Assignment 1

February 6, 2017

1 Assignment 1

Name: Julien Neves
Name: Matthieu Ranger
Date: February 6, 2017

2 Problem 1

```
In [1]: using Plots

        x = linspace(-1, 1, 500)

        # Function
        y = 1 - exp(-2x)

        # First-order
        t1 = 2x

        # Second-order
        t2 = 2x - 2x.^2

        plot(x, [y t1 t2],
        label=["Original Function" "First-Order" "Second-Order"])
```

See the plot at the end

3 Problem 2

3.1 Exercise 2.1

The speed of the computations will be similar. However, some functions may work while others won't if the polynomial has values a, b or c leaving a zero term in the denominator.

The following function only returns real roots of degree 2 polynomial inputs.

```
In [2]: function findRoot(a, b, c)
            if a!=0 && b!=0
```

```
            posRoot = (-b + sqrt(b^2 - 4*a*c))/2*a
            negRoot = (-b - sqrt(b^2 - 4*a*c))/2*a
        elseif a !=0 && (4*a*c)/b^2 < 1
            posRoot = (-b/(2*a)) * (1+ sqrt((1-4*a*c)/b^2))
            negRoot = (-b/(2*a)) * (1- sqrt((1-4*a*c)/b^2))
        end
        return [posRoot, negRoot]
    end
```

Out[2]: findRoot (generic function with 1 method)

3.2 Exercise 2.2

In [3]:
```
A = [54 14 11 2;
     14 50 4 29;
     11 4 55 22;
     2 29 22 95]

b = [1,1,1,1]

# This returns the solution to Ax=b
x = \(A, b)
print(x)
```

```
[0.0120453,0.0138627,0.0136088,0.00288944]
```

In [4]:
```
using IterativeSolvers

b = [1.0, 1.0, 1.0, 1.0]

# Solves Ax=b using Gauss-Jacobi method
x, result = jacobi(A, b, tol=0.0001)

print("results are: ", x)
result
```

```
results are: [0.0120329,0.0138428,0.013594,0.0028755]
```

Out[4]: IterativeSolvers.ConvergenceHistory{Float64,Array{Float64,1}}(false,0.0002,

In [5]:
```
b = [1.0, 1.0, 1.0, 1.0]

# Solves Ax=b using Gauss-Seidel method
x, result = gauss_seidel(A, b, tol=0.0001)

print("results are: ", x)
result
```

```
results are: [0.0120419,0.0138654,0.0136108,0.00288822]
```

```
Out[5]: IterativeSolvers.ConvergenceHistory{Float64,Array{Float64,1}}(true,0.0002,6
```

We can see from the results above, the Gauss-Jacobi method took 16 iterations, and the Gauss-Seidel method took 6.

4 Problem 3

For problem 3, we start by writing an iterative function where $x_{k+1} = g(x_k, b)$

```
In [6]: function iterate(g::Function, x0, b, tol=1e-4, maxit=1000)
            x = x0
            for i in 1:maxit
                x_old = copy(x)
                x = g(x, b)

                # Stop algorithm if absolute or relative tolerance are smaller than
                if abs(x - x_old) < tol || abs(1 - x_old/x) < tol
                    break
                end

                # Stop algorithm if we reach max iteration without convergence
                if i == maxit
                    x = "No convergence"
                end
            end
            return(x)
        end
```

```
Out[6]: iterate (generic function with 3 methods)
```

Then we use $g(x, b) = 1 - b + bx$

```
In [7]: g(x, b) = 1 - (b) + (b)*x

        print("input -2 fixed point at: ", iterate(g, 0.5, -2), "\n")
        print("input 0 fixed point at: ", iterate(g, 0.5, 0), "\n")
        print("input 1 fixed point at: ", iterate(g, 0.5, 1), "\n")
        print("input 2 fixed point at: ", iterate(g, 0.5, 2), "\n")

input -2 fixed point at: No convergence
input 0 fixed point at: 1.0
input 1 fixed point at: 0.5
input 2 fixed point at: No convergence
```

Note that for -2 and 2 there's no convergence. Therefore, for $g(x, b)$ with $b = 2, -2$, there's no fixed point.

5 Problem 4

```
In [8]: function newton(f::Function, df::Function , x0, tol = 1e-4 ,maxit = 10000)

            # Give error if inital guess is negative
            if x0 <= 0
                error("Need positive inital guess")
            end

            x = x0
            for i in 1:maxit

                x_old = copy(x)
                x = x - f(x) / df(x)

                # If new guess x is negative reduce step by half until new guess is
                while x <= 0
                    x = x_old + (x - x_old)/2
                end

                # Stop algorithm if absolute or relative tolerance are smaller than
                if abs(x - x_old) < tol || abs(1 - x_old/x) < tol
                    break
                end

                # Stop algorithm if we reach max iteration without convergence
                if i == maxit
                    x = "Reached maximum iteration"
                end
            end
            return(x)
        end
```

```
Out[8]: newton (generic function with 3 methods)
```

We try our algorithm for $f(x) = log(x)$ and starting value equal to 1000

```
In [9]: f(x) = log(x)
        df(x) = 1/x

        newton(f, df , 1000)
```

```
Out[9]: 1.0
```

Note that even if the starting guess is rather bad, we still converge to the right value of 1

6 Problem 5

6.1 Excercises 3.3

In this problem note that the implied volatility solve the problem $V(\sigma) = \bar{V}$, where $V(\sigma)$ is value function for some σ and $\bar{V}$ is some specified value of our option. Therefore, if we write the problem in the following way, $F(\sigma) = V(\sigma) - \bar{V} = 0$, we can solve it using Newton's method.

```
In [10]: using Distributions

         # Set value function
         function BSVal(S,K,tau,r,delta,sigma)
             d = (log(exp(-delta*tau)*S)-log(exp(-r*tau)*K))/(sigma*sqrt(tau))
             + 0.5*sigma*sqrt(tau)
             V = exp(-delta*tau)*S*cdf(Normal(), d)
             - exp(-r*tau)*K*cdf(Normal(), d-sigma*sqrt(tau))
             return(V)
         end

         # Set implied volatility function
         function ImpVol(S,K,tau,r,delta, V, tol = 1e-3, maxit = 1000)

             # Set starting value for sigma
             sigma = 1

             # Set function equal to the derivative of the value function
             function dBSVal(S,K,tau,r,delta,sigma)
                 d = (log(exp(-delta*tau)*S)-log(exp(-r*tau)*K))/(sigma*sqrt(tau))
                 + 0.5*sigma*sqrt(tau)
                 dV = S*exp(-delta*tau)*sqrt(tau/(2*pi))*exp(-0.5*d^2)
             end

             # Use Newton's method to solve for sigma
             for i in 1:maxit
                 ImpV = BSVal(S,K,tau,r,delta,sigma)
                 dImpV = dBSVal(S,K,tau,r,delta,sigma)

                 sigma = sigma - (ImpV - V)/dImpV

                 # Stop algorithm if absolute or relative tolerance are smaller tha
                 if abs(V - ImpV) < tol || abs(1 - ImpV/V) < tol
                     break
                 end

                 # Stop algorithm if we reach max iteration without convergence
                 if i == maxit
                     sigma = "No convergence"
                 end
             end
```

```
        return(sigma)
    end

    ImpVol(1, 1.1, 1, 0.08, 0, 0.0728)
```

```
Out[10]: 0.20002397692365345
```

Thus for $V = 0.0728$, we have $\sigma = 0.20$.

6.2 Excercises 3.5

For this exercise, we used MATLAB and the CompEcon toolbox with the code:

```
function [fval, fjac] = fun(x)
    f1 = 200*x(1)*(x(2) - x(1)^2)-x(1)+1;
    f2 = 100*(x(1)^2 - x(2));

    fval = [f1; f2];

    df11 = 200*x(2) - 600*x(1)^2 - 1;
    df12 = 200*x(1);
    df21 = 200*x(1);
    df22 = -100;

    fjac = [df11 df12; df21 df22];
end

% Netwon
newton('fun',[0;2])
% Broyden
broyden('fun',[0;2])
```

The code gives the following output.

```
ans =

    1.0000
    1.0000


ans =

    1.0000
    1.0000
```

Hence, we have $x_1 = x_2 = 1$ as a solution in both cases.

6.3 Excercises 3.7

First note that for a CDF $F(x) \to U$, where $U \in [0, 1]$ and its inverse $F^{-1}(U) = x$, the problem can be recast as a root finding problem where $F(x) - U = 0$, where given a particular U, we want to find the corresponding x. This can be done using the Newton function written in problem 4.

```
In [11]: using Distributions


         function icdf(p::Float64, F::Function, x0=1.0, args...)
             # p::float [0,1] is a probability
             # F is a function (CDF)
             # x0 is an initial guess
             # F(x) should be formed as returning a tuple
             #       (CDF(x), pdf(x))
             # args... are args for F
             f(x) = F(x, args...)[1] - p
             fPrime(x) = F(x, args...)[2]
             result = newton(f, fPrime, x0)
             return result
         end

         function cdfnormal(x,mu,sigma)
             dist = Normal(mu, sigma)
             z=(x-mu)./sigma;
             F=cdf(dist,x);
             f=exp(-0.5*z.^2)./(sqrt(2*pi)*sigma)
             return F, f
         end

         test = 5

         testResult = test-icdf(cdfnormal(test,0,1)[1], cdfnormal, 3, 0, 1)

Out[11]: 4.966773659020873e-10
```

$4.97 * 10^{-10}$ is indeed close to 0