1 Calculus plots with Makie

The Makie.jl webpage says

> From the Jpanese word Maki-e, which is a technique to sprinkle lacquer with gold and silver powder. Data is basically the gold and silver of our age, so let's spread it out beautifully on the screen!

Makie itself is a metapackage for a rich ecosystem. We show how to use the interface provided by AbstractPlotting and the GLMakie backend to produce the familiar graphics of calculus. We do not discuss the MakieLayout package which provides a means to layout multiple graphics and add widgets, such as sliders and buttons, to a layout. We do not discuss MakieRecipes. For Plots, there are "recipes" that make some of the plots more straightforward. We do not discuss the AlgebraOfGraphics which presents an interface for the familiar graphics of statistics. The MakieGallery shows many exmaples of the use of Makie.

1.1 Scenes

Makie draws graphics onto a canvas termed a "scene" in the Makie documentation. There are GLMakie, WGLMakie, and CairoMakie backends for different types of canvases. In the following, we have used GLMakie. WGLMakie is useful for incorporating Makie plots into web-based technologies.

We begin by loading our two packages:

```
using AbstractPlotting
using GLMakie
#using WGLMakie; WGLMakie.activate!()
#AbstractPlotting.set_theme!(scale_figure=false, resolution = (480, 400))
```

The Makie developers have workarounds for the delayed time to first plot, but without utilizing these the time to load the package is lengthy.

A scene is produced with Scene() or through a plotting primitive:

```
scene = Scene()
```

We see next how to move beyond the blank canvas.

1.2 Points (scatter)

The task of plotting the points, say $(1, 2)$, $(2, 3)$, $(3, 2)$ can be done different ways. Most plotting packages, and Makie is no exception, allow the following: form vectors of the x and y values then plot those with scatter:

```
xs = [1,2,3]
ys = [2,3,2]
scatter(xs, ys)
```

The `scatter` function creates and returns a `Scene` object, which when displayed shows the plot.

The more generic `plot` function can also be used for this task.

1.2.1 Point2, Point3

When learning about points on the Cartesian plane, a "t"-chart is often produced:

```
 x | y
-----
 1 | 2
 2 | 3
 3 | 2
```

The `scatter` usage above used the columns. The rows are associated with the points, and these too can be used to produce the same graphic. Rather than make vectors of x and y (and optionally z) coordinates, it is more idiomatic to create a vector of "points." `Makie` utilizes a `Point` type to store a 2 or 3 dimensional point. The `Point2` and `Point3` constructors will be utilized.

`Makie` uses a GPU, when present, to accelerate the graphic rendering. GPUs employ 32-bit numbers. Julia uses an `f0` to indicate 32-bit floating points. Hence the alternate types `Point2f0` to store 2D points as 32-bit numbers and `Points3f0` to store 3D points as 32-bit numbers are seen in the documentation for Makie.

We can plot vector of points in as direct manner as vectors of their coordinates:

```
pts = [Point2(1,2), Point2(2,3), Point2(3,2)]
scatter(pts)
```

A typical usage is to generate points from some vector-valued function. Say we have a parameterized function `r` taking R into R^2 defined by:

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

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

Then broadcasting values gives a vector of vectors, each identified with a point:

```
ts = [1,2,3]
r.(ts)
```

```
3-element Array{Array{Float64,1},1}:
 [0.8414709848078965, 0.5403023058681398]
 [0.9092974268256817, -0.4161468365471424]
 [0.1411200080598672, -0.9899924966004454]
```

We can broadcast `Point2` over this to create a vector of `Point` objects:

```
pts = Point2.(r.(ts))
```

```
3-element Array{GeometryBasics.Point{2,Float64},1}:
 [0.8414709848078965, 0.5403023058681398]
 [0.9092974268256817, -0.4161468365471424]
 [0.1411200080598672, -0.9899924966004454]
```

These then can be plotted directly:

```
scatter(pts)
```

The ploting of points in three dimesions is essentially the same, save the use of `Point3` instead of `Point2`.

```
r(t) = [sin(t), cos(t), t]
ts = range(0, 4pi, length=100)
pts = Point3.(r.(ts))
scatter(pts)
```

To plot points generated in terms of vectors of coordinates, the component vectors must be created. The "`t`"-table shows how, simply loop over each column and add the corresponding x or y (or z) value. This utility function does exactly that, returning the vectors in a tuple.

```
unzip(vs) = Tuple([vs[j][i] for j in eachindex(vs)] for i in eachindex(vs[1]))
```

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

(The functionality is essentially a reverse of the `zip` function, hence the name.)

We might have then:

```
scatter(unzip(r.(ts))...)
```

where splatting is used to specify the `xs`, `ys`, and `zs` to `scatter`.

(Compare to `scatter(Point3.(r.(ts)))` or `scatter(Point3∘r).(ts)).`)

1.2.2 Attributes

A point is drawn with a "marker" with a certain size and color. These attributes can be adjusted, as in the following:

```
scatter(xs, ys, marker=[:x,:cross, :circle], markersize=25, color=:blue)
```

Marker attributes include

- `marker` a symbol, shape. A single value will be repeated. A vector of values of a matching size will specify a marker for each point.
- `marker_offset` offset coordinates
- `markersize` size (radius pixels) of marker

1.2.3 Text (`text`)

Text can be placed at a point, as a marker is. To place text the desired text and a position need to be specified.

For example:

```
pts = Point2.(1:5, 1:5)
scene = scatter(pts)
[text!(scene, "text", position=pt, textsize=1/i, rotation=2pi/i) for (i,pt) in
enumerate(pts)]
scene
```

The graphic shows that `position` positions the text, `textsize` adjusts the displayed size, and `rotation` adjusts the orientation.

Attributes for `text` include:

- `position` to indicate the position. Either a `Point` object, as above, or a tuple
- `align` Specify the text alignment through `(:pos, :pos)`, where `:pos` can be `:left`, `:center`, or `:right`.
- `rotation` to indicate how the text is to be rotated
- `textsize` the font point size for the text
- `font` to indicate the desired font

1.3 Curves

1.3.1 Plots of univariate functions

The basic plot of univariate calculus is the graph of a function f over an interval $[a, b]$. This is implemented using a familiar strategy: produce a series of representative values between a and b; produce the corresponding $f(x)$ values; plot these as points and connect the points with straight lines. The `lines` function of `AbstractPlotting` will do the last step.

By taking a sufficient number of points within $[a, b]$ the connect-the-dot figure will appear curved, when the function is.

To create regular values between `a` and `b` either the `range` function, the related `LinRange` function, or the range operator (`a:h:b`) are employed.

For example:

```
f(x) = sin(x)
a, b = 0, 2pi
xs = range(a, b, length=250)
lines(xs, f.(xs))
```

Or

```
f(x) = cos(x)
a, b = -pi, pi
xs = a:pi/100:b
lines(xs, f.(xs))
```

As with `scatter`, `lines` returns a `Scene` object that produces a graphic when displayed.

As with `scatter`, `lines` can can also be drawn using a vector of points:

```
lines([Point2(x, fx) for (x,fx) in zip(xs, f.(xs))])
```

(Though the advantage isn't clear here, this will be useful when the points are more naturally generated.)

When a `y` value is `NaN` or infinite, the connecting lines are not drawn:

```
xs = 1:5
ys = [1,2,NaN, 4, 5]
lines(xs, ys)
```

As with other plotting packages, this is useful to represent discontinuous functions, such as what occurs at a vertical asymptote.

Adding to a scene (`lines!`, `scatter!`, ...) To *add* or *modify* a scene can be done using a mutating version of a plotting primitive, such as `lines!` or `scatter!`. The names follow `Julia`'s convention of using an `!` to indicate that a function modifies an argument, in this case the scene.

Here is one way to show two plots at once:

```
xs = range(0, 2pi, length=100)
scene = lines(xs, sin.(xs))
lines!(scene, xs, cos.(xs))
```

We will see soon how to modify the line attributes so that the curves can be distinguished.

The following shows the construction details in the graphic, and that the initial scene argument is implicitly assumed:

```
xs = range(0, 2pi, length=10)
lines(xs, sin.(xs))
scatter!(xs, sin.(xs), markersize=10)
```

The current scene will have data limits that can be of interest. The following indicates how they can be manipulated to get the limits of the displayed `x` values.

```
xs = range(0, 2pi, length=200)
scene = plot(xs, sin.(xs))
rect = scene.data_limits[] # get limits for g from f
a, b = rect.origin[1],  rect.origin[1] + rect.widths[1]
```

```
(-0.9633175f0, 6.2831855f0)
```

In the output it can be discerned that the values are 32-bit floating point numbers *and* yield a slightly larger interval than specified in `xs`.

As an example, this shows how to add the tangent line to a graph. The slope of the tangent line being computed by `ForwardDiff.derivative`.

```
using ForwardDiff
f(x) = x^x
a, b= 0, 2
c = 0.5
xs = range(a, b, length=200)

tl(x) = f(c) + ForwardDiff.derivative(f, c) * (x-c)

scene = lines(xs, f.(xs))
lines!(scene, xs, tl.(xs), color=:blue)
```

Attributes In the last example, we added the argument `color=:blue` to the `lines!` call. This set an attribute for the line being drawn. Lines have other attributes that allow different ones to be distinguished, as above where colors indicate the different graphs.

Other attributes can be seen from the help page for `lines`, and include:

- `color` set with a symbol, as above, or a string
- `linestyle` available styles are set by a symbol, one of `:dash`, `:dot`, `:dashdot`, or `:dashdotdot`.
- `linewidth` width of line
- `transparency` the `alpha` value, a number between 0 and 1, smaller numbers for more transparent.

A legend can also be used to help identify different curves on the same graphic, though this is not illustrated. There are examples in the Makie gallery.

Scene attributes Attributes of the scene include any titles and labels, the limits that define the coordinates being displayed, the presentation of tick marks, etc.

The `title` function can be used to add a title to a scene. The calling syntax is `title(scene, text)`.

To set the labels of the graph, there are ”shorthand” functions `xlabel!`, `ylabel!`, and `zlabel!`. The calling pattern would follow `xlabel!(scene, "x-axis")`.

The plotting ticks and their labels are returned by the unexported functions `tickranges` and `ticklabels`. The unexported `xtickrange`, `ytickrange`, and `ztickrange`; and `xticklabels`, `yticklabels`, and `zticklabels` return these for the indicated axes.

These can be dynamically adjusted using `xticks!`, `yticks!`, or `zticks!`.

```
pts = [Point2(1,2), Point2(2,3), Point2(3,2)]
scene = scatter(pts)
title(scene, "3 points")
ylabel!(scene, "y values")
xticks!(scene, xtickrange=[1,2,3], xticklabels=["a", "b", "c"])
```

To set the limits of the graph there are shorthand functions `xlims!`, `ylims!`, and `zlims!`. This might prove useful if vertical asymptotes are encountered, as in this example:

```
f(x) = 1/x
a,b = -1, 1
xs = range(-1, 1, length=200)
scene = lines(xs, f.(xs))
ylims!(scene, (-10, 10))
center!(scene)
```

1.3.2 Plots of parametric functions

A space curve is a plot of a function $f : R^2 \to R$ or $f : R^3 \to R$.

To construct a curve from a set of points, we have a similar pattern in both 2 and 3 dimensions:

```
r(t) = [sin(2t), cos(3t)]
ts = range(0, 2pi, length=200)
pts = Point2.(r.(ts))  # or (Point2∘r).(ts)
lines(pts)
```

Or

```
r(t) = [sin(2t), cos(3t), t]
ts = range(0, 2pi, length=200)
pts = Point3.(r.(ts))
lines(pts)
```

Alternatively, vectors of the x, y, and z components can be produced and then plotted using the pattern `lines(xs, ys)` or `lines(xs, ys, zs)`. For example, using `unzip`, as above, we might have done the prior example with:

```
xs, ys, zs = unzip(r.(ts))
lines(xs, ys, zs)
```

Tangent vectors (`arrows`) A tangent vector along a curve can be drawn quite easily using the `arrows` function. There are different interfaces for `arrows`, but we show the one which uses a vector of positions and a vector of ”vectors”. For the latter, we utilize the `derivative` function from `ForwardDiff`:

```
using ForwardDiff
r(t) = [sin(t), cos(t)] # vector, not tuple
ts = range(0, 4pi, length=200)
scene = Scene()
lines!(scene, Point2.(r.(ts)))

nts = 0:pi/4:2pi
us = r.(nts)
dus = ForwardDiff.derivative.(r, nts)

arrows!(scene, Point2.(us), Point2.(dus))
```

In 3 dimensions the differences are minor:

```
r(t) = [sin(t), cos(t), t] # vector, not tuple
ts = range(0, 4pi, length=200)
scene = Scene()
lines!(scene, Point3.(r.(ts)))

nts = pi:pi/4:3pi
us = r.(nts)
dus = ForwardDiff.derivative.(r, nts)

arrows!(scene, Point3.(us), Point3.(dus))
```

Attributes Attributes for `arrows` include

- `arrowsize` to adjust the size
- `lengthscale` to scale the size
- `arrowcolor` to set the color
- `arrowhead` to adjust the head
- `arrowtail` to adjust the tail

1.3.3 Implicit equations (2D)

The graph of an equation is the collection of all (x, y) values satisfying the equation. This is more general than the graph of a function, which can be viewed as the graph of the equation

$y = f(x)$. An equation in x-y can be graphed if the set of solutions to a related equation $f(x, y) = 0$ can be identified, as one can move all terms to one side of an equation and define f as the rule of the side with the terms.

The MDBM (Multi-Dimensional Bisection Method) package can be used for the task of characterizing when $f(x, y) = 0$. (Also `IntervalConstraintProgramming` can be used.) We first wrap its interface and then define a ”`plot`” recipe (through method overloading, not through `MakieRecipes`).

```
using MDBM
```

```
function implicit_equation(f, axes...; iteration::Int=4, constraint=nothing)

    axes = [axes...]

    if constraint == nothing
        prob = MDBM_Problem(f, axes)
    else
        prob = MDBM_Problem(f, axes, constraint=constraint)
    end

    solve!(prob, iteration)

    prob
end
```

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

The `implicit_equation` function is just a simplified wrapper for the `MDBM_Problem` interface. It creates an object to be plotted in a manner akin to:

```
f(x,y) = x^3 + x^2 + x + 1 - x*y        # solve x^3 + x^2 + x + 1 = x*y
ie = implicit_equation(f, -5:5, -10:10)
```

```
MDBM.MDBM_Problem{MDBM.MemF{typeof(Main.##WeaveSandBox#1565.f),MDBM.var"#17
#19",Float64,Bool,Tuple{Float64,Float64}},2,1,1,StaticArrays.SArray{Tuple{4
},StaticArrays.SArray{Tuple{2},Bool,1,2},1,4},StaticArrays.SArray{Tuple{3,4
},Float64,2,12},Int64,Float64,Tuple{MDBM.Axis{Float64},MDBM.Axis{Float64}}}
(MDBM.MemF{typeof(Main.##WeaveSandBox#1565.f),MDBM.var"#17#19",Float64,Bool
,Tuple{Float64,Float64}}(Main.##WeaveSandBox#1565.f, MDBM.var"#17#19"(), MD
BM.MDBMcontainer{Float64,Bool,Tuple{Float64,Float64}}[MDBM.MDBMcontainer{Fl
oat64,Bool,Tuple{Float64,Float64}}(-154.0, true, (-5.0, -10.0)), MDBM.MDBMc
ontainer{Float64,Bool,Tuple{Float64,Float64}}(-149.0, true, (-5.0, -9.0)),
MDBM.MDBMcontainer{Float64,Bool,Tuple{Float64,Float64}}(-144.0, true, (-5.0
, -8.0)), MDBM.MDBMcontainer{Float64,Bool,Tuple{Float64,Float64}}(-139.0, t
rue, (-5.0, -7.0)), MDBM.MDBMcontainer{Float64,Bool,Tuple{Float64,Float64}}
(-134.0, true, (-5.0, -6.0)), MDBM.MDBMcontainer{Float64,Bool,Tuple{Float64
,Float64}}(-129.0, true, (-5.0, -5.0)), MDBM.MDBMcontainer{Float64,Bool,Tup
le{Float64,Float64}}(-124.0, true, (-5.0, -4.0)), MDBM.MDBMcontainer{Float6
4,Bool,Tuple{Float64,Float64}}(-119.0, true, (-5.0, -3.0)), MDBM.MDBMcontai
ner{Float64,Bool,Tuple{Float64,Float64}}(-114.0, true, (-5.0, -2.0)), MDBM.
```

```
MDBMcontainer{Float64,Bool,Tuple{Float64,Float64}}(-109.0, true, (-5.0, -1.
0))  ...@*( MDBM.MDBMcontainer(*@{Float64,Bool,Tuple{Float64,Float64}}(151.0, true
, (5.0, 1.0)), MDBM.MDBMcontainer{Float64,Bool,Tuple{Float64,Float64}}(146.
0, true, (5.0, 2.0)), MDBM.MDBMcontainer{Float64,Bool,Tuple{Float64,Float64
}}(141.0, true, (5.0, 3.0)), MDBM.MDBMcontainer{Float64,Bool,Tuple{Float64,
Float64}}(136.0, true, (5.0, 4.0)), MDBM.MDBMcontainer{Float64,Bool,Tuple{F
loat64,Float64}}(131.0, true, (5.0, 5.0)), MDBM.MDBMcontainer{Float64,Bool,
Tuple{Float64,Float64}}(126.0, true, (5.0, 6.0)), MDBM.MDBMcontainer{Float6
4,Bool,Tuple{Float64,Float64}}(121.0, true, (5.0, 7.0)), MDBM.MDBMcontainer
{Float64,Bool,Tuple{Float64,Float64}}(116.0, true, (5.0, 8.0)), MDBM.MDBMco
ntainer{Float64,Bool,Tuple{Float64,Float64}}(111.0, true, (5.0, 9.0)), MDBM
.MDBMcontainer{Float64,Bool,Tuple{Float64,Float64}}(106.0, true, (5.0, 10.0
))], [18892]), MDBM.Axes{2,Tuple{MDBM.Axis{Float64},MDBM.Axis{Float64}}}(([
-5.0, -4.9375, -4.875, -4.8125, -4.75, -4.6875, -4.625, -4.5625, -4.5, -4.4
375  ...@*( 4.4375, 4.5, 4.5625, 4.625, 4.6875, 4.75, 4.8125, 4.875, 4.9375, 5.0], [-10.0,
-9.9375, -9.875, -9.8125, -9.75, -9.6875, -9.625, -9.5625, -9.5, -9.4375 (*@...@*( 9.4375,
9.5, 9.5625, 9.625, 9.6875, 9.75, 9.8125, 9.875, 9.9375, 10.0])),
MDBM.NCube(*@{Int64,Float64,2}[MDBM.NCube{Int64,Float64,2}([23, 3
16], [1, 1], [1.6063732892989169, 0.25275584507525284], true), MDBM.NCube{I
nt64,Float64,2}([23, 317], [1, 1], [1.3026770110598742, 0.205532987510733],
 true), MDBM.NCube{Int64,Float64,2}([23, 318], [1, 1], [0.9973904105883838,
 0.15779865481363942], true), MDBM.NCube{Int64,Float64,2}([23, 319], [1, 1]
, [0.6905013026761199, 0.10954668019693573], true), MDBM.NCube{Int64,Float6
4,2}([23, 320], [1, 1], [0.3819973811799878, 0.06077080938709818], true), M
DBM.NCube{Int64,Float64,2}([24, 310], [1, 1], [1.5730148223057674, 0.252441
93136240256], true), MDBM.NCube{Int64,Float64,2}([24, 311], [1, 1], [1.2635
163443540753, 0.2033503249592459], true), MDBM.NCube{Int64,Float64,2}([24,
312], [1, 1], [0.9523386316264015, 0.15370721710235083], true), MDBM.NCube{
Int64,Float64,2}([24, 313], [1, 1], [0.6394683680980245, 0.1035057125802353
], true), MDBM.NCube{Int64,Float64,2}([24, 314], [1, 1], [0.324892101175531
, 0.05273881477153529], true)  ...@*( MDBM.NCube(*@{Int64,Float64,2}([119, 316], [
1, 1], [0.5737847603815756, -0.10253884896868308], true), MDBM.NCube{Int64,
Float64,2}([119, 317], [1, 1], [0.9241595306873377, -0.16592313346062373],
true), MDBM.NCube{Int64,Float64,2}([119, 318], [1, 1], [1.2776102439518204,
 -0.23045619627359337], true), MDBM.NCube{Int64,Float64,2}([119, 319], [1,
1], [1.6341761101488126, -0.2961613662405615], true), MDBM.NCube{Int64,Floa
t64,2}([120, 315], [1, 1], [-1.6639239167933426, 0.28200587702155877], true
), MDBM.NCube{Int64,Float64,2}([120, 316], [1, 1], [-1.3398508634660045, 0.
22805972144102202], true), MDBM.NCube{Int64,Float64,2}([120, 317], [1, 1],
[-1.013129945313339, 0.17319397496295114], true), MDBM.NCube{Int64,Float64,
2}([120, 318], [1, 1], [-0.6837296229134533, 0.11739123110059835], true), M
DBM.NCube{Int64,Float64,2}([120, 319], [1, 1], [-0.35161787289425867, 0.060
633695953389216], true), MDBM.NCube{Int64,Float64,2}([120, 320], [1, 1], [-
0.01676217906067428, 0.002903178176581569], true)], StaticArrays.SArray{Tup
le{2},Bool,1,2}[[0, 0], [1, 0], [0, 1], [1, 1]], [-0.25 0.25 -0.25 0.25; -0
.25 -0.25 0.25 0.25; -0.25 -0.25 -0.25 -0.25])
```

The function definition is straightforward. The limits for `x` and `y` are specified in the above using ranges. This specifies the initial grid of points for the apdaptive algorithm used by `MDBM` to identify solutions.

To visualize the output, we make a new method for `plot` and `plot!`. There is a distinction between 2 and 3 dimensions. Below in two dimensions curve(s) are drawn. In three dimensions, scaled cubes are used to indicate the surface.

```
AbstractPlotting.plot(m::MDBM_Problem; kwargs...) = plot!(Scene(), m; kwargs...)
AbstractPlotting.plot!(m::MDBM_Problem; kwargs...) =
plot!(AbstractPlotting.current_scene(), m; kwargs...)
```

```
AbstractPlotting.plot!(scene::AbstractPlotting.Scene, m::MDBM_Problem; kwargs...) =
    plot!(Val(_dim(m)), scene, m; kwargs...)

_dim(m::MDBM.MDBM_Problem{a,N,b,c,d,e,f,g,h}) where {a,N,b,c,d,e,f,g,h} = N
```

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

Dispatch is used for the two different dimesions, identified through `_dim`, defined above.

```
# 2D plot
function AbstractPlotting.plot!(::Val{2}, scene::AbstractPlotting.Scene,
    m::MDBM_Problem; color=:black, kwargs...)

    mdt=MDBM.connect(m)
    for i in 1:length(mdt)
        dt=mdt[i]
        P1=getinterpolatedsolution(m.ncubes[dt[1]], m)
        P2=getinterpolatedsolution(m.ncubes[dt[2]], m)
        lines!(scene, [P1[1],P2[1]],[P1[2],P2[2]], color=color, kwargs...)
    end

    scene
end
```

```
# 3D plot
function AbstractPlotting.plot!(::Val{3}, scene::AbstractPlotting.Scene,
    m::MDBM_Problem; color=:black, kwargs...)

    positions = Point{3, Float32}[]
    scales = Vec3[]

    mdt=MDBM.connect(m)
    for i in 1:length(mdt)
        dt=mdt[i]
        P1=getinterpolatedsolution(m.ncubes[dt[1]], m)
        P2=getinterpolatedsolution(m.ncubes[dt[2]], m)

        a, b = Vec3(P1), Vec3(P2)
        push!(positions, Point3(P1))
        push!(scales, b-a)
    end

    cube = Rect{3, Float32}(Vec3(-0.5, -0.5, -0.5), Vec3(1, 1, 1))
    meshscatter!(scene, positions, marker=cube, scale = scales, color=color,
transparency=true, kwargs...)

    scene
end
```

We see that the equation `ie` has two pieces. (This is known as Newton's trident, as Newton was interested in this form of equation.)

```
plot(ie)
```

1.4 Surfaces

Plots of surfaces in 3 dimensions are useful to help understand the behavior of multivariate functions.

Surfaces defined through $z = f(x, y)$ The "peaks" function generates the logo for MATLAB. Here we see how it can be plotted over the region $[-5, 5] \times [-5, 5]$.

```
peaks(x,y) = 3*(1-x)^2*exp(-x^2 - (y+1)^2) - 10(x/5-x^3-y^5)*exp(-x^2-y^2)-
1/3*exp(-(x+1)^2-y^2)
xs = ys = range(-5, 5, length=25)
surface(xs, ys, peaks)
```

The calling pattern `surface(xs, ys, f)` implies a rectangular grid over the x-y plane defined by `xs` and `ys` with z values given by $f(x, y)$.

Alternatively a "matrix" of z values can be specified. For a function `f`, this is conveniently generated by the pattern `f.(xs, ys')`, the `'` being important to get a matrix of all x-y pairs through `Julia`'s broadcasting syntax.

```
zs = peaks.(xs, ys')
surface(xs, ys, zs)
```

To see how this graph is constructed, the points $(x, y, f(x, y))$ are plotted over the grid and displayed.

Here we downsample to illutrate

```
xs = ys = range(-5, 5, length=5)
pts = [Point3(x, y, peaks(x,y)) for x in xs for y in ys]
scatter(pts, markersize=25)
```

These points are connected. The `wireframe` function illustrates just the frame

```
wireframe(xs, ys, peaks.(xs, ys'), linewidth=5)
```

The `surface` call triangulates the frame and fills in the shading:

```
surface!(xs, ys, peaks.(xs, ys'))
```

Implicitly defined surfaces, $F(x, y, z) = 0$ The set of points (x, y, z) satisfying $F(x, y, z) = 0$ will form a surface that can be visualized using the `MDBM` package. We illustrate showing two nested surfaces.

```
r_2(x,y,z) = x^2 + y^2 + z^2 - 5/4 # a sphere
r_4(x,y,z) = x^4 + y^4 + z^4 - 1
xs = ys = zs = -2:2
```

```
m2,m4 = implicit_equation(r_2, xs, ys, zs), implicit_equation(r_4, xs, ys, zs)

plot(m4, color=:yellow)
plot!(m2, color=:red)
```

Parametrically defined surfaces A surface may be parametrically defined through a function $r(u, v) = (x(u, v), y(u, v), z(u, v))$. For example, the surface generated by $z = f(x, y)$ is of the form with $r(u, v) = (u, v, f(u, v))$.

The `surface` function and the `wireframe` function can be used to display such surfaces. In previous usages, the `x` and `y` values were vectors from which a 2-dimensional grid is formed. For parametric surfaces, a grid for the `x` and `y` values must be generated. This function will do so:

```
function parametric_grid(us, vs, r)
    n,m = length(us), length(vs)
    xs, ys, zs = zeros(n,m), zeros(n,m), zeros(n,m)
    for (i, u_i) in enumerate(us)
        for (j, v_j) in enumerate(vs)
            x,y,z = r(u_i, v_j)
            xs[i,j] = x
            ys[i,j] = y
            zs[i,j] = z
        end
    end
    (xs, ys, zs)
end
```

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

With the data suitably massaged, we can directly plot either a `surface` or `wireframe` plot.

For example, a sphere can be parameterized by $r(u, v) = (\sin(u)\cos(v), \sin(u)\sin(v), \cos(u))$ and visualized through:

```
r(u,v) = [sin(u)*cos(v), sin(u)*sin(v), cos(u)]
us = range(0, pi, length=25)
vs = range(0, pi/2, length=25)
xs, ys, zs = parametric_grid(us, vs, r)

scene = Scene()
surface!(scene, xs, ys, zs)
wireframe!(scene, xs, ys, zs)
```

A surface of revolution for $g(u)$ revolved about the z axis can be visualized through:

```
g(u) = u^2 * exp(-u)
r(u,v) = (g(u)*sin(v), g(u)*cos(v), u)
us = range(0, 3, length=10)
vs = range(0, 2pi, length=10)
xs, ys, zs = parametric_grid(us, vs, r)
```