Hexagon Construction

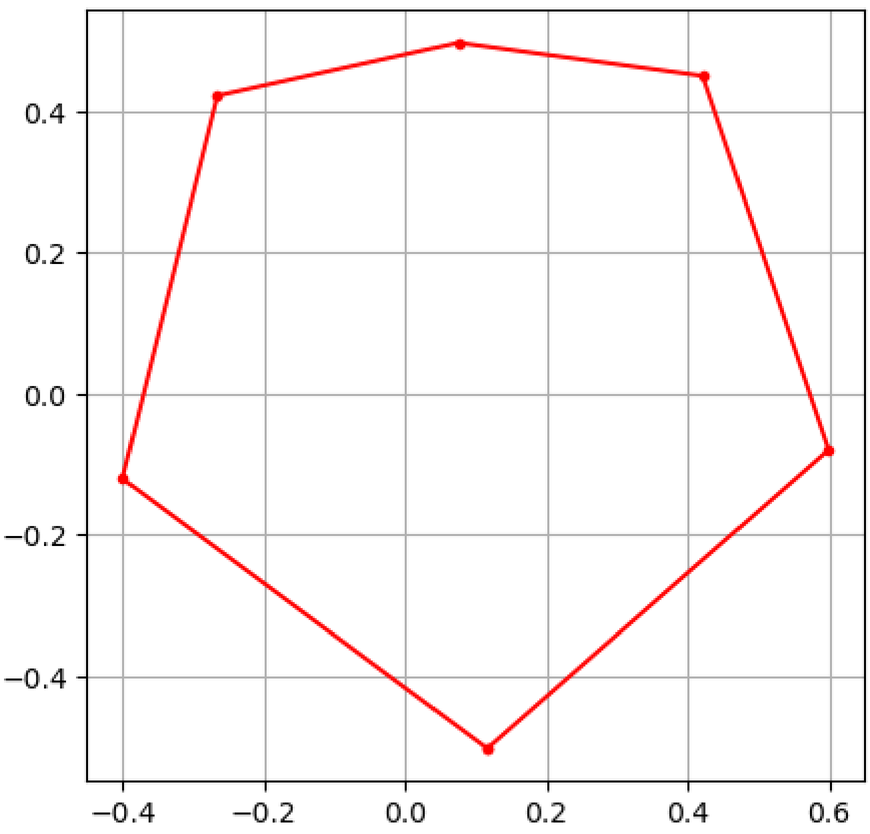
To the end of restricting the hexagons diameter to no greater than one, we first note that the diameter will be equal to the distance of the two verticies that are of greatest distance from each other, assuming a convex hexagon. This can be proven with some vector calculus by first assuming that the diameter is found between two points, both of which are not verticies. Then we derive a contradiction by showing that some vertex as a parameterized vector equation has a norm which is that of the candidate diameters norm plus some strictly positive value. Then we note that the diameter must have some vertex as an endpoint. So now we find the two verticies with the greatest distance between them. We trace out the circle described by fixing an endpoint of the candidate diameter and rotating it 2π radians. We see that the entirety of the hexagon is either inside the circle or touching it. So the candidate diameter is in fact the diameter.

In [102]:
```julia
using JuMP, Ipopt
n = 6
m = Model(solver = IpoptSolver(print_level=0))

@variable(m, x[1:n] )
@variable(m, y[1:n] )

@NLobjective(m, Max, 0.5*sum( x[i]*y[i+1]-y[i]*x[i+1] for i=1:n-1)  +  0.5*(x[n]*y[1]-y[n]*x[1]) )

#enforce the diameter constraint
for i = 1:n
    for j = i+1:n
            @NLconstraint(m, sqrt((x[j] - x[i])^2 + (y[j] - y[i])^2) <= 1)
    end
end

# add ordering constraint to the vertices
for i = 1:n-1
    @NLconstraint(m, x[i]*y[i+1]-y[i]*x[i+1] >= 0 )
end
@NLconstraint(m, x[n]*y[1]-y[n]*x[1] >= 0 )

srand(0)
setvalue(x,rand(n))
setvalue(y,rand(n))

status = solve(m)
println(status)
println("Optimal area: ", getobjectivevalue(m))
getvalue([x y]);
```

```
Optimal
Optimal area: 0.674981441395265
```

Here we plot the optimal hexagon. The note found here: http://www.math.ucsd.edu/~ronspubs/75_02_hexagon.pdf (http://www.math.ucsd.edu/~ronspubs/75_02_hexagon.pdf) gives assurance we have found the optimal hexagon.

In [103]:
```julia
using PyPlot
xopt = getvalue([x; x[1]])
yopt = getvalue([y; y[1]])
figure(figsize=[5,5])
plot( xopt, yopt, "r.-" )
axis("equal");
grid();
```



Fertilizer Influence Model

Here we use non-linear least squares to find the optimal coefficents for fitting the function to the given data.

In [90]:
```
using JuMP, Ipopt
```

In [91]:
```
m = Model(solver=IpoptSolver(print_level=0));
```

In [92]:
```
@variable(m, X[1:3]);
```

In [93]:
```
output  = [ 127, 151, 379, 421, 460, 426 ];
input   = [-5, -3, -1, 1, 3, 5];

srand()
#use the hint
setvalue(X[1], 500 + (rand()*10))
setvalue(X[2], -200 + (rand()*10))
setvalue(X[3], -1 + (rand()*10))
```

In [94]:
```
@NLobjective(m, Min, sum((output[i] -  (X[1] + X[2]*exp(X[3]*input[i])))^2 for i = 1:6))
```

In [95]:
```
solve(m)
```

Out[95]:
```
:Optimal
```

In [96]:
```
A = getvalue(X)
```

Out[96]:
```
3-element Array{Float64,1}:
  523.305
 -156.948
   -0.199665
```

In [97]:
```
using PyPlot
```

In [98]:
```
figure(figsize=(10,4))
```

Out[98]:
```
PyPlot.Figure(PyObject <matplotlib.figure.Figure object at 0x7f488da684d0>)
```

In [99]:
```
grid()
plot(input, (A[1] + A[2]*exp(A[3]*input)),"r")
scatter(input,output);
```