# 1 2D and 3D plots in Julia with Plots

This covers plotting the typical 2D and 3D plots in Julia with the `Plots` package.

```
using Plots
using LinearAlgebra, ForwardDiff
import PyPlot
import Contour: contours, levels, level, lines, coordinates
```

We will make use of some helper functions that will simplify plotting. These will be described in more detail in the following:

```
xs_ys(vs) = Tuple(eltype(vs[1])[vs[i][j] for i in 1:length(vs)] for j in eachindex(firs
xs_ys(v,vs...) = xs_ys([v, vs...])
xs_ys(r::Function, a, b, n=100) = xs_ys(r.(range(a, stop=b, length=n)))

function arrow!(p, v; kwargs...)
  if length(p) == 2
     quiver!(xs_ys([p])..., quiver=Tuple(xs_ys([v])); kwargs...)
  elseif length(p) == 3
    # 3d quiver needs support
    # https://github.com/JuliaPlots/Plots.jl/issues/319#issue-159652535
    # headless arrow instead
    plot!(xs_ys(p, p+v)...; kwargs...)
end
end
```

We will also use the `ForwardDiff` for derivatives and use the "prime" notation:

```
using ForwardDiff
function D(f, n::Int=1)
    n < 0 && throw(ArgumentError("n is a non-negative integer"))
    n == 0 && return f
    n == 1 && return t -> ForwardDiff.derivative(f, float(t))
    D(D(f), n-1)
end
Base.adjoint(r::Function) = D(r)
```

We will need to manipulate contours directly, so pull in the `Contours` package, using `import` to avoid name collisions and explicitly listing the methods we will use:

```
import Contour: contours, levels, lines, coordinates
```

Finally, we need some features for vectors:

```
using LinearAlgebra
```

## 1.1 Parametrically described curves in space

Let $r(t)$ be a vector-valued function with values in $R^d$, $d$ being 2 or 3. A familiar example is the equation for a line that travels in the direction of $\vec{v}$ and goes through the point $P$: $r(t) = P + t \cdot \vec{v}$. A *parametric plot* over $[a, b]$ is the collection of all points $r(t)$ for $a \leq t \leq b$.

In `Plots`, parameterized curves can be plotted through two interfaces, here illustrated for $d = 2$: `plot(f1, f2, a, b)` or `plot(xs, ys)`. The former is convenient for some cases, but typically we will have a function `r(t)` which is vector-valued, as opposed to a vector of functions. As such, we only discuss the latter.

An example helps illustrate. Suppose $r(t) = \langle \sin(t), 2\cos(t) \rangle$ and the goal is to plot the full ellipse by plotting over $0 \leq t \leq 2\pi$. As with plotting of curves, the goal would be to take many points between `a` and `b` and from there generate the $x$ values and $y$ values.

Let's see this with 5 points, the first and last being identical due to the curve:

```
r(t) = [sin(t), 2cos(t)]
ts = range(0, stop=2pi, length=5)
```

Then we can create the 5 points easily through broadcasting:

```
vs = r.(ts)
```

This returns a vector of points (stored as vectors). The plotting function wants two collections: the set of $x$ values for the points and the set of $y$ values. The data needs to be generated differently or reshaped. The function `xs_ys` above takes data in this style and returns the desired format, returning a tuple with the $x$ values and $y$ values pulled out:

```
xs_ys(vs)
```

To plot this, we "splat" the tuple so that `plot` gets the arguments separately:

```
plot(xs_ys(vs)...)
```

This basic plot is lacking, of course, as there are not enough points. Using more initially is a remedy. Rather than generate the `ts` separately, `xs_ys` has a simple frontend `xs_ys(r, a, b)` which does this work itself:

```
plot(xs_ys(r, 0, 2pi, 100)...)
```

### 1.1.1 Plotting a space curve in 3 dimensions

A parametrically described curve in 3D is similarly created. For example, a helix is described mathematically by $r(t) = \langle sin(t), cos(t), t \rangle$. Here we graph two turns:

```
r(t) = [sin(t), cos(t), t]
plot(xs_ys(r, 0, 4pi)...)
```