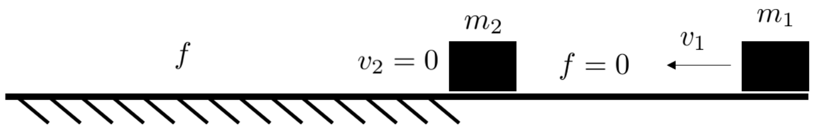
Lab Assignment #3
Due 2/14/2020 at 11:59pm on bCourses

The assignment will be graded by an autograder which will check the outputs of your functions. For your functions to be scored properly, it is important that the names of your functions **exactly match** the names specified in the problem statements, and input and output variables to each function are in the **correct order** (i.e. use the given function header exactly). Instructions for submitting your assignment are included at the end of this document.

1 Conservation of Momentum

`function t = collisionPrediction(m1, m2, v1, f)`		
Input	Type	Description
`m1`	`1x1 double`	Mass of box 1 (kilograms)
`m2`	`1x1 double`	Mass of box 2 (kilograms)
`v1`	`1x1 double`	Velocity of box 1 (m/s)
`f`	`1x1 double`	Coefficient of friction
Output	Type	Description
`t`	`1x1 double`	Time in motion after collision

Details



Box 1 (with mass m_1) is sliding with a velocity v_1 on a smooth surface and is going to hit a static box 2 (with mass m_2). Suppose there is a perfectly inelastic collision between the two boxes (i.e., the two boxes will combine into one body) and momentum is conserved. Thus, after collision the velocity of the combined boxes v can be described by:

$$v = \frac{m_1 \cdot v_1}{m_1 + m_2}$$

The combined boxes will then continue to slide onto a coarse surface with coefficient of friction f, which is given by the equation:

$$F_f = f \cdot G$$

where F_f is the frictional force, $G = (m_1 + m_2) \cdot g$ is the combined weight of the boxes, and f is the friction coefficient. Based on Newton's Second Law of Motion, the deceleration is given by $a = f \cdot g$. In this problem, you are required to find the time t when the combined box stops. You will write the function `collisionPrediction` that has the inputs `m1`, `m2`, `f`, and `v1` and outputs the time `t` it takes for the combined boxes to stop.

Tips

- $g = 9.81$ m/s^2

2 House Rental

`function rent = rentPrediction(time)`		
Input	Type	Description
`time`	`1x12 double`	An array composed of 0s and 1s, with each element indicating whether you are going to rent that month or not
Output		
`rent`	`1x1 double`	The maximum amount of money you can earn from plan A or plan B

Details

In this problem, you are going to rent your house over a period of 12 months but you haven't decided which months to rent and which months not to rent. There are two renting plans. You will write a function to determine which plan provides you with more income based on the months you choose to rent. The variable t indicates the t^{th} month.

Plan A is defined by

$$\text{Rent price} = \begin{cases} \$500, & 1 \le t < 3 \\ \$600, & 3 \le t < 4 \\ \$550, & 4 \le t < 5 \\ \$650, & 5 \le t < 8 \\ \$700, & 8 \le t < 9 \\ \$800, & 9 \le t < 11 \\ \$850, & 11 \le t \le 12 \end{cases} \tag{1}$$

Plan B is defined by

$$\text{Rent price} = \begin{cases} \$1000, & 1 \le t < 6 \\ \$200, & 6 \le t \le 12 \end{cases} \tag{2}$$

Tips

- The input `time` is a 1-by-12 array which represents the months you want to rent the house. For example, if you want to rent the house in 1^{st}, 3^{rd} and 8^{th} month, the input `time` should be $[1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0]$.
- You might find element-wise multiplication to be helpful in this question. You can check out the `.*`.
- The dot operator (in the tip above) can also be used for other arithmetic operations.
- You might also find the sum function in MATLAB to be helpful.

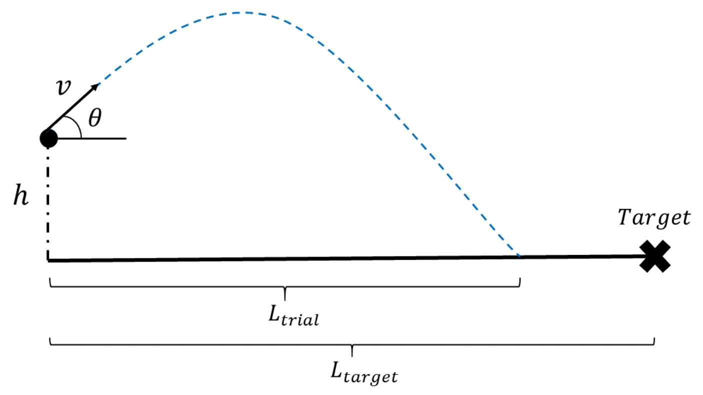
3 Object Launch Game

`function [v, theta] = targetPractice(L_target, height, tol)`		
Input	Type	Description
`L_target`	`1x1 double`	Distance to target (m)
`height`	`1x1 double`	Height the shot is taken from (m)
`tol`	`1x1 double`	Allowable error tolerance (m)
Output		
`v`	`1x1 double`	Initial velocity (v > 0) utilized to hit the target
`theta`	`1x1 double`	Launch angle utilized to hit target (degree) (0 < θ < 90)

Details

An object is launched with an initial velocity (v) and angle (θ) and follows a parabolic trajectory. In this problem, you will program a game in which the player is to guess the initial velocity v and angle θ required to hit the target. Your function will accept a target distance, which is the distance from the shooter to the target (distance > 0); the height from which the shot will be taken (height $\geq$ 0); and an error tolerance that determines the farthest the shot can be from the target to finish the game. In a loop, use MATLAB's built-in function input to prompt the user for the following inputs:

1. Input a positive initial velocity (v). A clear and meaningful prompt should be displayed.
2. Input an angle that the shot is taken at ($0 < \theta < 90$). A clear and meaningful prompt should be displayed.



The formula that determines horizontal distance traveled based on an initial velocity and angle is:

$$L_{trial} = v_x \cdot \left(\frac{v_y + \sqrt{v_y^2 + 2gh}}{g}\right)$$

where v_x and v_y are the x and y components of the initial velocity v, respectively.

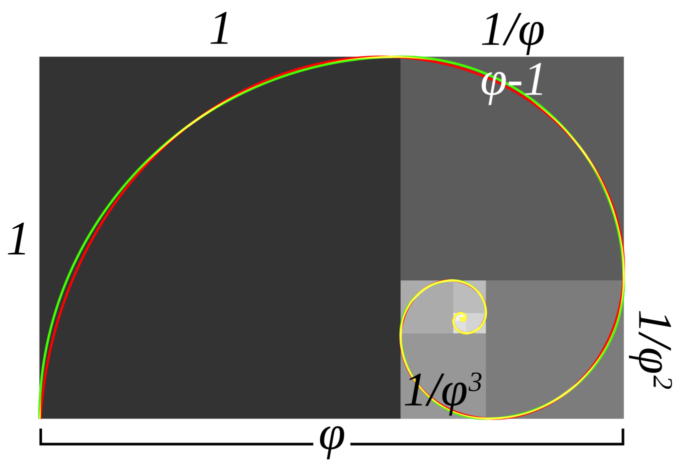
If the computed distance L_{trial} satisfies the `tol`, i.e., $|L_{trial} - L_{target}| \leq tol$, then your function should output the v and θ that achieved the hit. However, if $|L_{trial} - L_{target}| > tol$, your function should continue to prompt the user (via another input prompt) to put in another guess. The function will only give an output when the correct velocity and angle have been guessed by the player.

4 Golden Spiral - *Iterative*

`function ratio = myGoldenRatioIterative(m)`		
Input	Type	Description
`m`	`1x1 double`	A positive integer
Output		
`ratio`	`1x1 double`	The m^{th} Golden Ratio

Golden Ratio Formula

In geometry, a golden spiral (shown in the following image) is a logarithmic spiral whose growth factor is φ, the golden ratio. That is, a golden spiral gets wider (or further from its origin) by a factor of φ for every quarter turn it makes.



To approximate the golden spiral, you can start with approximating the golden ratio φ. One method to approximate the golden ratio φ is using the ratio between consecutive elements in a special F array. The approximation will approach the golden ratio as the length of the F array approaches infinity.

First, let's define the special F array. The k^{th} number $F(k)$ is defined as such: $F(1) = 1$, $F(2) = 1$, and $F(k) = F(k-1) + F(k-2)$ for $k > 2$. For example $F(3) = 2$, $F(4) = 3$ and $F(5) = 5$.

Next, let's define the m^{th} Golden Ratio to be $(m) = \frac{F(m+1)}{F(m)}$, $m \geq 1$, where $F(m)$ is the m^{th} element in the F array.

Use an **iterative** method to calculate the golden ratio of a given positive number `m` according to the above rule. You will need to calculate $F(m)$ and $F(m+1)$ before you calculate the golden ratio.

5 Golden Spiral - *Recursive*

`function F_k = myFArray(k)`		
Input	Type	Description
`k`	`1x1 double`	A positive integer indicating the k^{th} number
Output		
`F_k`	`1x1 double`	The k^{th} F array element

`function ratio2 = myGoldenRatioRecursive(m)`		
Input	Type	Description
`m`	`1x1 double`	A positive integer indicating the m^{th} number
Output		
`ratio2`	`1x1 double`	The m^{th} Golden Ratio

Details

Recall the Golden Ratio approximation in the last problem. This time, you will implement it again, but with recursion. You will first **recursively** calculate the k^{th} F array number in function `myFArray`.(You will get **zero (0)** points if you use an iterative method in `myFArray`.) Then you will build a function `myGoldenRatioRecursive` to calculate the golden ratio using function `myFArray`.

Part II: Computing Time

`function timeArrayGolden = goldenRatioComputing(n)`		
Input	Type	Description
`n`	`1x1 double`	A positive integer
Output		
`timeArrayGolden`	`2xn double`	1st row: computing times of Golden Ratio by `myGoldenRatioIterative` 2nd row: computing times of Golden Ratio by `myGoldenRatioRecursive`.

Details

Now we want to compare the time it takes to calculate the golden ratio with the iterative method and the recursive method. In this function, you will use iteration to call `myGoldenRatioIterative` and `myGoldenRatioRecursive` to approximate the 1^{st} to the n^{th} golden ratio. In each iteration, you will use the MATLAB built-in functions `tic` and `toc` to get the comuptation time of the two functions for each m ranging from 1 to n. The results will be stored in `timeArrayGolden`.

Then, you must use the MATLAB built-in function `plot` to plot the computing time of each method vs. the corresponding m. The y-axis of the plot should be the computing time and the x-axis should be array `[1:n]`. The graph will have two curves, one for each method. One suggestion for n is 20.

Submission Guidelines

When you are ready to test one or more of your function(s), download the `Lab03Tester.m` script from bCourses and run it in the same directory as your function(s). This will run a series of test cases for your function(s) and display the outputs. You should check these results with the example outputs displayed. Note that the series of tests that the script will run is not exhaustive and that it is up to you to make sure that your function fully satisfies the requirements of the problem. If you want to run all the test cases for one function, select that part of the tester script and push the `Run Section` button. If you wish to only check a single test case, you may do so by highlighting the appropriate lines in MATLAB and then pressing F9 (on Windows and linux) and **`shift + fn + F7`** (on a Mac) to run the selected text in the Command Window.

The script will also pack your function into a zip file for final submission. Your function file names and headers should match the examples given in the problem statement. If they do not match, the tester script will not be able to find them and will issue a warning. If you do not correct this problem, your missing function(s) will not be able to be graded and **you will receive 0 points**, so be sure to check carefully! Make sure to upload the `Lab03.zip` file to bCourses before the deadline. You may resubmit your zip file by re-uploading to bCourses before the deadline, but please be aware that only the most recent zip submission is considered for grading. That said, it is critical that you include your most recent version of **all** of your functions in each new zip submission.

Your final zip file for this assignment should contain the following:

- `collisionPrediction.m`
- `rentPrediction.m`
- `targetPractice.m`
- `myGoldenRatioIterative.m`
- `myFArray.m`
- `myGoldenRatioRecursive.m`
- `goldenRatioComputing.m`