Normal form

- 1^{st}-order DE (or system of DEs)
- 1^{st} deriv. (for each dep. var.) is solved for

$$y' = f(t, y)$$

An idea: Antidiff. both sides

$$\int y'(t)\,dt = \underbrace{\int f(t, y(t))\,dt}_{\text{generally, cannot do w/out knowing } y(t)}$$

$\downarrow$

$y(t)$

Special instance where this works: Separable DEs

Can be written as $M(y)\dfrac{dy}{dt} = \underbrace{N(t)}$

$$f(t,y) = \frac{N(t)}{M(y)}$$

Ex.)

$$y' = 2x^2y^2 \qquad (x \text{ not } t \text{ as ind.})$$

$$\frac{1}{y^2}\frac{dy}{dx} = 2x^2 \qquad (\text{have separation})$$

$$\int \frac{1}{y^2(x)}\, y'(x)\,dx = \int 2x^2\,dx$$

By substitution

$$\int y^{-2}\,dy = \int 2x^2\,dx$$

$$-y^{-1} + C_1 = \frac{2}{3}x^3 + C_2$$

$$y^{-1} = -\frac{2}{3}x^3 + C$$

$$\frac{1}{y} = \frac{-2}{3}x^3 + C$$

$$y(x) = \frac{1}{-\frac{2}{3}x^3 + C}$$

Answer - family of answers (one for each C)

Different idea:

$$y' = 2x^2y^2, \quad \text{w/ IC} \quad y(1) = 1$$

$$y(x_0) = y_0$$

Know $\quad y' = \lim_{h\to 0} \frac{y(x+h) - y(x)}{h}$

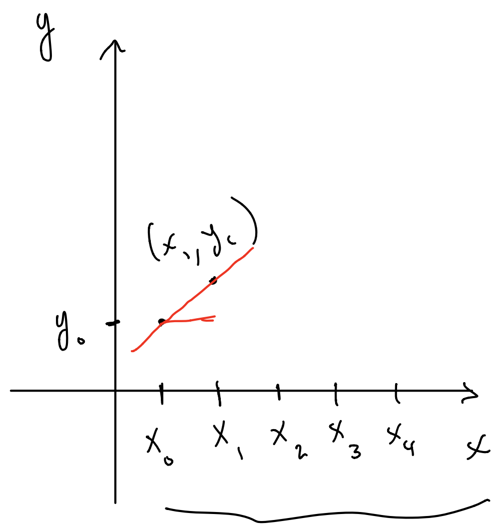
If we take fixed, small h, then

$$y'(x) \approx \frac{y(x+h) - y(x)}{h}$$

or, after algebra

$$y(x+h) \approx y(x) + h\,y'(x)$$

Idea: Euler's Method



make grid points evenly spaced by $h = \Delta x$

$$y(x_0 + \Delta x) \approx y(x_0) + \Delta x \underbrace{y'(x_0)}$$

from normal form, $y'(x_0) = f(x_0, y_0)$

$$y_1 = \underbrace{y_0}_{\text{init ht.}} + \underbrace{\Delta x\, f(x_0, y_0)}_{\text{rise}}$$

($x_0, y_0, f, \Delta x$ are all known at outset)

Now repeat! Have x_1, y_1, Δx, f

$$y(x_1) \approx y_1$$

$$y_2 = y_1 + \Delta x\, f(x_1, y_1)$$

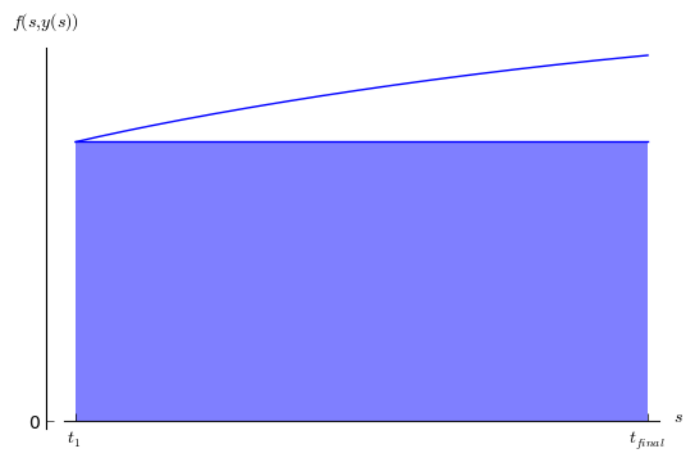
Numerical Solutions

Very often we would be happy with an approximate solution, say, one that aims not to tell us the value of $y(t)$ at all times, but rather at just some final time t_{final}. It follows from Equation (5) that

$$y(t_{\text{final}}) \;=\; y_0 + \int_{t_0}^{t_{\text{final}}} f(s, y(s))\, ds, \tag{7}$$

an expression that, as we have already observed, contains an integral we cannot calculate *exactly*. However, a number of methods have been proposed for calculating this integral *approximately*.

Euler's Method



The first idea we pose for approximating the integral in Equation (7) (corresponding to the full area under the curve $f(s, y(s))$ depicted at right) is just the left-hand Riemann sum method from Calculus. An extremely crude (and poor) approximation arises using just one rectangle: knowing the value of $f(s, y(s))$ at the left-most point $s = t_0$, we act as if $f(s, y(s))$ remains equal to $f(t_0, y_0)$ throughout the interval $[t_0, t_{\text{final}}]$. (See the figure.) Using this crude approximation we would get

$$y(t_{\text{final}}) \doteq y_0 + (t_{\text{final}} - t_0) f(t_0, y_0).$$

Section 2.7 of the text explains this approach in more detail and, in the process, describes an alternate way, via the idea of following a tangent line to the curve $y(t)$ instead of the curve itself, of understanding what is being done here.

While a poor approximation for $y(t_{\text{final}})$ is enough of a reason on its own to justify using more rectangles, there is also the fact that, while we didn't expect to get a full description of $y(t)$ (i.e., one we may evaluate at any time t between t_0 and t_{final}) using an approach like this, we would like to wind up with something more than just *two values* of y, one at t_0 (correct, but it was handed to us already before we did any work!), and one at t_{final} (which is only coarsely approximated). So, let us partition up the time interval $[t_0, t_{\text{final}}]$ into N subintervals of length $h = (t_{\text{final}} - t_0)/N$, so that

$$t_0 < t_1 < t_2 < \cdots < t_N = t_{\text{final}}, \qquad \text{with each} \qquad t_{j+1} - t_j = h.$$

We then

$$\begin{aligned} \text{set } y_1 &= y_0 + hf(t_0, y_0) \approx y_0 + \int_{t_0}^{t_1} f(s, y(s))\,ds = y(t_1), \\ \text{set } y_2 &= y_1 + hf(t_1, y_1), \\ &\vdots \\ \text{set } y_N &= y_{N-1} + hf(t_{N-1}, y_{N-1}). \end{aligned}$$

This is called **Euler's Method**, and it is simply choosing a stepsize $h > 0$ and calculating repeatedly

$$y_{j+1} = y_j + hf(t_j, y_j), \qquad j = 0, 1, 2, \ldots,$$

until we have reached a satisfactory final time t_{final}. By choosing h small, we obtain points (t_0, y_0), (t_1, y_1), ..., $(t_{\text{final}}, y_{\text{final}})$ as (horizontally) close to each other as we please, all of which approximately lie on the desired solution curve $y(t)$. See the applet at http://ocw.mit.edu/ans7870/18/18.03/s06/tools/EulerMethod.html.

The Runge-Kutta Method

Euler's Method is easily understood, but for it to yield good precision generally requires the step size h to be extremely small, thus making it slow. (In actual fact, the realities of storing numbers in a machine bring on ill effects of a different sort when h is too small!) One can do significantly better (without a great deal extra work) approximating the area under a curve via piecewise quadratic functions (Simpson's Rule) rather than via piecewise constant functions (left-hand method). This fact, coupled with some technical details from Numerical Analysis, yields a method for approximation of integrals like

$$\int_{t_j}^{t_{j+1}} f(s, y(s))\,ds$$

known as the **4^{th} order Runge-Kutta** method which is far better than Euler's method at solving the same 1st order problems. The formulas are a good deal more complicated as well, and we do not provide them here. There are a number of applets that carry out RK4; one is found at http://www.csun.edu/~hcmth018/RK.html. The next example gives an implementation in Sage.

Example 3:

Use the 4^{th} order RK method to find an approximate value of the solution of

$$y' = 3e^{-4t} - 2y, \quad y(0) = 1,$$

at $t = 4$ using 40 steps (i.e., 40 subintervals, so $h = 0.1$), and plot the result. Note that the true solution is

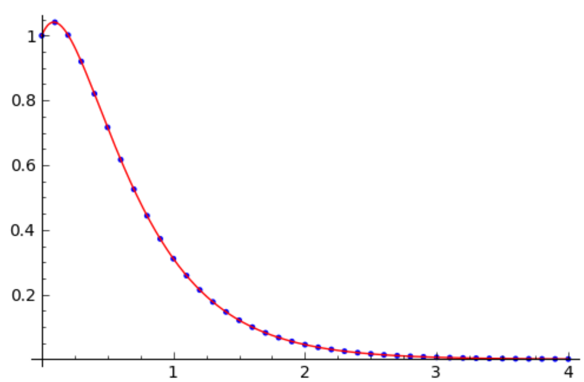
$$y(t) = \frac{1}{2}\left(5e^{-2t} - 3e^{-4t}\right).$$

In SAGE, we first define the function

```
def rk(f, y0, t0, tFin, numSteps):
    var('t y')
    h = (tFin - t0)/numSteps
    w = []
    t = t0
    y = n(y0)
    w.append((t, y))
    for i in range(1, numSteps+1):
        K1 = f(t, y)
        K2 = f(t + h/2, y + h*K1/2)
        K3 = f(t + h/2, y + h*K2/2)
        K4 = f(t + h, y + h*K3)
        y = n(y + (K1 + 2*K2 + 2*K3 + K4) * h/6)
        t = t0 + i*h
        w.append((n(t), n(y)))
    return w
```

This is the generic Runge-Kutta algorithm. To apply it to the specifics of our problem, we need only

```
var('t y')
f(t,y) = 3*exp(-4*t) - 2*y
p1 = list_plot(rk(f, 1, 0, 4, 40), plotjoined=True,color='blue')
p2 = plot( (5*exp(-2*t) - 3*exp(-4*t))/2, (t,0,4))
show(p1 + p2)
```



The red curve is the true solution, while the blue dots come from the RK4 method. These latter seem to stick quite closely to the true solution.

■