}(t)\\ \vdots\\ y_n^{(1)}(t) \end{pmatrix}$$

and the vector-valued function **f** as

$$\mathbf{f}\left(t, \mathbf{y}(t)\right) = \begin{pmatrix} f_1\left(t, y_1(t), y_2(t), \ldots, y_n(t)\right)\\ f_2\left(t, y_1(t), y_2(t), \ldots, y_n(t)\right)\\ \vdots\\ f_n\left(t, y_1(t), y_2(t), \ldots, y_n(t)\right) \end{pmatrix}$$

and the initial vector as

$$\mathbf{y}_0 = \begin{pmatrix} y_{0,1}\\ y_{0,2}\\ \vdots\\ y_{0,n} \end{pmatrix}$$

then we can write this system of initial-value problems as

$$\mathbf{y}^{(1)}(t) = \mathbf{f}(t, \mathbf{y}(t))$$
$$\mathbf{y}(t_0) = \mathbf{y}_0$$

Because this is vector-based, it is now possible to implement these the same functions using the same tools; for example, for Euler's method,

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}(t_k, \mathbf{y}_k)$$

while the 4th-order Runge-Kutta method calculates four slopes, each of which is a vector:

$$\begin{aligned}
\mathbf{s}_0 &= \mathbf{f}(t_k, \mathbf{y}_k) \\
\mathbf{s}_1 &= \mathbf{f}\left(t_k + \frac{h}{2}, \mathbf{y}_k + \frac{h}{2}\mathbf{s}_0\right) \\
\mathbf{s}_2 &= \mathbf{f}\left(t_k + \frac{h}{2}, \mathbf{y}_k + \frac{h}{2}\mathbf{s}_1\right) \\
\mathbf{s}_3 &= \mathbf{f}(t_k + h, \mathbf{y}_k + h\mathbf{s}_2) \\
\mathbf{y}_{k+1} &= \mathbf{y}_k + \frac{h}{6}(\mathbf{s}_0 + 2\mathbf{s}_1 + 2\mathbf{s}_2 + \mathbf{s}_3)
\end{aligned}$$

You will note that in each case, each of the vectors is *n*-dimensional, and so we are either performing scalar arithmetic, such as $t_k + \frac{h}{2}$; or vector addition or scalar multiplication, such as $\mathbf{y}_k + \frac{h}{6}(\mathbf{s}_0 + 2\mathbf{s}_1 + 2\mathbf{s}_2 + \mathbf{s}_3)$. Consequently, we need only rewrite our functions to use our vector class `vec<N>`; however, with templates, it gets even easier.

Let’s take a look at Euler’s method. All vectors are highlighted in blue, and the type is abstracted using templates.

```
template <typename vector_type>
std::tuple< std::vector<double>, std::vector<T>, std::vector<T> >
  euler( vector_type f(double, vector_type ),
         double t0, vector_type y0, double tf,
         std::size_t n ) {
    assert( n > 0 );

    double h{(tf - t0)/n};

    std::<double>  t(n + 1);
    std::vector<vector_type>  y(n + 1);
    std::vector<vector_type> dy(n + 1);

    y[0] = y0;

    for ( std::size_t k{0}; k < n; ++k ) {
         t[k] = t0 + k*h;

        vector_type s0{f( t[k], y[k] )};
        dy[k] = s0;

        y[k + 1] = y[k] + h*s0;
    }

     t[n] = tf;
    dy[n] = f( tf, y[n] );

    return std::make_tuple( t, y, dy );
}
```

With iterative or adaptive methods, it was previously necessary to calculate the difference between two values, e.g., $\left|y_{k+1} - z_{k+1}\right|$. When we have a system of initial-value problems, we must calculate the difference between two vectors, and thus the user must provide a vector norm. Thus, we will add a final argument that is a function that calculates the norm of either a scalar or vector, and give it the default value of std::abs; however, if the user is using a vector class, the user can provide a norm function.

```
template <typename vector_type>
std::tuple< std::vector<double>, std::vector<vector_type>, std::vector<vector_type> >
  adaptive_dormand_prince( vector_type f( double, vector_type ),
                           double t0, vector_type y0, double tf,
                           double h, double eps_abs, double norm( vector_type ) = std::abs );
```

As an example, consider the fox-and-rabbit problem presented at

https://mathinsight.org/introducing_rabbit_predators

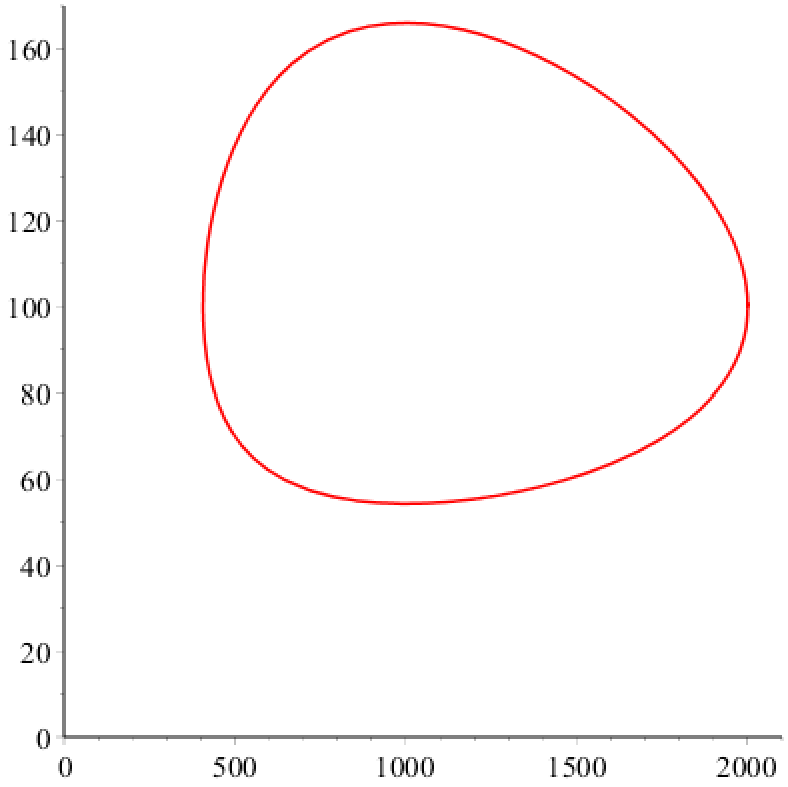
This gives the system of two differential equations that describe at a very simple level the relationship between a population of foxes and rabbits:

$$\begin{aligned}\frac{d}{dt}r(t) &= r(t) - 0.01r(t)f(t)\\ \frac{d}{dt}f(t) &= -f(t) + 0.0005r(t)f(t)\end{aligned}$$

with the initial conditions

$$\begin{aligned}r(0) &= 2000\\ f(0) &= 100\end{aligned}$$

After approximately 18.5 years, this returns to a point very close to the initial state.



To model this we must define the vector-valued function describing the slope:

```
        vec<2> f( double t, vec<2> y ) {
            return vec<2>{      y[0] -   0.01*y[0]*y[1],
                          -0.5*y[1] + 0.0005*y[0]*y[1] };
        }

        auto result = adaptive_dormand_prince( f, 0, vec<2>{2000, 100}, 18.5, 0.1, 1e-3,
                                           reinterpret_cast<double(*)(vec<2>)>(norm<2>) );
```

This table shows the *t* values and a pair representing the number of rabbits and foxes per generation. You will note that there seems to be an 18 ½ generation cycle. This is a consequence of the simplified differential equation. This looping behaviour is no different than the looping behavior of the second-order differential equation $y^{(2)}(t)+y(t)=0$.

```
0,         (2000,     100      )
0.0720791, (1997.37,  103.666  )
0.144193,  (1989.4,   107.449  )
0.2162,    (1976.01,  111.32   )
0.288708,  (1957.04,  115.292  )
0.362428,  (1932.2,   119.379  )
0.43828,   (1900.95,  123.604  )
0.517582,  (1862.45,  128.004  )
0.602531,  (1815.07,  132.649  )
0.697761,  (1755.24,  137.706  )
0.8198,    (1669.97,  143.832  )
0.915401,  (1598,     148.26   )
1.01126,   (1522.76,  152.297  )
1.10015,   (1451.39,  155.631  )
1.18623,   (1381.81,  158.447  )
1.27184,   (1313,     160.82   )
1.35847,   (1244.51,  162.773  )
1.44735,   (1176.14,  164.3    )
1.53972,   (1107.72,  165.378  )
1.63704,   (1039.09,  165.968  )
1.74123,   ( 970.008, 166.004  )
1.85518,   ( 899.986, 165.385  )
1.9841,    ( 827.942, 163.935  )
2.13867,   ( 751.483, 161.281  )
2.28606,   ( 688.199, 157.976  )
2.43289,   ( 633.806, 154.085  )
2.56943,   ( 590.287, 150.052  )
2.70192,   ( 553.937, 145.855  )
2.83345,   ( 523.004, 141.49   )
2.9656,    ( 496.574, 136.977  )
3.09938,   ( 474.073, 132.339  )
3.23556,   ( 455.116, 127.599  )
3.37476,   ( 439.436, 122.78   )
3.51755,   ( 426.859, 117.909  )
3.66442,   ( 417.284, 113.007  )
3.81586,   ( 410.676, 108.1    )
3.97238,   ( 407.068, 103.211  )
4.13448,   ( 406.561,  98.3638)
4.30275,   ( 409.34,   93.5821)
4.47784,   ( 415.689,  88.8883)
4.66055,   ( 426.015,  84.3052)
4.85185,   ( 440.897,  79.8557)
5.05292,   ( 461.138,  75.5646)
5.26526,   ( 487.854,  71.4607)
5.49073,   ( 522.607,  67.5796)
5.73189,   ( 567.638,  63.9672)
 5.99316, ( 626.447,  60.6798)
 6.28635, ( 706.14,   57.7721)
 6.54774, ( 790.604,  55.895 )
 6.77046, ( 873.303,  54.8536)
 6.9685,  ( 955.492,  54.3869)
 7.14824, (1037.21,   54.3663)
 7.32973, (1126.44,   54.7688)
 7.49605, (1213.73,   55.547 )
 7.65182, (1299.68,   56.6674)
 7.80071, (1384.86,   58.129 )
 7.94611, (1469.9,    59.9629)
 8.09215, (1555.9,    62.2514)
 8.2465,  (1645.61,   65.2064)
 8.39878, (1730.66,   68.7161)
 8.51646, (1792.31,   71.8667)
 8.62021, (1842.46,   74.9808)
 8.72445, (1887.8,    78.4411)
 8.81906, (1923.61,   81.8766)
 8.90662, (1951.48,   85.3089)
 8.9892,  (1972.51,   88.7672)
 9.06812, (1987.35,   92.2687)
 9.14435, (1996.41,   95.8251)
 9.21861, (1999.94,   99.4451)
 9.29155, (1998.07,  103.136 )
 9.36373, (1990.86,  106.907 )
 9.43571, (1978.24,  110.765 )
 9.50811, (1960.08,  114.722 )
 9.58161, (1936.1,   118.791 )
 9.65709, (1905.8,   122.995 )
 9.7358,  (1868.4,   127.367 )
 9.81975, (1822.42,  131.971 )
 9.91303, (1764.75,  136.951 )
10.0286,  (1685.17,  142.817 )
10.1265,  (1612.29,  147.428 )
10.2249,  (1535.56,  151.65  )
10.3147,  (1463.64,  155.092 )
10.4011,  (1393.85,  157.99  )
10.4866,  (1324.94,  160.437 )
10.573,   (1256.42,  162.463 )
10.6614,  (1188.04,  164.065 )
10.7531,  (1119.64,  165.225 )
10.8494,  (1051.06,  165.903 )
10.9522,  ( 982.078, 166.041 )
11.0642,  ( 912.272, 165.546 )
11.19,    ( 840.707, 164.259 )
11.339,   ( 765.222, 161.855 )
11.49,    ( 698.628, 158.602 )
11.6404,  ( 641.535, 154.709 )
11.7781,  ( 596.579, 150.696 )
11.911,   ( 559.254, 146.525 )
12.0425,  ( 527.544, 142.186 )
12.1745,  ( 500.45,  137.695 )
12.308,   ( 477.362, 133.074 )
12.4437,  ( 457.869, 128.348 )
12.5824,  ( 441.689, 123.541 )
12.7246,  ( 428.633, 118.676 )
12.8708,  ( 418.591, 113.777 )
13.0215,  ( 411.517, 108.869 )
13.1772,  ( 407.432, 103.976 )
13.3384,  ( 406.429,  99.1209)
13.5056,  ( 408.677,  94.3274)
13.6796,  ( 414.441,  89.6183)
13.8611,  ( 424.11,   85.0163)
14.051,   ( 438.23,   80.5443)
14.2504,  ( 457.565,  76.2266)
14.4609,  ( 483.175,  72.0911)
14.6841,  ( 516.54,   68.1723)
14.9226,  ( 559.77,   64.5142)
15.1803,  ( 616.093,  61.1719)
15.4668,  ( 691.679,  58.202 )
15.7454,  ( 779.53,   56.0879)
15.9718,  ( 862.177,  54.9568)
16.1727,  ( 944.449,  54.4216)
16.3546,  (1026.21,   54.3462)
16.5384,  (1115.75,   54.699 )
16.7063,  (1203.26,   55.4344)
16.8632,  (1289.34,   56.514 )
17.0127,  (1374.58,   57.9334)
17.1583,  (1459.6,    59.7193)
17.304,   (1545.39,   61.9456)
17.4566,  (1634.36,   64.7987)
17.6181,  (1724.99,   68.4554)
17.7371,  (1787.71,   71.6088)
17.8418,  (1838.68,   74.7238)
17.947,   (1884.85,   78.1913)
18.0422,  (1921.31,   81.629 )
18.1302,  (1949.71,   85.0604)
18.2131,  (1971.2,    88.5161)
18.2923,  (1986.48,   92.0141)
18.3687,  (1995.94,   95.5662)
18.443,   (1999.87,   99.1814)
18.5,     (1999.17,  102.046 )
```

Solving a higher-order IVP and systems of higher-order IVPs

A higher order initial-value problem involves a differential equation with a higher derivative together with initial conditions. For example,

$$\frac{d^2}{dt^2}y(t)+5\frac{d}{dt}y(t)+6y(t)=\sin(t)$$

or

$$y^{(2)}(t)+5y^{(1)}(t)+6y(t)=\sin(t)$$

with both an initial place and velocity:

$$\begin{aligned}y(t_0)&=y_0\\ y^{(1)}(t_0)&=y_0^{(1)}\end{aligned}$$

In this case, if we define

$$\mathbf{w}(t)=\begin{pmatrix}w_0(t)\\ w_1(t)\end{pmatrix}=\begin{pmatrix}y(t)\\ y^{(1)}(t)\end{pmatrix}.$$

Given a vector of functions of *t*, we can define the derivative of that vector by taking the derivative of each of the entries:

$$\mathbf{w}^{(1)}(t)=\begin{pmatrix}w_0^{(1)}(t)\\ w_1^{(1)}(t)\end{pmatrix}=\begin{pmatrix}y^{(1)}(t)\\ y^{(2)}(t)\end{pmatrix}.$$

Note, however, that $w_0^{(1)}(t)=w_1(t)$, while $y^{(2)}(t)+5y^{(1)}(t)+6y(t)=\sin(t)$ so $w_1^{(1)}(t)+5w_1(t)+6w_0(t)=\sin(t)$, and therefore we have that

$$\begin{aligned}w_0^{(1)}(t)&=w_1(t)\\ w_1^{(1)}(t)&=\sin(t)-5w_1(t)-6w_0(t)\end{aligned}$$

Therefore, we can define the vector-valued function

$$\mathbf{f}(t,\mathbf{w}(t))=\begin{pmatrix}w_1(t)\\ \sin(t)-5w_1(t)-6w_0(t)\end{pmatrix}$$

so that our initial value problem may now be written as

$$\mathbf{w}^{(1)}(t)=\mathbf{f}(t,\mathbf{w}(t))$$

The initial values are $\mathbf{w}(t_0)=\mathbf{w}_0=\begin{pmatrix}y_0\\ y_0^{(1)}\end{pmatrix}$.

Thus, our second-order initial-value problem is reduced to a system of two first-order initial-value problems:

$$\begin{aligned}\mathbf{w}^{(1)}(t) &= \mathbf{f}(t, \mathbf{w}(t))\\ \mathbf{w}(t_0) &= \mathbf{w}_0\end{aligned}$$

Thus, we can implement these using the functions we've already written:

```
        vec<2> f( double t, vec<2> w ) {
            return vec<2>{ w[1],
                           std::sin(t) - 5*w[1] - 6*w[0] };
        }

        auto result = adaptive_dormand_prince( f, 0, vec<2>{2000, 100}, 18.5, 0.1, 1e-3,
                                               reinterpret_cast<double(*)(vec<2>)>(norm<2>) );
```

In general, if we have a higher-order initial-value problem, we can almost always write it as

$$y^{(n)}(t) = f\left(t, y(t), y^{(1)}(t), y^{(2)}(t), \ldots, y^{(n-1)}(t)\right).$$

We could re-interpret the subsequent arguments as a vector:

$$y^{(n)}(t) = f\left(t, \begin{pmatrix} y(t) \\ y^{(1)}(t) \\ y^{(2)}(t) \\ \vdots \\ y^{(n-1)}(t) \end{pmatrix}\right) = f(t, \mathbf{w}(t)).$$

We thus have that

$$\mathbf{w}(t) = \begin{pmatrix} y(t) \\ y^{(1)}(t) \\ y^{(2)}(t) \\ \vdots \\ y^{(n-1)}(t) \end{pmatrix} = \begin{pmatrix} w_0(t) \\ w_1(t) \\ w_2(t) \\ \vdots \\ w_{n-1}(t) \end{pmatrix}.$$

Thus, the derivative of this vector is

$$\mathbf{w}^{(1)}(t) = \begin{pmatrix} w_0^{(1)}(t) \\ w_1^{(1)}(t) \\ w_2^{(1)}(t) \\ \vdots \\ w_{n-2}^{(1)}(t) \\ w_{n-1}^{(1)}(t) \end{pmatrix} = \begin{pmatrix} y^{(1)}(t) \\ y^{(2)}(t) \\ y^{(3)}(t) \\ \vdots \\ y^{(n-1)}(t) \\ y^{(n)}(t) \end{pmatrix} = \begin{pmatrix} w_1(t) \\ w_2(t) \\ w_3(t) \\ \vdots \\ w_{n-1}(t) \\ f(t, \mathbf{w}(t)) \end{pmatrix} = \mathbf{f}(t, \mathbf{w}(t))$$

Such a higher-order initial-value problem would have *n* initial conditions:

$$\mathbf{w}_0 = \begin{pmatrix} y_0 \\ y_0^{(1)} \\ y_0^{(2)} \\ \vdots \\ y_0^{(n-2)} \\ y_0^{(n-1)} \end{pmatrix}$$

For example, if we have an 8th-order initial-value problem and the 8th-derivative is described by the function

```
double f( double t, double y0, double y1, double y2, double y3, double y4, double y5, double y6, double y7 );
```

where `y`*`k`* is the value of the *k*th derivative, we can implement it as follows:

```
vec<8> f_vec( double t, vec<8> w ) {
    return vec<2>{ w[1],
                   w[2],
                   w[3],
                   w[4],
                   w[5],
                   w[6],
                   w[7],
                   f( t, w[0], w[1], w[2], w[3], w[4], w[5], w[6], w[7] ) };
}

auto result = adaptive_dormand_prince( f_vec, 0,
                                       vec<8>{0.2, 0.5, -0.3, 0.7, 0.4, 1.2, 0.8, 0.3},
                                       18.5, 0.1, 1e-3,
                                       reinterpret_cast<double(*)(vec<8>)>(norm<8>) );
```

Problems

A. Given the initial-value problem

$$y^{(1)}(t) = -0.5y(t)$$
$$y(1) = 1.2$$

1. Apply Euler's method to approximate $y(5)$ using $n = 4$ (answer: 1.2, 0.6, 0.3, 0.15, 0.075).
2. Apply Huen's method to approximate $y(5)$ using $n = 2$ (answer: 1.2, 0.6, 0.3).
3. Apply the 4th-order Runge Kutta method to approximate $y(5)$ using $n = 1$ (answer: 1.2, 0.4).
4. You have applied Euler's method twice to approximate $y(5)$ using $n = 8$ and $n = 16$. You get the following two sequences of approximations:

   $\mathbf{y}$ = 1.2, 0.9, 0.675, 0.5062, 0.3797, 0.2848, 0.2136, 0.1602, 0.1201

   $\mathbf{z}$ = 1.2, 1.05, 0.9188, 0.8039, 0.7034, 0.6155, 0.5386, 0.4712, 0.4123, 0.3608, 0.3157, 0.2762, 0.2417, 0.2115, 0.1851, 0.1619, 0.1417

   where $y_0 = 1.2$, $y_1 = 0.9$, …, $y_8 = 0.1201$ and $z_0 = 1.2$, $z_1 = 1.05$, …, $z_{16} = 0.1417$. What is a reasonable approximation of error of $z_{16} = 0.1417$ (answer: 0.0216), and use extrapolation to find a better approximation of $y(5)$ using both z_{16} and y_8 (answer: 0.1633). Similarly, what is the best approximation of $y(3)$ (answer: 0.4449)?
5. You have applied the 4th-order Runge Kutta method twice to approximate $y(5)$ using $n = 2$ and $n = 4$. You get the following two sequences of approximations:

   $\mathbf{y}$ = 1.2, 0.45, 0.1688

   $\mathbf{z}$ = 1.2, 0.7281, 0.4418, 0.2681, 0.1627

   What is a reasonable approximation of error of $z_4 = 0.1627$ (answer: –0.00040666⋯), and use extrapolation to find a better approximation of $y(5)$ using both y_2 and z_4 (answer: 0.16229333⋯) Similarly, what is the best approximation of $y(3)$ (answer: 0.44125333⋯)?
6. Perform one step of the adaptive Euler-Heun method to approximate $y(3)$ using $h = 2$. Assume the goal is to approximate $y(5)$. What is the value of a at the next step if $\varepsilon_{\text{abs}} = 0.1$? Must you repeat the calculation, or can you continue (answer: $a = 0.041666\cdots$, so repeat)? What is the step size at the next step (answer: 0.075)?
7. Perform one step of the backward-Euler method to approximate $y(5)$ with $h = 4$ until the difference is 0.01. Hint: does it converge using fixed-point interation? If not, what else can you do?

B. Given the initial-value problem

$$y^{(1)}(t) = -0.1y(t) + 0.1t$$
$$y(0) = 0.7$$

1. Apply Euler's method to approximate $y(4)$ using $n = 4$ (answer: 0.7, 0.63, 0.667, 0.8003, 1.02027)
2. Apply Huen's method to approximate $y(4)$ using $n = 2$ (answer: 0.7, 0.774, 1.19468)
3. Apply the 4th-order Runge Kutta method to approximate $y(4)$ using $n = 1$ (answer: 1.17328)
4. You have applied Euler's method twice to approximate $y(4)$ using $n = 8$ and $n = 16$. You get the following two sequences of approximations:

   y = **0**.7, 0.6650, 0.6568, 0.6739, 0.7152, 0.7795, 0.8655, 0.9722, 1.0986

   z = 0.7, 0.6825, 0.6717, 0.6674, 0.6695, 0.6777, 0.6920, 0.7122, 0.7382, 0.7697, 0.8067, 0.8491, 0.8966, 0.9492, 1.0067, 1.0690, 1.1360

   What is a reasonable approximation of error of $z_{16} = 1.1360$ (answer: 0.0374), and use extrapolation to find a better approximation of $y(4)$ using both y_8 and z_{16} (answer: 1.1734). Similarly, what is the best approximation of $y(2)$ (answer: 0.7612)?
5. You have applied the 4th-order Runge Kutta method twice to approximate $y(4)$ using $n = 2$ and $n = 4$. You get the following two sequences of approximations:

   y = **0**.7, 0.7604, 1.1725

   z = 0.7, 0.6818, 0.7604, 0.9268, 1.1724

   What is a reasonable approximation of error of $z_4 = 1.1724$ (answer: 0.000006667), and use extrapolation to find a better approximation of $y(4)$ using both y_2 and z_4 (answer: 1.172393333). Similarly, what is the best approximation of $y(2)$ (answer: 0.7604)?
6. Perform one step of the adaptive Euler-Heun method to approximate $y(2)$ using $h = 2$. What is the value of s at the next step. Assume your goal is to estimate $y(10)$. What is the value of a at the next step if $\varepsilon_{\text{abs}} = 0.1$? Must you repeat the calculation, or can you continue (answer: $a = 0.0258\cdots$, so repeat)? What is the step size at the next step (answer: repeat with $h = 0.04651\cdots$)?
7. Perform one step of the backward-Euler method to approximate $y(4)$ with $h = 4$ until the difference is 0.01 (answer: 1.64440192).

C. Given the initial-value problem

$$\begin{aligned} x^{(1)}(t) &= x(t) + y(t) \\ y^{(1)}(t) &= x(t) - y(t) \\ x(0) &= 0.5 \\ y(0) &= -0.5 \end{aligned}$$

1. Apply Euler's method to approximate $x(4)$ and $y(4)$ using n = 4 (answer: $\begin{pmatrix} 0.5 \\ -0.5 \end{pmatrix}, \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}, \begin{pmatrix} 1.5 \\ 0.5 \end{pmatrix}, \begin{pmatrix} 3.5 \\ 1.5 \end{pmatrix}, \begin{pmatrix} 8.5 \\ 3.5 \end{pmatrix}$).

2. Apply Huen's method to approximate $x(4)$ and $y(4)$ using n = 2 (answer: $\begin{pmatrix} 0.5 \\ -0.5 \end{pmatrix}, \begin{pmatrix} 2.5 \\ -0.5 \end{pmatrix}, \begin{pmatrix} 16.5 \\ 3.5 \end{pmatrix}$).

3. Apply the 4th-order Runge Kutta method to approximate $x(4)$ and $y(4)$ using n = 1 (answer: $\begin{pmatrix} 29.8\bar{3} \\ -4.5 \end{pmatrix}$).

4. If $\mathbf{z}(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$ and you have applied Euler's method twice to approximate $x(4)$ and $y(4)$ using n = 8 and n = 16. You get the following two sequences of approximations:

$$\mathbf{z} = \left(\begin{pmatrix} 0.5 \\ -0.5 \end{pmatrix}, \begin{pmatrix} 0.5 \\ 0 \end{pmatrix}, \begin{pmatrix} 0.75 \\ 0.25 \end{pmatrix}, \begin{pmatrix} 1.25 \\ 0.5 \end{pmatrix}, \begin{pmatrix} 2.125 \\ 0.875 \end{pmatrix}, \begin{pmatrix} 3.625 \\ 1.5 \end{pmatrix}, \begin{pmatrix} 6.188 \\ 2.562 \end{pmatrix}, \begin{pmatrix} 10.56 \\ 4.375 \end{pmatrix}, \begin{pmatrix} 18.03 \\ 7.469 \end{pmatrix} \right)$$

$$\tilde{\mathbf{z}} = \left(\begin{pmatrix} 0.5 \\ -0.5 \end{pmatrix}, \begin{pmatrix} 0.5 \\ -0.25 \end{pmatrix}, \begin{pmatrix} 0.5625 \\ -0.0625 \end{pmatrix}, \begin{pmatrix} 0.6875 \\ 0.09375 \end{pmatrix}, \begin{pmatrix} 0.8828 \\ 0.2422 \end{pmatrix}, \begin{pmatrix} 1.164 \\ 0.4023 \end{pmatrix}, \begin{pmatrix} 1.556 \\ 0.5928 \end{pmatrix}, \begin{pmatrix} 2.093 \\ 0.8335 \end{pmatrix}, \begin{pmatrix} 2.824 \\ 1.148 \end{pmatrix}, \right.$$
$$\left. \begin{pmatrix} 3.818 \\ 1.567 \end{pmatrix}, \begin{pmatrix} 5.164 \\ 2.130 \end{pmatrix}, \begin{pmatrix} 6.987 \\ 2.888 \end{pmatrix}, \begin{pmatrix} 9.456 \\ 3.913 \end{pmatrix}, \begin{pmatrix} 12.80 \\ 5.299 \end{pmatrix}, \begin{pmatrix} 17.32 \\ 7.174 \end{pmatrix}, \begin{pmatrix} 23.45 \\ 9.711 \end{pmatrix}, \begin{pmatrix} 31.74 \\ 13.14 \end{pmatrix} \right)$$

What is a reasonable approximation of the error of $\tilde{z}_{16}$ (answer: $\begin{pmatrix} 13.71 \\ 5.671 \end{pmatrix}$), and use extrapolation to find a better approximation of $x(4)$ and $y(4)$ using both z_8 and $\tilde{z}_{16}$ (answer: $\begin{pmatrix} 45.45 \\ 18.81 \end{pmatrix}$). Similarly, what is the best approximation of $x(2)$ and $y(2)$ (answer: $\begin{pmatrix} 3.523 \\ 1.421 \end{pmatrix}$)?

5. You have applied the 4th-order Runge Kutta method twice to approximate $x(4)$ and $y(4)$ using n = 2 and n = 4. You get the following two sequences of approximations:

$$\mathbf{z} = \left(\begin{pmatrix} 0.5 \\ -0.5 \end{pmatrix}, \begin{pmatrix} 3.833 \\ 0.8333 \end{pmatrix}, \begin{pmatrix} 51.17 \\ 20.39 \end{pmatrix} \right)$$

$$\tilde{\mathbf{z}} = \left(\begin{pmatrix} 0.5 \\ -0.5 \end{pmatrix}, \begin{pmatrix} 1.083 \\ 0.25 \end{pmatrix}, \begin{pmatrix} 4.125 \\ 1.653 \end{pmatrix}, \begin{pmatrix} 16.64 \\ 6.877 \end{pmatrix}, \begin{pmatrix} 67.41 \\ 27.92 \end{pmatrix} \right)$$

What is a reasonable approximation of error of $\tilde{z}_4$ (answer: $\begin{pmatrix} 1.083 \\ 0.502 \end{pmatrix}$), and use extrapolation to find a better approximation of $x(4)$ and $y(4)$ using both z_2 and $\tilde{z}_4$ (answer: $\begin{pmatrix} 68.49 \\ 28.42 \end{pmatrix}$). Similarly, what is the best approximation of $x(2)$ and $x(2)$ (answer: $\begin{pmatrix} 4.144 \\ 1.708 \end{pmatrix}$)?

6. Perform one step of the adaptive Euler-Heun method to approximate $x(2)$ and $y(2)$ using $h = 2$ ($\begin{pmatrix} 0.5 \\ 1.5 \end{pmatrix}$ and $\begin{pmatrix} 2.5 \\ -0.5 \end{pmatrix}$). Assume your goal is to estimate $x(20)$ and $y(20)$. What is the value of a at the next step if $\varepsilon_{\text{abs}} = 0.1$ (answer: 0.00176)? Must you repeat the calculation, or can you continue (answer: repeat)? What is the step size at the next step (answer: 0.00318)?

D. Given the initial-value problem

$$\begin{aligned} x^{(2)}(t) &= x^{(1)}(t)\,x(t) - 1 \\ x(0) &= 1 \\ x^{(1)}(0) &= 0.5 \end{aligned}$$

1. Apply Euler's method to approximate $x(4)$ using $n = 4$. What is an approximation of the derivative at $t = 4$?
   (Answer: $\begin{pmatrix}1\\0.5\end{pmatrix},\begin{pmatrix}1.5\\0\end{pmatrix},\begin{pmatrix}1.5\\-1\end{pmatrix},\begin{pmatrix}0.5\\-3.5\end{pmatrix},\begin{pmatrix}-3\\-6.25\end{pmatrix}$ so $x(4) \approx -3$ and $x^{(1)}(4) \approx -6.25$.)
2. Apply Huen's method to approximate $x(4)$ using $n = 2$. What is an approximation of the derivative at $t = 4$?
   (Answer: $\begin{pmatrix}1\\0.5\end{pmatrix},\begin{pmatrix}1\\-2\end{pmatrix},\begin{pmatrix}-9\\18\end{pmatrix}$ so $x(4) \approx -9$ and $x^{(1)}(4) \approx 18$.)
3. Apply the 4th-order Runge Kutta method to approximate $x(4)$ using $n = 1$. What is an approximation of the derivative at $t = 4$?
   (Answer: $\begin{pmatrix}1\\0.5\end{pmatrix},\begin{pmatrix}-6.3\bar{3}\\25.8\bar{3}\end{pmatrix}$ so $x(4) \approx -6.333$ and $x^{(1)}(4) \approx 25.833$.)
4. You have applied Euler's method twice to approximate $x(4)$ using $n = 8$ and $n = 16$. You get the following two sequences of approximations:

$$\mathbf{w} = \left(\begin{pmatrix}1\\0.5\end{pmatrix},\begin{pmatrix}1.25\\0.25\end{pmatrix},\begin{pmatrix}1.375\\-0.0938\end{pmatrix},\begin{pmatrix}1.3281\\-0.6582\end{pmatrix},\begin{pmatrix}0.9990\\-1.5953\end{pmatrix},\begin{pmatrix}0.2014\\-2.8922\end{pmatrix},\begin{pmatrix}-1.2447\\-3.6834\end{pmatrix},\begin{pmatrix}-3.0834\\-1.8910\end{pmatrix},\begin{pmatrix}-4.0319\\0.5272\end{pmatrix}\right)$$

$$\tilde{\mathbf{w}} = \left(\begin{pmatrix}1\\0.5\end{pmatrix},\begin{pmatrix}1.125\\0.375\end{pmatrix},\begin{pmatrix}1.2188\\0.2305\end{pmatrix},\begin{pmatrix}1.2764\\0.0507\end{pmatrix},\begin{pmatrix}1.2890\\-0.1831\end{pmatrix},\begin{pmatrix}1.2433\\-0.4922\end{pmatrix},\begin{pmatrix}1.1202\\-0.8951\end{pmatrix},\begin{pmatrix}0.8964\\-1.3958\end{pmatrix},\begin{pmatrix}0.5475\\-1.9586\end{pmatrix},\right.$$
$$\left.\begin{pmatrix}0.0578\\-2.4767\end{pmatrix},\begin{pmatrix}-0.5613\\-2.7625\end{pmatrix},\begin{pmatrix}-1.2520\\-2.6248\end{pmatrix},\begin{pmatrix}-1.9802\\-2.0533\end{pmatrix},\begin{pmatrix}-2.4215\\-1.3238\end{pmatrix},\begin{pmatrix}-2.7524\\-0.7724\end{pmatrix},\begin{pmatrix}-2.9455\\-0.4909\end{pmatrix},\begin{pmatrix}-3.0683\\-0.3794\end{pmatrix}\right)$$

   What is a reasonable approximation of error of $\tilde{w}_{16}$ ($\begin{pmatrix}0.9636\\-0.9066\end{pmatrix}$), and use extrapolation to find a better approximation of $x(4)$ using both w_8 and $\tilde{w}_{16}$ ($\begin{pmatrix}-2.1047\\-1.2860\end{pmatrix}$ so $x(4) \approx -2.1047$). Similarly, what is the best approximation of $x(2)$ ($\begin{pmatrix}-0.4515\\-0.3633\end{pmatrix}$ so $x(4) \approx -0.4515$)?
5. You have applied the 4th-order Runge Kutta method twice to approximate $x(4)$ using $n = 2$ and $n = 4$. You get the following two sequences of approximations:

$$\mathbf{w} = \left(\begin{pmatrix}1\\0.5\end{pmatrix},\begin{pmatrix}0\\-1.6667\end{pmatrix},\begin{pmatrix}-5.5309\\-19.4472\end{pmatrix}\right)$$

$$\tilde{\mathbf{w}} = \left(\begin{pmatrix}1\\0.5\end{pmatrix},\begin{pmatrix}1.1647\\-0.3164\end{pmatrix},\begin{pmatrix}-0.0271\\-1.9491\end{pmatrix},\begin{pmatrix}-1.8079\\-1.3227\end{pmatrix},\begin{pmatrix}-2.6032\\-0.6198\end{pmatrix}\right)$$

   What is a reasonable approximation of error of w_2, and use extrapolation to find a better approximation of $x(4)$ using both w_2 and $\tilde{w}_4$. Similarly, what is the best approximation of $x(2)$?
6. Perform one step of the adaptive Euler-Heun method to approximate $x(2)$ using $h = 2$. What is the value of s at the next step. Must you repeat the calculation, or can you continue? What is the step size at the next step?

E. Explain why we can use the extrapolation in the iterative methods to get a better solution given two less-accurate solutions.
F. Explain how we derive the scaling factor used in the adaptive techniques.
G. Give two reasons why the adaptive techniques discussed are better than the iterative techniques.

Acknowledgments

Sidharth Baveja
Xiao Tong Mu
Hamza Baig