1 Quick introduction to calculus with Julia

Julia can be downloaded and used like other programming languages.

launch binder

Julia can be used through the internet for free using the mybinder.org service. To do so, click on the `CalcululsWithJulia.ipynb` file after launching Binder by clicking on the badge.

Here are some `Julia` usages to create calculus objects.

The `Julia` packages loaded below are all loaded when the `CalculusWithJulia` package is loaded.

A `Julia` package is loaded with the `using` command:

```
using LinearAlgebra
```

The `LinearAlgebra` package comes with a `Julia` installation. Other packages can be added. Something like:

```
using Pkg
Pkg.add("SomePackageName")
```

These notes have an accompanying package, `CalculusWithJulia`, that when installed, as above, also installs most of the necessary packages to perform the examples.

Packages need only be installed once, but they must be loaded into *each* session for which they will be used.

```
using CalculusWithJulia
using Plots
```

Packages can also be loaded through `import PackageName`. Importing does not add the exported objects of a function into the namespace, so is used when there are possible name collisions.

1.1 Types

Objects in `Julia` are "typed." Common numeric types are `Float64`, `Int64` for floating point numbers and integers. Less used here are types like `Rational{Int64}`, specifying rational numbers with a numerator and denominator as `Int64`; or `Complex{Float64}`, specifying a comlex number with floating point components. Julia also has `BigFloat` and `BigInt` for arbitrary precision types. Typically, operations use "promotion" to ensure the combination of types is appropriate. Other useful types are `Function`, an abstract type describing functions; `Bool` for true and false values; `Sym` for symbolic values (through `SymPy`); and `Vector{Float64}` for vectors with floating point components.

For the most part the type will not be so important, but it is useful to know that for some function calls the type of the argument will decide what method ultimately gets called. (This allows symbolic types to interact with Julia functions in an idiomatic manner.)

1.2 Functions

1.2.1 Definition

Functions can be defined four basic ways:

- one statement functions follow traditional mathematics notation:

```
f(x) = exp(x) * 2x
```

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

- multi-statement functions are defined with the `function` keyword. The `end` statement ends the definition. The last evaluated command is returned. There is no need for explicit `return` statement, though it can be useful for control flow.

```
function g(x)
  a = sin(x)^2
  a + a^2 + a^3
end
```

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

- Anonymous functions, useful for example, as arguments to other functions or as return values, are defined using an arrow, `->`, as follows:

```
fn = x -> sin(2x)
fn(pi/2)
```

1 . 2 2 4 6 4 6 7 9 9 1 4 7 3 5 3 2 e - 1 6

In the following, the defined function, `Derivative`, returns an anonymously defined function that uses a `Julia` package, loaded with `CalculusWithJulia`, to take a derivative:

```
Derivatve(f::Function) = x -> ForwardDiff.derivative(f, x)    # ForwardDiff is loaded in
CalculusWithJulia
```

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

(The D function of CalculusWithJulia implements something similar.)

- Anonymous function may also be created using the function keyword.

For mathematical functions $f : R^n \to R^m$ when n or m is bigger than 1 we have:

- When $n = 1$ and $m > 1$ we use a "vector" for the return value

```
r(t) = [sin(t), cos(t), t]
```

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

(An alternative would be to create a vector of functions.)

- When $n > 1$ and $m = 1$ we use multiple arguments or pass the arguments in a container. This pattern is common, as it allows both calling styles.

```
f(x,y,z) = x*y + y*z + z*x
f(v) = f(v...)
```

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

Some functions need to pass in a container of values, for this the last definition is useful to expand the values. Splatting takes a container and treats the values like individual arguments.

Alternatively, indexing can be used directly, as in:

```
f(x) = x[1]*x[2] + x[2]*x[3] + x[3]*x[1]
```

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

- For vector fields $(n, m > 1)$ a combination is used:

```
F(x,y,z) = [-y, x, z]
F(v) = F(v...)
```

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

1.2.2 Calling a function

Functions are called using parentheses to group the arguments.

```
f(t) = sin(t)*sqrt(t)
sin(1), sqrt(1), f(1)
```

```
(0.8414709848078965, 1.0, 0.8414709848078965)
```

When a function has multiple arguments, yet the value passed in is a container holding the arguments, splatting is used to expand the arguments, as is done in the definition `F(v) = F(v...)`, above.

1.2.3 Multiple dispatch

`Julia` can have many methods for a single generic function. (E.g., it can have many different implementations of addiion when the `+` sign is encountered.) The *type*s of the arguments and the number of arguments are used for dispatch.

Here the number of arguments is used:

```
Area(w, h) = w * h    # area of rectangle
Area(w) = Area(w, w)  # area of square using area of rectangle defintion
```

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

Calling `Area(5)` will call `Area(5,5)` which will return `5*5`.

Similarly, the definition for a vector field:

```
F(x,y,z) = [-y, x, z]
F(v) = F(v...)
```

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

takes advantage of multiple dispatch to allow either a vector argument or individual arguments.

Type parameters can be used to restrict the type of arguments that are permitted. The `Derivative(f::Function)` definition illustrates how the `Derivative` function, defined above, is restricted to `Function` objects.

1.2.4 Keyword arguments

Optional arguments may be specified with keywords, when the function is defined to use them. Keywords are separated from positional arguments using a semicolon, `;`:

```
circle(x; r=1) = sqrt(r^2 - x^2)
circle(0.5), circle(0.5, r=10)
```

```
(0.8660254037844386, 9.987492177719089)
```

The main (but not sole) use of keyword arguments will be with plotting, where various plot attribute are passed as `key=value` pairs.

1.3 Symbolic objects

The add-on `SymPy` package allows for symbolic expressions to be used. Symbolic values are defined with `@vars`, as below.

```
using SymPy
@vars x y z  # no comma as done here, though @vars(x,y,z) is also available
x^2 + y^3 + z
```

$$x^2 + y^3 + z$$

Assumptions on the variables can be useful, particularly with simplification, as in

```
@vars x y z real=true
```

```
(x, y, z)
```

Symbolic expressions flow through `Julia` functions symbolically

```
sin(x)^2 + cos(x)^2
```

$$\sin^2(x) + \cos^2(x)$$

Numbers are symbolic once `SymPy` interacts with them:

```
x - x + 1  # 1 is now symbolic
```

$$1$$

The number `PI` is a symbolic `pi`. a

```
sin(PI), sin(pi)
```

```
(0, 1.2246467991473532e-16)
```

Use `Sym` to create symbolic numbers, `N` to find a `Julia` number from a symbolic number:

```
1 / Sym(2)
```

$$\frac{1}{2}$$

```
N(PI)
```

```
π@*( = 3.1415926535897...
```

Many generic `Julia` functions will work with symbolic objects through multiple dispatch (e.g., `sin`, `cos`, ...). Sympy functions that are not in `Julia` can be accessed through the `sympy` object using dot-call notation:

```
sympy.harmonic(10)
```

$$\frac{7381}{2520}$$

Some Sympy methods belong to the object and a called via the pattern `object.method(...)`. This too is the case using SymPy with `Julia`. For example:

```
A = [x 1; x 2]
A.det()   # determinant of symbolic matrix A
```

1.4 Containers

We use a few different containers:

- Tuples. These are objects grouped together using parentheses. They need not be of the same type

```
x1 = (1, "two", 3.0)
```

```
(1, "two", 3.0)
```

Tuples are useful for programming. For example, they are uesd to return multiple values from a function.

- Vectors. These are objects of the same type (typically) grouped together using square brackets, values separated by commas:

```
x2 = [1, 2, 3.0]  # 3.0 makes theses all floating point
```

```
3-element Array{Float64,1}:
 1.0
 2.0
 3.0
```

Unlike tuples, the expected arithmatic from Linear Algebra is implemented for vectors.

- Matrices. Like vectors, combine values of the same type, only they are 2-dimensional. Use spaces to separate values along a row; semicolons to separate rows:

```
x3 = [1 2 3; 4 5 6; 7 8 9]
```

```
3×@*(3 Array(*@{Int64,2}:
 1  2  3
 4  5  6
 7  8  9
```

- Row vectors. A vector is 1 dimensional, though it may be identified as a column of two dimensional matrix. A row vector is a two-dimensional matrix with a single row:

```
x4 = [1 2 3.0]
```

```
1×@*(3 Array(*@{Float64,2}:
 1.0  2.0  3.0
```

These have *indexing* using square brackets:

```
x1[1], x2[2], x3[3]
```

```
(1, 2.0, 7)
```

Matrices are usually indexed by row and column:

```
x3[1,2] # row one column two
```

```
2
```

For vectors and matrices - but not tuples, as they are immutable - indexing can be used to change a value in the container:

```
x2[1], x3[1,1] = 2, 2
```

```
(2, 2)
```

Vectors and matrices are arrays. As hinted above, arrays have mathematical operations, such as addition and subtraction, defined for them. Tuples do not.

Destructuring is an alternative to indexing to get at the entries in certain containers:

```
a,b,c = x2
```

```
3-element Array{Float64,1}:
 2.0
 2.0
 3.0
```

1.4.1 Structured collections

An arithmetic progression, $a, a+h, a+2h, ..., b$ can be produced *efficiently* using the range operator `a:h:b`:

```
5:10:55  # an object that describes 5, 15, 25, 35, 45, 55
```

```
5:10:55
```

If `h=1` it can be omitted:

```
1:10     # an object that describes 1,2,3,4,5,6,7,8,9,10
```

```
1:10
```

The `range` function can *efficiently* describe n evenly spaced points between `a` and `b`:

```
range(0, pi, length=5)  # range(a, stop=b, length=n) for version 1.0
```

```
0.0:0.7853981633974483:3.141592653589793
```

This is useful for creating regularly spaced values needed for certain plots.

1.5 Iteration

The `for` keyword is useful for iteration, Here is a traditional for loop, as `i` loops over each entry of the vector `[1,2,3]`:

```
for i in [1,2,3]
  print(i)
end
```

```
123
```

Technical aside: For assignment within a for loop at the global level, a `global` declaration may be needed to ensure proper scoping.

List comprehensions are similar, but are useful as they perform the iteration and collect the values:

```
[i^2 for i in [1,2,3]]
```

```
3-element Array{Int64,1}:
 1
 4
 9
```

Comprehesions can also be used to make matrices

```
[1/(i+j) for i in 1:3, j in 1:4]
```

```
3×@*(4 Array(*@{Float64,2}:
 0.5       0.333333  0.25      0.2
 0.333333  0.25      0.2       0.166667
 0.25      0.2       0.166667  0.142857
```

(The three rows are for `i=1`, then `i=2`, and finally for `i=3`.)

Comprehensions apply an *expression* to each entry in a container through iteration. Applying a function to each entry of a container can be facilitated by:

- Broadcasting. Using `.` before an operation instructs `Julia` to match up sizes (possibly extending to do so) and then apply the operation element by element:

```
xs = [1,2,3]
sin.(xs)   # sin(1), sin(2), sin(3)
```

```
3-element Array{Float64,1}:
 0.8414709848078965
 0.9092974268256817
 0.1411200080598672
```

This example pairs off the value in `bases` and `xs`:

```
bases = [5,5,10]
log.(bases, xs)  # log(5, 1), log(5,2), log(10, 3)
```

This example broadcasts the scalar value for the base with `xs`:

```
log.(5, xs)
```

```
3-element Array{Float64,1}:
 0.0
 0.43067655807339306
 0.6826061944859854
```

Row and column vectors can fill in:

```
ys = [4 5] # a row vector
f(x,y) = (x,y)
f.(xs, ys)    # broadcasting a column and row vector makes a matrix, then applies f.
```

```
3×@*(2 Array(*@{Tuple{Int64,Int64},2}:
 (1, 4)  (1, 5)
 (2, 4)  (2, 5)
 (3, 4)  (3, 5)
```

This should be contrasted to the case when both `xs` and `ys` are (column) vectors, as then they pair off:

```
f.(xs, [4,5])
```

- The `map` function is similar, it applies a function to each element:

```
map(sin, [1,2,3])
```

```
3-element Array{Float64,1}:
 0.8414709848078965
 0.9092974268256817
 0.1411200080598672
```

Many different computer languages implement `map`, broadcasting is less common. `Julia`'s use of the dot syntax to indicate broadcasting is reminiscent of MATLAB, but is quite different.

1.6 Plots

The following commands use the `Plots` package. The `Plots` package expects a choice of backend. We will use both `plotly` and `gr` (and occasionally `pyplot()`).

```
using Plots
pyplot()        # select pyplot. Use `gr()` for GR; `plotly()` for Plotly
```

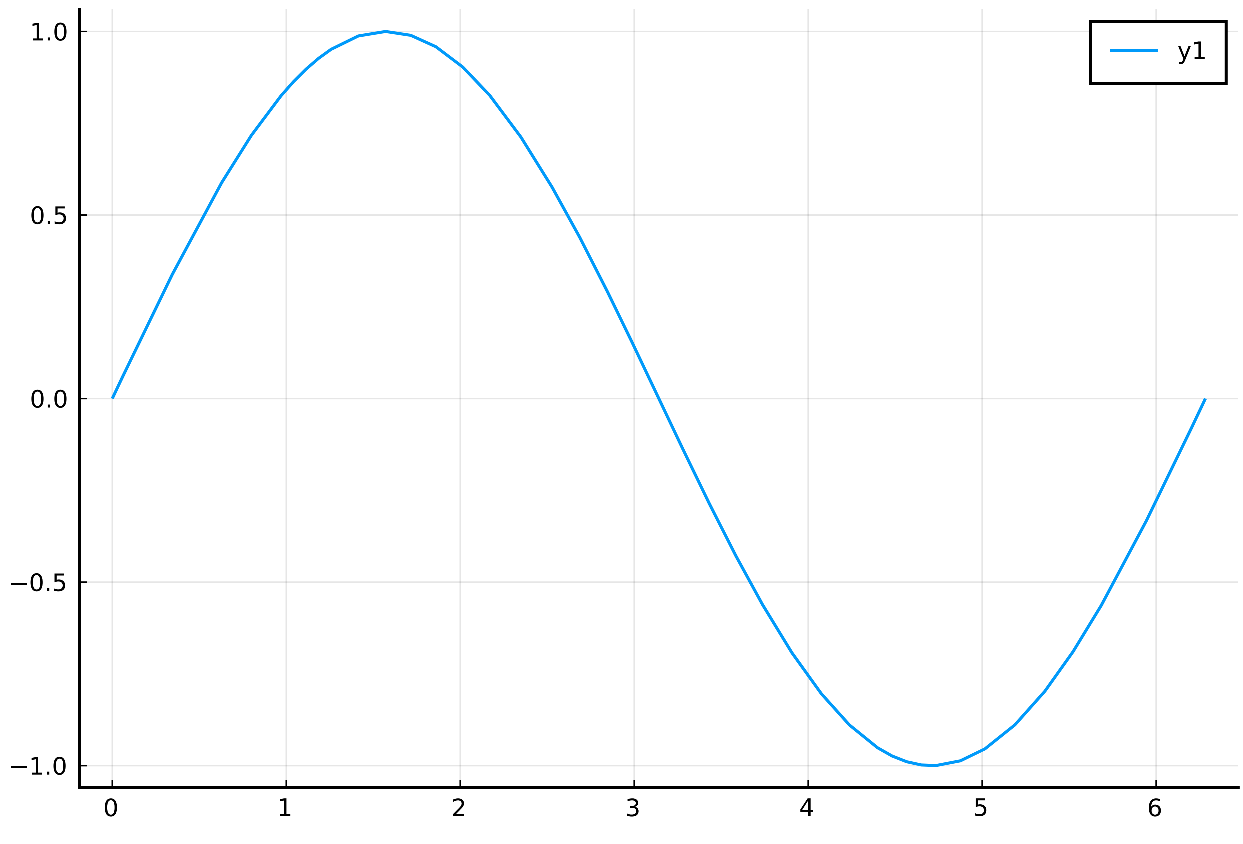
```
Plots.PyPlotBackend()
```

The `plotly` backend and `gr` backends are available by default. The `plotly` backend is has some interactivity, `gr` is for static plots. The `pyplot` package is used for certain surface plots, when `gr` can not be used.

Plotting a univariate function $f : R \to R$

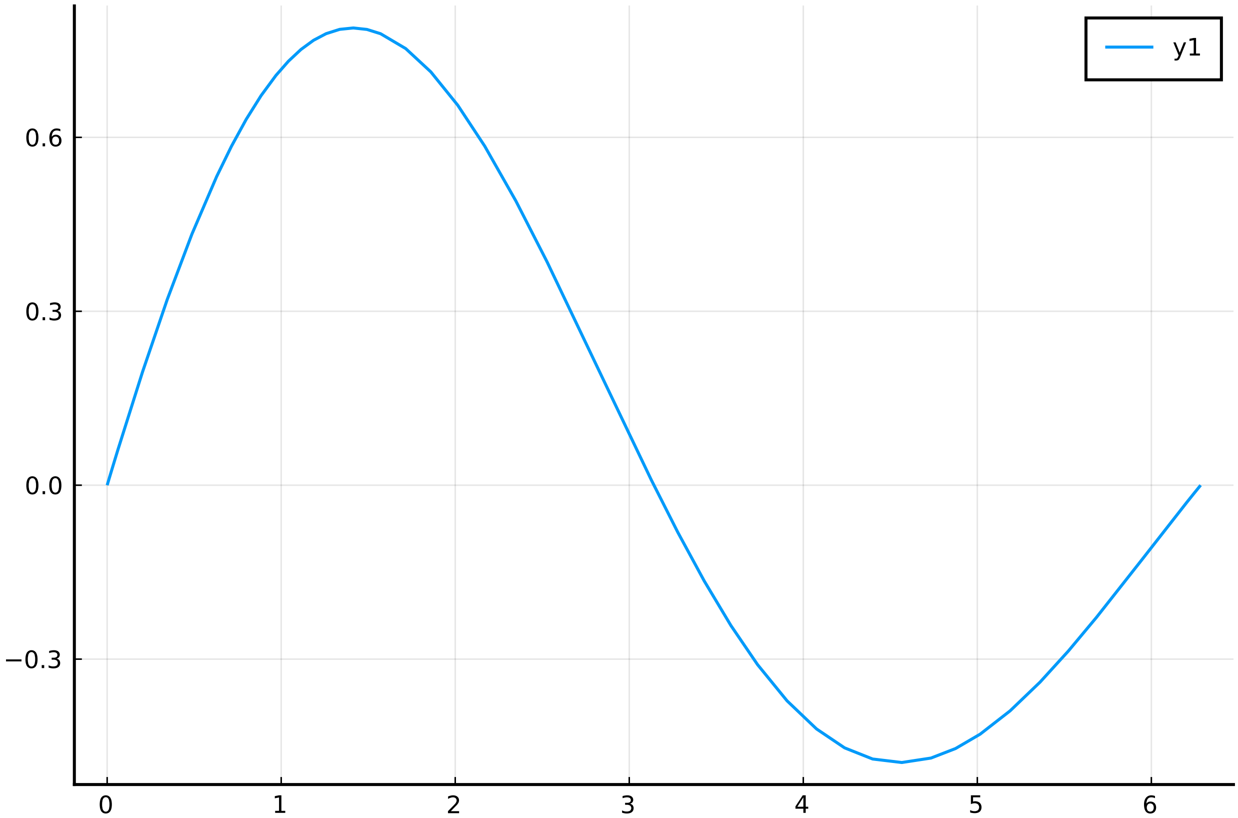
- using `plot(f, a, b)`

```
plot(sin, 0, 2pi)
```



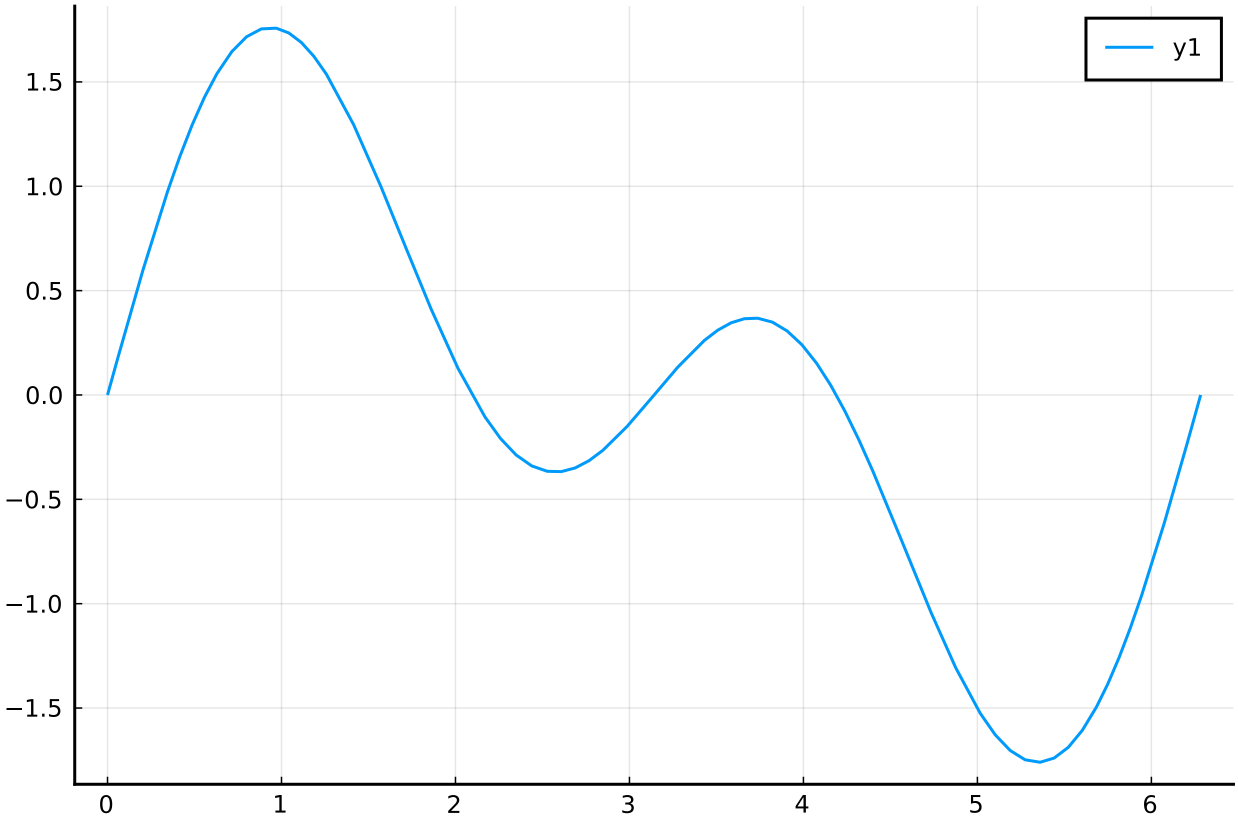
Or

```
f(x) = exp(-x/2pi)*sin(x)
plot(f, 0, 2pi)
```



Or with an anonymous function

```
plot(x -> sin(x) + sin(2x), 0, 2pi)
```



The time to first plot can be lengthy! This can be removed by creating a custom `Julia` image, but that is not introductory level stuff. As well, standalone plotting packages offer quicker first plots, but the simplicity of `Plots` is preferred. Subsequent plots are not so time consuming, as the initial time is spent compiling functions so their re-use is speedy.

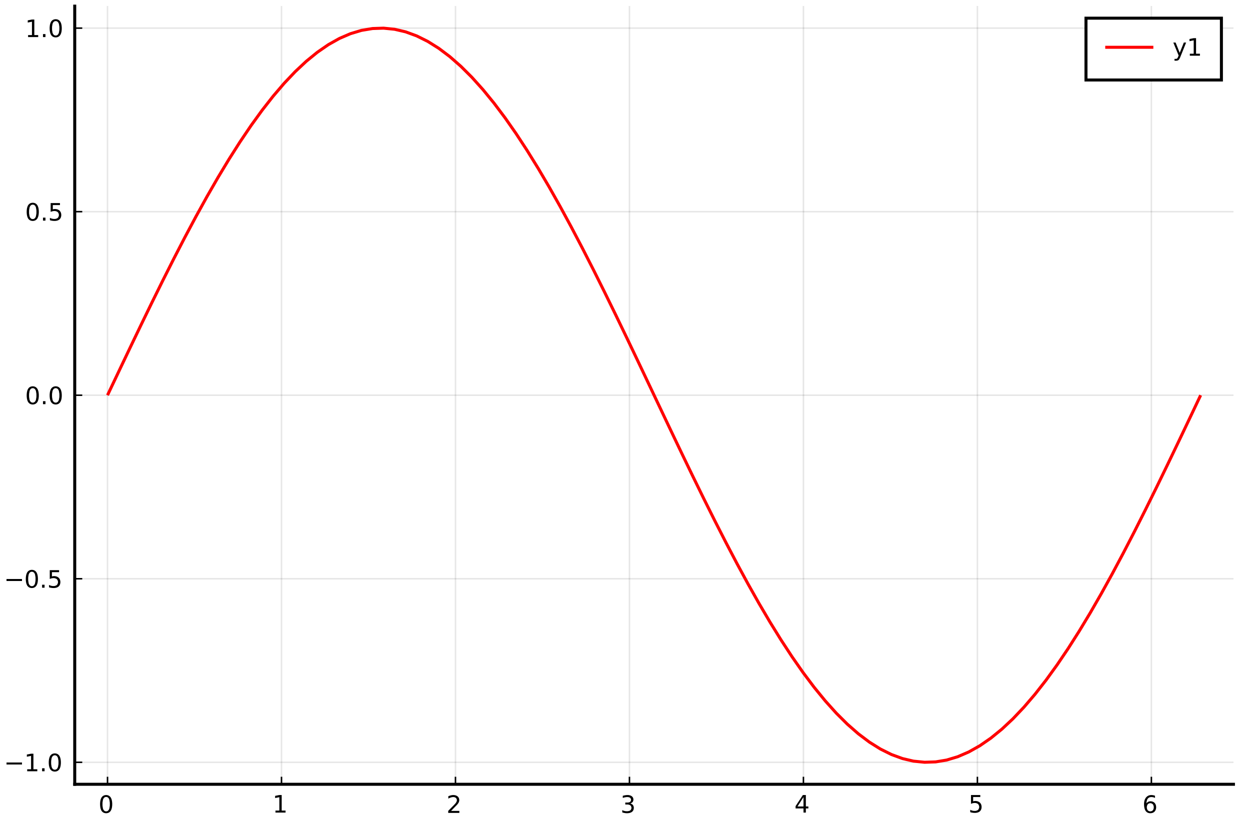
Arguments of interest include

Attribute	Value
`legend`	A boolean, specify `false` to inhibit drawing a legend
`aspect_ratio`	Use `:equal` to have x and y axis have same scale
`linewidth`	Ingters greater than 1 will thicken lines drawn
`color`	A color may be specified by a symbol (leading `:`). E.g., `:black`, `:red`, `:blue`

- using `plot(xs, ys)`

The lower level interface to `plot` involves directly creating x and y values to plot:

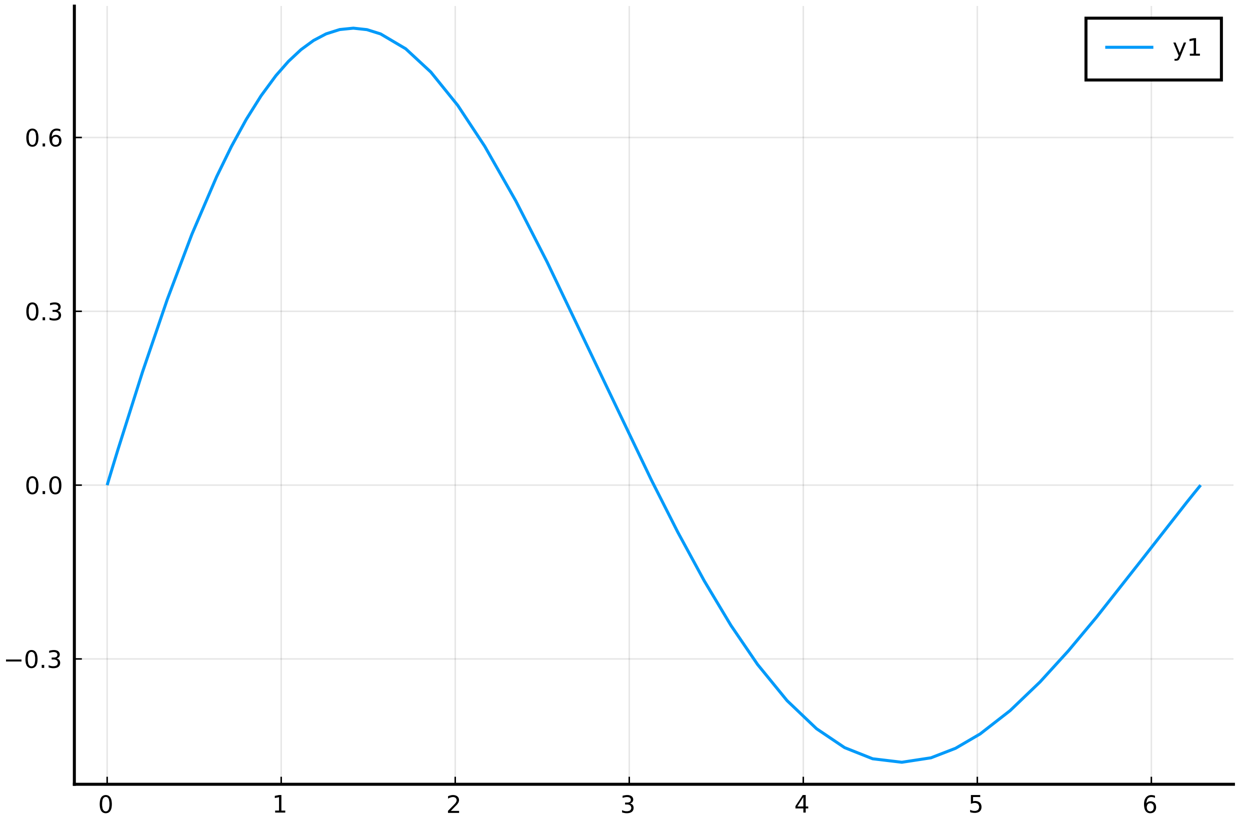
```
xs = range(0, 2pi, length=100)
ys = sin.(xs)
plot(xs, ys, color=:red)
```



- plotting a symbolic expression

A symbolic expression of single variable can be plotted as a function is:

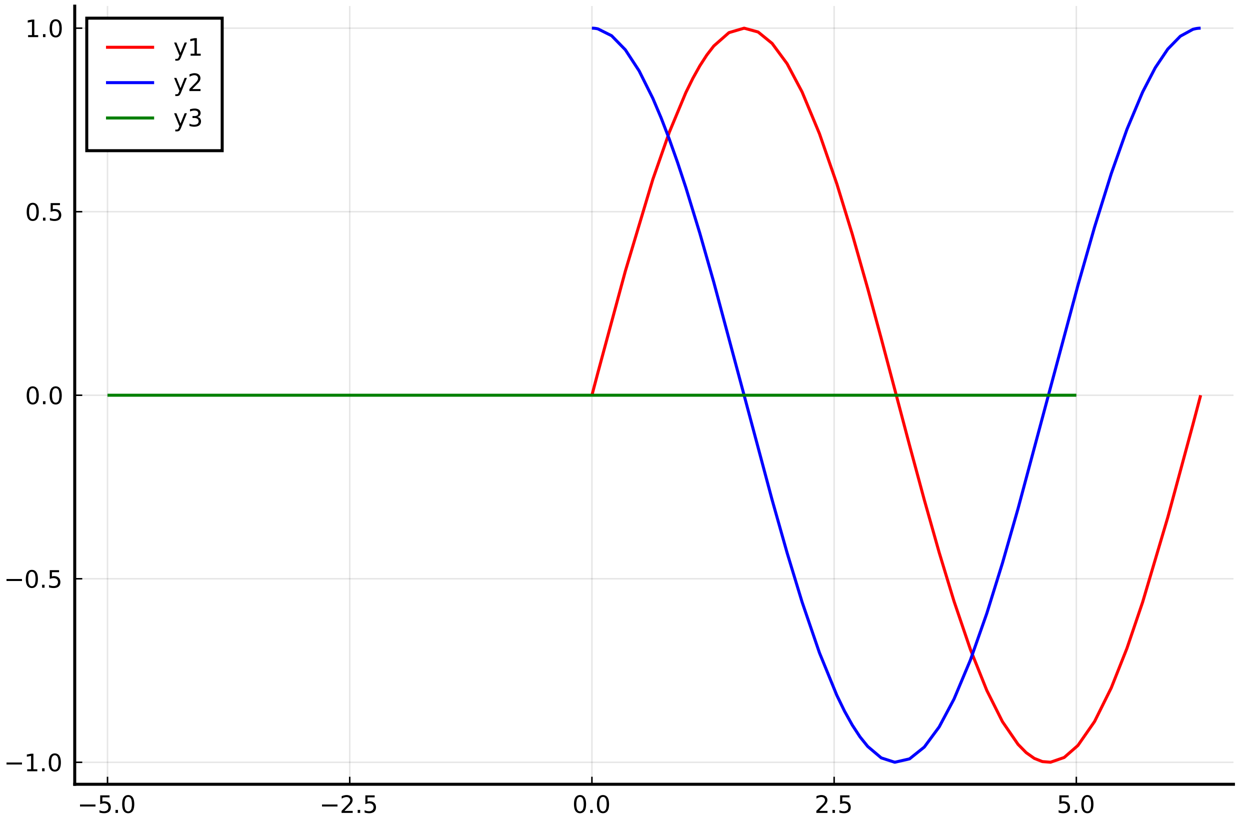
```
@vars x
plot(exp(-x/2pi)*sin(x), 0, 2pi)
```



- Multiple functions

The `!` Julia convention to modify an object is used by the `plot` command, so `plot!` will add to the existing plot:

```
plot(sin, 0, 2pi, color=:red)
plot!(cos, 0, 2pi, color=:blue)
plot!(zero, color=:green)  # no a, b then inherited from graph.
```



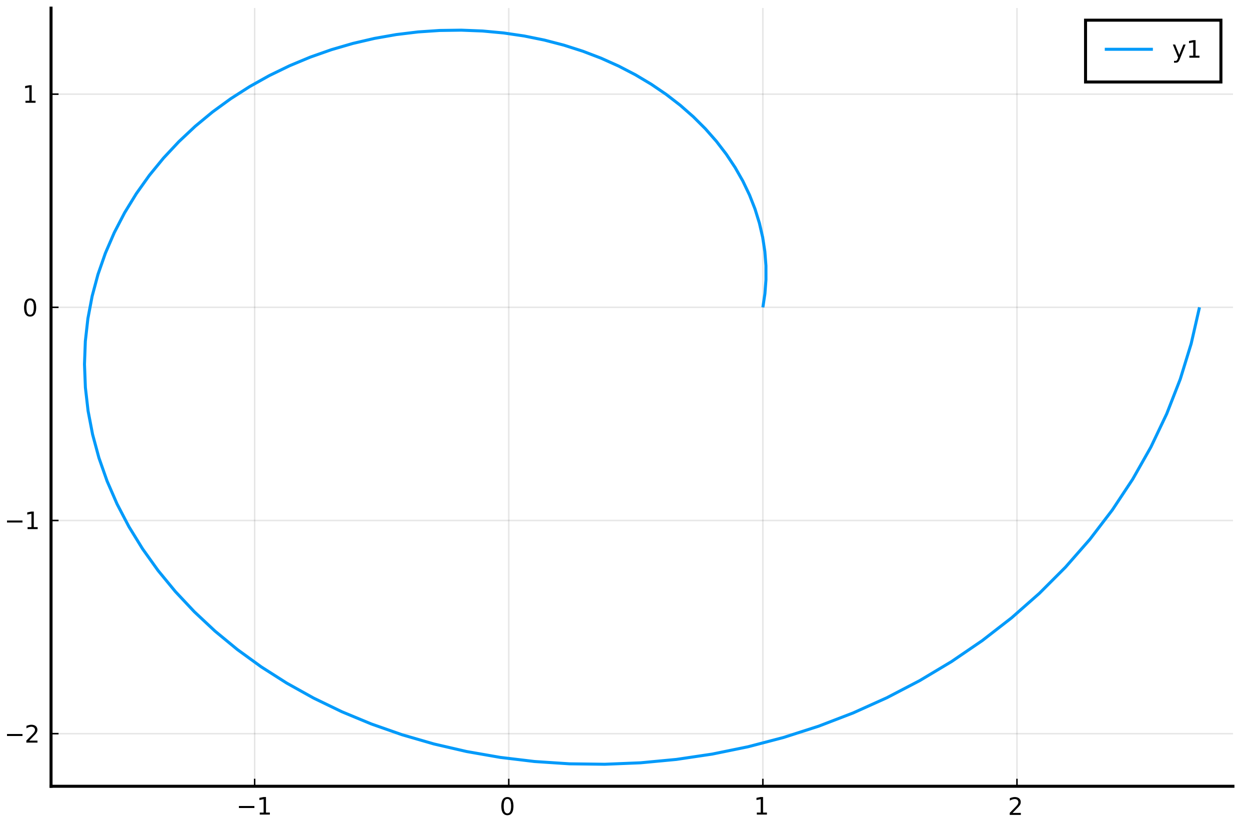
The `zero` function is just 0 (more generally useful when the type of a number is important, but used here to emphasize the x axis).

Plotting a parameterized (space) curve function $f : R \to R^n$, $n = 2$ or 3

- Using `plot(xs, ys)`

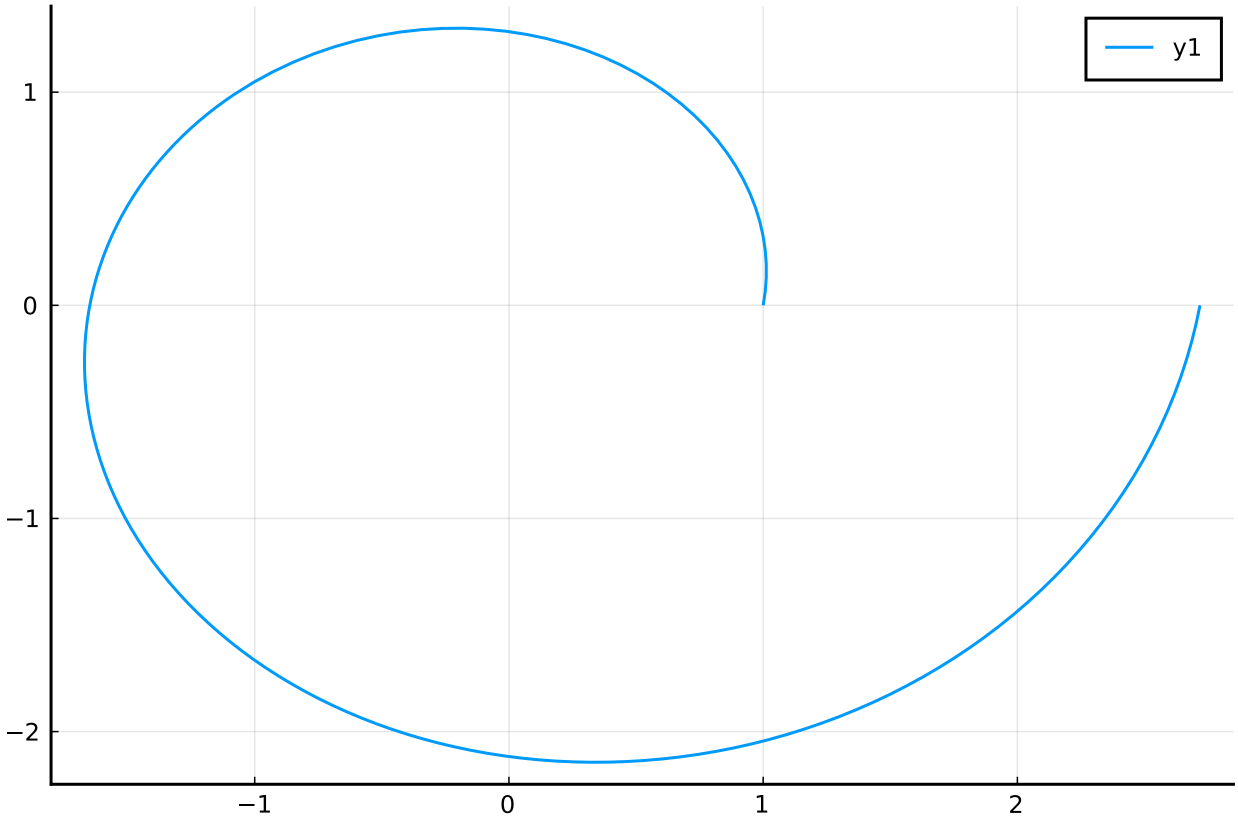
Let $f(t) = e^{t/2\pi}\langle\cos(t), \sin(t)\rangle$ be a parameterized function. Then the t values can be generated as follows:

```
ts = range(0, 2pi, length = 100)
xs = [exp(t/2pi) * cos(t) for t in ts]
ys = [exp(t/2pi) * sin(t) for t in ts]
plot(xs, ys)
```



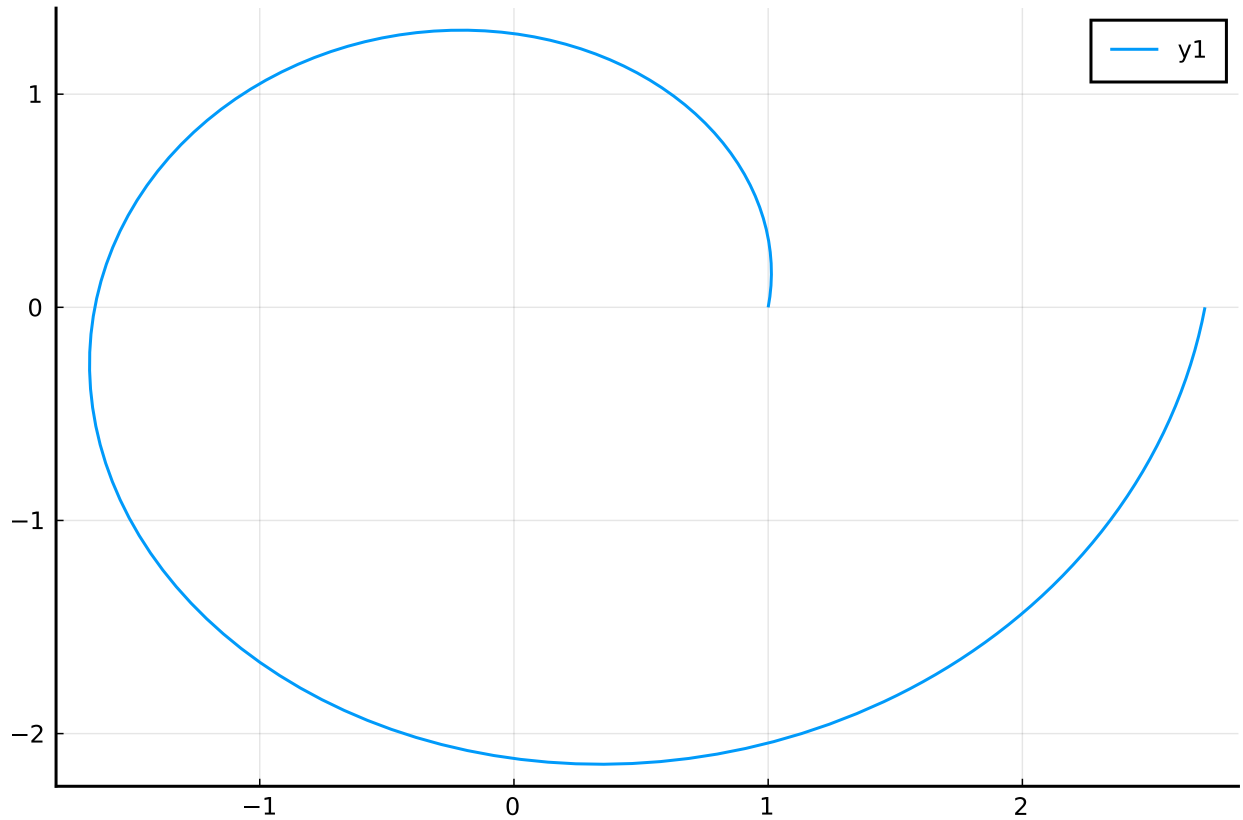
- using `plot(f1, f2, a, b)`. If the two functions describing the components are available, then

```
f1(t) = exp(t/2pi) * cos(t)
f2(t) = exp(t/2pi) * sin(t)
plot(f1, f2, 0, 2pi)
```



- Using `plot_parametric_curve`. If the curve is described as a function of `t` with a vector output, then the `CalculusWithJulia` package provides `plot_parametric_curve` to produce a plot:

```
r(t) = exp(t/2pi) * [cos(t), sin(t)]
plot_parametric_curve(r, 0, 2pi)
```



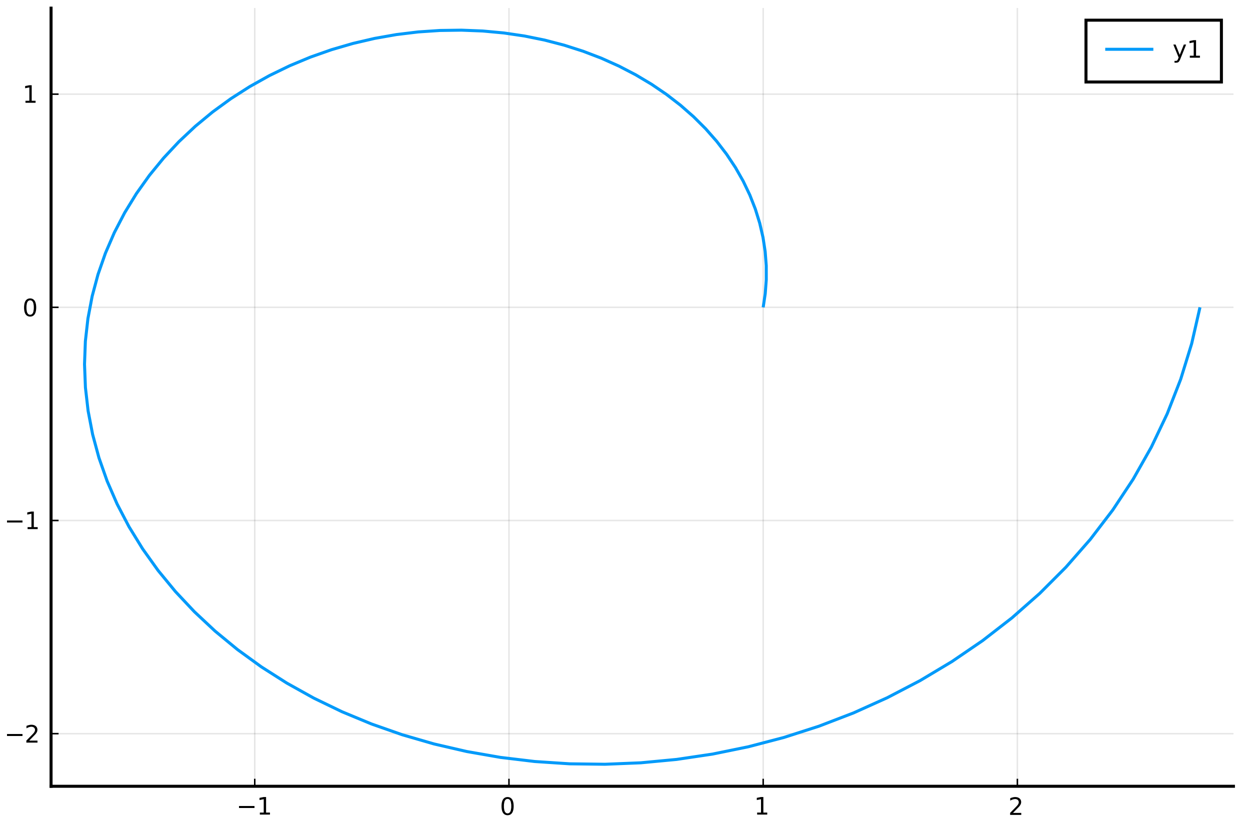
The low-level approach doesn't quite work as easily as desired:

```
ts = range(0, 2pi, length = 4)
vs = r.(ts)
```

```
4-element Array{Array{Float64,1},1}:
 [1.0, 0.0]
 [-0.6978062125430444, 1.2086358139617603]
 [-0.9738670205273388, -1.6867871593690715]
 [2.718281828459045, -6.657870280805568e-16]
```

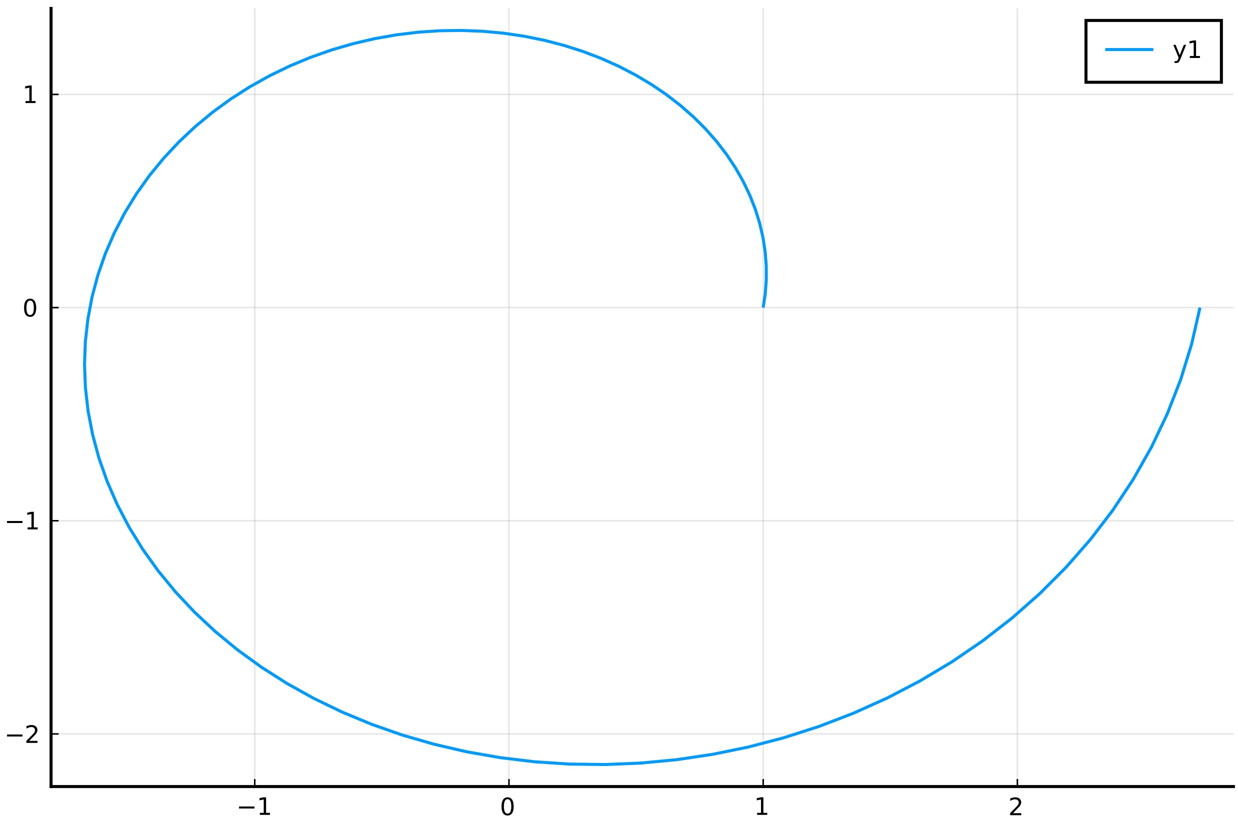
As seen, the values are a vector of vectors. To plot a reshaping needs to be done:

```
ts = range(0, 2pi, length = 100)
vs = r.(ts)
xs = [vs[i][1] for i in eachindex(vs)]
ys = [vs[i][2] for i in eachindex(vs)]
plot(xs, ys)
```



This approach is faciliated by the `unzip` function in `CalculusWithJulia` (and used internally by `plot_parametric_curve`):

```
plot(unzip(vs)...)
```



- Plotting an arrow

An arrow in 2D can be plotted with the `quiver` command. We show the `arrow(p, v)` (or `arrow!(p,v)` function) from the `CalculusWithJulia` package, which has an easier syntax (`arrow!(p, v)`, where `p` is a point indicating the placement of the tail, and `v` the vector to represent):

```
gr()
plot_parametric_curve(r, 0, 2pi)
t0 = pi/8
arrow!(r(t0), r'(t0))
```

The `GR` package makes nicer arrows that `Plotly`.

Plotting a scalar function $f : R^2 \to R$

The `surface` and `contour` functions are available to visualize a scalar function of 2 variables:

- A surface plot

```
plotly()    # The `plotly` backend allows for rotation by the mouse; otherwise the
`camera` argument is used
f(x, y) = 2 - x^2 + y^2
xs = ys = range(-2,2, length=25)
surface(xs, ys, f)
```

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

The function generates the z values, this can be done by the user and then passed to the `surface(xs, ys, zs)` format:

```
surface(xs, ys, f.(xs, ys'))
```

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

- A contour plot

The `contour` function is like the `surface` function.

```
contour(xs, ys, f)
```

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

```
contour(xs, ys, f.(xs, ys'))
```

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

- An implicit equation. The constraint $f(x, y) = c$ generates an implicit equation. While `contour` can be used for this type of plot - by adjusting the requested contours - the `ImplicitEquations` package can as well, and, perhaps. is easier. This package is loaded with `CalculusWithJulia`; loading it by itself will lead to naming conflicts with `SymPy`, so best not to do so. `ImplicitEquations` plots predicates formed by `Eq`, `Le`, `Lt`, `Ge`, and `Gt` (or some unicode counterparts). For example to plot when $f(x, y) = \sin(xy) - \cos(xy) \leq 0$ we have:

```
f(x,y) = sin(x*y) - cos(x*y)
plot(Le(f, 0))     # or plot(f ≦ 0) using \leqq[tab] to create that symbol
```

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

Plotting a parameterized surface $f : R^2 \to R^3$

The `plotly` (and `pyplot`) backends allow plotting of parameterized surfaces.

The low-level `surface(xs,ys,zs)` is used, and can be specified directly as follows:

```
X(theta, phi) = sin(phi)*cos(theta)
Y(theta, phi) = sin(phi)*sin(theta)
Z(theta, phi) = cos(phi)
thetas = range(0, pi/4, length=20)
phis = range(0, pi, length=20)
surface(X.(thetas, phis'), Y.(thetas, phis'), Z.(thetas, phis'))
```

The function `parametric_grid` from the `CalculusWithJulia` package will prepare the `xs`, `ys`, and `zs` to pass to `surface` when a vector-valued function is involved:

```
Phi(theta, phi) = [sin(phi) * cos(theta), sin(phi) * sin(theta), cos(phi)]
thetas, phis = range(0, pi/4, length=15), range(0, pi, length=20)
xs, ys, zs = parametric_grid(thetas, phis, Phi)
surface(xs, ys, zs)
wireframe!(xs, ys, zs)
```

Plotting a vector field $F : R^2 \to R^2$. The `CalculusWithJulia` package provides `vectorfieldplot`, used as:

```
gr()  # better arrows than plotly()
F(x,y) = [-y, x]
vectorfieldplot(F, xlim=(-2, 2), ylim=(-2,2), nx=10, ny=10)
```

```
Error: UndefVarError: .. not defined
```

There is also `vectorfieldplot3d`.

1.7 Limits

Limits can be investigated numerically by forming tables, eg.:

```
xs = [1, 1/10, 1/100, 1/1000]
f(x) = sin(x)/x
[xs f.(xs)]
```

```
4×@*(2 Array(*@{Float64,2}:
 1.0    0.841471
 0.1    0.998334
 0.01   0.999983
 0.001  1.0
```

Symbolically, `SymPy` provides a `limit` function:

```
@vars x
limit(sin(x)/x, x => 0)
```

$$1$$

Or

```
@vars h x
limit((sin(x+h) - sin(x))/h, h => 0)
```

$$\cos\left(x\right)$$

1.8 Derivatives

There are numeric and symbolic approaches to derivatives. For the numeric approach we use the `ForwardDiff` package, which performs automatic differentiation.

Derivatives of univariate functions

Numerically, the `ForwardDiff.derivative(f, x)` function call will find the derivative of the function `f` at the point `x`:

```
ForwardDiff.derivative(sin, pi/3) - cos(pi/3)
```

0.0

The `CalculusWithJulia` package overides the `'` (`adjoint`) syntax for functions to provide a derivative which takes a function and returns a function, so its usage is familiar

```
f(x) = sin(x)
f'(pi/3) - cos(pi/3)  # or just sin'(pi/3) - cos(pi/3)
```

0.0

Higher order derivatives are possible as well,

```
f(x) = sin(x)
f''''(pi/3) - f(pi/3)
```

0.0

Symbolically, the `diff` function of `SymPy` finds derivatives.

```
@vars x
f(x) = exp(-x)*sin(x)
ex = f(x)  # symbolic expression
diff(ex, x)   # or just diff(f(x), x)
```

$$-e^{-x}\sin{\left(x \right)} + e^{-x}\cos{\left(x \right)}$$

Higher order derivatives can be specified as well

```
diff(ex, x, x)
```

$$-2e^{-x}\cos{\left(x \right)}$$

Or with a number:

```
diff(ex, x, 5)
```

$$4\left(\sin{\left(x \right)} - \cos{\left(x \right)}\right)e^{-x}$$

The variable is important, as this allows parameters to be symbolic

```
@vars mu sigma x
diff(exp(-((x-mu)/sigma)^2/2), x)
```

$$-\frac{(-2\mu + 2x)\, e^{-\frac{(-\mu+x)^2}{2\sigma^2}}}{2\sigma^2}$$

partial derivatives

There is no direct partial derivative function provided by `ForwardDiff`, rather we use the result of the `ForwardDiff.gradient` function, which finds the partial derivatives for each variable. To use this, the function must be defined in terms of a point or vector.

```
f(x,y,z) = x*y + y*z + z*x
f(v) = f(v...)               # this is needed for ForwardDiff.gradient
ForwardDiff.gradient(f, [1,2,3])
```

```
3-element Array{Int64,1}:
 5
 4
 3
```

We can see directly that $\partial f/\partial x = \langle y + z \rangle$. At the point $(1, 2, 3)$, this is 5, as returned above.

Symbolically, `diff` is used for partial derivatives:

```
@vars x y z
ex = x*y + y*z + z*x
diff(ex, x)     # ∂f/∂x
```

$$y + z$$

Gradient

As seen, the `ForwardDiff.gradient` function finds the gradient at a point. In `CalculusWithJulia`, the gradient is extended to return a function when called with no additional arguments:

```
f(x,y,z) =  x*y + y*z + z*x
f(v) = f(v...)
gradient(f)(1,2,3) - gradient(f, [1,2,3])
```

```
3-element Array{Int64,1}:
 0
 0
 0
```

The ∇ symbol, formed by entering `\nabla[tab]`, is mathematical syntax for the gradient, and is defined in `CalculusWithJulia`.

```
∇(f)(1,2,3)    # same as gradient(f, [1,2,3])
```

```
3-element Array{Int64,1}:
 5
 4
 3
```

In `SymPy`, there is no gradient function, though finding the gradient is easy through broadcasting:

```
@vars x y z
ex = x*y + y*z + z*x
diff.(ex, [x,y,z])  # [diff(ex, x), diff(ex, y), diff(ex, z)]
```

$$\begin{bmatrix} y+z \\ x+z \\ x+y \end{bmatrix}$$

The `CalculusWithJulia` package provides a method for `gradient`:

```
gradient(ex, [x,y,z])
```

$$\begin{bmatrix} y+z \\ x+z \\ x+y \end{bmatrix}$$

The ∇ symbol is an alias. It can guess the order of the free symbols, but generally specifying them is needed. This is done with a tuple:

```
∇((ex, [x,y,z]))  # for this, ∇(ex) also works
```

$$\begin{bmatrix} y+z \\ x+z \\ x+y \end{bmatrix}$$

Jacobian

The Jacobian of a function $f : R^n \to R^m$ is a $m \times n$ matrix of partial derivatives. Numerically, `ForwardDiff.jacobian` can find the Jacobian of a function at a point:

```
F(u,v) = [u*cos(v), u*sin(v), u]
F(v) = F(v...)    # needed for ForwardDiff.jacobian
pt = [1, pi/4]
ForwardDiff.jacobian(F , pt)
```

```
3×@*(2 Array(*@{Float64,2}:
 0.707107  -0.707107
 0.707107   0.707107
 1.0        0.0
```

Symbolically, the `jacobian` function is a method of a *matrix*, so the calling pattern is different. (Of the form `object.method(arguments...)`.)

```
@vars u v
ex = F(u,v)
ex.jacobian([u,v])
```

$$\begin{bmatrix} \cos\left(v\right) & -u\sin\left(v\right) \\ \sin\left(v\right) & u\cos\left(v\right) \\ 1 & 0 \end{bmatrix}$$

As the Jacobian can be identified as the matrix with rows given by the transpose of the gradient of the component, it can be computed directly, but it is more difficult:

```
@vars u v real=true
vcat([diff.(ex, [u,v])' for ex in F(u,v)]...)
```

```
3×@*(2 Array(*@{Any,2}:
 cos(v)  -u·@*(sin(v)sin(v) u(*@·@*(cos(v)1 0
```

Divergence

Numerically, the divergence can be computed from the Jacobian by adding the diagonal elements. This is a numerically inefficient, as the other partial derivates must be found and discarded, but this is generally not an issue for these notes. The following uses `tr` (the trace from the `LinearAlgebra` package) to find the sum of a diagonal.

```
F(x,y,z) = [-y, x, z]
F(v) = F(v...)
pt = [1,2,3]
tr(ForwardDiff.jacobian(F , pt))
```

1

The `CalculusWithJulia` package provides `divergence` to compute the divergence and provides the $\nabla \cdot$ notation (`\nabla[tab]\cdot[tab]`):

```
divergence(F, [1,2,3])
(∇⋅F)(1,2,3)     # not ∇⋅F(1,2,3) as that evaluates F(1,2,3) before the divergence
```

1 . 0

Symbolically, the divergence can be found directly:

```
@vars x y z
ex = F(x,y,z)
sum(diff.(ex, [x,y,z]))     # sum of [diff(ex[1], x), diff(ex[2],y), diff(ex[3], z)]
```

$$1$$

The `divergence` function can be used for symbolic expressions:

```
divergence(ex, [x,y,z])
∇⋅(F(x,y,z), [x,y,z])    # For this, ∇ ⋅ F(x,y,z) also works
```

$$1$$

Curl

The curl can be computed from the off-diagonal elements of the Jacobian. The calculation follows the formula. The `CalculusWithJulia` package provides `curl` to compute this:

```
F(x,y,z) = [-y, x, 1]
F(v) = F(v...)
curl(F, [1,2,3])
```

```
3-element Array{Float64,1}:
  0.0
 -0.0
  2.0
```

As well, if no point is specified, a function is returned for which a point may be specified using 3 coordinates or a vector

```
curl(F)(1,2,3), curl(F)([1,2,3])
```

```
([0.0, -0.0, 2.0], [0.0, -0.0, 2.0])
```

Finally, the $\nabla \times$ (`\nabla[tab]\times[tab]` notation is available)

```
(∇×F)(1,2,3)
```

```
3-element Array{Float64,1}:
  0.0
 -0.0
  2.0
```

For symbolic expressions, we have

```
\ensuremath{\nabla}\ensuremath{\times}F(1,2,3)
```

(Do note the subtle difference in the use of parentheses between the numeric and the symbolic. For the symbolic, `F(x,y,z)` is evaluated *before* being passed to ∇×, where as for the numeric approach `∇×F` is evaluated *before* passing a point to compute the value there.)

1.9 Integrals

Numeric integration is provided by the `QuadGK` package, for univariate integrals, and the `HCubature` package for higher dimensional integrals.

Integrals of univariate functions

A definite integral may be computed numerically using `quadgk`

```
using QuadGK
quadgk(sin, 0, pi)
```

```
(2.0, 1.7905676941154525e-12)
```

The answer and an estimate for the worst case error is returned.

If singularities are avoided, improper integrals are computed as well:

```
quadgk(x->1/x^(1/2), 0, 1)
```

```
(1.9999999845983916, 2.376251192458876 5e-8)
```

SymPy provides the `integrate` function to compute both definite and indefinite integrals.

```
@vars a x real=true
integrate(exp(a*x)*sin(x), x)
```

$$\frac{ae^{ax}\sin\left(x\right)}{a^{2}+1}-\frac{e^{ax}\cos\left(x\right)}{a^{2}+1}$$

Like `diff` the variable to integrate is specified.

Definite integrals use a tuple, `(variable, a, b)`, to specify the variable and range to integrate over:

```
integrate(sin(a + x), (x, 0, PI))   #  ∫_0^PI sin(a+x) dx
```

$$2\cos(a)$$

2D and 3D iterated integrals

Two and three dimensional integrals over box-like regions are computed numerically with the `hcubature` function from the `HCubature` package. If the box is $[x_1, y_1] \times [x_2, y_2] \times \cdots \times [x_n, y_n]$ then the limits are specified through tuples of the form $(x_1, x_2, \ldots, x_n)$ and $(y_1, y_2, \ldots, y_n)$.

```
f(x,y) = x*y^2
f(v) = f(v...)
hcubature(f, (0,0), (1, 2))  # computes ∫_01∫_02 f(x,y) dy dx
```

```
(1.3333333333333333, 4.440892098500626e-16)
```

The calling pattern for more dimensions is identical.

```
f(x,y,z) = x*y^2*z^3
f(v) = f(v...)
hcubature(f, (0,0,0), (1, 2,3)) # computes ∫_01∫_02∫_03 f(x,y,z) dz dy dx
```

```
(27.0, 0.0)
```

The box-like region requirement means a change of variables may be necessary. For example, to integrate over the region $x^2 + y^2 \leq 1; x \geq 0$, polar coordinates can be used with (r, θ) in $[0, 1] \times [-\pi/2, \pi/2]$. When changing variables, the Jacobian enters into the formula, through

$$\iint_{G(S)} f(\vec{x}) dV = \iint_S (f \circ G)(\vec{u}) |\det(J_G)(\vec{u})| dU.$$

Here we implement this:

```
f(x,y) = x*y^2
f(v) = f(v...)
Phi(r, theta) = r * [cos(theta), sin(theta)]
Phi(rtheta) = Phi(rtheta...)
integrand(rtheta) = f(Phi(rtheta)) * det(ForwardDiff.jacobian(Phi, rtheta))
hcubature(integrand, (0.0,-pi/2), (1.0, pi/2))
```

```
(0.13333333333904923, 1.9853799966359355e-9)
```

In `CalculusWithJulia` a `fubini` function is provided to compute numeric integrals over regions which can be described by curves represented by functions. E.g., for this problem:

```
fubini(f, (x -> -sqrt(1-x^2), x -> sqrt(1-x^2)), (0, 1))
```

0 . 1 3 3 3 3 3 3 3 3 3 3 2 7 7 5 7 6 2

This function is for convenience, but is not performant.

Symbolically, the `integrate` function allows additional terms to be specified. For example, the above could be done through:

```
@vars x y real=true
integrate(x * y^2, (y, -sqrt(1-x^2), sqrt(1-x^2)), (x, 0, 1))
```

$$\frac{2}{15}$$

Line integrals

A line integral of f parameterized by $\vec{r}(t)$ is computed by:

$$\int_a^b (f \circ \vec{r})(t) \| \frac{dr}{dt} \| dt.$$

For example, if $f(x, y) = 2 - x^2 - y^2$ and $r(t) = 1/t\langle \cos(t), \sin(t)\rangle$, then the line integral over $[1, 2]$ is given by:

```
f(x,y) = 2 - x^2 - y^2
f(v) = f(v...)
r(t) = [cos(t), sin(t)]/t
integrand(t) = (f∘r)(t) * norm(r'(t))
quadgk(integrand, 1, 2)
```

```
(1.2399213772953277, 4.525271268818187e-9)
```

To integrate a line integral through a vector field, say $\int_C F \cdot \hat{T} ds = \int_C F \cdot \vec{r}'(t) dt$ we have, for example,

```
F(x,y) = [-y, x]
F(v) = F(v...)
r(t) =  [cos(t), sin(t)]/t
integrand(t) = (F∘r)(t) ⋅ r'(t)
quadgk(integrand, 1, 2)
```

```
(0.5, 2.1134927141730486e-10)
```

Symbolically, there is no real difference from a 1-dimensional integral. Let $\phi = 1/\|r\|$ and integrate the gradient field over one turn of the helix $\vec{r}(t) = \langle \cos(t), \sin(t), t\rangle$.

```
@vars x y z t real=true
phi(x,y,z) = 1/sqrt(x^2 + y^2 + z^2)
r(t) = [cos(t), sin(t), t]
∇phi = diff.(phi(x,y,z), [x,y,z])
∇phi_r = subs.(∇phi, x.=> r(t)[1], y.=>r(t)[2], z.=>r(t)[3])
rp = diff.(r(t), t)
ex = simplify(∇phi_r ⋅ rp )
```

$$-\frac{t}{(t^2+1)^{\frac{3}{2}}}$$

Then

```
integrate(ex, (t, 0, 2PI))
```

$$-1 + \frac{1}{\sqrt{1+4\pi^2}}$$

Surface integrals

The surface integral for a parameterized surface involves a surface element $\|\partial\Phi/\partial u \times \partial\Phi/\partial v\|$. This can be computed numerically with:

```
Phi(u,v) = [u*cos(v), u*sin(v), u]
Phi(v) = Phi(v...)

function SE(Phi, pt)
   J = ForwardDiff.jacobian(Phi, pt)
   J[:,1] × J[:,2]
end

norm(SE(Phi, [1,2]))
```

1.4142135623730951

To find the surface integral ($f = 1$) for this surface over $[0,1] \times [0, 2\pi]$, we have:

```
hcubature(pt -> norm(SE(Phi, pt)), (0.0,0.0), (1.0, 2pi))
```

```
(4.442882938158366, 2.6645352591003757e-15)
```

Symbolically, the approach is similar:

```
@vars u v real=true
ex = Phi(u,v)
J = ex.jacobian([u,v])
SurfEl = norm(J[:,1] × J[:,2]) |> simplify
```

$$\sqrt{2}\,|u|$$

Then

```
integrate(SurfEl, (u, 0, 1), (v, 0, 2PI))
```

$$\sqrt{2}\pi$$

Integrating a vector field over the surface, would be similar:

```
F(x,y,z) = [x, y, z]
ex = F(Phi(u,v)...) ⋅ (J[:,1] × J[:,2])
integrate(ex, (u,0,1), (v, 0, 2PI))
```

$$0$$