Ordinary Differential Equations (ODEs)

Differential equations describe lots of things in pretty much every modern field including physics, chemistry, engineering, finance etc....

They describe the rate of change generally of some quantity be it position, momentum, energy and so on. This is expressed as a derivative of some function of that quantity.

The simplest type of this is when your equation is a function of one variable. This is called an ordinary differential equation.

$$\frac{d^n}{dt^n} f(t) = g(t)$$

The simplest type of ODE is when the derivative is a first derivative, for example

$$f'(t) = t$$

Does this equation have a unique solution? No, we need an initial condition

$$f(t_0) = f_0$$

In general though, a good question to ask is whether a solution actually exists for any right hand side $g(t)$? Furthermore, if a solution exists is it unique?

Here are some cases:

$$f'(t) = \sqrt{f(t)}, f(0) = 0$$

$$f'(t) = f(t)^2, f(0) = 1$$

Do these have unique solutions? Turns out the first case does not have a unique solution, the second case does not have a global solution.

For the first case, can solve by just good old integration

$$\int \frac{1}{\sqrt{f}} df = \int t dt$$
$$2\sqrt{f(t)} = t$$
$$f(t) = \frac{t^2}{4}$$

Which is a solution. However so is the solution $f(t) = 0$. So which one is correct?

For the second case

$$\int \frac{1}{f^2} df = \int t dt$$
$$-\frac{1}{f(t)} = t - 1$$

$$f(t) = \frac{1}{1-t}$$

so the solution does not exist at $t = 1$. So then what do we do?

We are thus interested in knowing when we can expect the ODE to be uniquely solvable. This is called well-posedness. There is a very well known theorem for well-posedness of ODEs called the Picard-Lindelof theorem

Picard-Lindelof Theorem

If the right hand side $g(t)$ is Lipschitz continuous, then a unique solution to the ODE exists.

Lipschitz continuity: A function f is Lipschitz continuous if there exists some constant L such that for all x, y, $|f(x) - f(y)| < K|x - y|$.

The fact that such a theorem exists is actually pretty amazing - for context there doesn't exist anything like this for PDEs (when f is a function of more than one variable). A lot of the problems in modern mathematics are on the topic of showing well-posedness of various PDEs (such as the Navier-Stokes equations).

For problems in this class, we will assume that they are well-posed so that we can actually even attempt to solve them computationally.

The idea for solving these computationally lies in basic calculus, namely the FTC

$$f'(t) = g(t)$$
$$f(t) = f(t_0) + \int_{t_0}^{t} f'(t)dt$$
$$f(t) = f(t_0) + \int_{t_0}^{t} g(t)dt$$

If we knew the value of that integral, then we could calculate $f(t)$ exactly. Obviously we can't though in general which is where numerics come in. The goal then is to approximate that integral.

The first thing you might say is that the integral is roughly approximated by $g(t_0) * (t - t_0)$ given our knowledge of left Riemann sums. This gives

$$f(t) \approx f(t_0) + g(t_0)(t - t_0)$$

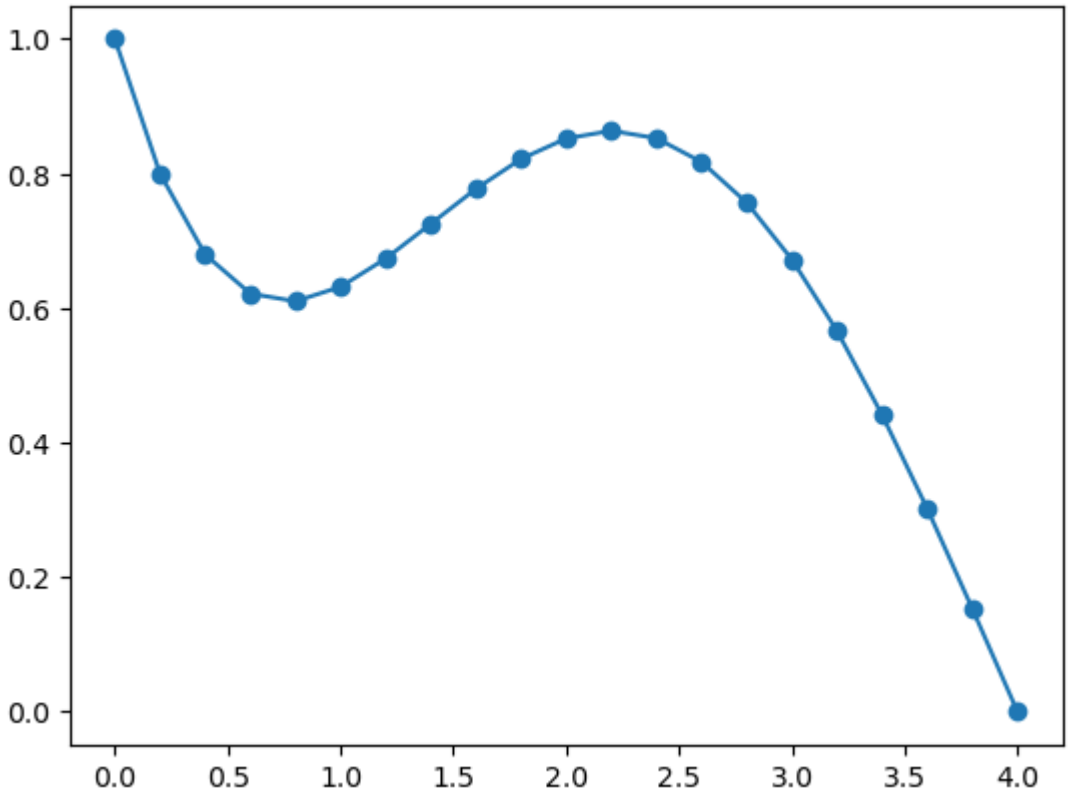
This is known as the forward Euler method.

The idea then is to use small timesteps h such that if we know the solution at t_0, we can then use it to get the solution at $t_0 + h$, then use that to get the solution at $t_0 + 2h$, and so on.

In [3]:
```
using PyPlot

function euler(f, y0, h, N, t0=0.0)
    t = t0 .+ h*(0:N)
    y = zeros(N+1, length(y0))

    y[1,:] .= y0
    for n = 1:N
        y[n+1,:] = y[n,:] + h * f(t[n], y[n,:])
    end

    return t,y
end
```

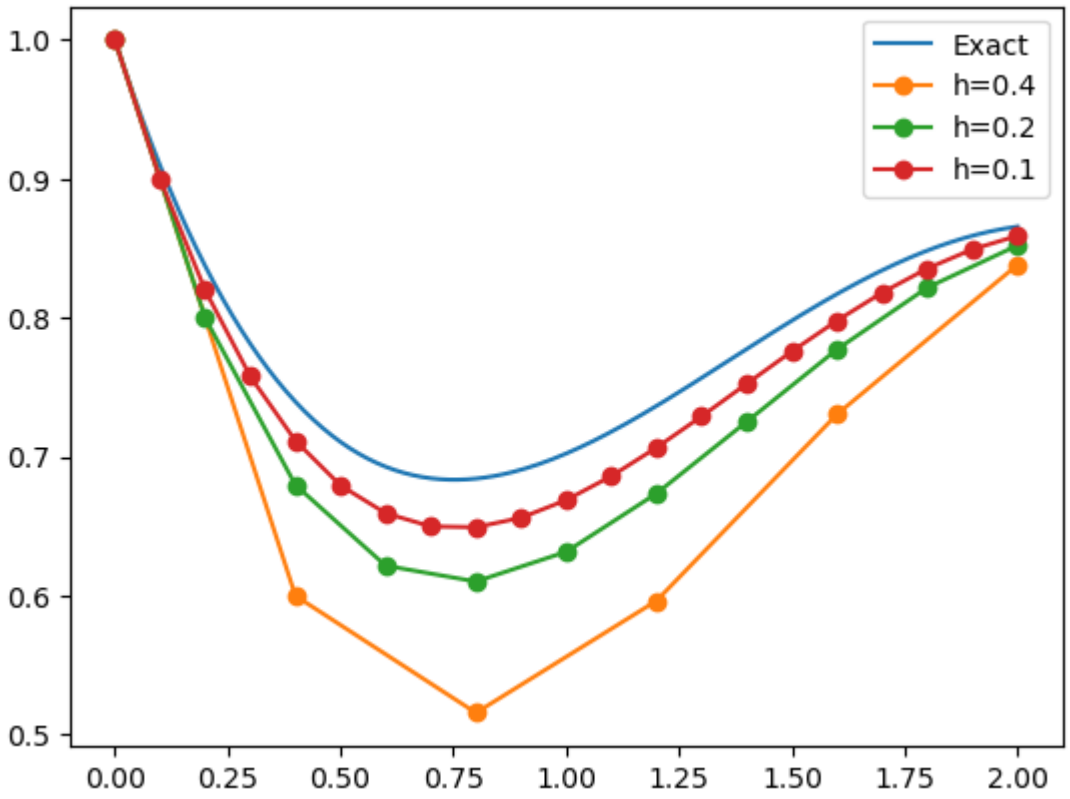
Out[3]: euler (generic function with 2 methods)

In [4]:
```
f(t,y) = -y .+ sin(t)
t,y = euler(f, 1, 0.2, 20)
plot(t, y, "-o");
```



How accurate is the approximation? We can check by trying different timestep sizes h:

In [14]:
```
yexact(t) = exp(-t) + (sin(t) - cos(t) + exp(-t)) / 2
tt = 0:0.01:2
plot(tt, yexact.(tt))
for h = [0.4, 0.2, 0.1]
    t,y = euler(f, 1, h, round(Int, 2/h))
    plot(t, y, "-o")
end
legend(("Exact", "h=0.4", "h=0.2", "h=0.1"));
```



Unsurprisingly, it seems to get more accurate with smaller timesteps. How do we characterise how fast it gets more accurate? This is called convergence - we can check at the final time for instance how far from the real solution it is.

In [15]:
```
yexact(t) = exp(-t) + (sin(t) - cos(t) + exp(-t)) / 2
tt = 0:0.01:2
error = []
for h = [0.4, 0.2, 0.1]
    t,y = euler(f, 1, h, round(Int, 2/h))
    push!(error, abs(y[end]-yexact(t[end])));
end
error
```

Out[15]:
```
3-element Vector{Any}:
 0.0275012128275961
 0.013522098820862971
 0.006647820551018313
```

It seems the error is roughly halving at each step as the timestep halves. This is called linear convergence.

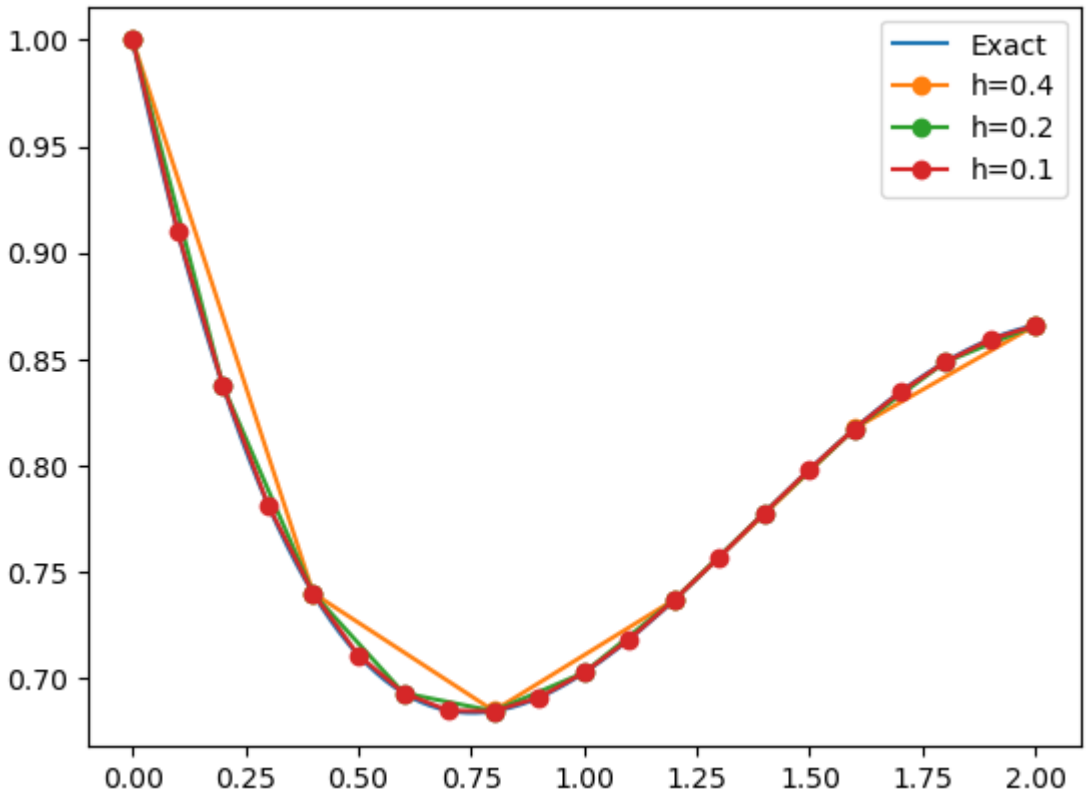
There are lots of methods to approximate that integral such as linear multistep methods, backwards differentiation formulae, etc. One famous type are the Runge-Kutta methods, none more so than RK4 which is sometimes known as the RK method. You can find its description on Wikipedia and we won't go into its derivation for this class.

In [16]:
```
function rk4(f, y0, h, N, t0=0)
    t = t0 .+ h*(0:N)
    y = zeros(N+1, length(y0))

    y[1,:] .= y0
    for n = 1:N
        k1 = h * f(t[n], y[n,:])
        k2 = h * f(t[n] + h/2, y[n,:] + k1/2)
        k3 = h * f(t[n] + h/2, y[n,:] + k2/2)
        k4 = h * f(t[n] + h, y[n,:] + k3)
        y[n+1,:] = y[n,:] + (k1 + 2k2 + 2k3 + k4) / 6
    end

    return t,y
end
```

Out[16]:
```
rk4 (generic function with 2 methods)
```

In [23]:
```
yexact(t) = exp(-t) + (sin(t) - cos(t) + exp(-t)) / 2
tt = 0:0.01:2
plot(tt, yexact.(tt))
error = []
for h = [0.4, 0.2, 0.1]
    t,y = rk4(f, 1, h, round(Int, 2/h))
    plot(t, y, "-o")
    push!(error, abs(y[end]-yexact(t[end])));
end
legend(("Exact", "h=0.4", "h=0.2", "h=0.1"));
display(error)
```

```
3-element Vector{Any}:
 2.01614610818579e-5
 1.34056601042154e-7
 1.9447003896111426e-8
```



It seems to be way more accurate. It's called RK4 as it is in fact 4th order accurate. This means as you halve the timestep your error should decrease by $\frac{1}{2^4}$. In your HW you will code up the 5th order accurate version.

We are brushing a lot of details under the rug here. In practice there are more nuances to this than simply accuracy, there is also something called stability that essentially asks even if a method is technically accurate can we still expect it to fail due to numerical error. This is somewhat covered in 128A but more so in 228A/B if you are interested.

A final note, ODEs are something that we as a species have gotten quite good at solving. The real challenge nowadays is on solving PDEs, as they difficult and also what makes the world go round.

Solving higher derivatives

A trick that actually works extremely well for solving ODEs with more than one derivative is to split the system by introducing a new variable for the derivative such that it becomes first order. For example

$$f''(t) = g(t)$$

becomes

$$h'(t) = g(t)$$
$$f'(t) = h(t)$$

which you can now solve as a system using the same techniques as above.

Structs

Structs are Julia's way of making new types. You have already encountered the built in types such as Integers, Floats, etc.... But what if you wanted to make your own?

In [31]:
```
struct Point
    x
    y
    norm
    Point(x, y, norm=sqrt(x*x+y*y)) = new(x,y,norm)
end
```

This creates a struct called Point that has three fields, x,y,norm. The last line is the constructor, it says that you create a point with arguments x,y which are to be inputted, but the norm need not be as it has a default value indicated by the = sign.

In [32]:
```
pt1 = Point( 0.1, 0.2 )
```

Out[32]:
```
Point(0.1, 0.2, 0.223606797749979)
```

You cannot change values of a struct once they are set. They are so called immutable values. This is for speed of the code.

In [33]:
```
pt1.x = 0.2
```

```
setfield! immutable struct of type Point cannot be changed

Stacktrace:
 [1] setproperty!(x::Point, f::Symbol, v::Float64)
   @ Base ./Base.jl:34
 [2] top-level scope
   @ In[33]:1
 [3] eval
   @ ./boot.jl:360 [inlined]
 [4] include_string(mapexpr::typeof(REPL.softscope), mod::Module, cod
e::String, filename::String)
   @ Base ./loading.jl:1116
```

To get the values though you just use the `.`

In [34]:
```
pt1.x
```

Out[34]:
```
0.1
```

Structs can act to simplify codes a lot to make them more readable. They exist in other languages too under different names such as classes. These form the basis of what is known as object-orientated programming.

In [ ]:
```

```