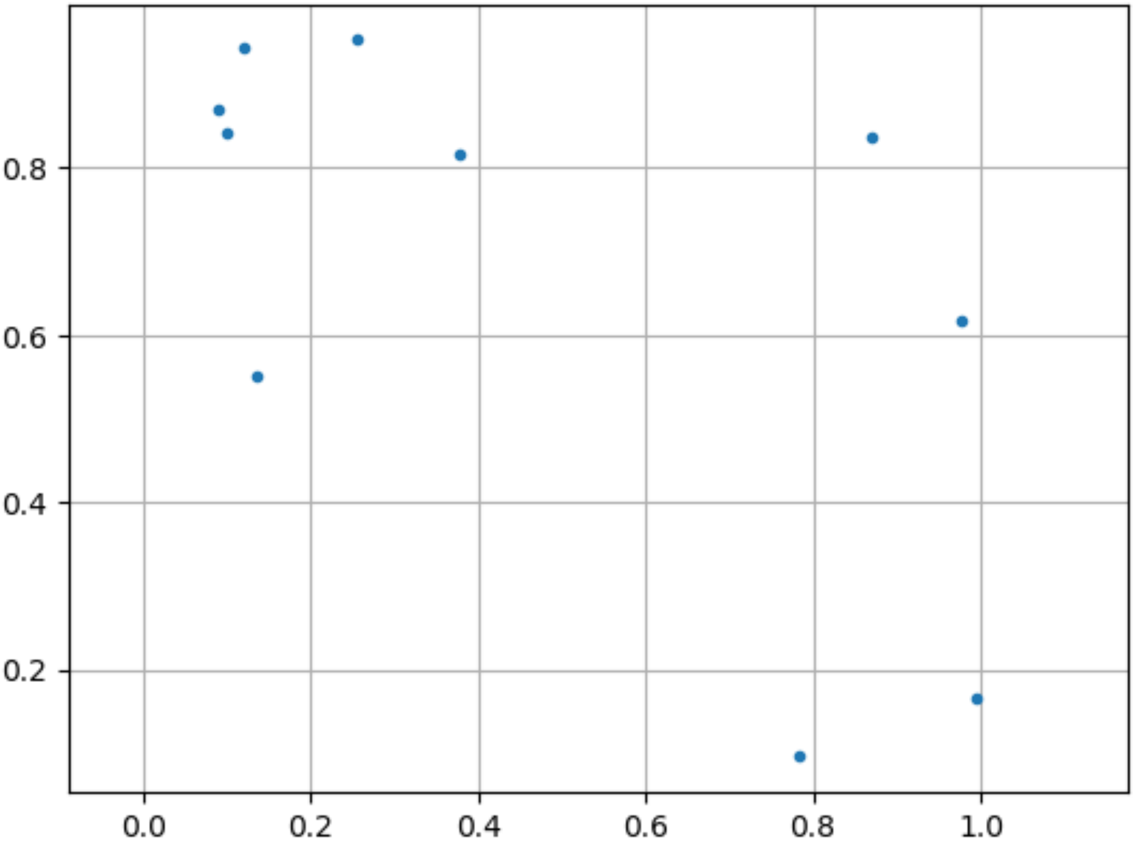
In [8]:
```
using PyPlot
```

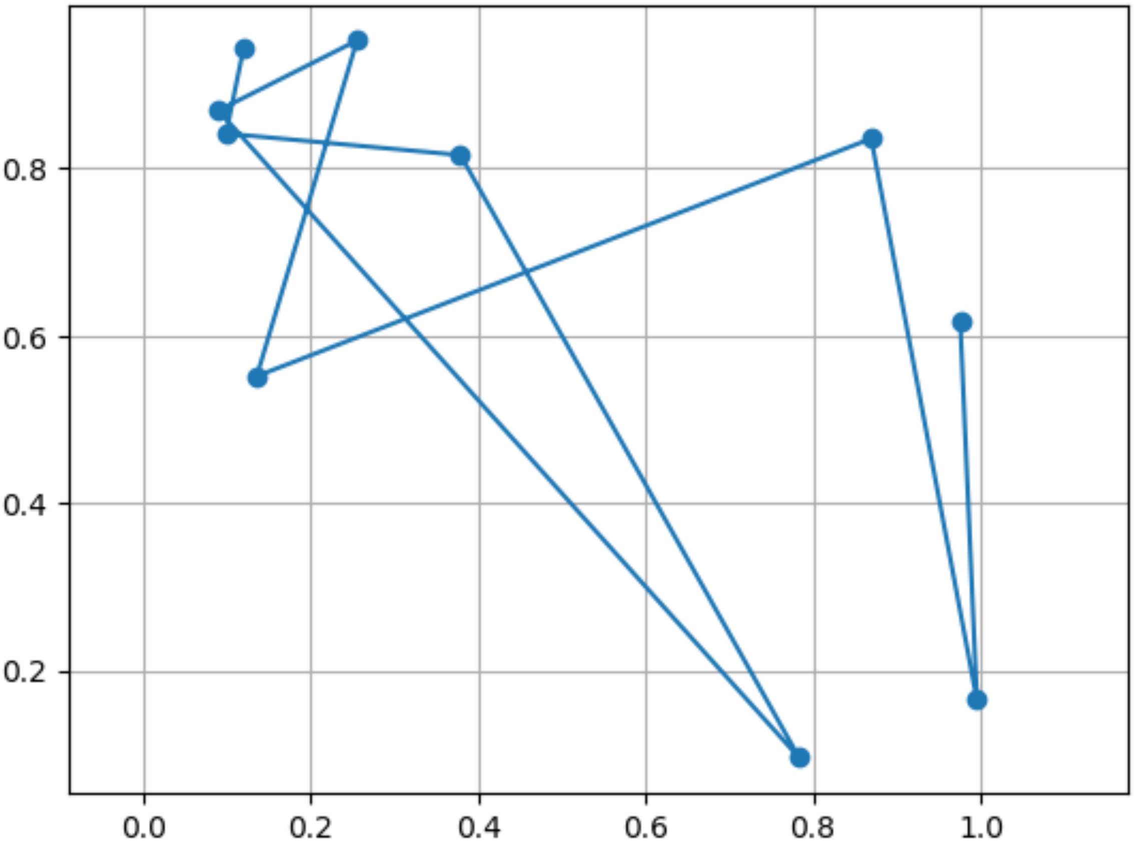
Points/Lines in 2D

In [2]:
```
p = rand(10,2)
plot( p[:,1], p[:,2], ".")#Marker type is important
axis("equal"); grid(true);
```



In [3]:

```
plot( p[:,1], p[:,2], "o-")#Marker type is important
axis("equal"); grid(true);
```



In [4]:

```
function clockwise_oriented(p1, p2, p3)
    # Return true if the line-segment between points p1,p2 is clockwis
    # oriented to the line-segment between points p1,p3
    cross = (p3[2] - p1[2]) * (p2[1] - p1[1]) - (p3[1] - p1[1]) * (p2[
    return cross > 0
end
```

Out[4]:

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

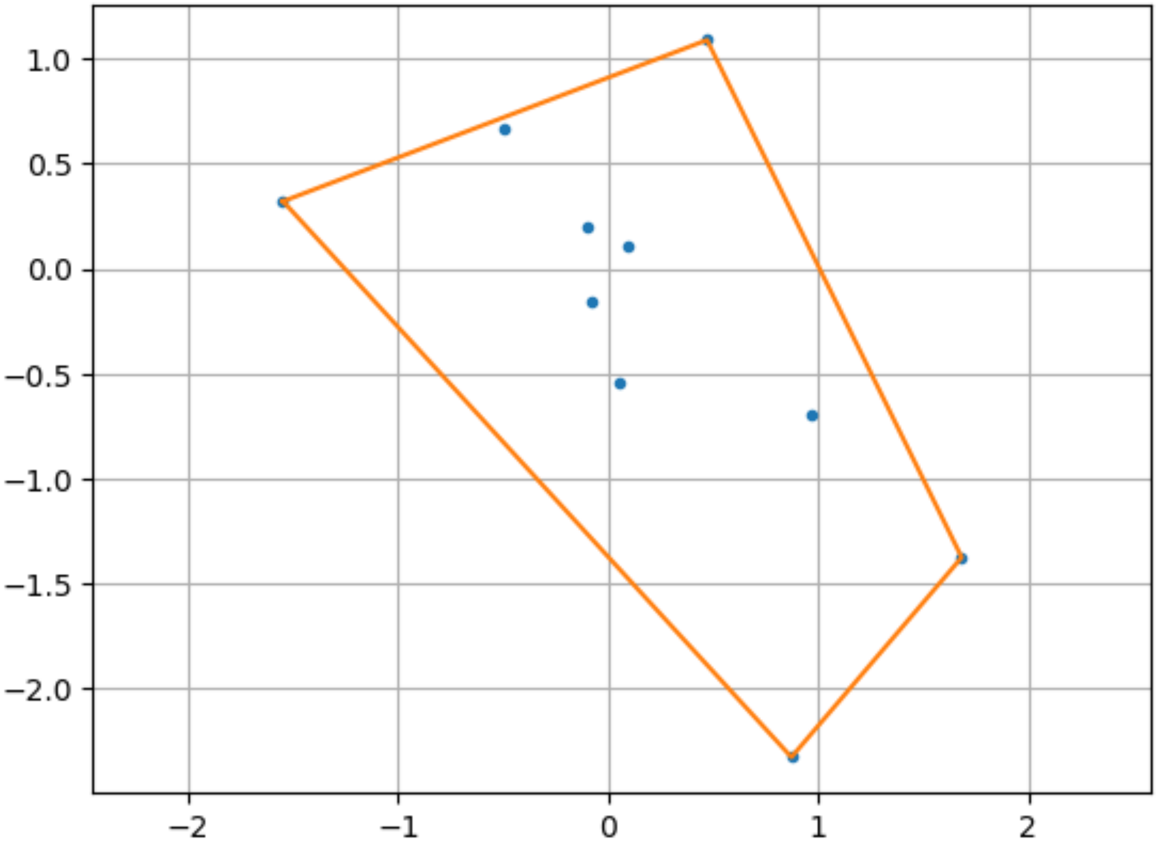
In [5]:

```
function convex_hull(p)
    # Find the nodes on the convex hull of the point array p using
    # the Jarvis march (gift wrapping) algorithm

    _, pointOnHull = findmin(first.(p)) # Start at left-most point
    hull = [pointOnHull] # Output: Vector of node indices on the conve

    while length(hull) ≤ 1 || hull[1] != hull[end] # Loop until close
        nextPoint = hull[end] % length(p) + 1 # First candidate, any p
        for j = 1:length(p) # Consider all other points
            if clockwise_oriented(p[hull[end]], p[nextPoint], p[j]) #
                nextPoint = j
            end
        end
        push!(hull, nextPoint) # Update current point
    end
    return hull
end
```

Out[5]:

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

In [6]:
```
p = [ randn(2) for i = 1:10 ]
hull = convex_hull(p)
plot(first.(p), last.(p), ".")
plot(first.(p[hull]), last.(p[hull]))
axis("equal"); grid(true);
```



The way to understand the convex hull intuitively is this - imagine stretching a rubber band around a bunch of pins at the location of the points. The convex hull is the shape the rubber band will end up taking.

A shape whose convex hull is equal to itself is convex (there are loads of definitions of convexity). This is actually a very important feature for many applciations (optimisation, solution of numerical PDEs, rendering, ...).

Interpolation

Why is this a bad idea? I stole/adapted the code from https://www.matecdev.com/posts/julia-interpolation.html (https://www.matecdev.com/posts/julia-interpolation.html)

In [1]:
```
function LagrangeInterp1D( fvals, xnodes, barw, t )
    numt = 0
    denomt = 0

    for j = 1 : length( xnodes )
        tdiff = t - xnodes[j]
        numt = numt + barw[j] / tdiff * fvals[j]
        denomt = denomt + barw[j] / tdiff

        if ( abs(tdiff) < 1e-15 )
            numt = fvals[j]
            denomt = 1.0
            break
        end
    end

    return numt / denomt

end
```

Out[1]:
```
LagrangeInterp1D (generic function with 1 method)
```

In [5]:
```
# Equispaced points
EquispacedNodes(n) = [2*(j/n-0.5) for j=0:n]
EquispacedBarWeights(n) = [ (-1)^j * binomial(n,j) for j=0:n ]
```

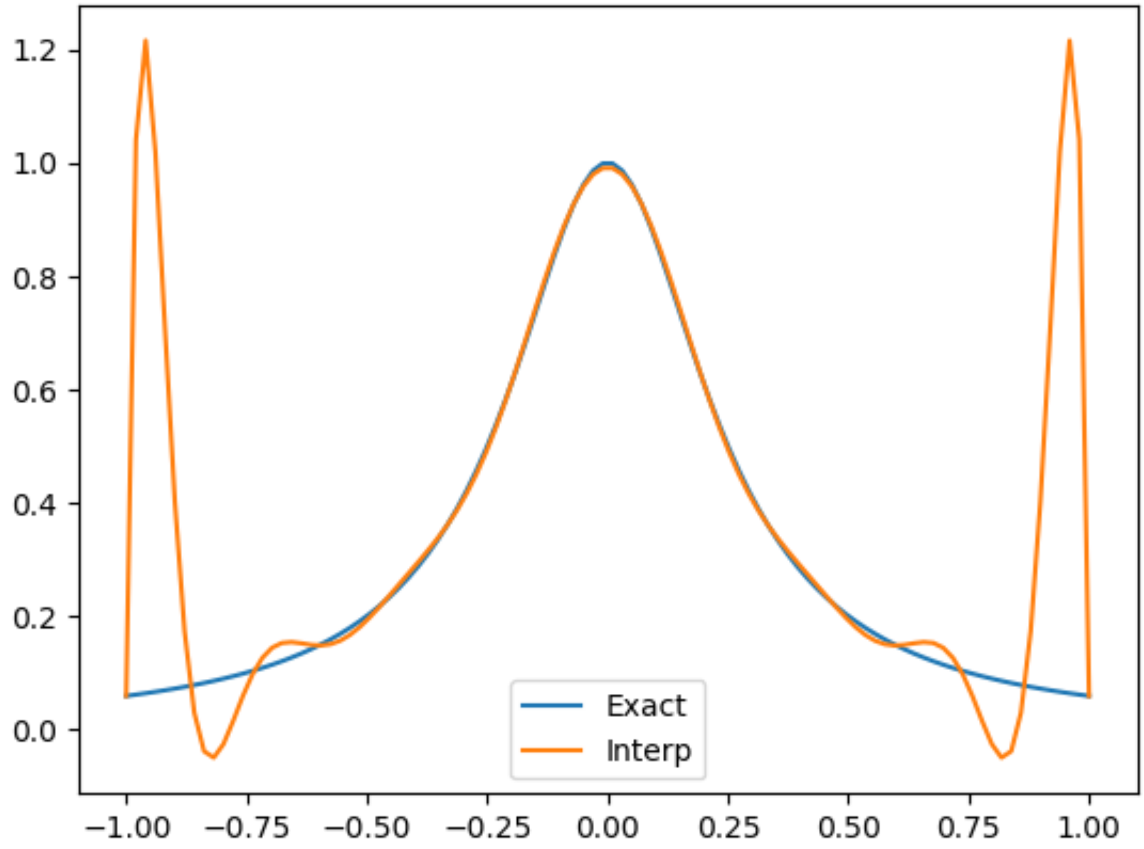
Out[5]:
```
EquispacedBarWeights (generic function with 1 method)
```

In [6]:
```
f(x) = 1/(1 + 16*x^2)
```

Out[6]:
```
f (generic function with 1 method)
```

In [11]:

```
# Sampling
n = 15;
xnodes = EquispacedNodes(n);
w = EquispacedBarWeights(n);
f_sample = f.(xnodes);

# Interpolation
t = LinRange(-1,1,100)
f_interp = [LagrangeInterp1D( f_sample, xnodes, w, t[j] ) for j=1:leng
plot(t,f.(t), label="Exact")
plot(t,f_interp, label="Interp")
legend();
```



Out[11]:

```
PyObject <matplotlib.legend.Legend object at 0x7fc44050f610>
```

This is called Runge's phenomenon, whereby high order interpolatory polynomials can oscillate wildly even though they are meant to be higher order accurate approximations of a function.

Bezier Curves

Bezier curves are an example of a parametric curve used to approximate a shape smoothly using polynomials. They were introduced for construction of cars in fact and are the basis of a lot of types of curves used in real practical applications.

In your homework you will explore a version of this known as a cubic spline to draw a car.

Here is a demo of sort of how Bezier curves work:

In [7]:
```
function eval_bezier(ctrlpts, x)
    #ctrlpts: control points
    #x: value at which to evaluate
    n = length(ctrlpts)-1
    val = [0.0,0.0]
    for i = 0:n
        val += binomial(n, i) * (1-x)^(n-i) * x^i * ctrlpts[i+1]
    end

    return val
end
```

Out[7]:
```
eval_bezier (generic function with 1 method)
```

In [8]:

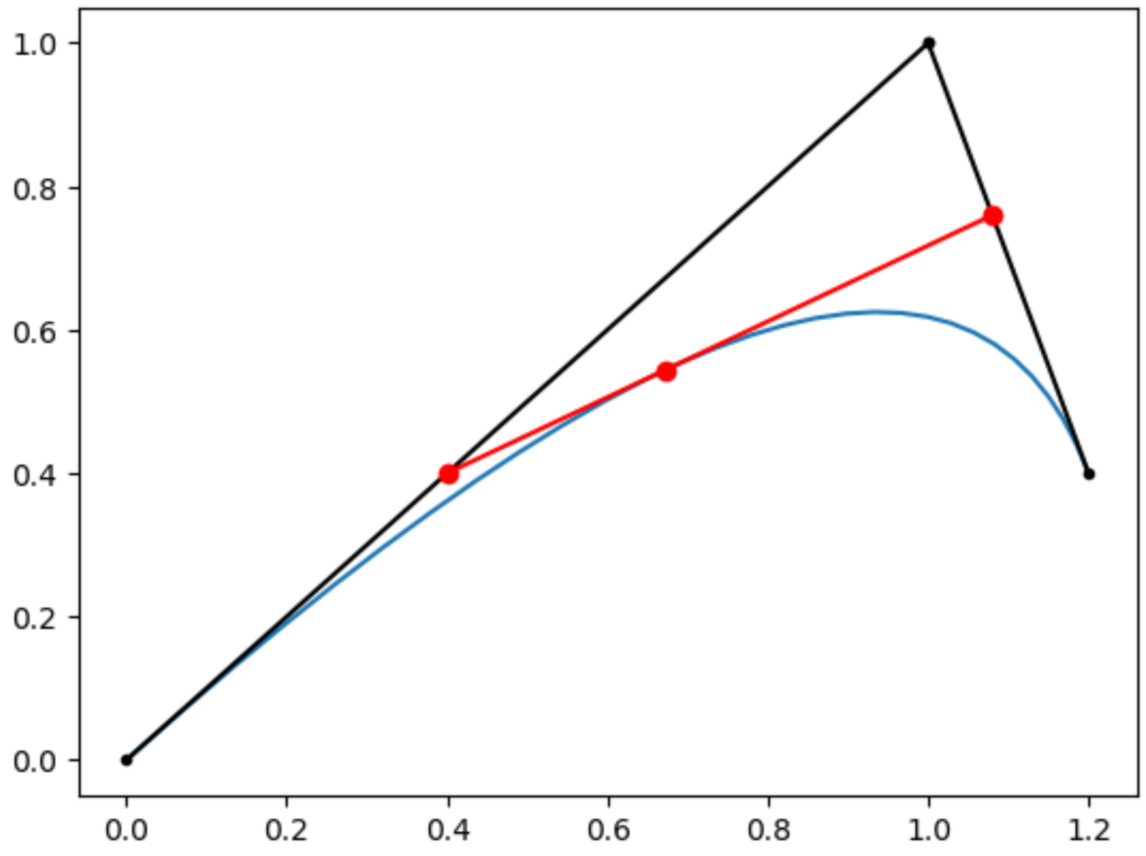
```
function quad_demo()
    #Things to set
    t = 0.4 #Parameter at which to evaluate
    nsmooth = 30; #Smoothness of graph
    #Control points
    ctrlpt0 = [0,0]
    ctrlpt1 = [1,1]
    ctrlpt2 = [1.2,0.4]

    ctrlpts = [ctrlpt0, ctrlpt1, ctrlpt2]

    """DON'T TOUCH BELOW HERE"""

    #Bezier part, evaluate Bezier Curve
    xvals = LinRange(0,1,nsmooth)
    xxvals = []
    yyvals = []
    for x in xvals
        vals = eval_bezier(ctrlpts, x)
        push!( xxvals, vals[1] )
        push!( yyvals, vals[2] )
    end
    #Plot graph
    plot( xxvals, yyvals )

    #Plot control points and control polygon
    for i in 1:length(ctrlpts)-1
        plot( [ctrlpts[i][1], ctrlpts[i+1][1]],[ctrlpts[i][2], ctrlpts
    end

    #DeCasteljau Algorithm
    ptA = ctrlpt0 + t * (ctrlpt1 - ctrlpt0)
    ptB = ctrlpt1 + t * (ctrlpt2 - ctrlpt1)
    plot(ptA[1], ptA[2], "ro")
    plot(ptB[1], ptB[2], "ro")
    plot([ptA[1], ptB[1]],[ptA[2], ptB[2]],"r-")
    ptC = ptA + t * (ptB - ptA)
    plot(ptC[1], ptC[2], "ro")

end
```

Out[8]:

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

Bezier curves and other spline based curves define a control polygon, shown in black below, of the shape you roughly want your final product to look like. It may/may not depending on user preference be interpolatory, and just guides the shape of your curve.

In [9]:
```
quad_demo();
```



So you see that the first red point is t along the first line, the second red point is t along the second line, and the middle red point is t along the line connecting the first two points. This is exactly how Bezier curves are defined.

Bezier curves are the building blocks for something called B-Splines (basis splines) which form a basis for the spline functions used in practice. You will be implementing in your hw a cubic spline (probably the most common one used).

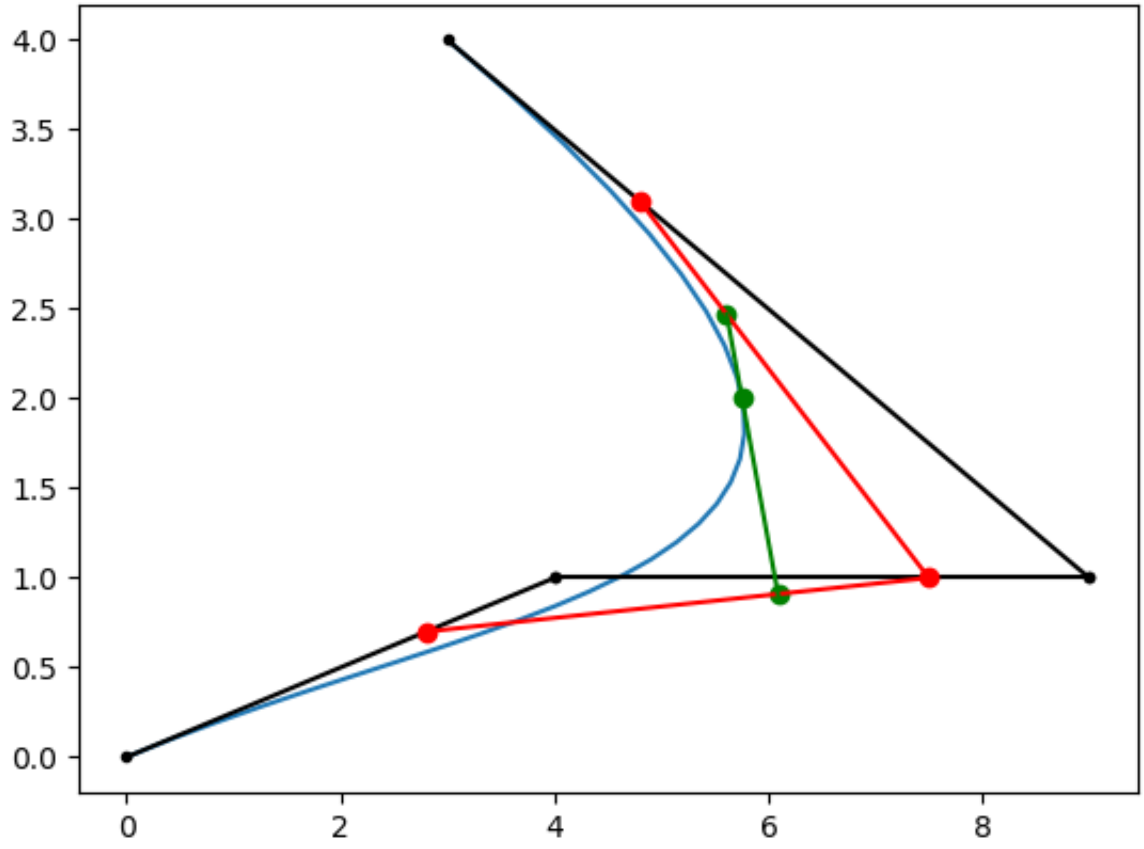
The example below shows a cubic Bezier curve, which is similar to the cubic spline you will be doing but slightly different. (The cubic spline is interpolatory for one)

In [10]:
```
function cubic_demo()
    #Things to set
    t = 0.7 #Parameter at which to evaluate
    nsmooth = 30; #Smoothness of graph
    #Control points
    ctrlpt0 = [0,0]
    ctrlpt1 = [4,1]
    ctrlpt2 = [9,1]
    ctrlpt3 = [3,4]
```

```
    ctrlpts = [ctrlpt0, ctrlpt1, ctrlpt2, ctrlpt3]

    """DON'T TOUCH BELOW HERE"""

    #Bezier part, evaluate Bezier Curve
    xvals = LinRange(0,1,nsmooth)
    xxvals = []
    yyvals = []
    for x in xvals
        val = eval_bezier(ctrlpts, x)
        push!( xxvals, val[1] )
        push!( yyvals, val[2] )
    end
    #Plot graph
    plot( xxvals, yyvals )

    #Plot control points and control polygon
    for i = 1:length(ctrlpts)-1
        plot([ctrlpts[i][1], ctrlpts[i+1][1]],[ctrlpts[i][2], ctrlpts[
    end

    #DeCasteljau Algorithm
    ptA = ctrlpt0 + t * (ctrlpt1 - ctrlpt0)
    ptB = ctrlpt1 + t * (ctrlpt2 - ctrlpt1)
    plot(ptA[1], ptA[2], "ro")
    plot(ptB[1], ptB[2], "ro")
    ptC = ctrlpt2 + t * (ctrlpt3 - ctrlpt2)
    plot(ptC[1], ptC[2], "ro")
    ptD = ptA + t * (ptB - ptA)
    ptE = ptB + t * (ptC - ptB)
    plot(ptD[1], ptD[2], "go")
    plot(ptE[1], ptE[2], "go")
    ptF = ptD + t * (ptE - ptD)
    plot(ptF[1], ptF[2], "go")

    plot([ptA[1], ptB[1]],[ptA[2], ptB[2]],"r-")
    plot([ptB[1], ptC[1]],[ptB[2], ptC[2]],"r-")
    plot([ptD[1], ptE[1]],[ptD[2], ptE[2]],"g-")

end
```

```
Out[10]: cubic_demo (generic function with 1 method)
```

In [11]:
```
cubic_demo();
```



Splines

Splines are piecewise polynomial representations of curves such that they can approximate a shape well but do not oscillate like polynomial interpolants.

In [12]:
```
using Dierckx
```

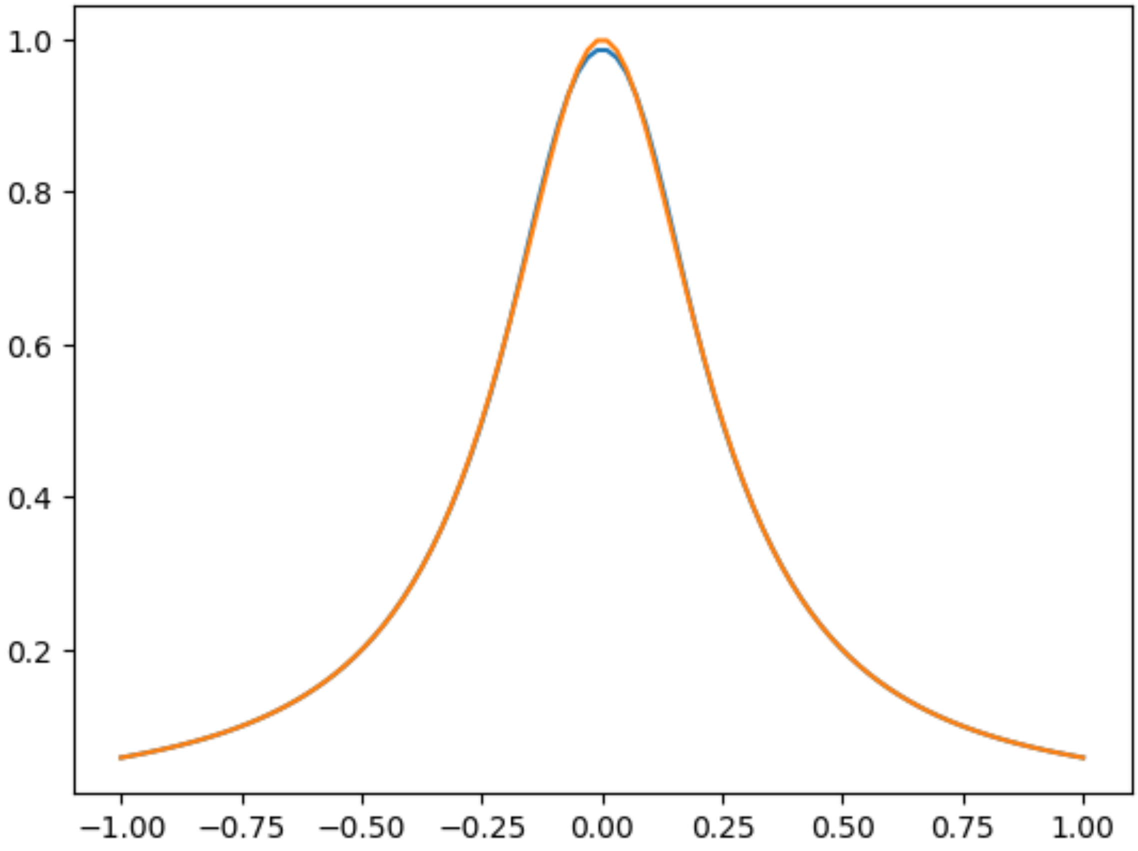
For demo purposes I'm using a package, derivation of these splines can be found on wikipedia or wolfram. Or you can just come and ask me (they can actually be very efficient derived from the Bezier curves above)

In [13]:
```
spl = Spline1D(xnodes, f_sample)
```

Out[13]:
```
Spline1D(knots=[-1.0,-0.733333 … 0.733333,1.0] (14 elements), k=3, ex
trapolation="nearest", residual=0.0)
```

In [18]:

```
xeval = LinRange(xnodes[1], xnodes[end], 100)
yeval = [spl(x) for x in xeval]
plot( xeval, yeval );
plot(t,f.(t), label="Interp");
```



So we see that this actually does it much better without any oscillations. This is because the spline is composed of low degree polynomials so cannot oscillate as much.

Delaunay Triangulation

In [16]:

```
using PyCall
```

In 2D, geometry becomes more complex, and to discretise, you need to define "elements." For example this can be by triangulating your domain - one famous method that is pretty much guaranteed to work (remarkably) is the Delaunay triangulation.

Some interesting properties of Delaunay:

1) The circles around each 3 points forming a triangle do not contain circumcentre of any other triangle

2) The minimum angle is maximised

3) Guaranteed to terminate (this is very important - otherwise it could infinite loop or something)

In [13]:

```julia
function delaunay(p)
    tri = pyimport("matplotlib.tri")
    t = tri[:Triangulation](p[:,1], p[:,2])
    return Int64.(t[:triangles] .+ 1)
end
```

Out[13]:

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

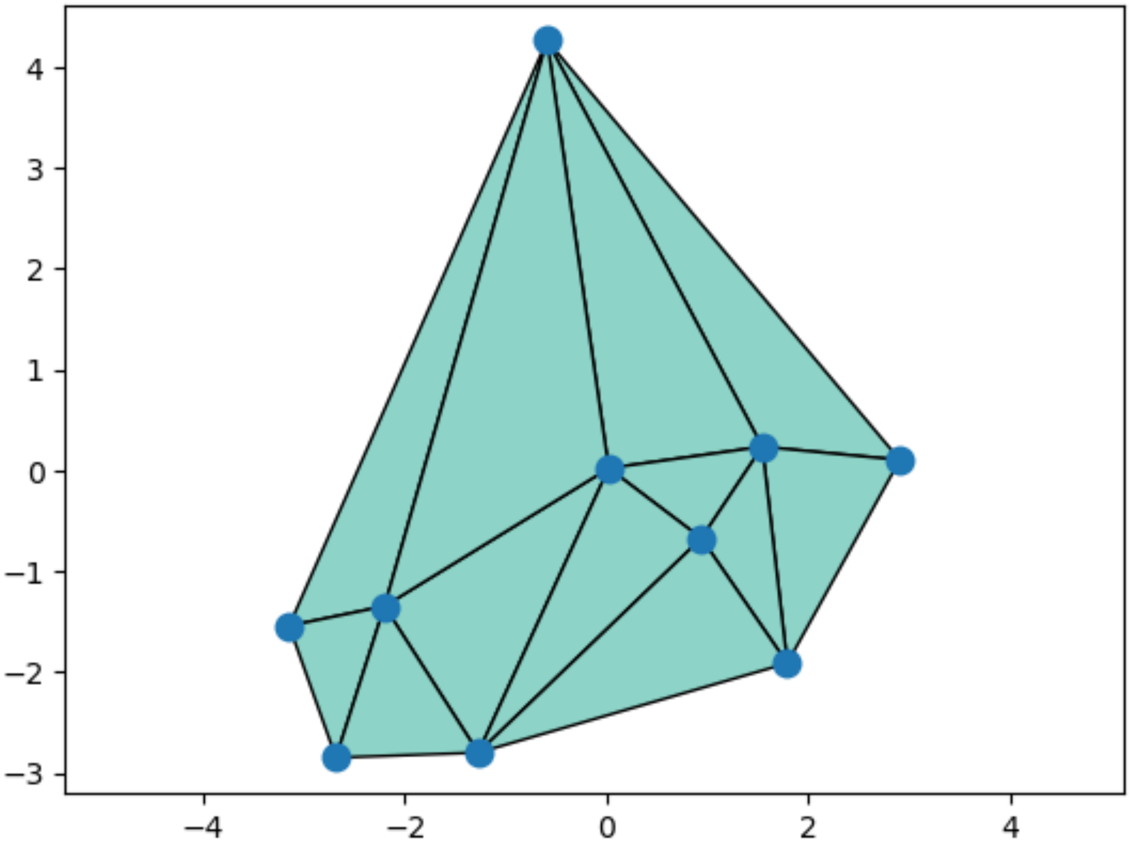
In [14]:

```julia
function tplot(p, t, u=nothing)
    clf()
    axis("equal")
    if u == nothing
        tripcolor(p[:,1], p[:,2], Array(t .- 1), 0*t[:,1],
                  cmap="Set3", edgecolors="k", linewidth=1)
    else
        tricontourf(p[:,1], p[:,2], Array(t .- 1), u[:], 20)
    end
    draw()
end
```

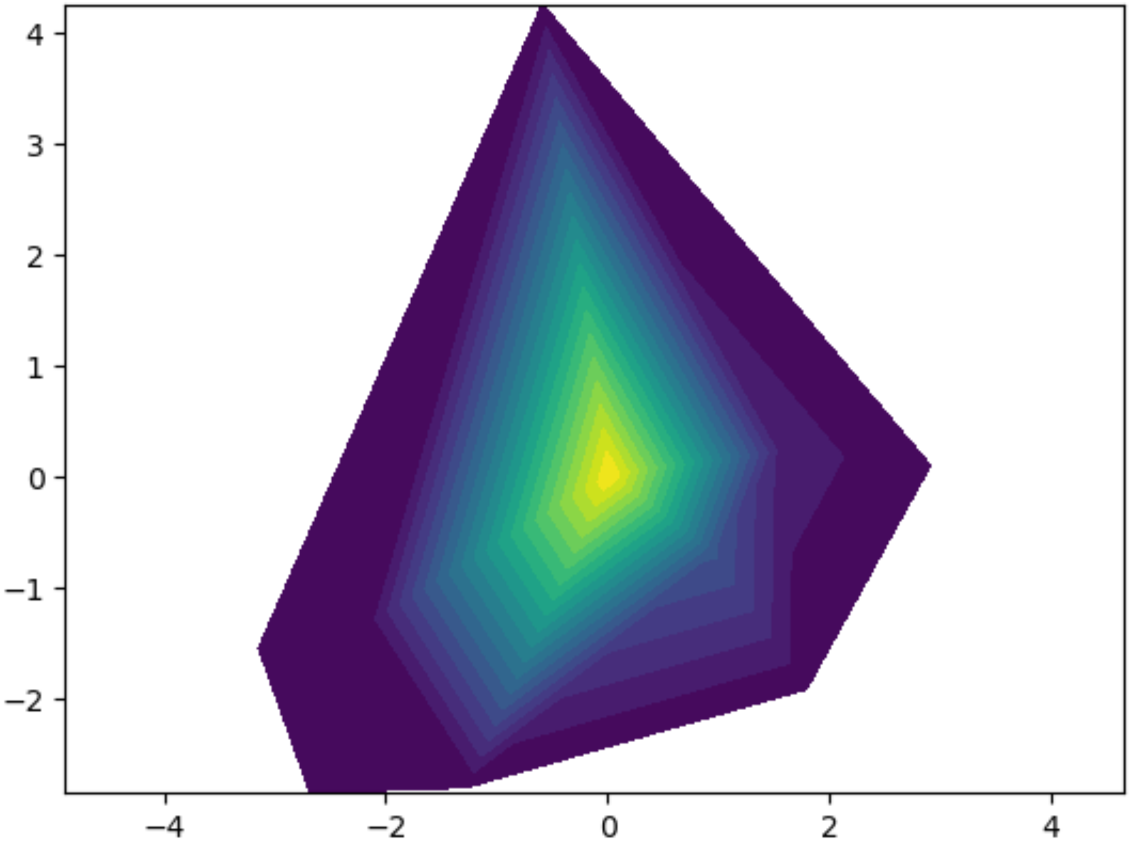
Out[14]:

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

In [15]:
```
p = randn(10,2)*2
t = delaunay(p)
tplot(p,t)
plot(p[:,1], p[:,2], ".", markersize=18), axis("equal");
```



In [16]:
```
### Example: Plot function on mesh as color contours
u = @. exp(-(p[:,1]^2 + p[:,2]^2))
tplot(p, t, u);
```



You might ask why not quadrilaterals? Some people do indeed prefer quadrilaterals (I'm one of them) but they can be very difficult to work with for certain applications. They are very widely used still however, for example in films (Pixar, ILM, ...) quads are very heavily used.

If you're interested - read about something called subdivision surfaces/NURBS.

In [ ]:
```
```