Thomas Rochais and Courtney Sheyko

Description of Matlab code

1 Introduction

Here we describe the code which simulates a spherical robot rolling on a surface, using contact as it's only way of guidance. The reader should refer to the section on assumptions, in the main paper, for restrictions on the surface type. First, we describe the main program 'Robotics.m', then a section is dedicated for each subroutine 'Robotics.m' uses. Furthermore, as a way to test our program we generated a few example surfaces. The way those surfaces were generated is described afterwards.

2 Storing the robot and the workspace

The robot, the surface, and the workspace are read from folders storing n $n \times n$ black and white '.png' files, and then stored by the program into $n \times n \times n$ 3-dimentional matrices. The white (or 1 values) correspond to the object, while the black (0 values) correspond to empty space. The robot is stored into the 'ROBOT' matrix, while the surface and the walls are combined into the 'WORKSPACE' matrix. One can either enter the surface and walls seperately or enter a full workspace containing both at once. Either way, the combined surface and walls will be stored into the same 'WORKSPACE' matrix.

3 Robotics.m

This is the main code. Everything is controled by this script, which makes use of a few other functions described in the following sections.

3.1 Loading the needed data into matrices

At first the user is asked to choose a folder from which to load the robot and workspace. For simplicity of the code, this folder must be located in the local directory. Also, the folders must have the form 'Bitmaps_n' where n is the size of the 3-dimentional WORKSPACE matrix which stores the workspace. Then the user has the option to choose between uploading the workspace all at once or the surface and walls seperately. Those are uploaded and stored into the WORKSPACE matrix as the user selects them. Afterwards, the robot stored into 'Bitmaps_n/Sphere_robot/' is uploaded into the ROBOT matrix. For simplicity, the uploaded robot is currently set to have it's lowest point located at the coordinates (n/2,n/2,0) with respect to the workspace. Thus, after being uploaded it it shifted to match a proper location on the uploaded surface.

3.2 Setting up the initial conditions

The initial coordinates in both the workspace and the robot reference frames are then initialized. The user is also asked to enter a final time for the robot to roll as well as angular velocities w_x and w_y. The other velocities (w_z, v_x, v_y, and v_z) are simply set to 0, since we are only concerned about the robot rolling on the surface without sliding. The physical interpretation of those variables is that if we scale the radius of the robot to be 1 unit of arbitrary [distance] and the final time to

be 1 unit of arbitrary [time], then the angular velocities have units of [rad/time] and the linear velocities have units of [distance/time]. Finally, using the known properties of the spherical robot, some matrices used in the contact equations are initialized right away.

3.3 Rolling

Then the robot is left to roll until the 'current_time' reaches 't_final'. This is the first loop. Within that loop, we have the second loop which let's the robot roll until a boundary is hit. At each time step the surface is approximated to that of a plane using the 'Get_slopes.m' function. Then a new contact point is computed using the 'Get_bigZ_bigC.m' and the 'Get_hit_cont_pts.m' functions. Furthermore, the program checks whether a boundary was hit using the output from 'Get_hit_cont_pts.m'. The same process repeats until a boundary is hit. When that happens, the codes jumps out of the second loop.

3.4 Bouncing and reorientating

After hitting a boundary, the code searches within r/10 to compute a normal and tangents to the wall at the point where the boundary was hit. Then using Snell's law, the bouncing angle is computed. This angle is converted into angular velocities, assuming no loss of kinetic energy. The robot is left to roll in that new bouncing direction for '*move*' number of time steps, where we currently have $move = r/5$. Afterwards, using the previously computed tangent vectors to the wall, the robot is reoriented into a direction parallel to that of the tangent plane to the wall, at the point where the boundary was hit. This is done by adjusting the angular velocities. Finally, the program loops back to the beginning of the first loop to repeat the same process over again.

3.5 Plotting

At last, when the final time is reached, several plots are output showing the path of the robot on the surface as well as a plot of the contact coordinates as a function of time.

4 Get_bigZ_bigC.m

This piece of code is straight forward computationally, but has all the math hidden in it. The computed matrices bigZ and bigC are such that $bigZ \cdot \dot{c_{sol}} = bigC$ The derivations to obtain those matrices follow from the change of coordinates between the real velocities V_{of} and the contact velocities V_{lol_f}, described in the change of coordinates section, and equation (5.28) of A Mathematical Introduction to Robotic Manipulation, by Murray, Li, and Sastry. Once those matrices are derived, the main 'Robotics.m' code can use Euler's approximation method: $c_{sol}(t+1) = \Delta t \cdot \dot{c_{sol}} + c_{sol}(t) = \Delta t \cdot bigZ^{-1} \cdot bigC + c_{sol}(t)$.

5 Get_hit_cont_pts.m

The purpose of this function is to find the new contact point with the surface after the robot has moved according to the contact equations. Furthermore, it looks for any contact point with the boundary: the hit_pt. Due to the discontinuous nature of the robot, surface, and walls, several matches between the robot and the workspace are at first allowed, with a maximum spacing of $delta = r/10$. If two matches are a distance greater than *delta* apart then they are considered to be a part of two different physical contact points. If more than two physical contact points are found

then the program terminates, since the contact equations do not allow for more than one contact point with the surface and one contact point with the walls. If only one contact point is found, then it is taken to be the new contact point between the robot and the surface. If there are exactly two contact points then the closest to the previous contact is the new contact point while the second one is the hit_pt, i.e. the point of contact with the boundary. Finally, if there is no contact point found between the robot and the workspace, contact has been lost. This arises in the case where Euler's approximation was not good enough, or the approximation of the surface to that of plane was too inaccurate. To make up for that, the robot is shifted in the x, y, and z directions by up to $delta\_move = round(r/10)$ until a contact point can be found. If after that no contact point is found the program terminates.

6 Get_slopes.m

This program uses the contact point between the robot and the surface and looks at nearby points (up to r/5 away) to determine the partial derivatives z_x and z_y to the surface, at that particular contact point. Unfortunately, due to the discontinuous nature of the surface, the program might erroneously find an infinite partial derivative. In this case, the partial derivative is reset to its previously found value. Thus, the robot rolls on as if there were no change in the partial derivative. Those partial derivatives are returned, and can be used by the main code to approximate the local surface to that of a plane.

7 Generating surfaces

Surfaces can be generated using the Bitmaps.m code found in each 'Bitmaps_n' folder. There, the spherical robot is generated as well as a few sample surfaces. These objects are saved as n 2-dimentional $n \times n$ images (slices) into different folders. The variable 'n' is the size of the 3-dimentional matrices which will store the robot and the workspace. The variable '$size\_workspace$' is a measure of the physical size of the workspace and is defined as $size\_workspace = n/r$ where r is the radius of the spherical robot. Finally, new surfaces and walls can easily be generated by defining them in their parametric form: (X,Y,Z) = (f(s,t),g(s,t),h(s,t)).