1 Euler's method

Consider the differential equation:

$$y'(x) = y(x) \cdot x, \quad y(1) = 1,$$

which can be solved with `SymPy`:

```
using CalculusWithJulia   # loads `SymPy`, `Roots`
using Plots
@vars x y
u = SymFunction("u")
x0, y0 = 1, 1
F(y,x) = y*x

dsolve(u'(x) - F(u(x), x))
```

$$u(x) = C_1 e^{\frac{x^2}{2}}$$

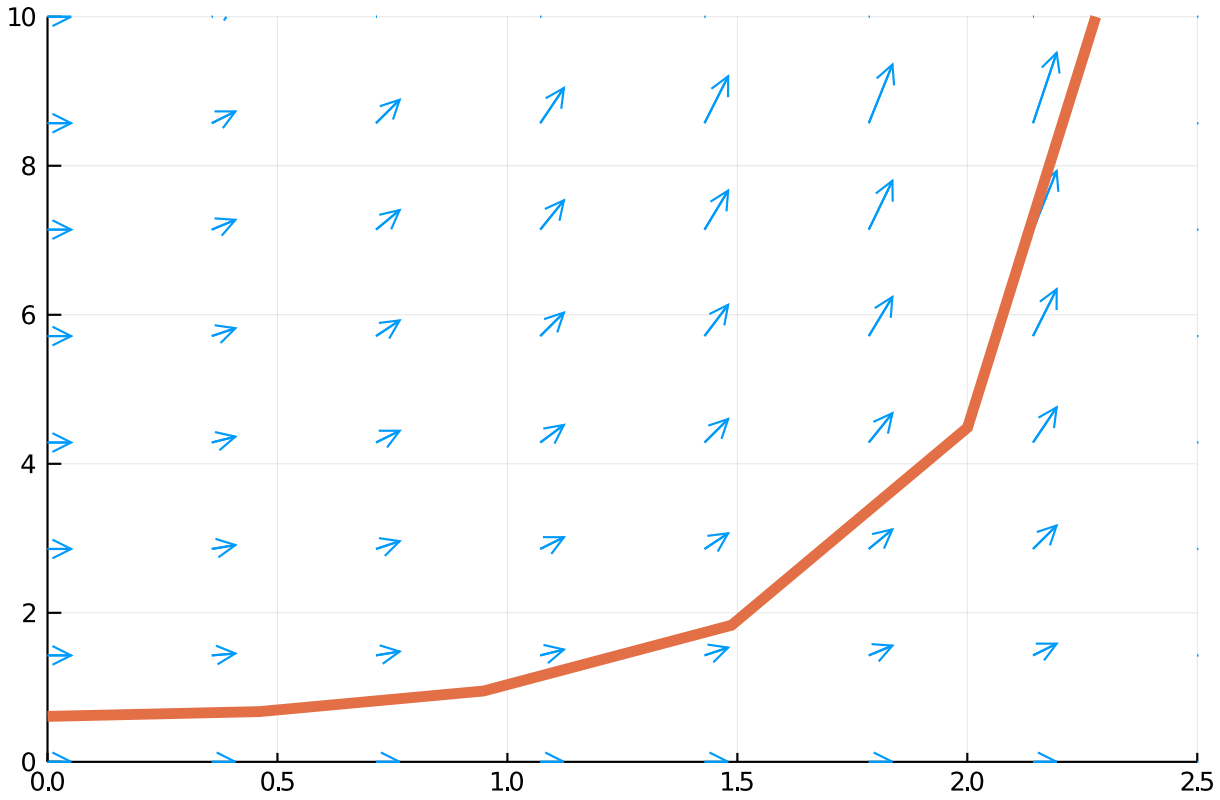
With the given initial condition, the solution becomes:

```
out = dsolve(u'(x) - F(u(x),x), u(x), ics=(u, x0, y0))
```

$$u(x) = \frac{e^{\frac{x^2}{2}}}{e^{\frac{1}{2}}}$$

Plotting this solution over the slope field

```
p = plot(legend=false)
vectorfieldplot!((x,y) -> [1, F(x,y)], xlims=(0, 2.5), ylims=(0, 10))
plot!(rhs(out),  linewidth=5)
```



we see that the vectors that are drawn seem to be tangent to the graph of the solution. This is no coincidence, the tangent lines to integral curves are in the direction of the slope field.

What if the graph of the solution were not there, could we use this fact to *approximately* reconstruct the solution?

That is, if we stitched together pieces of the slope field, would we get a curve that was close to the actual answer?

XXX can not include '.gif' file here

The illustration suggests the answer is yes, let's see. The solution is drawn over x values 1 to 2. Let's try piecing together 5 pieces between 1 and 2 and see what we have.

The slope-field vectors are *scaled* versions of the vector `[1, F(y,x)]`. The `1` is the part in the direction of the x axis, so here we would like that to be 0.2 (which is $(2-1)/5$. So our vectors would be `0.2 * [1, F(y,x)]`. To allow for generality, we use `h` in place of the specific value 0.2.

Then our first pieces would be the line connecting (x_0, y_0) to

$$\langle x_0, y_0 \rangle + h \cdot \langle 1, F(y_0, x_0) \rangle.$$

The above uses vector notation to add the piece scaled by h to the starting point. Rather than continue with that notation, we will use subscripts. Let x_1, y_1 be the postion of the tip of the vector. Then we have:

$$x_1 = x_0 + h, \quad y_1 = y_0 + hF(y_0, x_0).$$

With this notation, it is easy to see what comes next:

$$x_2 = x_1 + h, \quad y_2 = y_1 + hF(y_1, x_1).$$

We just shifted the indices forward by 1. But graphically what is this? It takes the tip of the first part of our "stitched" together solution, finds the slope filed there (`[1, F(y,x)]`) and then uses this direction to stitch together one more piece.

Clearly, we can repeat. The nth piece will end at:

$$x_{n+1} = x_n + h, \quad y_{n+1} = y_n + hF(y_n, x_n).$$

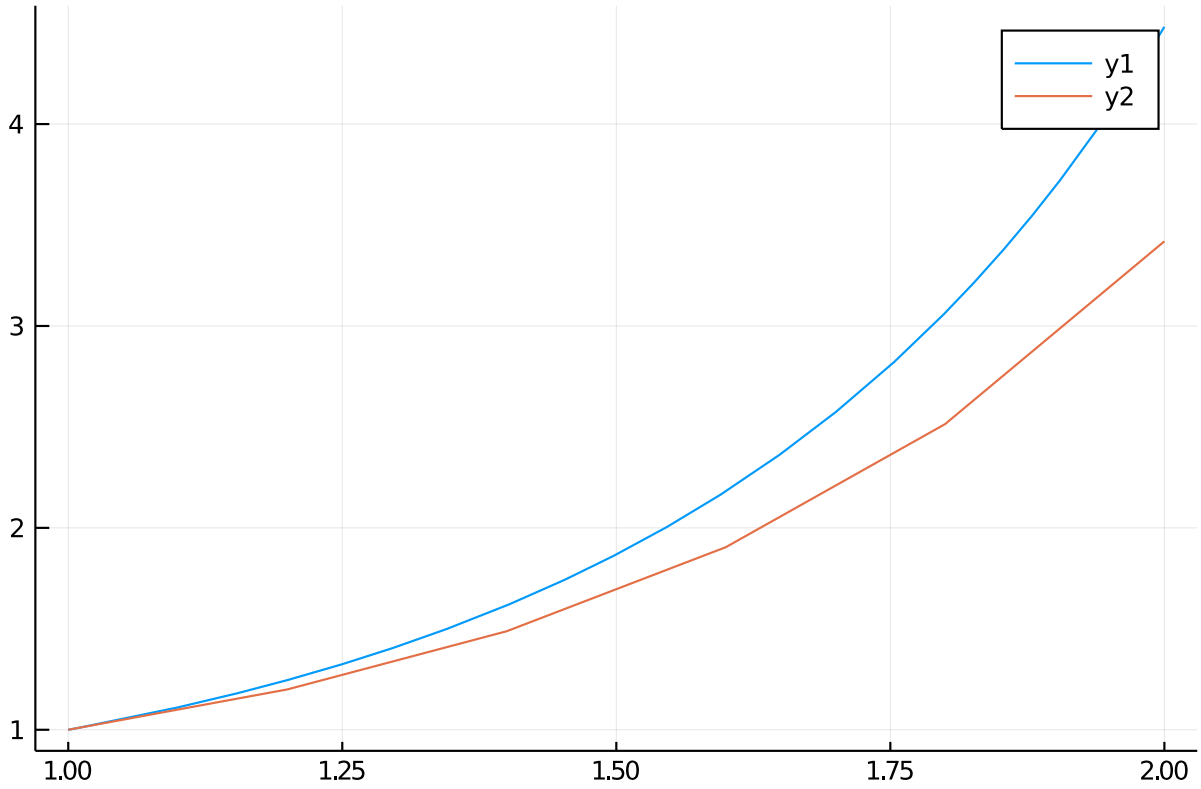
For our example, we can do some numerics. We want $h = 0.2$ and 5 pieces, so values of y at $x_0 = 1, x_1 = 1.2, x_2 = 1.4, x_3 = 1.6, x_4 = 1.8$, and $x_5 = 2$.

Below we do this in a loop. We have to be a bit careful, as in `Julia` the vector of zeros we create to store our answers begins indexing at 1, and not 0.

```
n=5
h = (2-1)/n
xs = zeros(n+1)
ys = zeros(n+1)
xs[1] = x0   # index is off by 1
ys[1] = y0
for i in 1:n
  xs[i + 1] = xs[i] + h
  ys[i + 1] = ys[i] + h * F(ys[i], xs[i])
end
```

So how did we do? Let's look graphically:

```
plot(exp(-1/2)*exp(x^2/2), x0, 2)
plot!(xs, ys)
```



Not bad. We wouldn't expect this to be exact - due to the concavity of the solution, each step is an underestimate. However, we see it is an okay approximation and would likely be better with a smaller h. A topic we pursue in just a bit.

Rather than type in the above command each time, we wrap it all up in a function. The inputs are n, $a = x_0$, $b = x_n$, y_0, and, most importantly, F. The output is massaged into a function through a call to `linterp`, rather than two vectors. The `linterp` function we define below just finds a function that linearly interpolates between the points and is `NaN` outside of the range of the x values:

```
function linterp(xs, ys)
    function(x)
        ((x < xs[1]) || (x > xs[end])) && return NaN
        for i in 1:(length(xs) - 1)
            if xs[i] <= x < xs[i+1]
                l = (x-xs[i]) / (xs[i+1] - xs[i])
                return (1-l) * ys[i] + l * ys[i+1]
            end
        end
        ys[end]
    end
end
```

```
linterp (generic function with 1 method)
```

With that, here is our function to find an approximate solution to $y' = F(y, x)$ with initial condition:

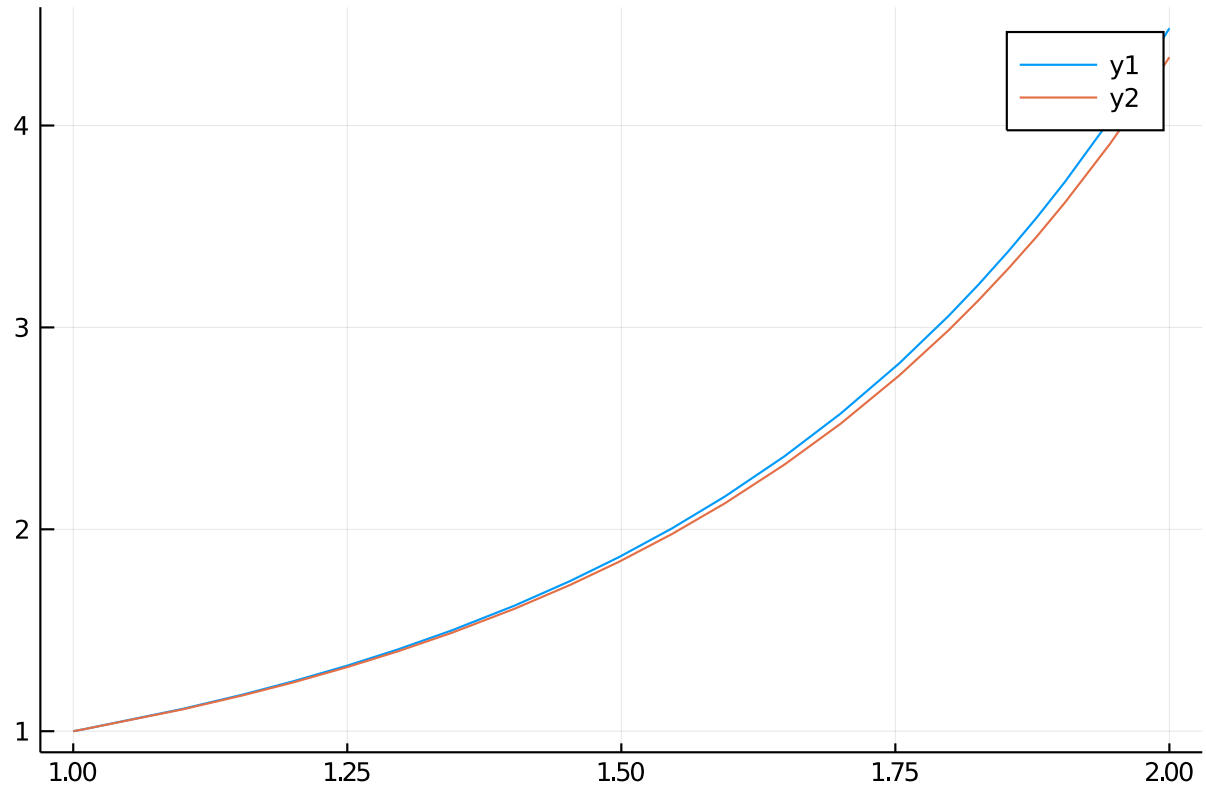
```
function euler(F, x0, xn, y0, n)
  h = (xn - x0)/n
  xs = zeros(n+1)
  ys = zeros(n+1)
  xs[1] = x0
  ys[1] = y0
  for i in 1:n
    xs[i + 1] = xs[i] + h
    ys[i + 1] = ys[i] + h * F(ys[i], xs[i])
  end
  linterp(xs, ys)
end
```

```
euler (generic function with 1 method)
```

With `euler`, it becomes easy to explore different values.

For example, we thought the solution would look better with a smaller h (or larger n). Instead of $n = 5$, let's try $n = 50$:

```
u = euler(F, 1, 2, 1, 50)
plot(exp(-1/2)*exp(x^2/2), x0, 2)
plot!(u, x0, 2)
```



It is more work for the computer, but not for us, and clearly a much better approximation to the actual answer is found.

Figure 1: Figure from first publication of Euler's method. From [Gander and Wanner](http://www.unige.ch/ gander/Preprints/Ritz.pdf).

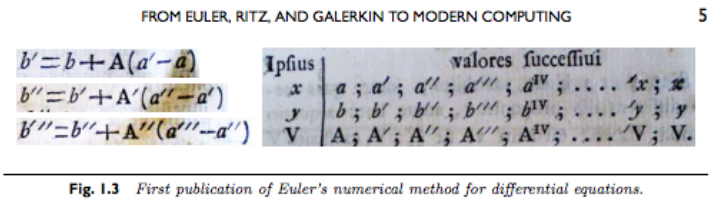


Fig. I.3 *First publication of Euler's numerical method for differential equations.*

1.1 The Euler method

The name of our function reflects the mathematician associated with the iteration:

$$x_{n+1} = x_n + h, \quad y_{n+1} = y_n + h \cdot F(y_n, x_n),$$

to approximate a solution to the first-order, ordinary differential equation with initial values: $y'(x) = F(y, x)$.

The Euler method uses linearization. Each "step" is just an approximation of the function value $y(x_{n+1})$ with the value from the tangent line tangent to the point (x_n, y_n).

Each step introduces an error. The error in one step is known as the *local truncation error* and can be shown to be about equal to $1/2 \cdot h^2 \cdot f''(x_n)$ assuming y has 3 or more derivatives.

The total error, or more commonly, *global truncation error*, is the error between the actual answer and the approximate answer at the end of the process. It reflects an accumulation of these local errors. This error is *bounded* by a constant times h. Since it gets smaller as h gets smaller in direct proportion, the Euler method is called *first order*.

Other, somewhat more complicated, methods have global truncation errors that involve higher powers of h - that is for the same size h, the error is smaller. In analogy is the fact that Riemann sums have error that depends on h, whereas other methods of approximating the integral have smaller errors. For example, Simpson's rule had error related to h^4. So, the Euler method may not be employed if there is concern about total resources (time, computer, ...), it is important for theoretical purposes in a manner similar to the role of the Riemann integral.

In the examples, we will see that for many problems the simple Euler method is satisfactory, but not always so. The task of numerically solving differential equations is not a one-size-fits-all one. In the following, a few different modifications are presented to the basic Euler method, but this just scratches the surface of the topic.

1.2 Examples

Example Consider the initial value problem $y'(x) = x+y(x)$ with initial condition $y(0) = 1$. This problem can be solved exactly. Here we approximate over $[0, 2]$ using Euler's method.

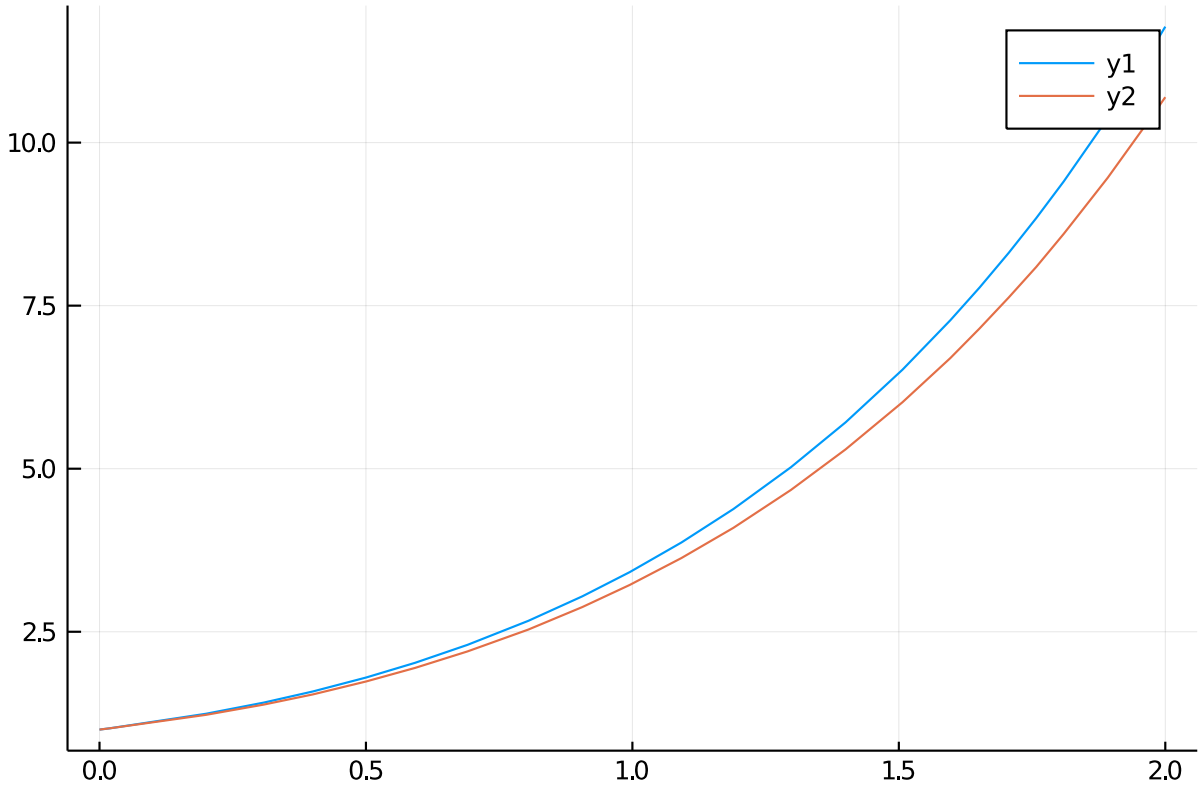
```
F(y,x) = x + y
x0, xn, y0 = 0, 2, 1
f = euler(F, x0, xn, y0, 25)
```

```
f(xn)
```

```
10.696950392438628
```

We graphically compare our approximate answer with the exact one:

```
plot(f, x0, xn)
u = SymFunction("u")
out = dsolve(u'(x) - F(u(x),x), u(x), ics = (u, x0, y0))
plot(rhs(out), x0, xn)
plot!(f, x0, xn)
```



From the graph it appears our value for `f(xn)` will underestimate the actual value of the solution slightly.

Example The equation $y'(x) = \sin(x \cdot y)$ is not separable, so need not have an easy solution. `SymPy` will return a power series *approximation*. Let's look at comparing an approximate answer given by the Euler method and to that one returned by `SymPy`.

First, the `SymPy` solution:

```
@vars x
u = SymFunction("u")
F(y,x) = sin(x*y)
eqn = u'(x) - F(u(x), x)
out = dsolve(eqn)
```

$$u(x) = C_1 + \frac{C_1 x^2}{2} + \frac{C_1 x^4 \left(3 - C_1^2\right)}{24} + O\left(x^6\right)$$

If we assume $y(0) = 1$, we can continue:

```
out = dsolve(eqn, u(x), ics=(u, 0, 1))
```

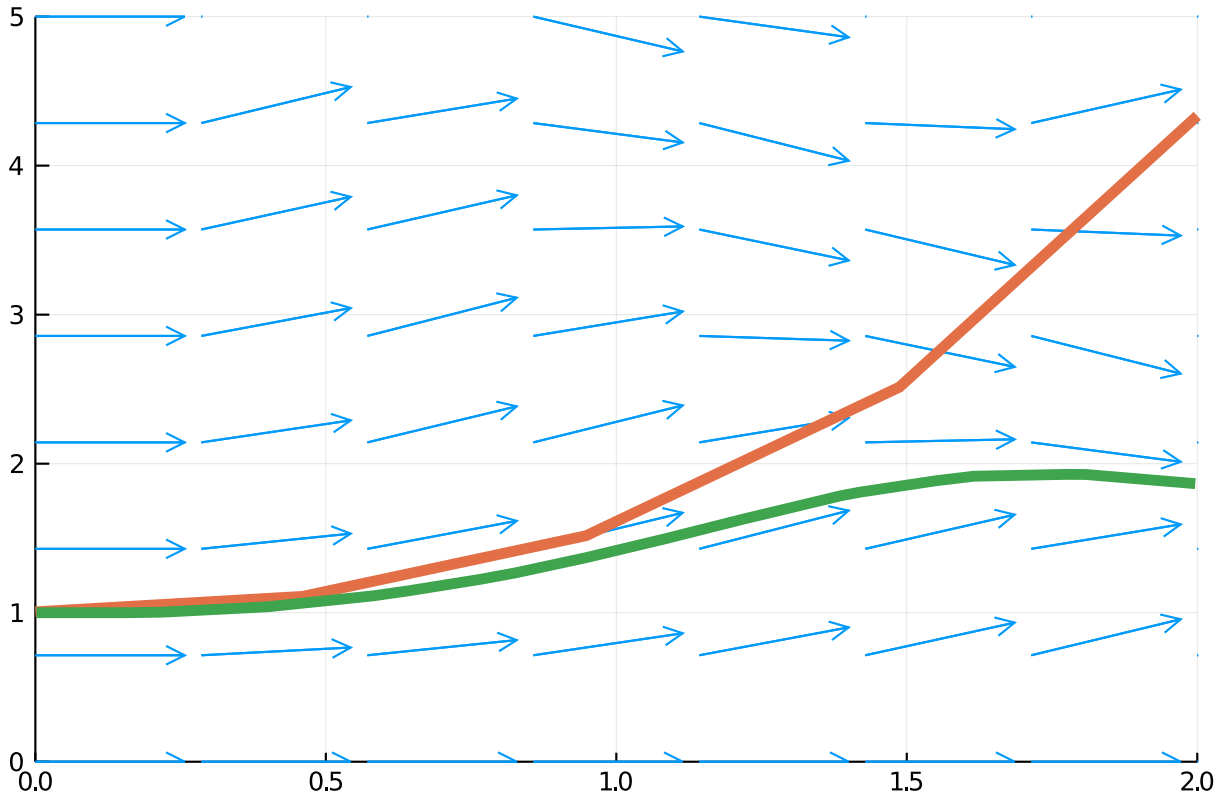
$$u(x) = 1 + \frac{x^2}{2} + \frac{x^4}{12} + O\left(x^6\right)$$

The approximate value given by the Euler method is

```
x0, xn, y0 = 0, 2, 1

p = plot(legend=false)
vectorfieldplot!((x,y) -> [1, F(y,x)], xlims=(x0,xn), ylims=(0,5))
plot!(rhs(out).removeO(),  linewidth=5)

u = euler(F, x0, xn, y0, 10)
plot!(u, linewidth=5)
```



We see that the answer found from using a polynomial series matches that of Euler's method for a bit, but as time evolves, the approximate solution given by Euler's method more closely tracks the slope field.

Example The Brachistochrone problem was posed by Johann Bernoulli in 1696. It asked for the curve between two points for which an object will fall faster along that curve than