1 Scalar functions

Consider a function $f : R^n \rightarrow R$. It has multiple arguments for its input (an $x_1, x_2, \ldots, x_n$) and only one, *scalar*, value for an output. Some simple examples might be:

$$f(x, y) = x^2 + y^2 \tag{1}$$
$$g(x, y) = x \cdot y \tag{2}$$
$$h(x, y) = \sin(x) \cdot \sin(y) \tag{3}$$

For two examples from real life consider the elevation Point Query Service (of the USGS) returns the elevation in international feet or meters for a specific latitude/longitude within the United States. The longitude can be associated to an x coordinate, the latitude to a y coordinate, and the elevation a z coordinate, and as long as the region is small enough, the x-y coordinates can be thought to lie on a plane. (A flat earth assumption.)

Similarly, a weather map, say of the United States, may show the maximum predicted temperature for a given day. This describes a function that take a position (x, y) and returns a predicted temperature (z).

Mathematically, we may describe the values (x, y) in terms of a point, $P = (x, y)$ or a vector $\vec{v} = \langle x, y \rangle$ using the identification of a point with a vector. As convenient, we may write any of $f(x, y)$, $f(P)$, or $f(\vec{v})$ to describe the evaluation of f at the value x and y

Before proceeding with how to define such functions in `Julia`, we load our package:

```
using CalculusWithJulia
using Plots
```

Returning to the task at hand, in `Julia`, defining a scalar function is straightforward, the syntax following mathematical notation:

```
f(x,y) = x^2 + y^2
g(x,y) = x * y
h(x,y) = sin(x) * sin(y)
```

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

To call a scalar function for specific values of x and y is also similar to the mathematical case:

```
f(1,2), g(2, 3), h(3,4)
```

```
(5, 6, -0.10679997423758245)
```

It may be advantageous to have the values as a vector or a point, as in `v=[x,y]`. Splatting can be used to turn a vector or tuple into two arguments:

```
v = [1,2]
f(v...)
```

5

Alternatively, the function may be defined using a vector argument:

```
f(v) = v[1]^2 + v[2]^2
```

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

A style required for other packages within the `Julia` ecosystem.

More verbosely, but avoiding index notation, we can use multiline functions:

```
function g(v)
    x, y = v
    x * y
end
```

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

Then we have

```
f(v), g([2,3])
```

```
(5, 6)
```

More elegantly, and the approach we will use, is to mirror the mathematical notation through multiple dispatch. If we define `f` for multiple variables, say with:

```
f(x,y) = x^2 - 2x*y^2
```

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

The we can define an alternative method with just a single variable and use splatting to turn it into multiple variables:

```
f(v) = f(v...)
```

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

The we can call `f` with a vector or point:

```
f([1,2])
```

```
-7
```

or by passing in the individual components:

```
f(1,2)
```

```
-7
```

Following a calculus perspective, we take up the question of how to visualize scalar functions within `Julia`? Further, how to describe the change in the function between nearby values?

1.1 Visualizing scalar functions

Suppose for the moment that $f : R^2 \rightarrow R$. The equation $z = f(x, y)$ may be visualized by the set of points in 3-dimensions $\{(x, y, z) : z = f(x, y)\}$. This will render as a surface, and that surface will pass a "vertical line test", in that each (x, y) value corresponds to at most one z value. We will see alternatives for describing surfaces beyond through a function of the form $z = f(x, y)$. These are similar to how a curve in the x-y plane can be described by a function of the form $y = f(x)$ but also through an equation of the form $F(x, y) = c$ or through a parametric description, such as is used for planar curves. For now though we focus on the case where $z = f(x, y)$.

In `Julia`, plotting such a surface requires a generalization to plotting a univariate function where, typically, a grid of evenly spaced values is given between some a and b, the corresponding y or $f(x)$ values are found, and then the points are connected in a dot-to-dot manner.

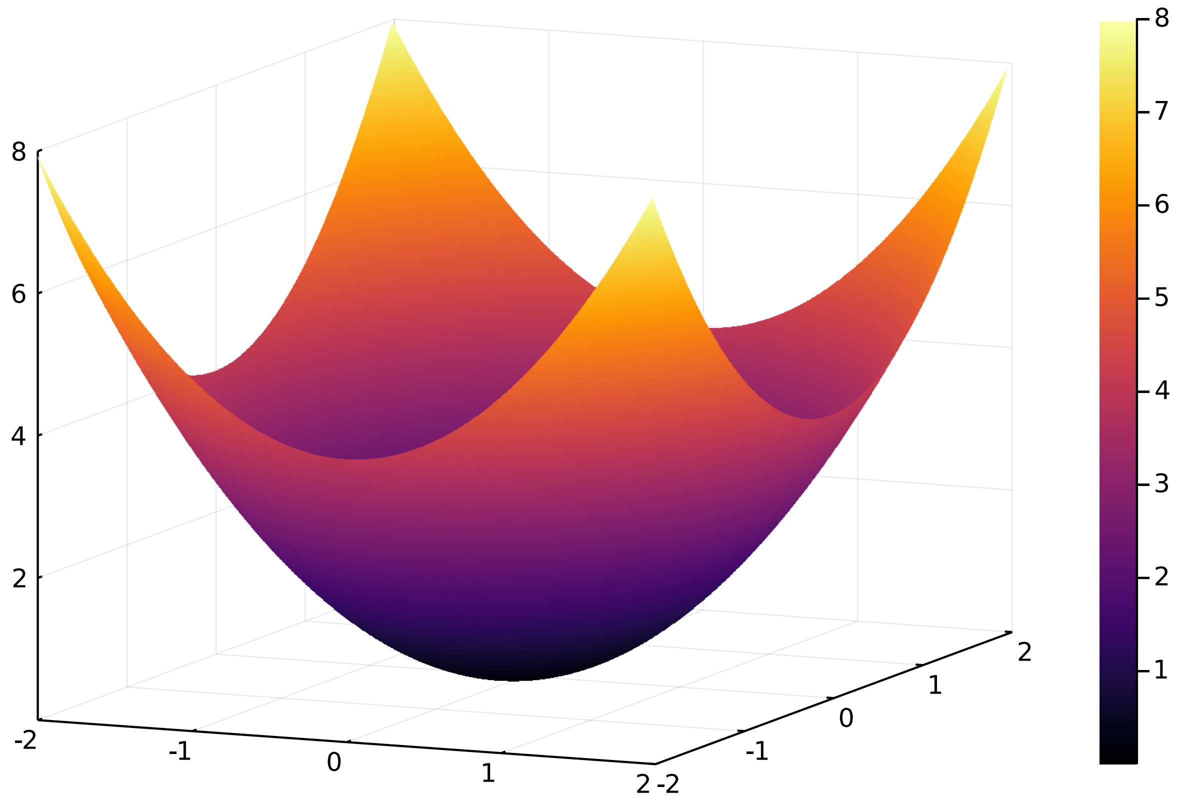
Here, a two-dimensional grid of x-y values needs specifying, and the corresponding z values found. As the grid will be assumed to be regular only the x and y values need specifying, the set of pairs can be computed. The z values, it will be seen, are easily computed. This cloud of points is plotted and each cell in the x-y plane is plotted with a surface giving the x-y-z, 3-dimensional, view. One way to plot such a surface is to tessalate the cell and then for each triangle, represent a plane made up of the 3 boundary points.

Here is an example:

```
f(x, y) = x^2 + y^2

xs = range(-2, 2, length=100)
ys = range(-2, 2, length=100)

surface(xs, ys, f)
```



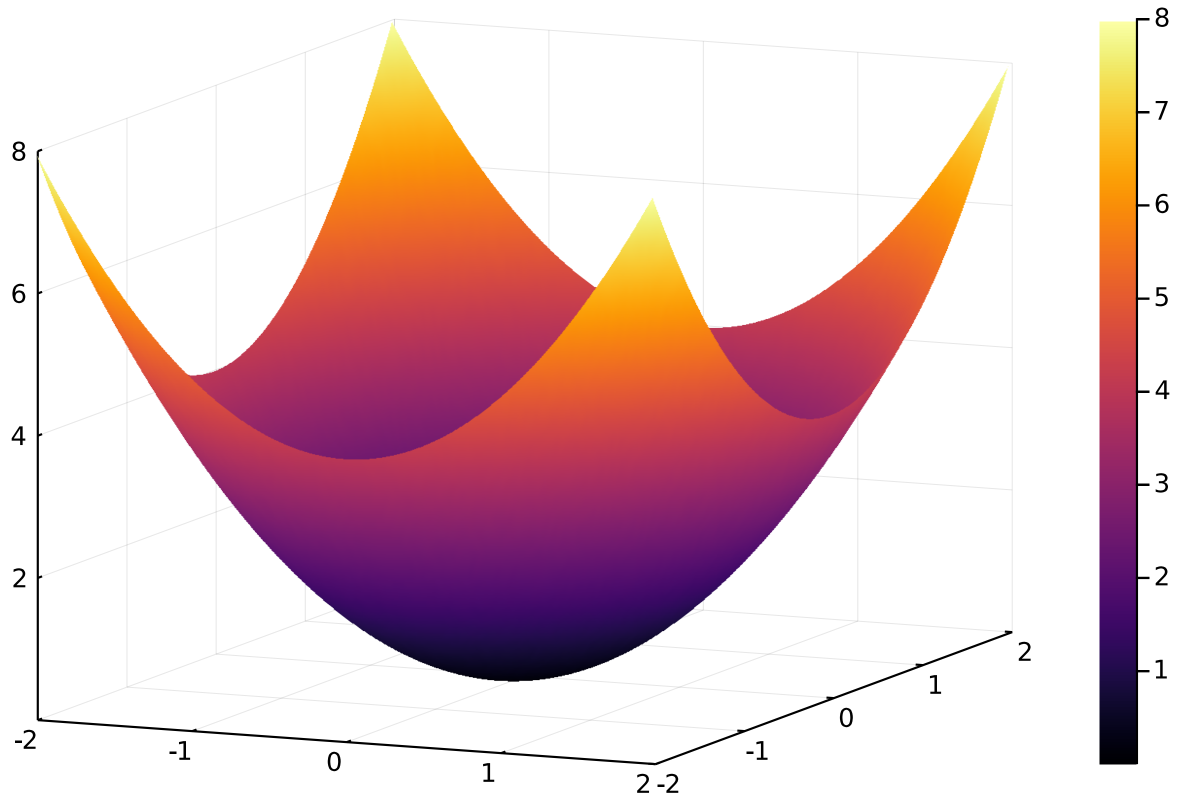
The `surface` function will generate the surface.

Using `surface` as a function name is equivalent to `plot(xs, ys, f, seriestype=:surface)`.

We can also use `surface(xs, ys, zs)` where `zs` is not a vector, but rather a *matrix* of values corresponding to a grid described by the `xs` and `ys`. A matrix is a rectangular collection of values indexed by row and column through indices `i` and `j`. Here the values in `zs` should satisfy: the ith row and jth column entry should be $z_{ij} = f(x_i, y_j)$ where x_i is the ith entry from the `xs` and y_j the jth entry from the `ys`.

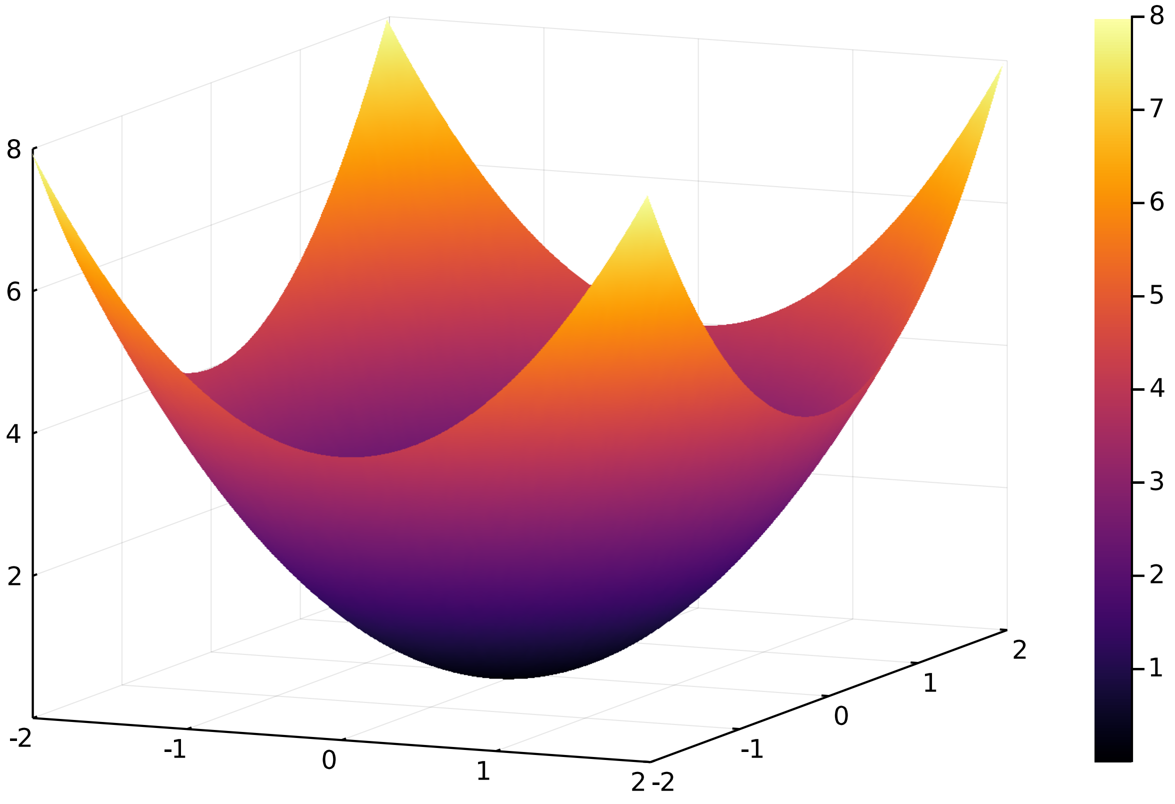
We can generate this using a comprehension:

```
zs = [f(x,y) for y in ys, x in xs]
surface(xs, ys, zs)
```



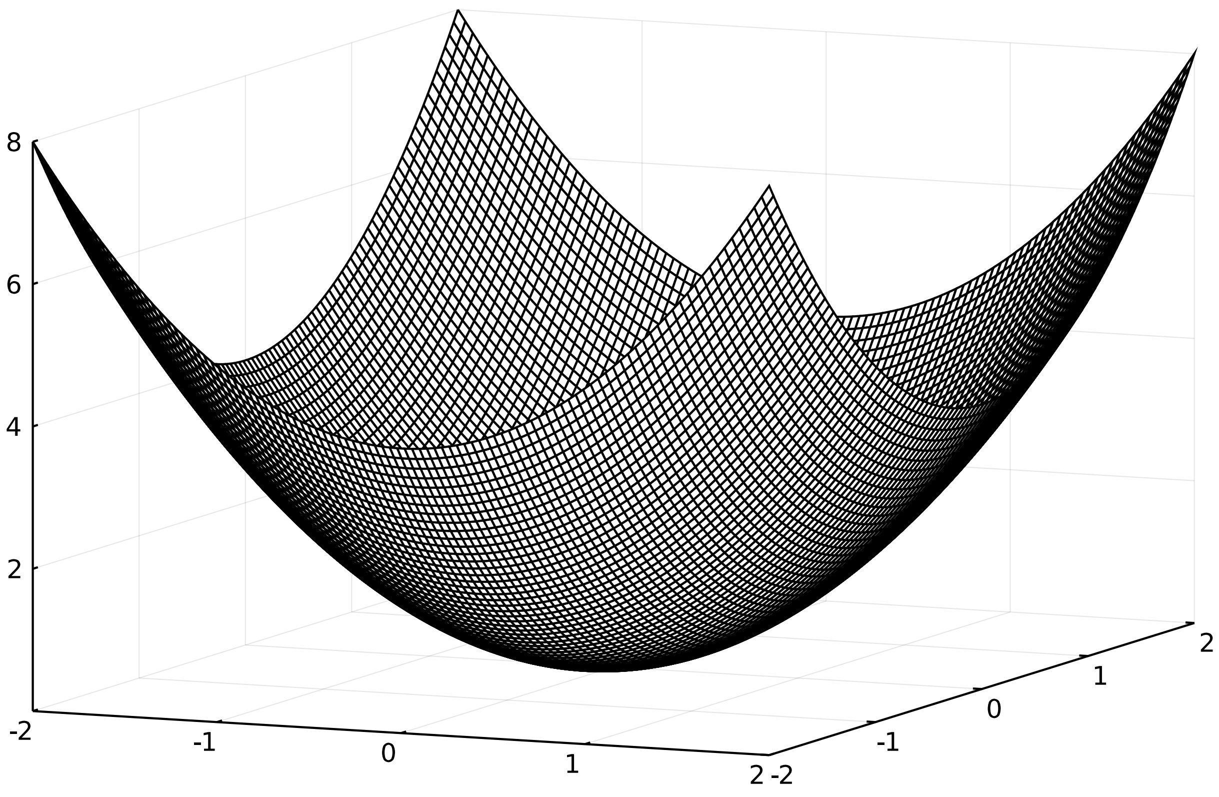
If remembering that the y values go first, and then the x values in the above is too hard, then an alternative can be used. Broadcasting `f.(xs,ys)` may not make sense, were the `xs` and `ys` not of commensurate lengths, and when it does, this call pairs off `xs` and `ys` values and passes them to `f`. What is desired here is different, where for each `xs` value there are pairs for each of the `ys` values. The syntax `xs'` can ve viewed as creating a *row* vector, where `xs` is a *column* vector. Broadcasting will create a *matrix* of values in this case. So the following is identical to the above:

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



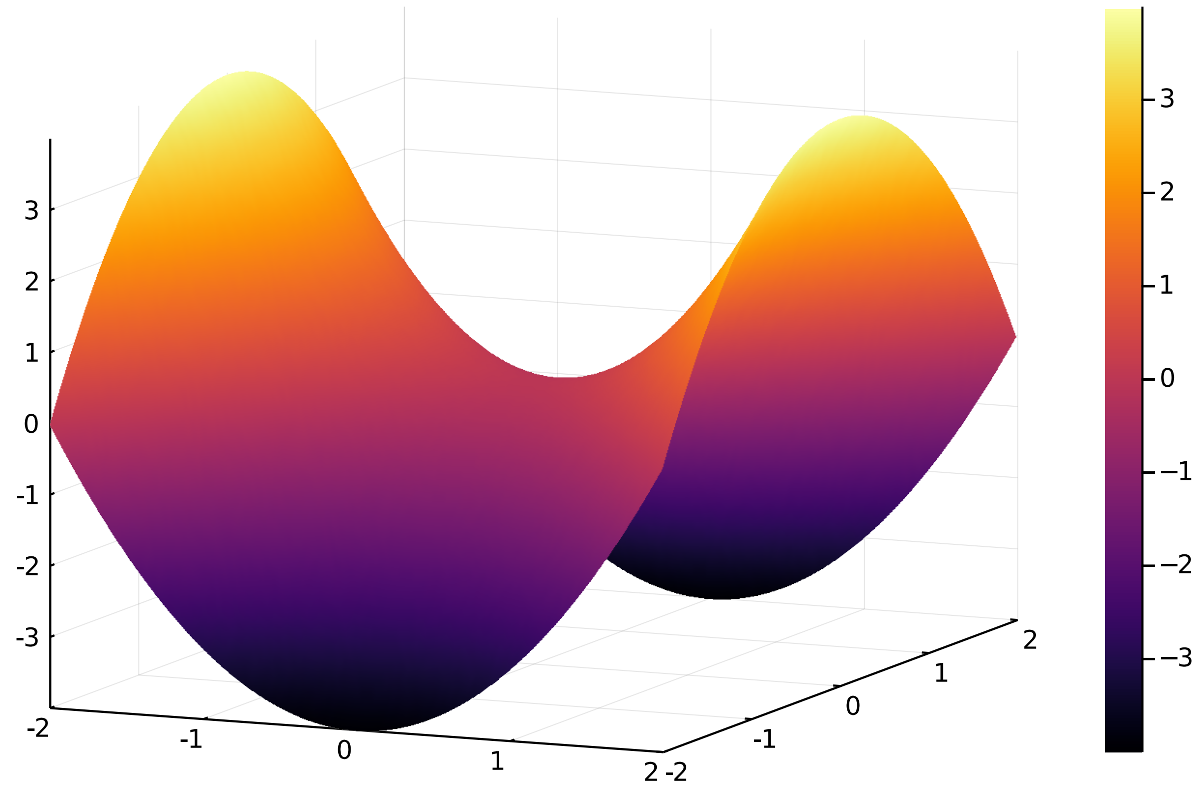
An alternate to `surface` is `wireframe`, which doesn't use shading, but rather shows the grid in the x-y plane mapped to the surface:

```
wireframe(xs, ys, f)    # gr() or pyplot() wireplots render better than plotly()
```



Example The surface $f(x, y) = x^2 - y^2$ has a "saddle," as this shows:

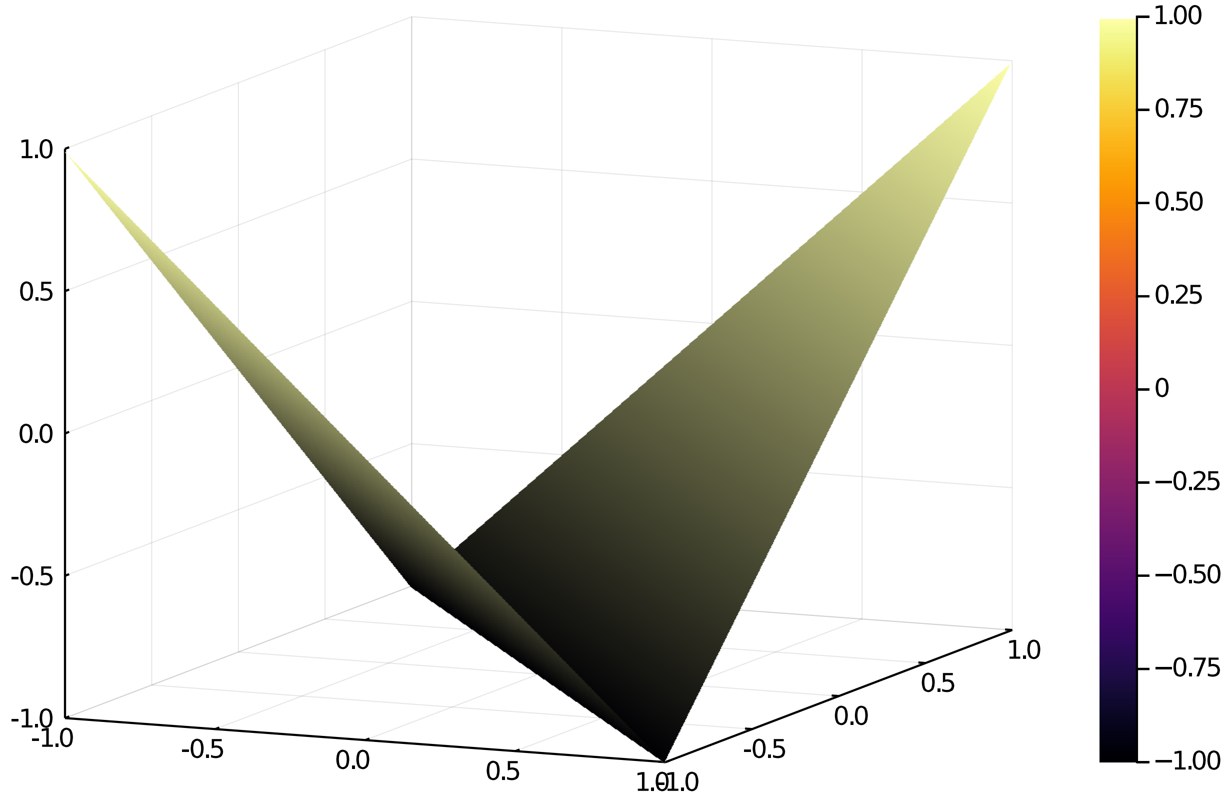
```
f(x,y) = x^2 - y^2
xs = ys = range(-2, 2, length=100)
surface(xs, ys, f)
```



Example As mentioned. In plots of univariate functions, a dot-to-dot algorithm is followed. For surfaces, the two dots are replaced by four points, which over determines a plane. Some choice is made to partition that rectangle into two triangles, and for each triangle, the 3 resulting points determines a plane, which can be suitably rendered.

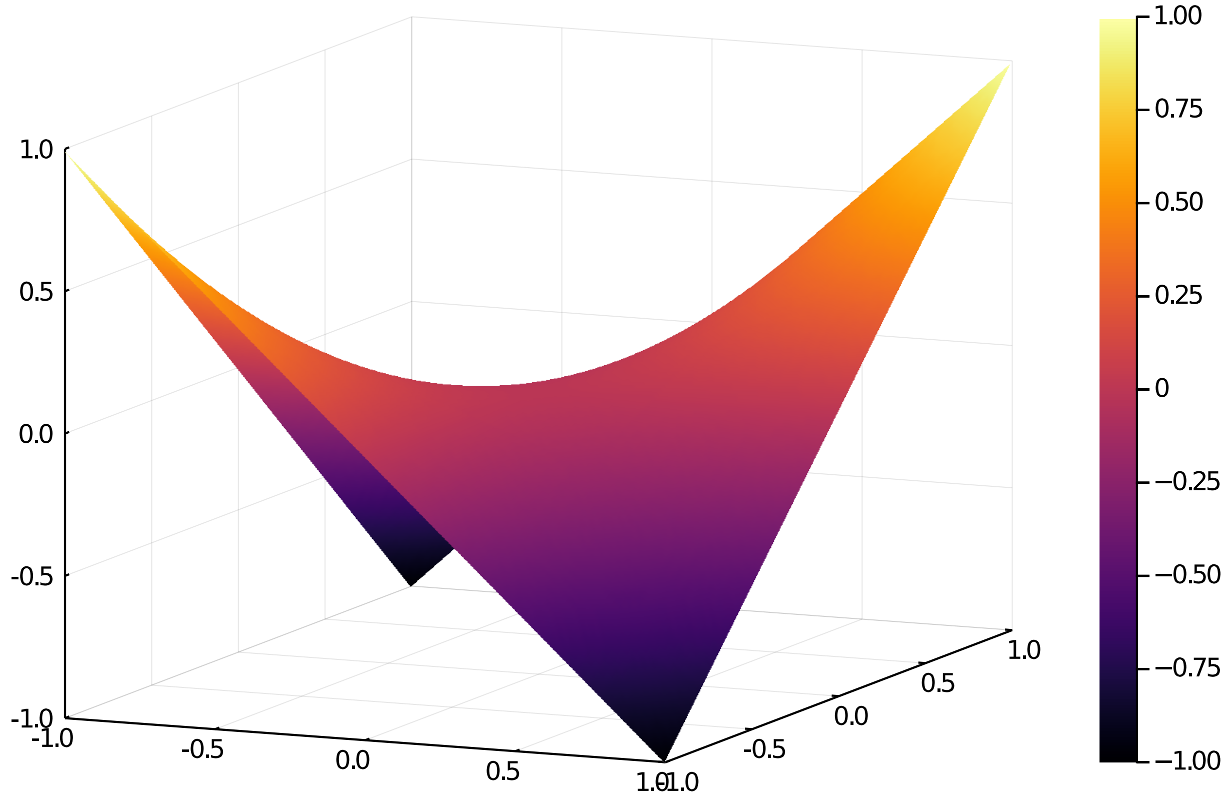
We can see this in the `GR` toolkit by forcing the surface to show just one cell, as the `xs` and `ys` below only contain 2 values:

```
gr()
xs = [-1,1];ys = [-1,1]
f(x,y) = x*y
surface(xs, ys, f.(xs', ys))
```



Compare this, to the same region, but with many cells to represent the surface:

```
xs = ys = range(-1, 1, length=100)
surface(xs, ys, f.(xs', ys))
```



1.1.1 Contour plots

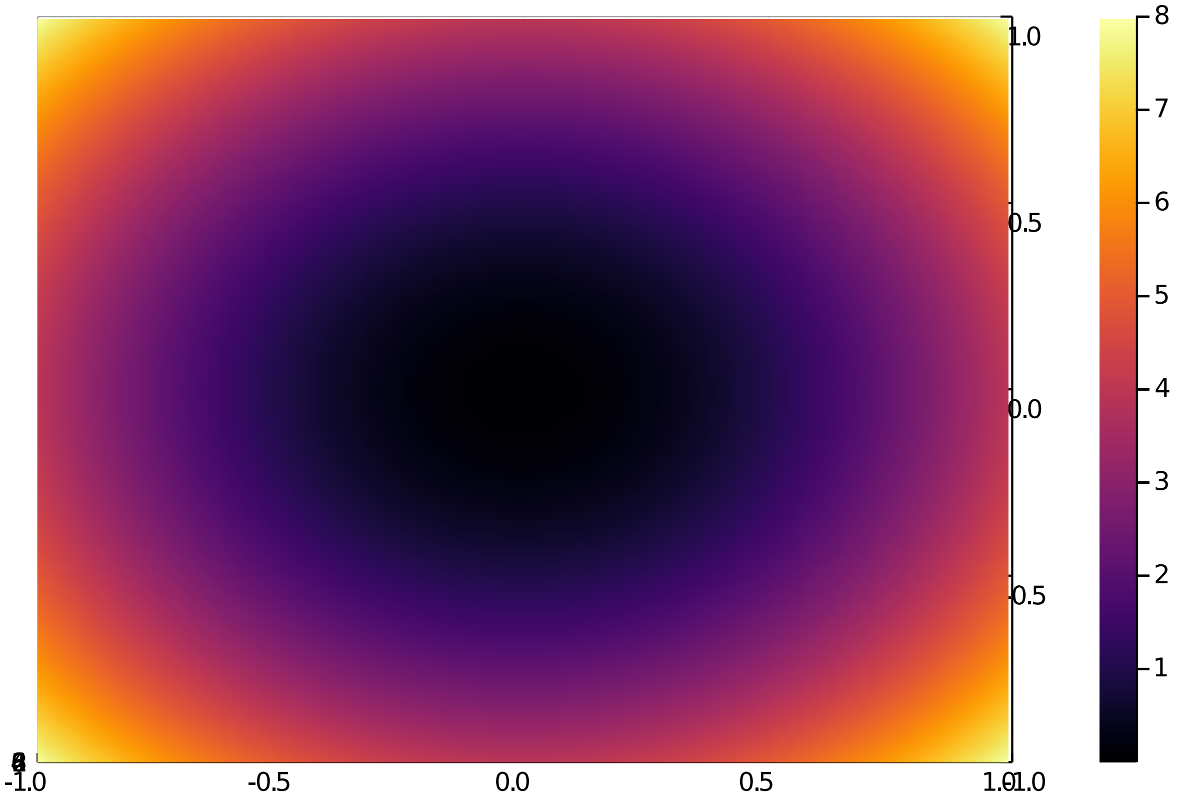
Consider the example of latitude, longitude, and elevation data describing a surface. The following graph is generated from such data, which was retrieved from the USGS website for a given area. The grid points are chosen about every 150m, so this is not too fine grained.

```
using JSON
SC = JSON.parsefile("data/somocon.json")  # a local file
xs, ys, zs =  [float.(SC[i]) for i in ("xs", "ys","zs")]
gr()  # use GR backend
surface(xs, ys, zs)
```

```
Error: SystemError: opening file "data/somocon.json": No such file or direc
tory
```

This shows a bit of the topography. If we look at the region from directly above, the graph looks different:

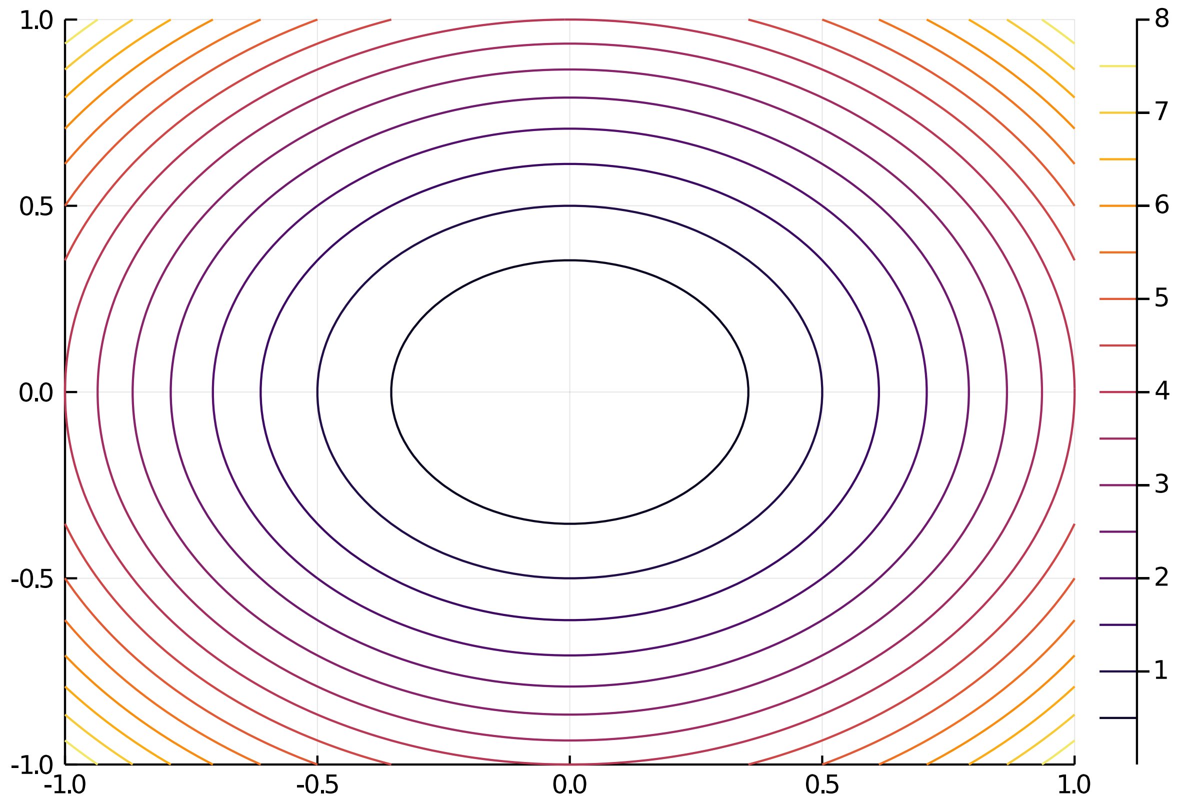
```
surface(xs, ys, zs, camera=(0, 90))
```



The rendering uses different colors to indicate height. A more typical graph, that is somewhat similar to the top down view, is a *contour* map.

For a scalar function, Define a *level curve* as the solutions to the equations $f(x, y) = c$ for a given c. (Or more generally $f(\vec{x}) = c$ for a vector if dimension 2 or more.) Plotting a selection of level curves yields a *contour* graph. These are produced with `contour` and called as above. For example, we have:

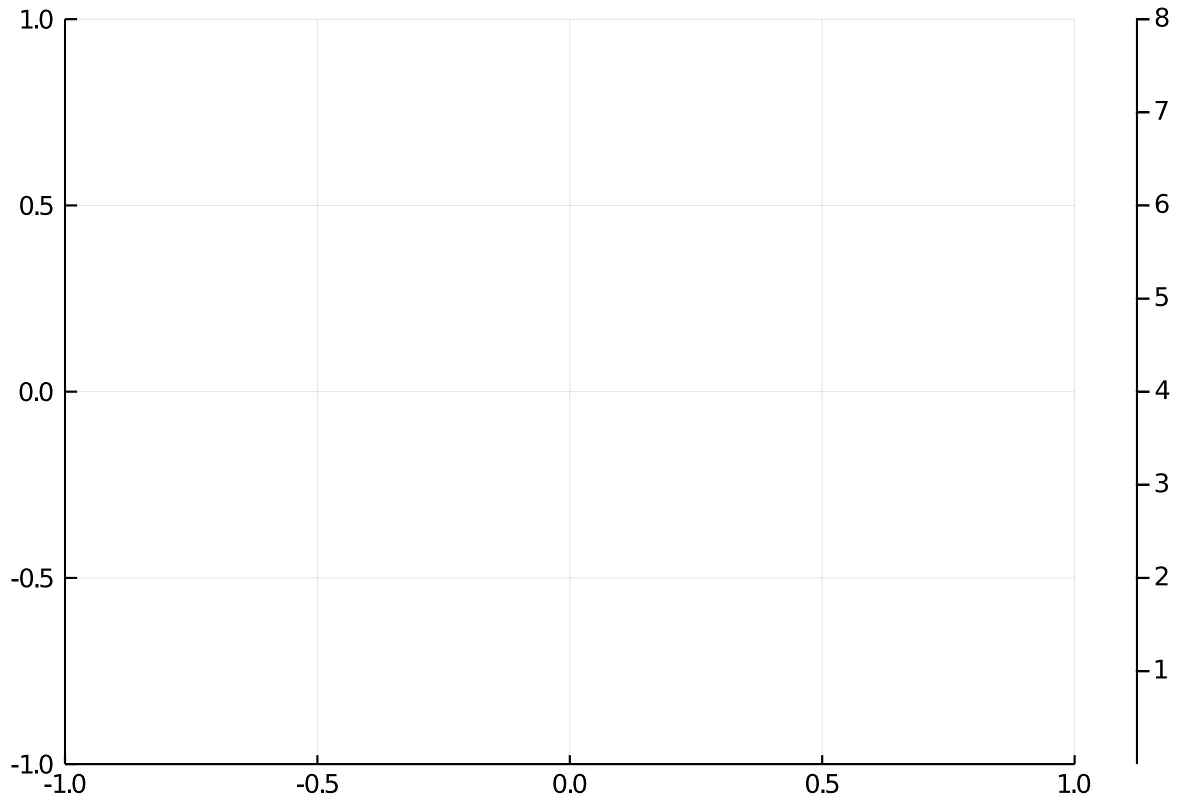
```
contour(xs, ys, zs)
```



Were one to walk along one of the contour lines, then there would be no change in elevation. The areas of greatest change in elevation - basically the hills - occur where the different contour lines are closest. In this particular area, there is a river that runs from the upper right through to the lower left and this is flanked by hills.

The c values for the levels drawn may be specified through the `levels` argument:

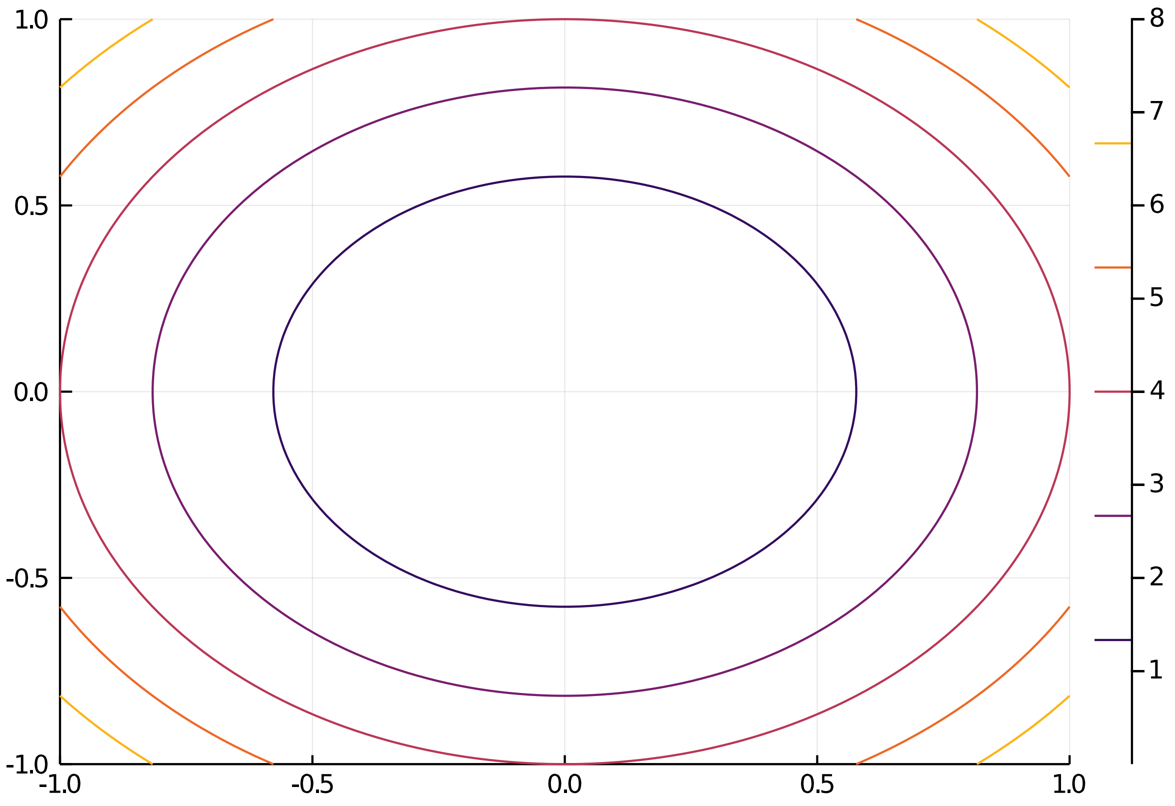
```
contour(xs, ys, zs, levels=[50,75,100, 125, 150, 175])
```



That shows the 50m, 75m, ... contours.

If a fixed number of evenly spaced levels is desirable, then the `nlevels` argument is available.

```
contour(xs, ys, zs, nlevels = 5)
```



If a function describes the surface, then the function may be passed as the third value:

```
plotly()
f(x, y) = sin(x) - cos(y)
xs = range(0, 2pi, length=100)
ys = range(-pi, pi, length = 100)
contour(xs, ys, f)
```

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

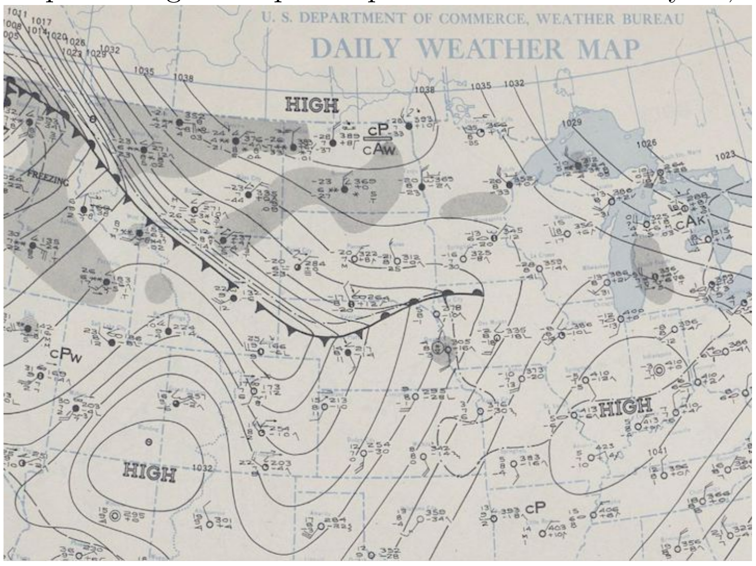
Example An informative graphic mixes both a surface plot with a contour plot. The `PyPlot` package can be used to generate one, but such graphs are not readily made within the `Plots` framework. Here is a workaround, where the contours are generated through the `Contours` package. This shows how to add a contour at a fixed level (0 below). As no hidden line algorithm is used to hide the contour line if the surface were to cover it, a transparency is specified through `alpha=0.5`:

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

function surface_contour(xs, ys, f; offset=0)
  p = surface(xs, ys, f, legend=false, fillalpha=0.5)

  ## we add to the graphic p, then plot
  zs = [f(x,y) for x in xs, y in ys]  # reverse order for use with Contour package
  for cl in levels(contours(xs, ys, zs))
    lvl = level(cl) # the z-value of this contour level
```

Figure 1: Image from [weather.gov](https://www.weather.gov/unr/1943-01-22) of a contour map showing atmospheric pressures from January 22, 1943 in Rapid City, South Dakota.



```
    for line in lines(cl)
        _xs, _ys = coordinates(line) # coordinates of this line segment
        _zs = offset * _xs
        plot!(p, _xs, _ys, _zs, alpha=0.5)        # add curve on x-y plane
    end
  end
  p
end

xs = ys = range(-pi, stop=pi, length=100)
f(x,y) = 2 + sin(x) - cos(y)

surface_contour(xs, ys, f)
```

```
Plot{Plots.PlotlyBackend() n=25}
```

We can see that at the minimum of the surface, the contour lines are nested closed loops with decreasing area.

Example The figure shows a weather map from 1943 with contour lines based on atmospheric pressure. These are also know as *isolines*.

This day is highlighted as ”The most notable temperature fluctuations occurred on January 22, 1943 when temperatures rose and fell almost 50 degrees in a few minutes. This phenomenon was caused when a frontal boundary separating extremely cold Arctic air from warmer Pacific air rolled like an ocean tide along the northern and eastern slopes of the Black Hills.”

This frontal boundary is marked with triangles and half circles along the thicker black line. The tight spacing of the contour lines above that marked line show a big change in pressure in a short distance.