Part 1 Newton's method Matlab implementation

The following code is my implementation of Newton's method for approximating the solution of a system of two equations in Matlab. The mathematical logic behind this code is based off of Newton's method as it was taught in class, making sure to not invert the Jacobian matrix.

Algorithm 1: newton's method code

```
function x = newton(F, maxiter, acc, x)
    syms x1 x2;
    J = jacobian(F, [x1,x2]);
    i = 1;
    while i < maxiter
        Ji = double(subs(J, {x1, x2}, {x(1), x(2)}));
        Fi = double(-1*subs(F, {x1, x2}, {x(1), x(2)})).';
        y = Ji \ Fi;
        x = x+y;
        err = norm(y,2);
        if err < acc  break; end
        i = i + 1;
    end
end
```

This function takes in four arguments. The "F" argument is a row vector containing the system of equations of interest. The "maxiter" argument is an integer that determines how many times the while loop will run. The "acc" argument allows the function to exit the while loop if a desired level of accuracy between the approximated solution and the real solution. Finally, the "x" argument is the initial guess that Newton's method starts with. If the system has more than one solution, this algorithm will find the solution closest to the initial guess, because the algorithm only converges toward one solution at a time.

Part 2 Testing Newton's method, quadratic convergence

Since Newton's method approximates the solution of a set of equations of interest, it makes sense to test the code against an existing Matlab function that also computes the solution of a set of equations. I compared the output of my Newton's method implementation to the output from the Matlab function fsolve, which is designed to solve a system of nonlinear equations. We expect to see agreement between the approximated solutions from my code and from fsolve if my code has been written properly. Additionally, I augmented the above newton's method code in order to keep track of the norm of the iteration update y, and then to plot these values on a logarithmic scale in order to see the convergence trend. Below is the updated Newton's method code.

Algorithm 2: updated Newton's method code

```
function x = fig_newton(F, max_iter, acc)
    syms x1 x2;
    x = zeros(2,1);
    J = jacobian(F, [x1,x2]);
    i = 1;
    errors = [];
    while i < max_iter
        Ji = double(subs(J, {x1, x2}, {x(1), x(2)}));
        Fi = double(-1*subs(F, {x1, x2}, {x(1), x(2)})).';
        y = Ji \ Fi;
        x = x+y;
        err = norm(y,2);
        errors = [errors err];
        if err < acc  break; end
        i = i + 1;
    end
    if i >= max_iter
        disp("max iterations reached")
    end

    semilogy(errors)
    title(string(F))
    xlabel("iteration")
    ylabel("error")
    grid on
end
```

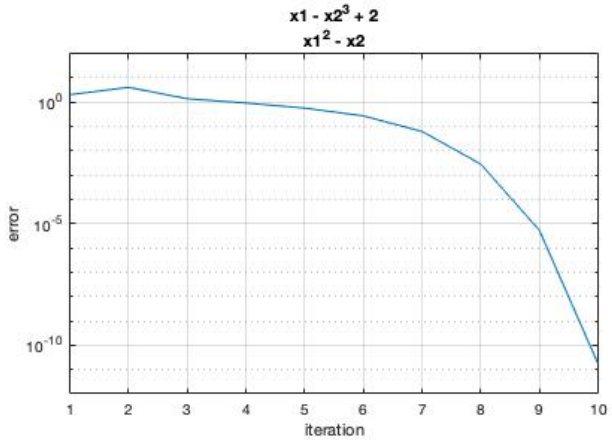
Now we define two systems of equations with which to test my implementation of newton's method, compare them to Matlab's fsolve, and observe the rate of convergence graphically.

system 1:

$$0 = x_1 - x_2^3 + 2$$
$$0 = x_1^2 - x_2$$

```
function F = somefunc(x)
    F(1) = x(1)-x(2)^3 + 2;
    F(2) = x(1)^2 - x(2);
end
```

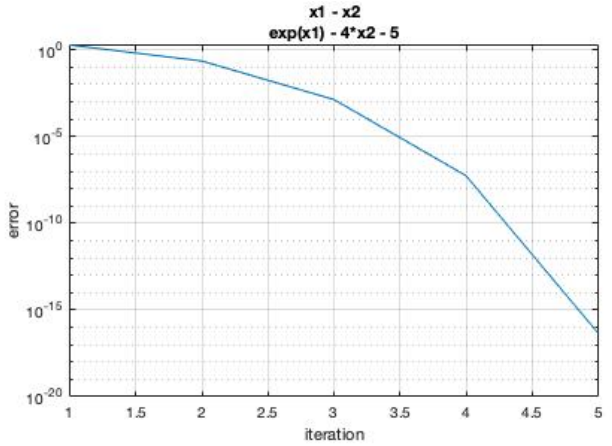
```
>> F1 = [x1-x2^3 + 2, x1^2 - x2];
>> fig_newton(F1, 100, 1e-8)

ans =
    -1
     1

>> fsolve(@somefunc, [0,0])

ans =
   -1.0000    1.0000
```



system 2:

$$0 = x_1 - x_2$$
$$0 = e^{x_1} - 4x_2 - 5$$

```
function F = somefunc(x)
    F(1) = x(1)-x(2);
    F(2) = exp(x(1)) - 4*x(2)-5;
end

>> F1 = [x1-x2, exp(x1) - 4*x2 - 5];
>> fig_newton(F1, 100, 1e-8)
ans =
   -1.1726
   -1.1726

>> fsolve(@somefunc, [0,0])
ans =
   -1.1726    -1.1726
```



For both systems, the output of my Newton's method code and Matlab's fsolve are equivalent, showing that my Newton's method is behaving correctly. Additionally, the plots showing error at each iteration shows that this algorithm has quadratic convergence for the systems that I chose.

Part 3 Two-gene network with cross activation

The following is Matlab code used to test my implementation on Newton's method on the two gene network of interest.

Algorithm 3: two gene network

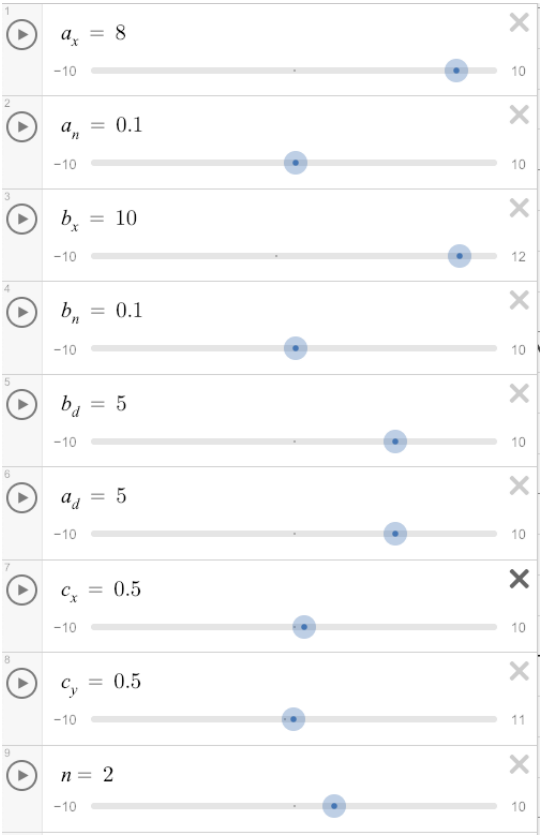
```
function x = twoGene
    aMin = 0.1;
    aMax = 8;
    aDeg = 5;
    bMin = 0.1;
    bMax = 10;
    bDeg = 5;
    eX = 0.5;
    eY = 0.5;
    n = 2;

    f1 = aMin - x1*aDeg + (aMax - aMin) * (x2^n)/(eY^n + x2^n);
    f2 = bMin - x2*bDeg + (bMax - bMin) * (x1^n)/(eX^n + x1^n);
    F = [f1, f2];

    x = fig_newton(F,100,1e-5);
end
```

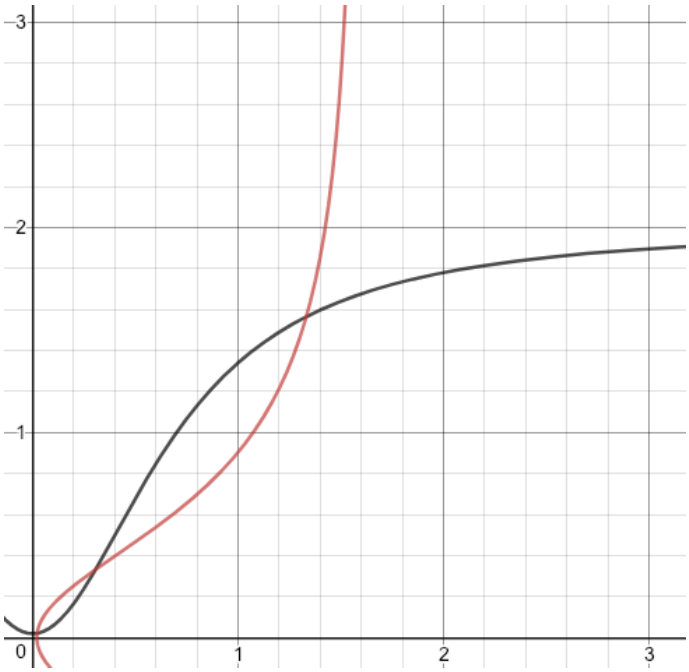
Next, we will run this code and once again compare my Newton's method output to the output of Matlab's fsolve on the same system. Running this code, we see the following output:

```
function F = myfunc(x)
    aMin = 0.1;
    aMax = 8;
    aDeg = 5;
    bMin = 0.1;
    bMax = 10;
    bDeg = 5;
    eX = 0.5;
    eY = 0.5;
    n = 2;

    F(1) = aMin - x(1)*aDeg + (aMax - aMin) * (x(2)^n)/(eY^n + x(2)^n);
    F(2) = bMin - x(2)*bDeg + (bMax - bMin) * (x(1)^n)/(eX^n + x(1)^n);
end

>> twoGene()
ans =
    0.0238
    0.0245

>> fsolve(@myfunc, [0,0])
ans =
    0.0238    0.0245
```

Once again, we see that the output from my Newton's method function and the output of fsolve for this system are identical, ensuring that my code correctly approximates the solution of this system.

We now aim to search for a parameter regime in this system that offers bistability. First, I manually looked for parameters that caused at least two solutions using the online graphing tool Desmos (see below figures). The following figures are the parameters that I used to obtain a bistable system, as well as the nullclines plotted together.





In order to determine stability of each fixed point, the fixed points were plugged into the Jacobian, and

then the trace and determinant were computed from the resulting matrices. I augmented my twoGene() function (above) for these additional computations. Additionally, I changed my function in order to take in a different starting point for the Newton's method algorithm. Previously, my implementation of Newton's method always began with 0,0 as the initial guess. However, this will always converge towards only one solution, the one that is closest to 0,0. We observe in the Desmos plot that one Solution is closer 1,1, so I will use that as a starting value in order to get the algorithm to converge towards the other fixed point. We once again compare the output of my implementation to the output of fsolve in order to test whether or not we obtain the expected output.

Algorithm 4: updated twoGene function

```
function x = twoGene(start)
    syms x1 x2;
    aMin = 0.1;
    aMax = 8;
    aDeg = 5;
    bMin = 0.1;
    bMax = 10;
    bDeg = 5;
    eX = 0.5;
    eY = 0.5;
    n = 2;

    f1 = aMin - x1*aDeg + (aMax - aMin) * (x2^n)/(eY^n + x2^n);
    f2 = bMin - x2*bDeg + (bMax - bMin) * (x1^n)/(eX^n + x1^n);
    F = [f1, f2];

    x = fig_newton(F,100,1e-5,start);
    J = jacobian(F, [x1,x2]);

    %evaluate the jacobian at the fixed point to determine stability:
    J_fp = double(subs(J, {x1,x2}, x.'));
    Trace = trace(J_fp)
    Det = det(J_fp)
end
```

```
>> twoGene([1;1])

Trace =
   -10

Det =
   24.2890

ans =
    1.4867
    1.7988

>> fsolve(@myfunc, [1,1])
ans =
    1.4867    1.7988

>> twoGene([0;0])
```

```
Trace =
   -10

Det =
   22.1155

ans =
    0.0238
    0.0245

>> fsolve(@myfunc, [0,0])
ans =
    0.0238    0.0245
```

My implementation of Newton's method found two solutions to this system by using two different starting points for the algorithm. For each solution, the trace and determinant of the Jacobian evaluated at the fixed point were computed. In both cases, the trace was negative, and the determinant was positive, indicating a stable solution, and thus, proving that this parameter regime allows bistability of the system.