E7: Introduction to Computer Programming for Scientists and Engineers
University of California at Berkeley, Spring 2017
Instructor: Lucas A. J. Bastien

Lab Assignment 08: Root Finding; Systems of Linear Algebraic Equations
Version: release

Due date: Friday March 17th 2017 at 12 pm (noon).

General instructions, guidelines, and comments:

- For each question, you will have to write and submit one or more Matlab functions. We provide a number of test cases that you can use to test your function. The fact that your function works for all test cases provided does not guarantee that it will work for all possible test cases relevant to the question. It is your responsibility to test your function thoroughly, to ensure that it will also work in situations not covered by the test cases provided. During the grading process, your function will be evaluated on a number of test cases, some of which are provided here, some of which are not.
- Submit on bCourses one m-file for each function that you have to write. The name of each file must be the name of the corresponding function, with the suffix `.m` appended to it. For example, if the name of the function is `my_function`, the name of the file that you have to submit is `my_function.m`. **Carefully check the name of each file that you submit.** Do not submit any zip file. If you re-submit a file that you have already submitted, bCourses may rename the file by adding a number to the file's name (*e.g.,* rename `my_function.m` into `my_function-01.m`). This behavior is okay and should be handled seamlessly by our grading system. Do not rename the file yourself as a response to this behavior.
- A number of optional Matlab toolboxes can be installed alongside Matlab to give it more functionality. All the functions that you have to write to complete this assignment can, however, be implemented without the use of any optional Matlab toolboxes. We encourage you to not use optional toolboxes to complete this assignment. All functions of the Matlab base installation will be available to our grading system, but functions from optional toolboxes may not. If one of your function uses a function that is not available to our grading system, you will loose all points allocated to the corresponding part of this assignment. To guarantee that you are not using a Matlab function from an optional toolbox that is not available to our grading system, use one or both of the following methods:
  - ◇ Only use functions from the base installation of Matlab.
  - ◇ Make sure that your function works on the computers of the 1109 Etcheverry Hall computer lab. All the functions available on these computers will be available to our grading system.

- For this assignment, the required submissions are:
  - ◇ `my_root_bisection.m`
  - ◇ `my_root_newtonraphson.m`
  - ◇ `my_buoyancy.m`
  - ◇ `my_reach_island.m`
  - ◇ `my_solve_system.m`
  - ◇ `my_treatment_plant.m`

Built-in functions that you may not use

You may not use Matlab's built-in functions `fzero`, `roots`, `fminbnd`, and `fminsearch` in this lab assignment. This rule applies to each question of this lab assignment.

Where to find information about the methods used in this lab?

The bisection and Newton-Raphson methods for root finding were covered in lecture L19 (Monday March 6$^{\text{th}}$). Chapter 16 of the textbook also covers these methods. Techniques to determine the number of solution(s) of systems of linear algebraic equations, and techniques to solve these systems were covered in lecture L20 (Wednesday March 8$^{\text{th}}$). Chapter 12 of the textbook also covers these techniques.

1. Generic root finding

1.1. Bisection method

Write a function with the following header:

```
function [n_iterations, root] = my_root_bisection(f, a, b, tolerance)
```

where:

- `f` is a function handle that represents a continuous real-valued function f defined on $\mathbb{R}$. `f` takes a single input argument that is a scalar of class `double` (other than `NaN`, `Inf`, and `-Inf`) and outputs a single output argument that is a scalar of class `double` (other than `NaN`, `Inf`, and `-Inf`).
- `a` and `b` are scalars of class `double` such that `(b > a)` and `(f(a)*f(b) < 0)`
- `tolerance` is a scalar of class `double` that is greater than zero and that is different from `NaN` and `Inf`.
- `root` is a scalar of class `double` that is the approximation of the root of the function f on the interval $[a, b]$, estimated using the bisection method on the interval $[a, b]$, with as few iterations as possible, and such that `abs(f(root)) <= tolerance`. You can assume that f has one and only one root in the interval $[a, b]$.
- `n_iterations` is the number of iterations of the bisection method used to estimate `root`.

In the bisection method, an iteration corresponds to the calculation of the midpoint of the interval of interest. If the first midpoint that you calculate can be considered a root according to the criterion mentioned above *i.e.* `abs(f(root)) <= tolerance`, then `n_iterations` should be equal to `1`. Note that, in this question, `n_iterations` cannot be equal to zero.

Test cases:

```
>> my_function = @(x) x.^2 - 5;
>> [n, root] = my_root_bisection(my_function, 2, 3, 1e-3)
  n =
    12
root =
    2.2361

>> f = @(var) cos(var/2);
>> [n, root] = my_root_bisection(f, 3, 4, 1e-5)
n =
    10
root =
    3.1416

>> f = @(x) cos(x)-x;
>> [n, root] = my_root_bisection(f, -10, 10, 1e-4)
n =
    15
root =
    0.7391

>> g = @(t) exp(cos(t)) - (t-3).^2;
>> [n, root] = my_root_bisection(g, 3, 4, 1e-5)
n =
    15
root =
    3.6454
```

1.2. Newton-Raphson method

Write a function with the following header:

```
function [n_iterations, root] = my_root_newtonraphson(f, df, r0, tolerance)
```

where:

- `f` is a function handle that represents a real-valued function f defined and differentiable on $\mathbb{R}$. `f` takes a single input argument that is a scalar of class `double` (other than `NaN`, `Inf`, and `-Inf`) and outputs a single output argument that is a scalar of class `double` (other than `NaN`, `Inf`, and `-Inf`).
- `df` is a function handle that represents the real-valued function f' (*i.e.* the derivative of

the function f). `df` takes a single input argument that is a scalar of class `double` (other than `NaN`, `Inf`, and `-Inf`) and outputs a single output argument that is a scalar of class `double` (other than `NaN`, `Inf`, and `-Inf`).

- `r0` is a scalar of class `double` that is neither `NaN`, `Inf`, nor `-Inf`.
- `tolerance` is a scalar of class `double` that is greater than zero and that is different from `NaN` and `Inf`.
- `root` is a scalar of class `double` that is the approximation of the root of the function f, estimated using the Newton-Raphson method starting with the initial guess `r0`, with as few iterations as possible, and such that `abs(f(root)) <= tolerance`. You can assume that, for each case with which your function will be tested, the Newton-Raphson method will converge to a value within the expected tolerance.
- `n_iterations` is the number of iterations of the Newton-Raphson method used to estimate `root`. In the Newton-Raphson method, an iteration corresponds to the calculation of a new guess for the root. If the initial guess `r0` is such that `abs(f(r0)) <= tolerance`, then `n_iterations` should be zero.

Test cases:

```
>> f = @(x) x.^2 - 5;
>> df = @(x) 2*x;
>> [n, root] = my_root_newtonraphson(f, df, 2, 1e-3)
n =
     2
root =
    2.2361

>> f = @(var) cos(var/2);
>> df = @(var) -0.5*sin(var/2);
>> [n, root] = my_root_newtonraphson(f, df, 3, 1e-5)
n =
     2
root =
    3.1416

>> f = @(x) cos(x)-x;
>> df = @(x) -sin(x)-1;
>> [n, root] = my_root_newtonraphson(f, df, 5, 1e-4)
n =
    21
root =
    0.7391

>> g = @(t) exp(cos(t)) - (t-3).^2;
>> dg = @(t) -exp(cos(t))*sin(t) - 2*(t-3);
>> [n, root] = my_root_newtonraphson(g, dg, 3, 1e-5)
n =
     8
root =
```

```
3.6454
```

2. Buoyant solid

Consider a non-deformable spherical solid of radius r_s and of uniform density ρ_s that is at equilibrium (*i.e.* not moving) in a liquid of density $\rho_l > \rho_s$, as illustrated by Figure 1. Since $\rho_l > \rho_s$, the solid does not sink completely into the liquid. Rather, a certain fraction of this solid is submerged while the rest is not (*i.e.* the rest is surrounded by air).

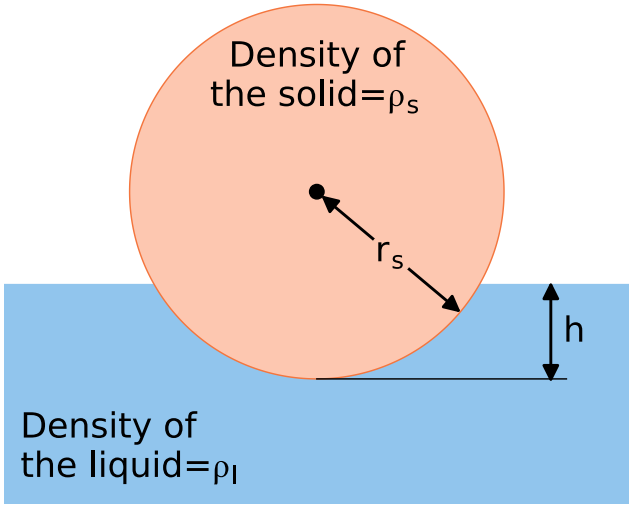

Figure 1: A non-deformable spherical solid of radius r_s and of uniform density ρ_s that is at equilibrium (*i.e.* not moving) in a liquid of density $\rho_l > \rho_s$. See main text for a more detailed description.

The three forces that are exerted on the solid are vertical and are:

- The weight of the solid. This force is directed downward and its magnitude is $F_w = m_s g = (4/3)\pi r_s^3 \rho_s g$ where m_s is the mass of the solid and g is the acceleration of gravity.
- The action of the liquid on the solid. This force is directed upward and its magnitude is, according to Archimedes' principle, $F_a = V_{\text{sub}} \rho_l g$, where V_{sub} is the volume of solid submerged in the liquid.
- The action of the air on the solid. This force is directed upward and, assuming that the density of the solid is much larger than the density of air, its magnitude is small compared to the magnitude of the weight of the solid itself. We neglect this force in this question.

The value of V_{sub} is given by:

$$V_{\text{sub}} = \pi h^2 (r_s - \frac{h}{3}) \tag{1}$$

where h is the depth of the submerged part of the solid. Newton's laws of motion indicate that, since the solid is at equilibrium:

$$F_w = F_a \tag{2}$$

which yields:

$$4r_s^3 \rho_s - h^2(3r_s - h)\rho_l = 0 \tag{3}$$

Let us define a function f on $\mathbb{R}$ such that, for any $h \in \mathbb{R}$:

$$f(h) = 4r_s^3 \rho_s - h^2(3r_s - h)\rho_l \tag{4}$$

In this question, you will write a function that estimates the depth h of the submerged part of the solid. More precisely, write a function with the following header:

```
function [depth] = my_buoyancy(radius, rho_solid, rho_liquid, tolerance)
```

where:

- `radius` is a scalar of class `double` that represents the radius r_s (in m) of the spherical solid.
- `rho_solid` is a scalar of class `double` that represents the density ρ_s (in kg m^{-3}) of the solid.
- `rho_liquid` is a scalar of class `double` that represents the density ρ_l (in kg m^{-3}) of the liquid in which the solid is at rest.
- `tolerance` is a scalar of class `double` that is positive and that is neither `NaN` nor `Inf`. `tolerance` is in units of kg.
- `depth` is a scalar of class `double` that represents the depth h (in m) of the submerged part of the solid. Your function should estimate this quantity by applying the Newton-Raphson root finding method to Equation 3, using r_s as an initial guess for h, and such that $|f($`depth`$)|$ `<= tolerance`. You can assume that `depth` > 0 and `depth` $< 2r_s$. You can also assume that, for each case with which your function will be tested, the Newton-Raphson method will converge to a value within the expected tolerance.

Hint: your function `my_buoyancy` can call your function `my_root_newtonraphson`.

Test cases:

```
>> [depth] = my_buoyancy(1, 0.7, 1, 1e-10)
depth =
    1.2735

>> [depth] = my_buoyancy(1, 0.5, 1, 1e-10)
depth =
     1

>> [depth] = my_buoyancy(0.5, 0.25, 0.75, 1e-5)
depth =
    0.3870

>> [depth] = my_buoyancy(0.5, 0.25, 0.3, 1e-3)
depth =
    0.7405
```

3. Reaching an island

Imagine that you are standing on a beach (which has a straight shore line) at a distance d_B from the water, as illustrated by Figure 2. You wish to reach an island that is located at a distance d_W from the shore line. Distance along the shore line is measured by the x-coordinate and distance from the shoreline is measured by the y-coordinate. You wish to determine the fastest way for you to reach the island by first running at constant speed v_B on the beach, and then by swimming to the island at constant speed v_W. You know that to achieve this goal, you have to first run in a straight line, enter the water, and then swim in a straight line. However, you are not sure at which location along the shore you should enter the water. The distance that you travel along the x-axis before entering the water is called a. The distance along the x-axis between your initial location and the island is called b. The angle formed between your running trajectory and the shoreline is called θ. In this question, we treat θ as an unsigned angle (*i.e.* it is always positive).

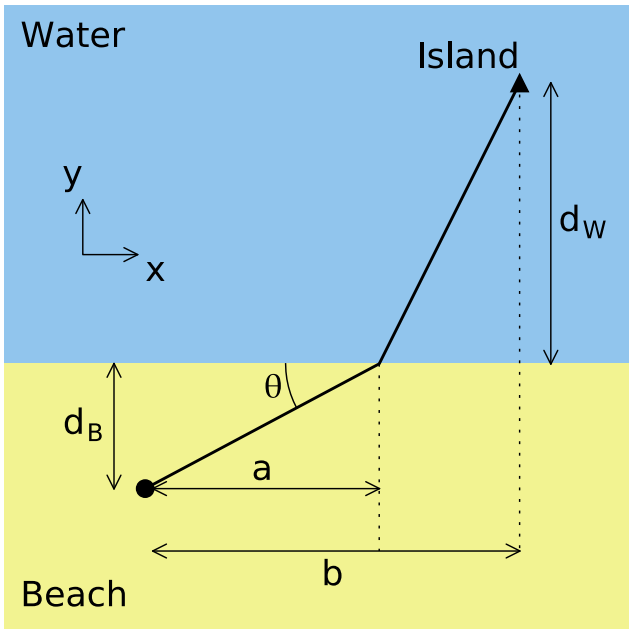

Figure 2: Illustration of your location (dot) versus the location of the island (triangle).

In this question, you will write a function that estimates the angle θ that makes you reach the island as fast as possible. More precisely, write a function will the following header:

```
function [theta] = my_reach_island(d_beach, d_water, b, v_beach, v_water)
```

where:

- `d_beach` is a scalar of class `double` that represents the distance d_B (in m) between your initial position and the shoreline.
- `d_water` is a scalar of class `double` that represents the distance d_W (in m) between the shoreline and the island.
- `b` is a scalar of class `double` that represents the distance (in m) along the x-axis between your initial location and the island.
- `v_beach` is a scalar of class `double` that represents the speed (in m s^{-1}) at which you run on the beach.
- `v_water` is a scalar of class `double` that represents the speed (in m s^{-1}) at which you swim in the water.
- `theta` is a scalar of class `double` that represents the angle (in degrees) formed between your running trajectory and the shoreline, and such that you reach the island in as little time as possible.

To write your function, you should:

1. On paper, derive the expression of the time $t(a)$ that it takes you to reach the island as

a function of the distance a. We will use the bisection method to find the value of a that minimizes $t(a)$.

2. Still on paper, calculate the derivative $t'(a)$ of this function with respect to a.
3. In the general case, $t'(a_m) = 0$ is a necessary but not a sufficient condition for the function t to have a minimum or maximum at $a = a_m$. In this question, however, you can assume that the function t has one and only one minimum in the interval $[0, b]$, and that this minimum occurs for the only value of a (let us call this value a_m) such that $t'(a_m) = 0$. Use the bisection method on t' to determine a_m such that $t'(a_m) < 10^{-6}$ s m^{-1}.
4. Calculate the value of θ that corresponds to $a = a_m$. Consider this estimate of θ as the value that allows you to reach the island in the minimum amount of time.

Hint: your function `my_reach_island` can call your function `my_root_bisection`.

Test cases:

```
>> theta = my_reach_island(50, 500, 200, 3, 0.5)
theta =
   22.2920

>> theta = my_reach_island(50, 1000, 200, 3, 0.5)
theta =
   37.1555

>> theta = my_reach_island(50, 1000, 200, 3, 2)
theta =
   74.1005

>> theta = my_reach_island(100, 1000, 200, 3, 2)
theta =
   75.1443
```

4. Systems of linear algebraic equations

Consider a system of m linear algebraic equations and n unknowns:

$$
\begin{aligned}
a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= b_1 \\
a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n &= b_2 \\
&\cdots \\
a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n &= b_m
\end{aligned}
$$

where $a_{i,j} \in \mathbb{R}$ and $b_i \in \mathbb{R}$, $i = \{1, 2, \ldots, m\}$, $j = \{1, 2, \ldots, n\}$ (*i.e.* the $a_{i,j}$'s and b_i's are real constant coefficients). The n unknowns are x_j, $j = \{1, 2, \ldots, n\}$, and we try to find solutions where all the x_j's are real numbers (*i.e.* $x_j \in \mathbb{R}$, $j = \{1, 2, \ldots, n\}$)

We can write this system of linear algebraic equations in matrix form as follows:

$$Ax = b \tag{5}$$

with:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \tag{6}$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{7}$$

$$b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \tag{8}$$

In this question, you will write a function that attempts to solve a system of linear algebraic equations as described above. More precisely, write a function with the following header:

```
function [n_solutions, x, square_error] = my_solve_system(a, b)
```

where:

- `a` is an $m \times n$ array of class `double` that represents the matrix A of the system to solve (*i.e.* A in Equation 5), such that `a(i,j)` in your function represents $a_{i,j}$. You can assume that $m > 0$ and $n > 0$, and that all elements of `a` are different from `NaN`, `Inf`, and `-Inf`.
- `b` is an $m \times 1$ array of class `double` that represents the matrix b of the system to solve (*i.e.* b in Equation 5), such that `b(i,1)` in your function represents b_i. You can assume that $m > 0$ and that all elements of `b` are different from `NaN`, `Inf`, and `-Inf`.
- `n_solutions` is a scalar of class `double` whose value is the number of distinct solutions of the system of linear algebraic equations to solve. Its only possible values are `0`, `1`, and `Inf`.
- `x` is an $n \times 1$ array of class `double`, whose value is:
  - `a\b` if the system to solve has zero solution. Note that in this case, `x` does **not** represent a solution of the system to solve. Rather, it represents the vector x such that the square error made on b when using x as a "solution" is minimized (see below for more detail).

- ◇ `a\b` if the system to solve has one and only one solution.
- ◇ `pinv(a)*b` if the system to solve has an infinite number of solutions. In this case `pinv(a)*b` corresponds to a specific solution among the infinite number of possible solutions.

- `square_error` is a scalar of class `double` that represents the square error s between (i) the left-hand side of Equation 5 using the value of `x` defined above and (ii) the right-hand side of Equation 5. If the system to solve has one or more solutions, you should set the value of `square_error` to zero (*i.e.* `square_error = 0`). If the system to solve has zero solution, then you should calculate `square_error` using the following formula:

$$s = \sum_{i=1}^{m} (a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,n}x_n - b_i)^2 \tag{9}$$

Note that if you were to use the formula shown in Equation 9 to calculate `square_error` when the system to solve has one or more solutions, you might observe, in some cases, that the calculated value is not exactly equal to zero. Theoretically, if `x` represents a solution of the system, the square error should indeed be exactly zero. This discrepancy occurs because of truncature and other numerical errors in the calculations performed by the computer, such that even though `x` is very, very close to be the exact solution of the system of equations, it may not be exactly the solution. In this question, however, you should "manually" set the value of `square_error` to exactly zero (*i.e.* `square_error = 0`) when the system to solve has one or more solutions.

Test cases:

```
>> a = [3, 6, 7; 0, 1, -5; 0, 1, 6];
>> b = [2; 5; 1];
>> [n_solutions, x, square_error] = my_solve_system(a, b)
n_solutions =
     1
x =
   -4.8485
    3.1818
   -0.3636
square_error =
     0

>> a = [32, 13, 46, 46; 41, -41, 47, -1; 42, 5, 48, -36; 84, 15, 96, -72];
>> b = [-8; 42; 46; 1];
>> [n_solutions, x, square_error] = my_solve_system(a, b)
n_solutions =
     1
x =
 -259.8886
  -18.2000
  211.1874
  -25.4257
square_error =
```

```
     0

>> a = [32, 13, 46, 46; 41, -41, 47, -1; 42, 5, 48, -36; 84, 10, 96, -72];
>> b = [-8; 42; 46; 92];
>> [n_solutions, x, square_error] = my_solve_system(a, b)
n_solutions =
   Inf
x =
    0.3140
   -0.3656
    0.2885
   -0.5775
square_error =
     0

>> a = [16, 35; 18, -47; 44, 26];
>> b = [1; 10; 7];
>> [n_solutions, x, square_error] = my_solve_system(a, b)
n_solutions =
     0
x =
    0.2383
   -0.1113
square_error =
    1.7519
```

5. Water treatment plant

Often, water treatment systems rely on multiple treatment tanks arranged in series and/or in parallel in order to remove particles, bacteria, chemicals, and other undesirable components from water. Consider a hypothetical treatment system that is designed to remove a specific pollutant from water, and that consists of 4 treatment tanks as illustrated by Figure 3, where each tank is represented by a rectangle that contains the number of the tank. For each of these tanks, an arrow going in the tank indicates water flowing into the tank and an arrow coming out of the tank indicates water flowing out of the tank. With each arrow is associated two numbers:

- The top number indicates the flow rate of water flowing at that point of the treatment system. The flow rate indicates how much water flows through a certain part of the treatment system per unit of time. For example, the flow rate of water entering tank number 1 is Q. As another example, the flow rate of water exiting tank number 3 is $(1 - xy)Q$. In this question, flow rates are measured in units of volume per time (*e.g.,* $\mathrm{m}^3\ \mathrm{s}^{-1}$).
- The bottom number indicates the concentration of pollutant in the water at that point of the treatment system. For example, the concentration of pollutant in the water entering tank number 1 is C_1. In this question, concentrations are measured in units of mass of pollutant per volume of water (*e.g.,* $\mathrm{g}\ \mathrm{m}^{-3}$).

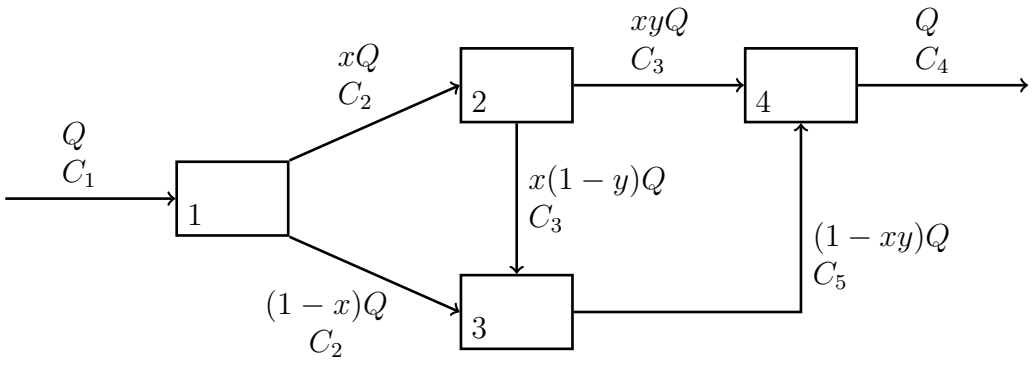

Figure 3: Illustration of the hypothetical treatment plant studied in this question. See the main text for details on how to read this figure.

Note that x and y are two scalars between 0 and 1 (both boundaries not included). Each tank removes some, but not all, of the pollutant present in the water. The rate at which tank number i removes pollutant from the water is equal to R_i. In this question, rates of pollutant removal are measured in units of mass of pollutant per unit time (*e.g.*, g s^{-1}).

In this question, we assume that the treatment system has reached steady-state, meaning that all the flow rates, concentrations, and rates of removal of the system are constant quantities. For each tank, the principle of conservation of mass dictates that:

$$Q_{\text{in}}C_{\text{in}} - Q_{\text{out}}C_{\text{out}} - R = 0 \tag{10}$$

where:

- Q_{in} and C_{in} are the flow rate and pollutant concentration, respectively, of the water entering the tank. If, for a given tank, water enters the tank through more than one stream, then $Q_{\text{in}}C_{\text{in}}$ should be the sum of the corresponding values calculated for each stream separately. For example, for tank number 3, $Q_{\text{in}}C_{\text{in}}$ is $(1-x)QC_2 + x(1-y)QC_3$.
- Q_{out} and C_{out} are the flow rate and pollutant concentration, respectively, of the water leaving the tank. If, for a given tank, water leaves the tank through more than one stream, then $Q_{\text{out}}C_{\text{out}}$ should be the sum of the corresponding values calculated for each stream separately. For example, for tank number 2, $Q_{\text{out}}C_{\text{out}}$ is $xyQC_3 + x(1-y)QC_3$.
- R is the rate of pollutant removal of the tank.

You can write one such equation for each tank.

In this question, you will write a function that calculates the pollutant concentration of the water at each point of the treatment system. More precisely, write a function with the following header:

```
function [concentrations] = my_treatment_plant(q, c_1, x, y, removals)
```

where:

- `q` is a scalar of class `double` that represents the flow rate of water entering tank number 1 (*i.e.* Q).
- `c_1` is a scalar of class `double` that represents the pollutant concentration in the water entering tank number 1 (*i.e.* C_1).
- `x` and `y` are scalars of class `double` that are greater than zero and smaller than 1.
- `removals` is a 4×1 array of class `double` whose elements represent the rates of pollutant removal from the corresponding tanks (*i.e.* `removals(i)` represents R_i).
- `concentrations` is a 5×1 array of class `double` whose elements represent the pollutant concentrations in the water at the corresponding points of the treatment system (*i.e.* `concentrations(i)` represents C_i).

You can assume that the units that will be given to the quantities represented by `q`, `c_1`, `x`, `y`, `removals`, and `concentrations`, are such that you do **not** have to do any unit conversion in this question. You can also assume that all these quantities will make physical sense (*e.g.*, the removal rates and concentrations will always be positive).

Hints:

- There are four unknown quantities in the system: C_2, C_3, C_4, and C_5, and you can write one "conservation of pollutant mass" equation for each tank, and thus obtain a system of linear algebraic equations.
- Although you can solve this system of equations "manually", we encourage you to write it in matrix form, and to use the methods seen in class to solve it.
- Your function `my_treatment_plant` can call your function `my_solve_system`.

Test cases:

```
>> q = 1000; c_1 = 0.4; x = 0.9; y = 0.1; removals = [20; 25; 15; 5];
>> concentrations = my_treatment_plant(q, c_1, x, y, removals)
concentrations =
    0.4000
    0.3800
    0.3522
    0.3350
    0.3388

>> q = 1000; c_1 = 0.4; x = 0.1; y = 0.9; removals = [20; 25; 15; 5];
>> concentrations = my_treatment_plant(q, c_1, x, y, removals)
concentrations =
    0.4000
    0.3800
    0.1300
    0.3350
    0.3608

>> q = 900; c_1 = 0.5; x = 0.9; y = 0.1; removals = [5; 15; 25; 20];
```

```
>> concentrations = my_treatment_plant(q, c_1, x, y, removals)
concentrations =
    0.5000
    0.4944
    0.4759
    0.4278
    0.4474

>> q = 900; c_1 = 0.5; x = 0.1; y = 0.9; removals = [5; 15; 25; 20];
>> concentrations = my_treatment_plant(q, c_1, x, y, removals)
concentrations =
    0.5000
    0.4944
    0.3278
    0.4278
    0.4621
```