1 Newton's method

The Babylonian method is an algorithm to find an approximate value for $\sqrt{k}$. It was described by the first-century Greek mathematician Hero of Alexandria.

The method starts with some initial guess, called x_0. It then applies a formula to produce an improved guess. This is repeated until the improved guess is accurate enough or it is clear the algorithm fails to work.

For the Babylonian method, the next guess, x_{i+1} derived from the current guess, x_i, is:

$$x_{i+1} = \frac{1}{2}(x_i + \frac{k}{x_i})$$

We use this algorithm to approximate the square root of 2, a value known to the Babylonians.

Start with x, then form $x/2 + 1/x$, from this again form $x/2 + 1/x$, repeat.

Let's look starting with $x = 2$ as a rational number:

```
x = 2//1
x = x//2 + 1//x
x, x^2.0
```

```
(3//2, 2.25)
```

Our estimate improved from something which squared to 4 down to something which squares to 2.25. A big improvement, but there is still more to come.

```
x = x//2 + 1//x
x, x^2.0
```

```
(17//12, 2.0069444444444446)
```

We now see accuracy until the third decimal point.

```
x = x//2 + 1//x
x, x^2.0
```

```
(577//408, 2.000006007304883)
```

This is now accurate to the sixth decimal point. That is about as far as we, or the Bablyonians, would want to go by hand. Using rational numbers quickly grows out of hand. The next step shows the explosion:

```
x = x//2 + 1//x
```

```
665857//470832
```

However, with the advent of floating point numbers, the method stays quite manageable:

```
x = 2.0
x = x/2 + 1/x   # 1.5, 2.25
x = x/2 + 1/x   # 1.4166666666666665, 2.006944444444444
x = x/2 + 1/x   # 1.4142156862745097, 2.0000060073048824
x = x/2 + 1/x   # 1.4142135623746899, 2.0000000000045106
x = x/2 + 1/x   # 1.414213562373095,  1.9999999999999996
x = x/2 + 1/x   # 1.414213562373095,  1.9999999999999996
```

```
1.414213562373095
```

We see that the algorithm - to the precision offered by floating point numbers - has resulted in an answer `1.414213562373095`. This answer is an *approximation* to the actual answer. Approximation is necessary, as $\sqrt{2}$ is an irrational number and so can never be exactly represented in floating point. That being said, we see that the value of $f(x)$ is accurate to the last decimal place, so our approximation is very close and is achieved in a few steps.

1.1 Newton's generalization

Let $f(x) = x^3 - 2x - 5$. The value of 2 is almost a zero, but not quite, as $f(2) = -1$. We can check that there are no *rational* roots. Though there is a method to solve the cubic it may be difficult to compute and will not be as generally applicable as some algorithm like the Babylonian method to produce an approximate answer.

Is there some generalization to the Babylonian method?

We know that the tangent line is a good approximation to the function at the point. Looking at this graph gives a hint as to an algorithm:

```
Plot{Plots.PlotlyBackend() n=4}
```

The tangent line and the function nearly agree near 2. So much so, that the intersection point of the tangent line with the x axis nearly hides the actual zero of $f(x)$ that is near 2.1.

That is, it seems that the intersection of the tangent line and the x axis should be an improved approximation for the zero of the function.

Let x_0 be 2, and x_1 be the intersection point of the tangent line at $(x_0, f(x_0))$ with the x axis. Then by the definition of the tangent line:

$$f'(x_0) = \frac{\Delta y}{\Delta x} = \frac{f(x_0)}{x_0 - x_1}.$$

This can be solved for x_1 to give $x_1 = x_0 - f(x_0)/f'(x_0)$. In general, if we had x_i and used the intersection point of the tangent line to produce x_{i+1} we would have Newton's method:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

We will use automatic derivatives, as possible, so load the `CalculusWithJulia` package which provides the `f'` notation for derivatives through the definition `Base.adjoint(f::Function)=x->Forward`
`float(x))`:

```
using CalculusWithJulia
```

With this, the algorithm above starting from 2 becomes:

```
x0 = 2
x1 = x0 - f(x0)/f'(x0)
```

```
2.1
```

We can see we are closer to a zero:

```
f(x0), f(x1)
```

```
(-1, 0.06100000000000083)
```

Trying again, we have

```
x2 = x1 - f(x1)/ f'(x1)
x2, f(x2), f(x1)
```

```
(2.094568121104185, 0.00018572317327247845, 0.06100000000000083)
```

And again:

```
x3 = x2 - f(x2)/ f'(x2)
x3, f(x3), f(x2)
```

```
(2.094551481698199, 1.7397612239733462e-9, 0.00018572317327247845)
```

```
x4 = x3 - f(x3)/ f'(x3)
x4, f(x4), f(x3)
```

```
(2.0945514815423265, -8.881784197001252e-16, 1.7397612239733462e-9)
```

We see now that $f(x_4)$ is within machine tolerance of 0, so we call x_4 an *approximate zero* of $f(x)$.

Newton's method. Let x_0 be an initial guess for a zero of $f(x)$. Iteratively define x_{i+1} in terms of the just generated x_i by: $x_{i+1} = x_i - f(x_i)/f'(x_i)$. Then for most functions and reasonable initial guesses, the sequence of points converges to a zero of f.

On the computer, we know that actual convergence will likely never occur, but accuracy to a certain tolerance can often be achieved.

In the example above, we kept track of the previous values. This is unnecessary if only the answer is sought. In that case, the update step can use the same variable:

```
x = 2                       # x0
x = x - f(x) / f'(x)        # x1
x = x - f(x) / f'(x)        # x2
x = x - f(x) / f'(x)        # x3
x = x - f(x) / f'(x)        # x4
```

```
2.0945514815423265
```

As seen above, the assignment will update the value bound to `x` using the previous value of `x` in the computation.

We implement the algorithm by repeating the step until either we converge or it is clear we won't converge. For good guesses and most functions, convergence happens quickly.

Newton looked at this same example in 1699 (B.T. Polyak, *Newton's method and its use in optimization*, European Journal of Operational Research. 02/2007; 181(3):1086-1096.) though his technique was slightly different as he did not use the derivative, *per se*, but rather an approximation based on the fact that his function was a polynomial (though identical to the derivative). Raphson (1690) proposed the general form, hence the usual name of the Newton-Raphson method.

1.2 Examples

Example: visualizing convergence This graphic demonstrates the method and the rapid convergence:

XXX can not include '.gif' file here

Example: numeric not algebraic For the function $f(x) = \cos(x) - x$, we see that SymPy can not solve symbolically for a zero:

```
@vars x real=true
solve(cos(x) - x, x)
```

```
Error: PyError ($(Expr(:escape, :(ccall(#= /Users/verzani/.julia/packages/P
yCall/zqDXB/src/pyfncall.jl:43 =# @pysym(:PyObject_Call), PyPtr, (PyPtr, Py
Ptr, PyPtr), o, pyargsptr, kw))))) <class 'NotImplementedError'>
NotImplementedError('multiple generators [x, cos(x)]\nNo algorithms are imp
lemented to solve equation -x + cos(x)')
  File "/Users/verzani/.julia/conda/3/lib/python3.7/site-packages/sympy/sol
vers/solvers.py", line 1174, in solve
    solution = _solve(f[0], *symbols, **flags)
  File "/Users/verzani/.julia/conda/3/lib/python3.7/site-packages/sympy/sol
vers/solvers.py", line 1748, in _solve
    raise NotImplementedError('\n'.join([msg, not_impl_msg % f]))
```

We can find a numeric solution, even though there is no closed-form answer. Here we try Newton's method:

```
f(x) = cos(x) - x
x = .5
x = x - f(x)/f'(x)  # 0.7552224171056364
x = x - f(x)/f'(x)  # 0.7391416661498792
x = x - f(x)/f'(x)  # 0.7390851339208068
x = x - f(x)/f'(x)  # 0.7390851332151607
x = x - f(x)/f'(x)
```

```
0.7390851332151607
```

This answer is close, to machine tolerance it produces 0.0:

```
x, f(x)
```

```
(0.7390851332151607, 0.0)
```

Example division as multiplication Newton-Raphson Division is a means to divide by multiplying.

Why would you want to do that? Well, even for computers division is harder (read slower) than multiplying. The trick is that p/q is simply $p \cdot (1/q)$, so finding a means to compute a reciprocal by multiplying will reduce division to multiplication. (This trick is used by yeppp, a high performance library for computational mathematics.)

Well suppose we have q, we could try to use Newton's method to find $1/q$, as it is a solution to $f(x) = x - 1/q$. The Newton update step simplifies to:

$$x - f(x)/f'(x) \quad \text{or} \quad x - (x - 1/q)/1 = 1/q$$

That doesn't really help, as Newton's method is just $x_{i+1} = 1/q$

- that is it just jumps to the answer, the one we want to compute by

some other means!

Trying again, we simplify the update step for a related function: $f(x) = 1/x - q$ with $f'(x) = -1/x^2$ and then one step of the process is:

$$x_{i+1} = x_i - (1/x_i - q)/(-1/x_i^2) = -qx_i^2 + 2x_i.$$

Now for q in the interval $[1/2, 1]$ we want to get a *good* initial guess. Here is a claim. We can use $x_0 = 48/17 - 32/17 \cdot q$. Let's check graphically that this is a reasonable initial approximation to $1/q$:

```
g(q) = 1/q
h(q) = 1/17 * (48 - 32q)
plot(g, 1/2, 1)
plot!(h)
```

```
Plot{Plots.PlotlyBackend() n=2}
```

It can be shown that we have for any q in $[1/2, 1]$ with initial guess $x_0 = 48/17 - 32/17 \cdot q$ that Newton's method will converge to 16 digits in no more than this many steps:

$$\log_2(\frac{53+1}{\log_2(17)}).$$

```
a = log2((53 + 1)/log2(17))
ceil(Integer, a)
```

```
4
```

That is 4 steps suffices.

For $q = 0.80$, to find $1/q$ using the above we have

```
q = 0.80
x = (48/17) - (32/17)*q
x = -q*x*x + 2*x
x = -q*x*x + 2*x
x = -q*x*x + 2*x
x = -q*x*x + 2*x
```

```
1.25
```

This method has basically 18 multiplication and addition operations for one division, so it naively would seem slower, but timing this shows the method is competitive with a regular division.

1.3 A function

In the previous example, a bound ensures convergence in 4 steps. In general, this is not the case with Newton's method where the algorithm is iterated until convergence. Having to repeat steps until something happens is a task best done by the computer. The `while` loop is a good way to repeat commands until some condition is met. With this, we present a simple function implementing Newton's method, we iterate until the update step gets really small (the `delta`) or the convergence takes more than 50 steps. (There are other reasonable choices that could be used to determine when the algorithm should stop.)

```
function nm(f, fp, x0)
  tol = 1e-14
  ctr = 0
  delta = Inf
  while (abs(delta) > tol) & (ctr < 50)
    delta = f(x0)/fp(x0)
    x0 = x0 - delta
    ctr = ctr + 1
  end

  ctr < 50 ? x0 : NaN
end
```

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

Examples

- Find a zero of $\sin(x)$ starting at $x_0 = 3$:

```
nm(sin, cos, 3)
```

```
3.141592653589793
```

This is an approximation for π, that historically found use, as the convergence is fast.

- Find a solution to $x^5 = 5^x$ near 2:

Writing a function to handle this, we have:

```
f(x) = x^5 - 5^x
```

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

We can find the derivative, but in this example will let the `D` function from the `Roots` package do so for us:

```
alpha = nm(f, f', 2)
alpha, f(alpha)
```

```
(1.764921914525776, 0.0)
```

1.3.1 Functions in the Roots package

Typing in the `nm` function might be okay once, but would be tedious if it was needed each time. The `Roots` package provides a `Newton` method. `Roots` is loaded with

```
using Roots
find_zero((sin, cos), 3, Roots.Newton())  # alternatively Roots.newton(sin,cos, 3)
```

```
3.141592653589793
```

Or, if a derivative is not specified, one can be computed using automatic differentiation:

```
find_zero((f, f'), 2, Roots.Newton())
```

```
1.764921914525776
```

The argument `verbose=true` will force a print out of a message summarizing each step.

More generally, the function `find_zero` provides a derivative-free algorithm for finding roots of functions, when started with an initial guess. It is similar to Newton's method in that only a good initial guess is needed. However, the algorithm, while slower in terms of function evaluations and steps, is engineered to be a bit more robust to the choice of initial estimate than Newton's method. (If it finds a bracket, it will use a bisection algorithm which is guaranteed to converge, but can be slower to do so.) Here we see how to call the function:

```
f(x) = cos(x) - x
x0 = 1
find_zero(f, x0)
```

```
0.7390851332151607
```

Compare to this related call which uses the bisection method:

```
find_zero(f, (0, 1))           ## [0,1] must be a bracketing interval
```

```
0.7390851332151607
```

For this example both give the same answer, but the bisection method is a bit more inconvenient as a bracketing interval must be pre-specified.

Example: intersection of two graphs Find the intersection point between $f(x) = \cos(x)$ and $g(x) = 5x$ near 0.

We have Newton's method to solve for zeros of $f(x)$, i.e. when $f(x) = 0$. Here we want to solve for x with $f(x) = g(x)$. To do so, we make a new function $h(x) = f(x) - g(x)$, for that is 0 when $f(x)$ equals $g(x)$:

```
f(x) = cos(x)
g(x) = 5x
h(x) = f(x) - g(x)
x0 = find_zero((h,h'), 0, Roots.Newton())
x0, h(x0), f(x0), g(x0)
```

```
(0.19616428118784215, 0.0, 0.9808214059392107, 0.9808214059392107)
```

Example: Finding c in Rolle's Theorem The function $f(x) = \sqrt{1 - \cos(x^2)^2}$ has a zero at 0 and one near 1.77.

```
f(x) = sqrt(1 - cos(x^2)^2)
plot(f, 0, 1.77)
```

```
Plot{Plots.PlotlyBackend() n=1}
```

As $f(x)$ is differentiable between 0 and a, Rolle's theorem says there will be value where the derivative is 0. Find that value.

This value will be a zero of the derivative. A graph shows it should be near 1.2, so we use that as a starting value to get the answer:

```
find_zero(f', 1.2)
```

```
1.2533141373155003
```

1.4 Convergence

Newton's method is famously known to have "quadratic convergence." What does this mean? Let the error in the ith step be called $e_i = x_i - \alpha$. Then Newton's method satisfies a bound of the type:

$$|e_{i+1}| \leq M_i \cdot e_i^2.$$

If M were just a constant and we suppose $e_0 = 10^{-1}$ then e_1 would be less than $M10^{-2}$ and e_2 less than $M^2 10^{-4}$, e_3 less than $M^3 10^{-8}$ and e_4 less than $M^4 10^{-16}$ which for $M = 1$ is basically the machine precision. That is for some problems, with a good initial guess it will take around 4 or so steps to converge.

The actual value of M depends on i and f, so the answer isn't always so easy. To see what M is, the basic assumption of f is such that this fact of linearization holds at each x_i with $f(x_i) \neq 0$:

$$f(x) = f(x_i) + f'(x_i) \cdot (x - x_i) + \frac{1}{2} f''(\xi) \cdot (x - x_i)^2.$$

The value ξ is from the mean value theorem and is between x and x_i.

Let $x = \alpha$, the zero of $f(x)$ that is being sought. Then $f(\alpha) = 0$ and $0 = f(x_i)/f'(x_i) + (\alpha - x_i) + 1/2 \cdot f''(\xi)/f'(x_i) \cdot (\alpha - x_i)^2$. For this value, we have

$$x_{i+1} - \alpha = x_i - \frac{f(x_i)}{f'(x_i)} - \alpha = (x_i - \alpha) + (\alpha - x_i) + \frac{1}{2} \frac{f''(\xi) \cdot (\alpha - x_i)^2}{f'(x_i)} = \frac{1}{2} \frac{f''(\xi)}{f'(x_i)} \cdot (x_i - \alpha)^2.$$

That is

$$|e_{i+1}| \leq \frac{1}{2} \frac{|f''(\xi)|}{|f'(x_i)|} e_i^2.$$

This convergence will be quadratic *if*:

- The initial guess x_0 is not too far from α, so e_0 is managed.
- The derivative at x_i is not too close to 0. (As it appears in the denominator). That is, the function can't be too flat, which should make sense, as then the tangent line is nearly parallel to the x axis and would intersect far away.
- The second derivative is not too big (in absolute value) near the zero. A large second derivative means the function is very concave, which means it is "turning" a lot. In this case, the function turns away from the tangent line quickly, so the tangent line's zero is not necessarily a better approximation to the actual zero, α.

The basic tradeoff: methods like Newton's are faster than the bisection method in terms of function calls, but are not guaranteed to converge, as the bisection method is.

What can go wrong when one of these isn't the case is illustrated next:

1.4.1 Poor initial step

XXX can not include '.gif' file here

XXX can not include '.gif' file here

1.4.2 The second derivative is too big

XXX can not include '.gif' file here

1.4.3 The tangent line at some x_i is flat

XXX can not include '.gif' file here

⊛ Example
Suppose α is a simple zero for $f(x)$. (The value α is a zero of multiplicity k if $f(x) = (x-\alpha)^k g(x)$ where $g(\alpha)$ is not zero.) A simple zero has multiplicity 1. If $f'(\alpha) \neq 0$ and the second derivative exists, then a zero α will be simple.) Around α, quadratic convergence should apply. However, consider the function $g(x) = f(x)^k$ for some integer $k \geq 2$. Then α is still a zero, but the derivative of g at α is zero, so the tangent line is basically flat. This will slow the convergence up. We can see that the update step $g'(x)/g(x)$ becomes $(1/k)f'(x)/f(x)$, so an extra factor is introduced.

The calculation that produces the quadratic convergence now becomes:

$$x_{i+1} - \alpha = (x_i - \alpha) - \frac{1}{k}(x_i - \alpha + \frac{f''(\xi)}{2f'(x_i)}(x_i - \alpha)^2) = \frac{k-1}{k}(x_i - \alpha) + \frac{f''(\xi)}{2kf'(x_i)}(x_i - \alpha)^2.$$

As $k > 1$, the $(x_i - \alpha)$ term dominates, and we see the convergence is linear with $|e_{i+1}| \approx (k-1)/k|e_i|$.

1.5 Questions

⊛ Question
Look at this graph with x_0 marked with a point:

```
Plot{Plots.PlotlyBackend() n=3}
```

If one step of Newton's method was used, what would be the value of x_1?

1. $$-2.224$$
2. $$-2.80$$
3. $$-0.020$$
4. $$0.355$$

⊛ Question
Look at this graph of some concave up $f(x)$ with initial point x_0 marked. Let c be the zero.

```
Plot{Plots.PlotlyBackend() n=3}
```

What can be said about x_1?

1. It must be $x_1 < x_0$
2. It must be $x_1 > c$
3. It must be $x_0 < x_1 < c$

⊛ Question
Look at this graph of some concave up $f(x)$ with initial point x_0 marked. Let c be the zero.

```
Plot{Plots.PlotlyBackend() n=3}
```

What can be said about x_1?

1. It must be $c < x_1 < x_0$
2. It must be $x_1 < c$
3. It must be $x_1 > x_0$

⊛ Question
Suppose $f(x)$ is concave up and we have the tangent line representation: $f(x) = f(c) + f'(c) \cdot (x-c) + f''(\xi)/2 \cdot (x-c)^2$. Explain why it must be that the graph of $f(x)$ lies on or *above* the tangent line.

1. As $f''(\xi) < 0$ it must be that $f(x) - (f(c) + f'(c) \cdot (x-c)) \geq 0$.
2. As $f''(\xi)/2 \cdot (x-c)^2$ is non-negative, we must have $f(x) - (f(c) + f'(c) \cdot (x-c)) \geq 0$.
3. This isn't true. The function $f(x) = x^3$ at $x = 0$ provides a counterexample

⊛ Question
Let $f(x) = x^2 - 3^x$. This has derivative $2x - 3^x \cdot \log(3)$. Starting with $x_0 = 0$, what does Newton's method converge on?

⊛ Question
Let $f(x) = \exp(x) - x^4$. There are 3 zeros for this function. Which one does Newton's method converge to when $x_0 = 2$?

⊛ Question
Let $f(x) = \exp(x) - x^4$. As mentioned, there are 3 zeros for this function. Which one does Newton's method converge to when $x_0 = 8$?

⊛ Question
Let $f(x) = \sin(x) - \cos(4 \cdot x)$.

Starting at $\pi/8$, solve for the root returned by Newton's method

⊛ Question
Using Newton's method find a root to $f(x) = \cos(x) - x^3$ starting at $x_0 = 1/2$.

⊛ Question
Use Newton's method to find a root of $f(x) = x^5 + x - 1$. Make a quick graph to find a reasonable starting point.

⊛ Question
Will Newton's method converge for the function $f(x) = x^5 - x + 1$ starting at $x = 1$?

1. Yes
2. No. The initial guess is not close enough
3. No. The second derivative is too big
4. No. The first derivative gets too close to 0 for one of the x_i

⊛ Question
Will Newton's method converge for the function $f(x) = 4x^5 - x + 1$ starting at $x = 1$?

1. Yes
2. No. The initial guess is not close enough
3. No. The second derivative is too big, or does not exist
4. No. The first derivative gets too close to 0 for one of the x_i

⊛ Question
Will Newton's method converge for the function $f(x) = x^{10} - 2x^3 - x + 1$ starting from 0.25?

1. Yes
2. No. The initial guess is not close enough
3. No. The second derivative is too big, or does not exist
4. No. The first derivative gets too close to 0 for one of the x_i

⊛ Question
Will Newton's method converge for $f(x) = 20x/(100x^2 + 1)$ starting at 0.1?

1. Yes
2. No. The initial guess is not close enough
3. No. The second derivative is too big, or does not exist
4. No. The first derivative gets too close to 0 for one of the x_i

⊛ Question
Will Newton's method converge to a zero for $f(x) = \sqrt{(1 - x^2)^2}$?

1. Yes
2. No. The initial guess is not close enough
3. No. The second derivative is too big, or does not exist
4. No. The first derivative gets too close to 0 for one of the x_i

⊛ Question
Use `find_zero` to find a root of $f(x) = 4x^4 - 5x^3 + 4x^2 - 20x - 6$ starting at $x_0 = 0$.

⊛ Question
Use `find_zero` to find a zero of $f(x) = \sin(x) - x/2$ that is *bigger* than 0.

⊛ Question
The Newton baffler (defined below) is so named, as Newton's method will fail to find the root for most starting points.

```julia
function newton_baffler(x)
    if ( x - 0.0 ) < -0.25
        0.75 * ( x - 0 ) - 0.3125
    elseif  ( x - 0 ) < 0.25
        2.0 * ( x - 0 )
    else
        0.75 * ( x - 0 ) + 0.3125
    end
end
```

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

Will `find_zero` find the zero at 0.0 starting at 1 using the default option for `order`?

1. Yes
2. No

Will Newton's method find the zero at 0.0 starting at 1?

1. Yes
2. No

Considering this plot:

```julia
plot(newton_baffler, -1.1, 1.1)
```

```
Plot{Plots.PlotlyBackend() n=1}
```

Starting with $x_0 = 1$, you can see why Newton's method will fail. Why?

1. The tangent lines for $|x| > 0.25$ intersect at x values with $|x| > 0.25$
2. It doesn't fail, it converges to 0
3. The first derivative is 0 at 1

⊛ Question
Consider this crazy function defined by:

```julia
import SpecialFunctions: erf
f(x) = cos(100*x)-4*erf(30*x-10)
```

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

(The `erf` function is the error function.)

Make a plot over the interval $[-3, 3]$ to see why it is called ”crazy”.

Does `find_zero` find a zero to this function starting from 0?

1. Yes
2. No

If so, what is the value?

If not, what is the reason?

1. The zero is a simple zero
2. The zero is not a simple zero
3. The function oscillates too much to rely on the tangent line approximation far from the zero
4. We can find an answer

Does `find_zero` find a zero to this function starting from 1?

1. Yes
2. No

If so, what is the value?

If not, what is the reason?

1. The zero is a simple zero
2. The zero is not a simple zero
3. The function oscillates too much to rely on the tangent line approximations far from the zero
4. We can find an answer

⊛ Question
Let $f(x) = \sin(x) - x/4$. Starting at $x_0 = 2\pi$ Newton's method will converge to a value, but it will take many steps. Using `verbose=true` when calling the `newton` function in the `Roots` package, how many steps does it take:

What is the zero that is found?

Is this the closest zero to the starting point, x_0?

1. Yes
2. No

⊛ Question
Quadratic convergence of Newton's method only applies to *simple* roots. For example, we can see (using the `verbose=true` argument to the `Roots` package's `newton` method, that it only takes 4 steps to find a zero to $f(x) = \cos(x) - x$ starting at $x_0 = 1$. But it takes many more steps to find the same zero for $f(x) = (\cos(x) - x)^2$.

How many?

⊛ Question: implicit equations
The equation $x^2 + x \cdot y + y^2 = 1$ is a rotated ellipse.

```
Plot{Plots.PlotlyBackend() n=1}
```

Can we find which point on its graph has the largest y value?

This would be straightforward *if* we could write $y(x) = \dots$, for then we would simply find the critical points and investiate. But we can't so easily solve for y interms of x. However, we can use Newton's method to do so:

```
function findy(x)
  fn = y -> (x^2 + x*y + y^2) - 1
  fp = y -> (x + 2y)
  find_zero((fn, fp), sqrt(1 - x^2), Roots.Newton())
end
```

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

For a *fixed* x, this solves for y in the equation: $F(y) = x^2 + x \cdot y + y^2 - 1 = 0$. It should be that (x, y) is a solution:

```
x = .75
y = findy(x)
x^2 + x*y + y^2  ## is this 1?
```

```
1.0000000000000002
```

So we have a means to find $y(x)$, but it is implicit. We can't readily find the derivative to find critical points. Instead we can use the approximate derivative with $h = 10^{-6}$:

```
yp(x) = (findy(x + 1e-6) - findy(x)) / 1e-6
```

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

Using `find_zero`, find the value x which maximizes `yp`. Use this to find the point (x, y) with largest y value.

1. $$(0.577, 0.577)$$
2. $$(-0.577, 1.155)$$
3. $$(0, -0.577)$$
4. $$(0, 0)$$

⊛ Question
In the last problem we used an *approximate* derivative in place of the derivative. This can introduce an error due to the approximation. Will this be true if we replace the derivative in Newton's method with an approximation? In general, this can often be done *but* the convergence can be *slower* and the sensitivity to a poor initial guess even greater.

Three common approximations are given by the difference quotient for a fixed h: $f'(x_i) \approx (f(x_i + h) - f(x_i))/h$; the secant line approximation: $f'(x_i) \approx (f(x_i) - f(x_{i-1}))/(x_i - x_{i-1})$; and the Steffensen approximation $f'(x_i) \approx (f(x_i + f(x_i)) - f(x_i))/f(x_i)$ (using $h = f(x_i)$).

Let's revisit the 4-step convergence of Newton's method to the root of $f(x) = 1/x - q$ when $q = 0.8$. Will these methods be as fast?

```
q = 0.8
xstar = 1.25 # q = 4/5 --> 1/q = 5/4
f(x) = 1/x - q
```

```
f (generic function with 2 methods)
```

Let's define the above approximations for a given `f`:

```
delta = 1e-6
secant_approx(x0,x1) = (f(x1) - f(x0)) / (x1 - x0)
diffq_approx(x0, h) = secant_approx(x0, x0+h)
steff_approx(x0) = diffq_approx(x0, f(x0))
```

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

Then using the difference quotient would look like:

```
x1 = 42/17 - 32/17*q
x1 = x1 - f(x1) / diffq_approx(x1, delta)   # |x1 - xstar| = 0.06511395862036995
x1 = x1 - f(x1) / diffq_approx(x1, delta)   # |x1 - xstar| = 0.003391809999860218; etc
```

```
1.2466081900001398
```

The Steffensen method would look like:

```
x1 = 42/17 - 32/17*q
x1 = x1 - f(x1) / steff_approx(x1)   # |x1 - xstar| = 0.011117056291670258
x1 = x1 - f(x1) / steff_approx(x1)   # |x1 - xstar| = 3.502579696146313e-5; etc.
```

```
1.2499649742030385
```

And the secant method like:

```
x1 = 42/17 - 32/17*q
x0 = x1 - delta # we need two initial values
x0, x1 = x1, x1 - f(x1) / secant_approx(x0, x1)   # |x1 - xstar| = 8.222358365284066e-6
x0, x1 = x1, x1 - f(x1) / secant_approx(x0, x1)   # |x1 - xstar| =
1.8766323799379592e-6; etc.
```

```
(1.1848855848819007, 1.235138592314222)
```

Repeat each of the above algorithms until `abs(x1 - 1.25)` is `0` (which will happen for this problem, though not in general). Record the steps.

- Does the difference quotient need *more* than 4 steps?

1. Yes
2. No

- Does the secant method need *more* than 4 steps?

1. Yes
2. No

- Does the Steffensen method need *more* than 4 steps?

1. Yes
2. No

All methods work quickly with this well-behaved problem. In general the convergence rates are slightly different for each, with the Steffensen method matching Newton's method and the difference quotient method being slower in general. All can be more sensitive to the initial guess.