**BME 313L: Introduction to Numerical Methods in Biomedical Engineering**

**Lab Report**

**Lab #6: Iterative Methods**

**Last name, First name: Liu, Vincent**

**EID: VL5649**

**Lab Section: 14035 (Tuesday 9:30-12:30)**

**Problem 1: Gauss-Seidel**

Consider the problem wherein a drug is being extracted from an organic phase into two aqueous phases. The volumetric flow rates of the two organic phases are denoted by *H* and *K*; and the volumetric flow rates of the two aqueous phases are denoted by *L* and *M*. Four different experiments are run by varying *H*, *K*, *L*, and *M* simultaneously, as tabulated below, in order to achieve the same exit concentration of the drug, *y*, *w*, *x*, and *z*, respectively. The total mass of solute entering each experiment is given by *F*. Estimate the exit concentrations of the drug in the four liquid streams.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Trial | *H* (in L/min) | *K* (in L/min) | *L* (in L/min) | *M* (in L/min) | *F*  (Total mass in g) |
| 1 | 100 | 125 | 125 | 62.5 | 6625 |
| 2 | 80 | 110 | 120 | 25 | 5290 |
| 3 | 140 | 80 | 120 | 100 | 7300 |
| 4 | 90 | 104.8 | 60 | 137.33 | 6539 |

The governing equation for mass distribution of the drug between the four liquid streams is shown below in terms of the total solute mass entering each experiment.

*Hy* + *Kw* + *Lx* + *Mz* = *F*

Solve this problem by implementing the Gauss-Seidel method. The use of the function GaussSeidel.m is not permitted. Consider the initial values as: *y* = 2, *w* = 7, *x* = 30, *z* = 30 and the approximate percentage error criterion as es = 8%. Report the following: (1) the equations used for updating the parameters using Gauss-Seidel method, and (2) print a table with 5 columns (iteration number and corresponding values of *y, w, x* and *z* at the end of that iteration) in the following format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Iteration | y | w | x | z |
| 1 | 1.250 | 7.636 | 29.284 | 28.174 |
| 2 | 2.491 | 7.930 | 29.162 | 27.190 |
| 3 | 2.890 | 7.996 | 29.472 | 26.743 |
| 4 | 2.701 | 7.897 | 30.132 | 26.654 |

**Things to discuss**: What is the property of Gauss-Seidel? (Pros and Cons)

**MATLAB code:**

clear all; close all; clc; %resets everything

A = [100 125 125 62.5;80 110 120 25;140 80 120 100;90 104.8 60 137.33]; %coefficient matrix

b = [6625;5290;7300;6539]; %right hand vector

n = length(A); %sets length, number can be used to designate end of matrix

ea = 1; %intialize error

nn = 0; %initializes counter

xi = [2;7;30;30]; %initial values

C = A; %duplicates matrix

while max(ea) > .08

nn = nn + 1; %updates counter

iter(nn) = nn; %vector for counter

for i = 1:n

C(i,i) = 0; %makes diagonals 0

end

for i = 1:n

d = C(i,:); %works by row

oldxi(i) = xi(i); %save old xi

xi(i) = (b(i) - d\*xi)/A(i,i); %Gauss-Seidal Method (updates as it goes)

ea(i) = abs((xi(i) - oldxi(i))/xi(i)); %calculates error

end

y(nn) = xi(1); %saves values for each iteration

w(nn) = xi(2);

xx(nn) = xi(3);

z(nn) = xi(4);

end

T = [iter;y;w;xx;z]; %matrix for table

fprintf('Iteration\t y\t\t w\t\t x\t\t z\n') %table formatting

fprintf('\t%d\t\t%.3f\t%.3f\t%.3f\t%.3f\n',T)

**MATLAB function:**

The purpose of this question was to model a hypothetical scenario in which we could gather some experimental data and tabulate it to work backwards and solve for the unknowns (concentrations of the drug in various phases in this case). To do so, we used an iterative method (Gauss-Seidal) to find the solutions to a system of equations given to us as part of the problem. To implement this, we could set this up as a loop within a loop so that it would continually update guesses for each variable and evaluate our guesses for each iteration. By setting up arrays along the way, we can then output values for every step along the way into a table.

clear all; close all; clc; %resets everything

This first line of code ensures that previous scripts and functions do not affect this script.

A = [100 125 125 62.5;80 110 120 25;140 80 120 100;90 104.8 60 137.33]; %coefficient matrix

b = [6625;5290;7300;6539]; %right hand vector

These 2 lines of code are the coefficient matrix and right hand vector given to us as part of the problem. The values were derived as part of the tabulated data.

n = length(A); %sets length, number can be used to designate end of matrix

This line of code creates a variable ‘n’ with the length of the coefficient matrix. This is useful/necessary because many of the operations later on will need to be repeated along the length of the matrix.

ea = 1; %intialize error

nn = 0; %initializes counter

These 2 lines of code initialize the error value and counter, respectively. By initializing the error value, we eliminate the possibility that the loop later (with error as a conditional) terminates preemptively, and by initializing the counter, we ensure that the counter starts at the correct value.

xi = [2;7;30;30]; %initial values

This line of code is a vector corresponding to the initial values or guesses for y, w, x, and z, given to us as part of the problem. By having initial values, it drastically decreases the number of iterations necessary in order to reach the desired error approximation.

C = A; %duplicates matrix

This line of code duplicates the original coefficient matrix. This makes it much easier to implement the Gauss Seidal method later on because you can remove a value from the matrix and still be able to work with it from the copy.

while max(ea) > .08

This line of code is a conditional while loop that repeats all of our Gauss Seidal calculations and stores them while the error (ea) is still below the desired threshold. Because there are 4 variables that we are calculating for that will be later stored into a vector, we want to make sure that all of the values are below the error threshold that we set, so we use the max of the error vector.

nn = nn + 1; %updates counter

iter(nn) = nn; %vector for counter

These 2 lines of code update and store the counter for number of iterations. This is necessary in order to create the tables too properly output our results for each iteration.

for i = 1:n

C(i,i) = 0; %makes diagonals 0

end

These 3 lines of code make the diagonal of the copy of the coefficient matrix 0. This makes the calculation for the Gauss Seidal method much easier later on because the variable corresponding to the variable being calculated is not used as part of the calculation. However, rather than making this variable 0 and having to reset it for every calculation, we can just set the diagonal (which is the value multiplied by the variable) so that the variable is omitted in calculations.

for i = 1:n

This line of code specifies that we work with values starting from the first going all the way to the variable defined earlier, the length of our matrix. This is useful because we know we will have the same number of variables as we do equations, and by having operations under the same loop we can work with specific values for specific variables.

d = C(i,:); %works by row

This line of code sets the row i, as the variable, d. This makes it so that we can work with a specific row in the coefficient matrix to solve for a specific variable.

oldxi(i) = xi(i); %save old xi

This line of code stores the previous x values so that we can calculate the approximate error later on.

xi(i) = (b(i) - d\*xi)/A(i,i); %Gauss-Seidal Method (updates as it goes)

This line of code implements the Gauss-Seidal Method for each of the variables along the matrix. This can be implemented in one line of code because of the efforts spent earlier to make a copy of the coefficient matrix and to set the diagonals as 0.

ea(i) = abs((xi(i) - oldxi(i))/xi(i)); %calculates error

This line of code calculates the approximate error using the previous value and the current value of x. Because there are 4 variables, it makes a vector with all 4 of the error approximations for each variable.

end

This line of code closes the for loop portion of the code, sectioning off each iteration.

y(nn) = xi(1); %saves values for each iteration

w(nn) = xi(2);

xx(nn) = xi(3);

z(nn) = xi(4);

These 4 lines of code store the values of y, w, x, and z, for each iteration, into separate vectors to be outputted into our table later.

end

This line of code closes the while loop portion of our code.

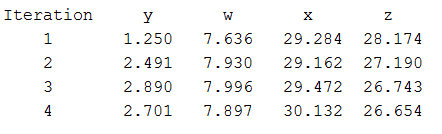
T = [iter;y;w;xx;z]; %matrix for table

fprintf('Iteration\t y\t\t w\t\t x\t\t z\n') %table formatting

fprintf('\t%d\t\t%.3f\t%.3f\t%.3f\t%.3f\n',T)

These last 3 lines of code make all of our vectors created earlier into a singular matrix and then output said matrix in table form.

**Results:**



**Discussion:**

As shown by the results, with an initial guess and 4 iterations of the Gauss Seidal Method, we can reach an maximum approximate error between all the terms of below 8%. With each successive iteration, the distance between approximations seems to decrease (which could be gauged if we outputted all of the corresponding approximate error values). By constantly updating the values used in calculations, the Gauss Seidal Method is constantly using the “best” or closest approximation to the value available, an advantage over other methods such as the jacobi iteration (albeit it might be slightly slower). Additionally, as an interative method, in the event that the script terminates early, there is still some approximation of the values corresponding to the unknowns.

From this problem, we learned how to use iterative methods in order to solve linear systems of equations. We reviewed implementing conditional statements in order to efficiently iterate calculations until a condition was met (error in this case). Furthermore, we reviewed implementing arrays without fixed values and general vector/matrix manipulation. Lastly, we refreshed our knowledge on how to format and output arrays as a table using the fprintf function in MATLAB.

**Problem 2: Successive Substitution**

Determine the solution of the following simultaneous nonlinear equations using the successive substitution method using initial guesses of x = y = 1.2 and approximate percentage error criterion es = 0.001%.



Report and comment on all the different equation posings/formats tried before obtaining the representation format that converges to a solution. As output from the converging format, print out a table with 5 columns (iteration number, x value at the end of iteration, y value at the end of iteration, approximate percentage error in X and approximate percentage error in Y) in the following format:

Iteration X Y Ea\_X Ea\_Y

1 2.14000 -8.26040 -6.32080 -101.22628

2 -4.18080 -155.19631 -134.03642 3071.54832

3 -138.21722 -88150.01691 -68908.29859 60811999.28828

You need to submit a script file that implements the successive substitution method.

**Things to discuss:** Report and comment on all the different equation posings/formats tried before obtaining the representation format that converges to a solution.

**MATLAB code:**

clear all; close all; clc; %resets everything

ex = 1; %initialize errors

ey = 1;

c = 0; %initialize counter

x = 1.2; %initial guesses

y = 1.2;

while ex > .00001 | ey > .00001

c = c + 1; %updates counter

iter(c) = c; %stores counter vector

yold = y; %temporarily stores last values

xold = x;

y = (x^2 - y)/(5\*x); %converges on 10 iter

x = sqrt(x - y + .5); %converges on 10 iter

xx(c) = x; %stores calculated solutions

yy(c) = y;

ex = abs((xx(c)-xold)/xx(c)); %calculates errors

ey = abs((yy(c)-yold)/yy(c));

eax(c) = ex \* 100; %stores errors

eay(c) = ey \* 100;

end

A = [iter;xx;yy;eax;eay]; %matrix for table

fprintf('Iteration\t X\t\t Y\t Ea\_X\t\t Ea\_Y\n') %table formatting

fprintf('\t%d\t %.5f\t%.5f\t %.5f\t%.5f\n',A)

**MATLAB function:**

The purpose of this function was to incorporate an interative into a system of nonlinear equations. Using the same strategy as that of the Gauss-Seidal method, we can solve for each of the unknowns and use them to compute successive values in a method known as successive substitution. To implement this, we have to solve for each of the unknowns using the equations, and we can iteratively reinput values until an error criterion is met (using a conditional while loop). By setting up arrays along the way, we can then output values for every step along the way into a table.

clear all; close all; clc; %resets everything

This first line of code ensures that previous scripts and functions do not affect this script.

ex = 1; %initialize errors

ey = 1;

These 2 lines of code initialize the error values for x and y. This ensure that the conditional while loop with the error criterion does not terminate because it interprets the initial error as 0,

c = 0; %initialize counter

x = 1.2; %initial guesses

y = 1.2;

These 3 lines of code initialize the counter and the initial guesses for x and y. By initializing the counter, we ensure that our iteration counter starts and stays at the correct value per iteration. The initial guesses of x and y provide values that can be plugged into the functions in order to generate successive values; otherwise, in some cases, some equations will continually just loop 0 if no initial guess is provided.

while ex > .00001 | ey > .00001

This line of code forms the while loop that continually iterates the successive substitution method until the error of both x and y falls below .00001 or .001%.

c = c + 1; %updates counter

iter(c) = c; %stores counter vector

These 2 lines of code update the counter and store it in an array, for each iteration of the while loop. By updating the counter each iteration, we can use the counter to store values for each iteration in arrays.

yold = y; %temporarily stores last values

xold = x;

These 2 lines of code temporarily store the x and y value before we update them so that we can calculate approximate error later on within the loop.

y = (x^2 - y)/(5\*x); %converges on 10 iter

x = sqrt(x - y + .5); %converges on 10 iter

These 2 lines of code update x and y using successive substitution method. The equations were derived by solving for a particular variable using the given nonlinear equations.

xx(c) = x; %stores calculated solutions

yy(c) = y;

These 2 lines of code store the updated x and y values for each iteration into arrays so that they can be outputted into the table at the end.

ex = abs((xx(c)-xold)/xx(c)); %calculates errors

ey = abs((yy(c)-yold)/yy(c));

eax(c) = ex \* 100; %stores errors1

eay(c) = ey \* 100;

These 4 lines of code calculate the approximate error for both the x and the y values, and then store them into arrays. Because I am working with the numerical rather than the percentage values of error, the values are multiplied by 100 to get the percentage before being stored into arrays.

end

This line of code closes the while loop.

A = [iter;xx;yy;eax;eay]; %matrix for table

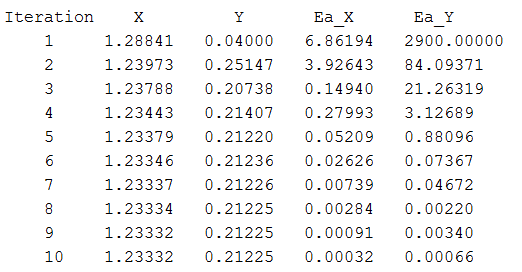
fprintf('Iteration\t X\t\t Y\t Ea\_X\t\t Ea\_Y\n') %table formatting

fprintf('\t%d\t %.5f\t%.5f\t %.5f\t%.5f\n',A)

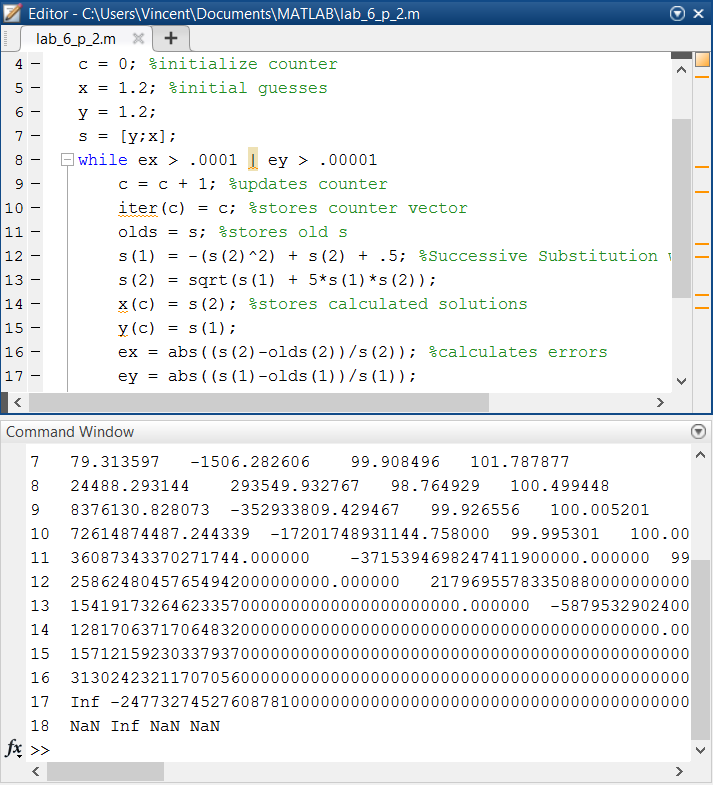
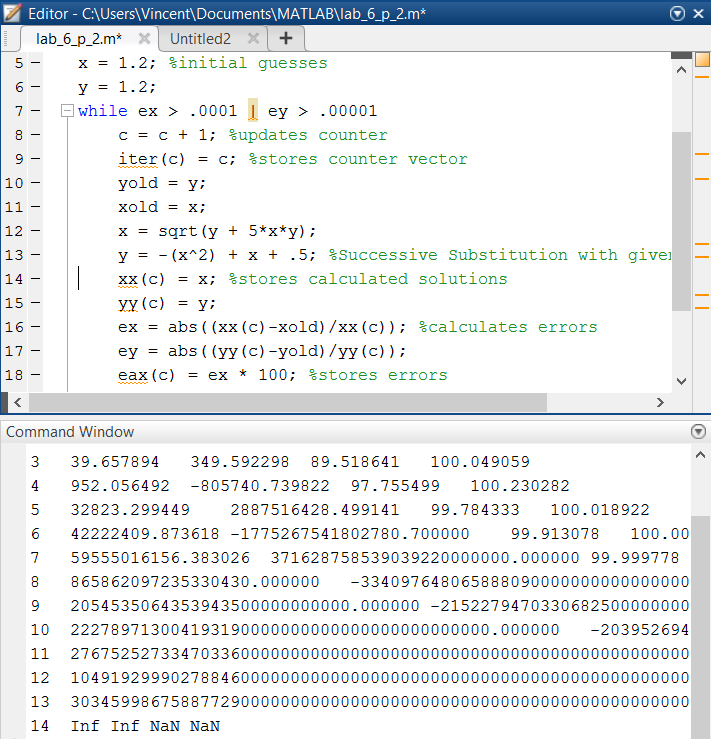
These last 3 lines of code generate a matrix using all of the arrays stored for each iteration and output it as a table.

**Results:**

(Actual Solution)



(Other attempts)

x = sqrt(x - y + .5);

y = (x^2 - y)/(5\*x);

%converges on 11 iterations (I forgot to screenshot)

**Discussion:**

As shown by the results, it is possible to solve for a nonlinear set of equations using the successive substitution method. By having initial guesses, solving for the equations in terms of variables, and successively plugging in ‘guesses’ for the solutions of the equation, we can approximate the solution very closely. This is not always the case, however. Depending on how the variables were solved for, the solution might converge very slowly, skipping around roots before finally arriving at the solution or possibly not converging at all (this can be seen by some of my other attempts at solving the problem)—this highlights the biggest flaw in the successive substitution method.

From this problem, we learned how to implement the successive substitution method in order to solve nonlinear systems of equations (which is different from our usual linear systems). We reviewed implementing conditional statements in order to efficiently iterate calculations until a condition was met (error in this case). Furthermore, we reviewed implementing arrays without fixed values using a counter. Lastly, we refreshed our knowledge on how to format and output arrays as a table using the fprintf function in MATLAB.

**Problem 3: Newton-Raphson**

Cell membrane receptors can bind to specific ligands, such as hormones and growth factors, and trigger intracellular signaling in mammalian cells (Lauffenburger and Linderman, 1993). Multimeric receptors can possess different units or epitopes that can bind to more than one ligand, thereby making the binding chemistry nonlinear. If we imagine a receptor that binds to three ligand molecules, and a second population of receptors that binds to two ligand molecules each, the following reversible binding reactions can be established:

Let *CA* be the concentration of trimeric receptor.

Let *CB* be the concentration of dimeric receptor.

Let *CL b*e the concentration of the ligand.

Let *CD* be the concentration of all receptor-bound ligands.



The initial concentrations of A, B, and L are given as: *CA,0* = 5000/cell; *CB,0* = 10000/cell; *CL,0* = 10-7 M (1 M = 1 mole/liter = 6.023 x 1023 molecules/liter. For 1 cell equivalent volume of 4 x 10-9 cm3, *CL,0* = 24000/cell). The values of equilibrium constants = 7.19x10-14 and =6x10-10 . Find the fractional occupancy (fraction of receptors occupied) *xA* and *xB* of the two receptors *A* and *B* respectively, at equilibrium.

Solve this problem by using the Newton-Raphson method (using the MATLAB function newtmult.m) and report: (1) the fractional occupancy values *xA* and *xB*, (2) the approximate percentage error at convergence, and (3) the number of iterations required for convergence. Besides your problem script file, you need to create and submit a MATLAB function ‘receptor\_ligand\_Initials.m’ that is required as input in the newtmult.m function (see help of newtmult.m function for more information). Use the initial guesses: *xA* = 0.5 and *xB* = 0.2, and approximate percentage error criterion as es = 1%. For calculating function derivatives, use the forward finite difference scheme with.

**MATLAB code:**

**Function:**

function [J,f]=receptor\_ligand\_func\_VL(x,varargin)

del=0.01; %specified change

df1dx1=(u(x(1)+del,x(2))-u(x(1),x(2)))/del; %partial forward difference approximations

df1dx2=(u(x(1),x(2)+del)-u(x(1),x(2)))/del;

df2dx1=(v(x(1)+del,x(2))-v(x(1),x(2)))/del;

df2dx2=(v(x(1),x(2)+del)-v(x(1),x(2)))/del;

J=[df1dx1 df1dx2;df2dx1 df2dx2]; %forms jacobian matrix

f1=u(x(1),x(2)); %function parameters

f2=v(x(1),x(2));

f=[f1;f2];

function f=u(x,y) %derived functions (solving for roots)

f =(5000\*x+10000\*y)/(((24000-15000\*x-20000\*y)^3)\*5000\*(1-x))-7.19e-14;

function f=v(x,y)

f =(5000\*x+10000\*y)/(((24000-15000\*x-20000\*y)^2)\*10000\*(1-y))-6e-10;

**Function:**

function [x,f,ea,iter]=newtmult\_VL(func,x0,es,maxit,varargin)

% newtmult: Newton-Raphson root zeroes nonlinear systems

% [x,f,ea,iter]=newtmult(func,x0,es,maxit,p1,p2,...):

% uses the Newton-Raphson method to find the roots of

% a system of nonlinear equations

% input:

% func = name of function that returns f and J

% x0 = initial guess

% es = desired percent relative error (default = 0.0001%)

% maxit = maximum allowable iterations (default = 50)

% p1,p2,... = additional parameters used by function

% output:

% x = vector of roots

% f = vector of functions evaluated at roots

% ea = approximate percent relative error (%)

% iter = number of iterations

if nargin<2,error('at least 2 input arguments required'),end

if nargin<3||isempty(es),es=0.0001;end %defaults error criterion if not included

if nargin<4||isempty(maxit),maxit=50;end %defaults max iteration if not included

iter = 0; %initializes counter

x=x0; %initializes guess

while (1)

[J,f]=func(x,varargin{:}); %plugs arguments into

dx=(J\f)';

x=x-dx; %calculates next x and automatically replaces it

iter = iter + 1; %updates counter

ea=100\*max(abs(dx./x)); %calculates error

if iter>=maxit||ea<=es, break, end %breaks on either reaching max iterations or if error criterion is reached

end

**Main script:**

clear all; close all; clc; %resets everything

[x,f,ea,iter] = newtmult\_VL(@receptor\_ligand\_func\_VL,[.5,.2],1,100); %calls receptor ligand function as an anonymous function into newmult

%initial guesses, error, and max iterations are built in.

fprintf('x\_a:\n%f\nx\_b:\n%f\n',x(1),x(2)) %output results

fprintf('approximate percent error at convergence:\n%f\n',ea)

fprintf('number of iterations for convergence:\n%d\n',iter)

**MATLAB function:**

The purpose of this problem was to model a situation in where we knew the initial concentrations and we were given a set of conditions for which would affect the equilibrium of the concentrations. In order to solve for this, we had to generate a system of equations that would model the equilibrium within the system. These functions would then need to be converted into a jacobian matrix so that we could solve for the unknowns. The fastest way that I could think of implementing these methods was using separate scripts that had different functions from each other. In the code that follows, the main scripts calls a function into another function: for the first function, a jacobian of the equilibrium equations is calculated and then passed into the second function, which takes jacbians and solves for the unknown variables.

del=0.01; %specified change

This line of code specifies the difference between steps in approximations for calculating derivatives and solving for the unknowns. This value was given to us as part of the problem.

df1dx1=(u(x(1)+del,x(2))-u(x(1),x(2)))/del; %partial forward difference approximations

df1dx2=(u(x(1),x(2)+del)-u(x(1),x(2)))/del;

df2dx1=(v(x(1)+del,x(2))-v(x(1),x(2)))/del;

df2dx2=(v(x(1),x(2)+del)-v(x(1),x(2)))/del;

J=[df1dx1 df1dx2;df2dx1 df2dx2]; %forms jacobian matrix

These 5 lines of code calculate the jacobian matrix as part of the Newton-Raphson multivariable (newtmult) iterative method of solving for variables. The first 4 lines calculate partial forward differences to approximate the partial derivatives of the functions. The 5th and last line then complies the ‘derivatives’ to form the jacobian matrix.

f1=u(x(1),x(2)); %function parameters

f2=v(x(1),x(2));

f=[f1;f2];

function f=u(x,y) %derived functions (solving for roots)

f =(5000\*x+10000\*y)/(((24000-15000\*x-20000\*y)^3)\*5000\*(1-x))-7.19e-14;

function f=v(x,y)

f =(5000\*x+10000\*y)/(((24000-15000\*x-20000\*y)^2)\*10000\*(1-y))-6e-10;

These 7 lines of code form the ‘function’ portion of the receptor\_ligand\_func file. The first 3 lines specify that the function is a function of 2 variables, x(1) and x(2) passed through 2 equations, u(x) and v(x). The 5th and 7th lines are the equations derived from the problem that take into account all of different factors and solving for will yield us our fractional occupancy of x(1) and x(2) which correspond to A and B.

iter = 0; %initializes counter

x=x0; %initializes guess

These first 2 lines of the newtmult function initialize the counter and the guess. The counter is set to 0 so that it can be updated for each iteration that follows. The initial guess is dependent on user input and is whatever value is called as part of the function.

while (1)

[J,f]=func(x,varargin{:}); %plugs arguments into

dx=(J\f)';

x=x-dx; %calculates next x and automatically replaces it

This line of code calculates the next x value and goes ahead and replaces it (the value is not stored as we do not need to output an array of values).

iter = iter + 1; %updates counter

This line of code keeps track of the number of iterations that have been run overall.

ea=100\*max(abs(dx./x)); %calculates error

This line of code calculates the approximate error of the current value.

if iter>=maxit||ea<=es, break, end %breaks on either reaching max iterations or if error criterion is reached

end

This last line of code in the newtmult function causes the infinite while loop to terminate upon falling below the desired error amount or reaching the maximum number of iterations.

clear all; close all; clc; %resets everything

This first line of code in the main script ensures that previous scripts and functions do not affect this script.

[x,f,ea,iter] = newtmult\_VL(@receptor\_ligand\_func\_VL,[.5,.2],1,100); %calls receptor ligand function as an anonymous function into newmult

%initial guesses, error, and max iterations are built in.

This line of code calls the ‘receptor\_lingand\_func\_VL’ function and passes it as an anonymous function (because of the @) to another function by the name of ‘newtmult\_VL’. Included in this line of code are the initial guesses to be used as part of the function, as well as the desired error and max number of iterations to run. The results are then stored as the variables x, f, ea, and iter as defined by the function.

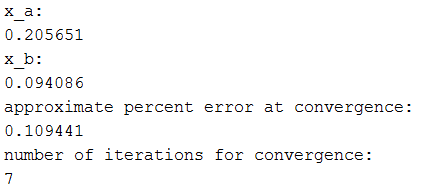
fprintf('x\_a:\n%f\nx\_b:\n%f\n',x(1),x(2)) %output results

fprintf('approximate percent error at convergence:\n%f\n',ea)

fprintf('number of iterations for convergence:\n%d\n',iter)

These last 3 lines of code format and print out the 3 results that were asked to output as part of the function.

**Results:**



**Discussion:**

As shown by the results, it took 7 iterations of the Newton-Raphson Multivariable method in order to converge to a value below the approximate error percentage specified of 1%. The accuracy of this method likely could have been improved by an alternative root finding method such as the modified secant or even the central difference approximation, and subsequently reduce the amount of iterations (this was not part of the problem).

From this problem, we learned how to implement the Newton Rhapson method that we had learned as a root solving method, in the context of linear systems of equations. We learned how to pass an entire function script as an anonymous function into another function. We reviewed calling functions from the main script. Furthermore, we reviewed different approximation methods for finding (partial) derivatives and used those to calculate a jacobian matrix. Lastly, we refreshed our knowledge on how to format and print values using the fprintf function in MATLAB.