Table of Contents

Parcial Ariadna Cortés

```
clear all;
close all;
format long;
```

Equilibrium position for charges

```
% plot the surface
r_sample = surface(linspace(0,2*pi,200));
figure(1)
plot(r_sample(1,:), r_sample(2,:),'-k', 'linewidth',1);
hold on

% We will do a newton solving for (sigma1, sigma2, sigma3)
% initial guess
x0 = [pi; pi/2; 3*pi/2];

% call the newton with epsilon tolerance and the Dots function
[xk,resd,it] = newtonn(x0,eps,50, @dots);

% The result is the last iteration of the newton
x = xk(end-2:end)

% We plot the results (plot appears at the end, after the auxiliar
% functions)
r_result = surface(x); % (x,y) position of the result
plot(r_result(1,:), r_result(2,:),'or', 'linewidth',1)
title('Equilibrium positions for $q_1$, $q_2$,
 $q_3$', 'Interpreter', 'latex')
xlabel('$x$', 'Interpreter', 'latex')
xlabel('$y$', 'Interpreter', 'latex')
```

Auxiliar codes

```
function r = surface(sigmas)
    % Input: sigma scalar
    % Output: position in the surface at that sigma
    x = exp(cos(sigmas));
    y = exp(sin(sigmas));
    r = [x; y];
end
```

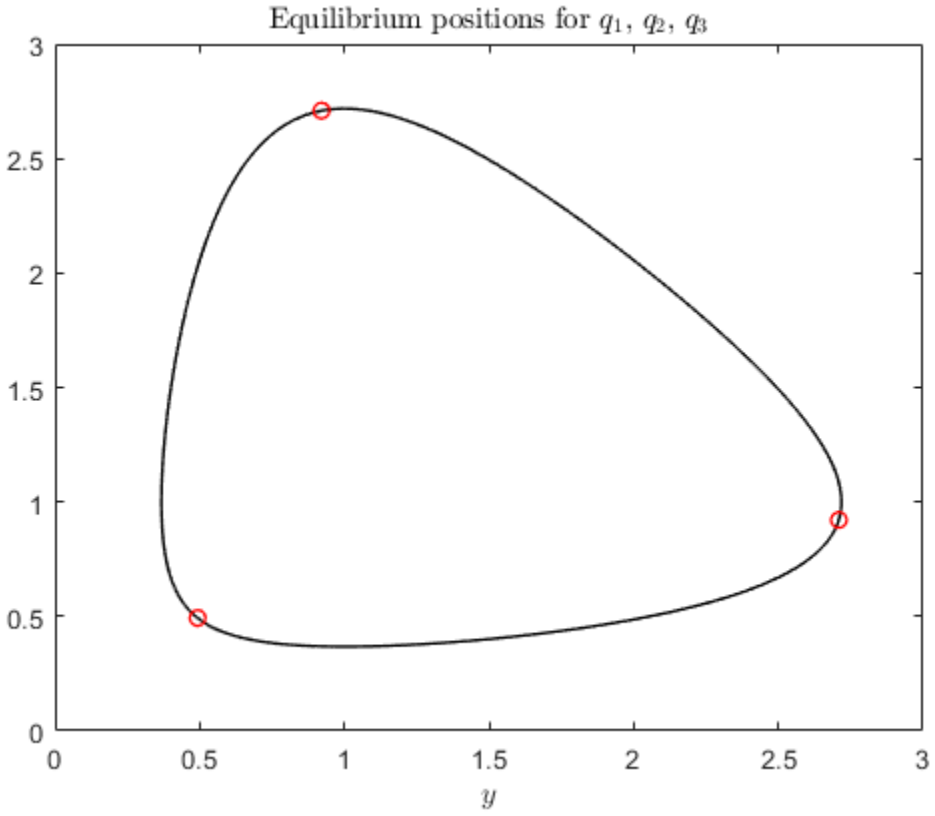
```
function d = gradSurface(sigma)
    % Input: sigma(scalar to evaluate the parcial derivative at
    % output: d -> partial derivatives evaluated at sigma
    x = -sin(sigma)*exp(cos(sigma));
    y = cos(sigma)*exp(sin(sigma));
    d = [x; y];
end

function F = coulomb(sigma1, sigma2)
    % Input: the positions of two particles
    % output: F the force between them
    c1 = exp(cos(sigma1)); c2 = exp(cos(sigma2));
    s1 = exp(sin(sigma1)); s2 = exp(sin(sigma2));
    d = sqrt((c1-c2)^2+(s1-s2)^2);
    d3 = d^3;

    F = [c1-c2, s1-s2];
    F = F/d3;
end

function y = dots(sigma)
    % Input: vector of sigmas (size 3)
    % Output: for each particle in sigma evaluate scalar product
 between
    % the force acting on it and the gradient of the surface

    F1 = coulomb(sigma(1),sigma(2)) + coulomb(sigma(1), sigma(3));
    F2 = coulomb(sigma(2),sigma(1)) + coulomb(sigma(2), sigma(3));
    F3 = coulomb(sigma(3),sigma(1)) + coulomb(sigma(3), sigma(2));
    Dot1 = dot(F1, gradSurface(sigma(1)));
    Dot2 = dot(F2, gradSurface(sigma(2)));
    Dot3 = dot(F3, gradSurface(sigma(3)));
    y = [Dot1; Dot2; Dot3];
end


% Codes from class
% Code 20: Newtonis method for n-dimensional systems
% Input: x0 - initial guess (column vector)
%        tol - tolerance so that ||x_{k+1} - x_{k} || < tol
%        itmax - max number of iterations
%        fun - function's name
% Output:   XK - iterated
%           resd: resulting residuals of iteration: ||F_k||
%           it:   number of required iterations to satisfy tolerance
function [XK,resd,it] = newtonn(x0,tol,itmax,fun)
    xk = [x0];
    resd = [norm(feval(fun,xk))];
    XK = [x0];
    it = 1;

    tolk = 1.0;
    n = length(x0);
```

```
    while it < itmax && tolk > tol
        Fk = feval(fun, xk);

        DFk = jac(fun, xk);
        [P,L,U] = pplu(DFk);

        dxk = plusolve(L,U,P,-Fk);

        xk = xk + dxk;
        XK = [XK xk];
        resd = [resd norm(Fk)];
        tolk = norm(XK(:, end)-XK(:, end-1));
        it = it + 1;
    end
end

% Code 13: PA = LU factorization (partial pivoting)
% Input: A (non-singular square matrix)
% Output: L (unit lower triangular matrix)
%         U (upper triangular matrix)
%         P (reordering vector)
function [P, L, U] = pplu(A)
    [m,n] = size(A);

    if m~=n
           error('not square matrix');
    end

    U = A;
    L = eye(n);

    P = [1:n]';

    for k = 1:n-1
        [~, imax] = max(abs(U(k:end,k)));
        imax = imax+k-1;
        i1 = [k, imax];
        i2 = [imax, k];

        U(i1,:) = U(i2,:); % Column k will be column imax and column
 imax will be column k
        P(k) = imax;

        L(i1,1:k-1) = L(i2, 1:k-1);

        for jj = [k+1:n]
            L(jj, k) = U(jj, k)/U(k, k);
            U(jj, k:n) = U(jj, k:n) - L(jj, k)*U(k,k:n);
        end
    end
end

% Code 14: PA = LU (Solver for Ax = b)
% Input:    L (unit lower triangular matrix)
```

```
%           U (upper triangular matrix)
%           P (reordering vector)
%           b (right-hand side)
% Output:   solution x
function x = plusolve(L, U, P, b)
    n = length(b);
    for k = 1:n-1
        b([k P(k)]) = b([P(k) k]);
    end
    y = fs(L, b);
    x = bs(U, y);
end

% Code 19: Computation of the Jacobian J
% Input:   F(x) : R^m ---> R^n
%          x : (m x 1)-vector ; F: (n x 1)-vector
% Output: DF(x) (n x m) Jacobian matrix at x
function DF = jac(F,x)
    f1 = feval(F,x);
    n = length(f1);
    m = length(x);

    DF = zeros(n,m);
    H = sqrt(eps)*eye(m);

    for j = 1:m
        f2 = feval(F,x+H(:,j));
        DF(:,j) = (f2 - f1)/H(j,j);
    end
end

% Code 12: Backward Substitution for Upper Triangular Systems
% Input:    U: Upp. Triangular non-singular square matrix
%           b: column right-hand side
% Output:   x: solution of Ux=b
function x = bs(U, b)
    x = 0*b;
    n = length(b);
    x(n) = b(n)/U(n,n);

    for ii = n-1:-1:1
        x(ii) = (b(ii)-U(ii, ii+1:n)*x(ii+1:n))/U(ii,ii);
    end
end

% Code 11: Forward Substitution for Lower Triangular Systems
% Input:    L: Low Triangular non-singular square matrix
%           b: column right-hand side
% Output:   x: solution of Lx=b
function x = fs(L, b)
    x = 0*b;
    n = length(b);
    x(1) = b(1)/L(1,1);
```

```
    for ii = 2:n
        x(ii) = (b(ii)-L(ii, 1:ii-1)*x(1:ii-1))/L(ii,ii);
    end
end
```

```
x =

   3.926990816987241   -0.082065999145743    7.936047633120226
```



Published with MATLAB® R2020b