Assignment 3: Simulation using Runge–Kutta

Tufts ME 149: Optimal Control For Robotics

Assigned: Jan. 30 — Due: Feb. 8

Introduction

In this assignment you will implement a general-purpose simulation function which implements a few different Runge–Kutta integration methods. You will then use this function to generate simulations for two different dynamical systems: a driven-damped pendulum and a passive double pendulum. The final part of the assignment is to create two plots, one for each system, showing the solution as well as the absolute error for each method.

Write-up

Instead of submitting a separate write-up, please fill-in the header block in `hw_03_studentName.m`. Like previous assignments, this includes your name, information about the assignment, collaborators and references, how long the assignment took to complete, and concludes with an outline of your code.

Deliverables

Please upload each of the following files to Trunk — you do not need to zip them to a single file before uploading. If you choose to create any additional Matlab files as part of you assignment, then please upload those as well. You do not need to include any files (or functions) that are part of the `codeLibrary` for this course.

- `hw_03_studentName.m` — main function for the assignment
- `runSimulation.m` — general simulation function using Runge–Kutta methods
- `hw_03_driven_damped_pendulum.pdf` — plot for the driven damped pendulum results
- `hw_03_passive_double_pendulum.pdf` — plot for the passive double pendulum results

See the following pages for additional details on how to create each file.

Details: `runSimulation.m`

```
function [zGrid, nEval] = runSimulation(dynFun, tGrid, zInit, method)
%  [zGrid, nEval] = runSimulation(dynFun, tGrid, zInit, method)
%
% Simulate a dynamical system, given an integration routine, time grid,
% and initial state. Select from several explicit Runge-Kutta methods.
%
% INPUTS:
%    dynFun = a function handle:  dz = dynFun(t, z)
%        IN:  t = [1, nTime] = row vector of time
%        IN:  z = [nState, nTime] = matrix of states corresponding to each time
%        OUT: dz = [nState, nTime] = time-derivative of the state at each point
%    tGrid = [1, nGrid] = time grid to evaluate the simulation
%    zInit = [nDim, 1] = initial state
%    method = string = name of the desired method
%      'euler' = Euler's method (first-order)
%      'heun' = Heun's method (second-order)
%      'midpoint' = the midpoint method (second-order)
%      'ralston' = Ralston's method (second-order)
%      'rk4' = "The" Runge--Kutta method (forth-order)
%
% OUTPUTS:
%   zGrid = [nDim, nGrid] = state at each point in tGrid. zGrid(:,1) = zInit
%   nEval = scalar = total number of calls to the dynamics function
%

%%%% TODO:  implement this function

end
```

Requirements

This function provides an interface that makes it easy to run a simulation with a variety of different integration methods. You need to implement Euler's method, at least one of the second-order methods, and "The" forth-order Runge–Kutta method. While working on your implementation please follow good programming practices.

Programming Challenge

The easiest way to implement this function is to have a separate sub-function for each of the three method orders (first, second and forth). You will receive full credit for this style of implementation.

An alternate implementation is to write a truely general-purpose sub-function that will work for *any* explicit Runge–Kutta method. Such a function would work by accepting the method parameters (in the form of a Butcher tableau) as arguments. The top level function would then simply set the correct method parameters before calling the sub-function. If you choose to implement this method, then try experimenting with more complicated methods (email me for examples).

Details: `hw_03_studentName.m`

```
function hw_03_studentName()
% Tufts ME 149 - Optimal Control for Robotics - HW 3
%
% Simulation using Runge--Kutta methods
%
% Assigned: Jan 30, 2018
% Due: Feb 8, 2018
%
% Student Name:  TODO
%
% Collaborators:  TODO:
%
% References:  TODO
%
% How long did this assignment take you to complete?
%   %%%% TODO
%
% Outline:
%
%   TODO
%
%

% Part One: driven-damped pendulum
analysisDrivenDampedPendulum();

% Part Two: passive double pendulum
analysisPassiveDoublePendulum();

end

%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~%

function analysisDrivenDampedPendulum()

%%%% TODO: implement this function

end

%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~%

function analysisPassiveDoublePendulum()

%%%% TODO: implement this function

end

%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~%
```

System Dynamics

For this assignment you will need to simulate two systems: the driven-damped pendulum and a passive double pendulum. You will find functions that implement the system dynamics for both of these systems in the code library for the course:
`ME149/codeLibrary/modelSystems/systemName/systemNameDynamics.m`

You can add the entire code library to your matlab path by running:
`ME149/codeLibrary/addLibraryToPath.m`

You are permitted (and encouraged) to use any file within the code library for your assignment. You do not need to upload these files to Trunk.

Double Pendulum Dynamics

The double pendulum dynamics have a different set of arguments than are required by the standard simulation functions. Use a function handle to pass through the correct arguments.

- **time** — doublePendulumDynamics does not use time, you can ommit it.
- **control** — doublePendulumDynamics accepts control torques, as well as the state. To make the system passive you will need to set both control torques to zero.
- **parameterss** — doublePendulumDynamics requires that you pass system parameters, which can all be set to one. Use `>> help doublePendulumDynamics` to find the parameter names.

Note: If you've taken a dynamics course, then I suggest that you take a look at the script for deriving the dynamics function using the Matlab symbolic toolbox. The techniques used there can be applied to any continuous dynamical system, with a few small modifications, provided that it lives in two-dimensions.

Solution with `ode45()`

You will need to compute the error associated with each simulation method that you implement in `runSimulation.m`. We will use Matlab's `ode45` function to approximation the true solution. To do this, use `ode45` to run a simulation for each system, and specify `'RelTol' = 1e-10` and `'AbsTol' = 1e-10` using the `options` argument and `odeset`.

Simulation Parameters

The driven-damped pendulum simulation should start at $t = 0$ and finish at $t = 20$ using 150 grid points. Its initial angle should be 1.2 radians and the initial angular rate should be 0.8 radians per second. The passive double pendulum simulation should have a duration of 10 seconds and use 100 grid points. The initial angles for first link should be 0.6 and the second link should be 0.9. Both links are initially stationary. Link lengths, masses, and gravity should all be set to one.

Plots

Create one figure for the driven-damped pendulum results and a second figure for the passive double pendulum results. Both figures should have the same formatting and outline.

The top row of subplots should show the state while the bottom row shows the error. Each subplot should have four curves: one for the solution (via `ode45`), and the remaining three for each method that you implemented in `runSimulation.m`: first-, second-, and forth-order Be sure to include axis labels and a legend for each subplot.

The error plots should show the difference between your simulations and `ode45`. You should plot the absolute value of the error, and the y-axis should be on a log scale. Since the tolerance for `ode45` is set to $1e-10$, it makes not sense to have any error values smaller than that: please clamp the error curves to have a minimum value of $1e-10$.

Following this outline, the driven-damped pendulum figure will have a two-by-two array of subplots, while the passive double pendulum figure will have two rows and four columns of subplots. Be sure to make your figures full-screen before saving them to pdf.