Assignment 1 Report

Gabriel Ruszala (gdr37) and Jessica Scheier (jls772)

September 24, 2024

1 Validating Rotations

1. *check_SOn()*: takes in an array and a float value of 0.01 called epsilon. It returns a boolean value. Let the array representing the matrix be denoted by R. There are three conditions for R to be valid. First, R must be a square matrix of either size 2x2 or 3x3 for SO(2) or SO(3). Second, this must be true, where T means transpose:

$$R^T * R = I$$

   This implementation uses epsilon as a parameter in the numpy method *allclose()* to determine how closely accepted the matrix should be. Finally, *check_SOn()* checks whether the determinant of R is equal to 1, similarly using the numpy library method *isclose()* to determine whether the determinant of R is equal to 1 within an absolute tolerance of epsilon.

2. *check_quaternion()*: takes in an array representing a vector and a float value of 0.01 called epsilon. It returns true or false. There are two conditions to be met for the quaternion to be valid:
   1. The vector must be a 4 dimensional vector.
   2. The vector's magnitude must be 1.

   For number 1, this method returns false if the array is not a 1-dimensional array of 4 elements or a 2-dimensional array of 4 rows and 1 column, using the *shape()* method. For number 2, the magnitude of the vector is calculated using the numpy library method *linalg.norm()*. Then, using the numpy method *isclose()* to determine whether the magnitude is close enough to the value 1, using epsilon as a parameter for tolerance. Finally, the result of this check is returned as a boolean value of true or false.

3. *check_SEn()*: takes in the same inputs and returns the same output as the previous two methods. For this rotation matrix to be valid, n must be 2 or 3 and the matrix must be square. The matrix is of size $(n+1)*(n+1)$, so the implementation subtracts 1 from the shape of the matrix to get n. Then, it checks if the shape is not square or if n is not 2 or 3 and returns false if so. Next, the matrix contains a translation vector and a rotation matrix in SO(n), so you have to extract the rotation matrix portion and

call *check_SOn()* to determine whether the rotation matrix is indeed valid. Finally, the last row is of the form [0, 0, ... 1] for homogeneity. The last row of the matrix is extracted and compared to an example array it is supposed to be of, using n and the numpy method *zeros()*. If this last row does not fit closely enough to the test row using epsilon and the numpy method *allclose()*, it returns false. Otherwise, it returns true as that is the last condition necessary to be checked.

1.1 Extra

1. *correct_SOn()*: this corrects the rotation matrix if deemed incorrect. First, check whether the matrix is correct using *check_SOn()*, providing it the matrix and epsilon. If it outputs true, return the matrix because it does not need correcting. If the matrix is not orthogonal, use singular value decomposition to orthogonalize the matrix. Get the *U*, *sigma*, and *Vt* components from the matrix using the *linalg.svd()* method from the numpy library. Compute the dot product between *U* and *Vt*. Then, calculate the determinant of the matrix. If the determinant of the matrix is -1, negate the last column to flip the determinant. The matrix is returned at the end of the method.

2. *correct_quaternion()*: this corrects the vector if deemed incorrect. First, check if the vector is a valid quaternion using the method *check_quaternion()* and if the result is true and it is determined to be correct, simply return the vector. If the vector fails the check, the vector should be normalized such that the magnitude is 1. First, the magnitude should be calculated using the numpy method *linalg.norm()*. If the magnitude is not equal to 1, divide the vector by its magnitude and return the resultant vector.

3. *correct_SEn()*: this corrects the matrix if deemed incorrect. First, get the size of the matrix using the *shape()* method. Extract the rotation matrix and correct it to be orthogonal by calling the method *correct_SOn()*. Then finally, ensure the last row of the matrix is of the form [0, 0, ... 1] for homogeneity and correct that by setting it to an array made up of zeros with a 1 at the end. The array of zeros is created using the numpy method *zeros()*. The corrected matrix is returned at the end of the method.

- 2x2 Test Case for *correct_SOn()*:

$$\begin{pmatrix} 0.79314706 & 0.38616734 \\ 0.16134404 & 0.81168602 \end{pmatrix}$$

check_SOn() outputs this matrix as False. Run *correct_SOn()*. This outputs the matrix as:

$$\begin{pmatrix} 0.99032932 & 0.13873661 \\ -0.13873661 & 0.99032932 \end{pmatrix}$$

check_SOn() outputs this matrix as True.

- 3x3 Test Case for *correct_SOn()*:

$$\begin{pmatrix} 0.50185332 & 0.03149489 & 0.67248774 \\ 0.52993567 & 0.60671833 & 0.18681610 \\ 0.02759163 & 0.97507689 & 0.73246010 \end{pmatrix}$$

check_SOn() outputs this matrix as False. Run *correct_SOn()*. This outputs the matrix as:

$$\begin{pmatrix} 0.53711175 & -0.34794612 & 0.76840384 \\ 0.77047304 & 0.57316649 & -0.27901874 \\ -0.34333985 & 0.74189869 & 0.57593757 \end{pmatrix}$$

check_SOn() outputs this matrix as True.

- Test Case for *check_quaternion()*:

$$\begin{pmatrix} 0.78175724 \\ 0.08413272 \\ 0.01788872 \\ 0.66339191 \end{pmatrix}$$

check_quaternion() outputs the vector as False. Run *correct_quaternion()*. This corrects the vector to:

$$\begin{pmatrix} 0.75980037 \\ 0.08176972 \\ 0.01738628 \\ 0.64475951 \end{pmatrix}$$

check_quaternion() outputs the vector as True.

- 2x2 Test Case for *correct_SEn()*:

$$\begin{pmatrix} 0.2564426 & 0.07159408 & 0.97111483 \\ 0.80524686 & 0.02267503 & 0.37787368 \\ 0.24581518 & 0.80671959 & 0.51262039 \end{pmatrix}$$

check_SEn() outputs the matrix as False. Run *correct_SEn()*. This corrects the matrix to:

$$\begin{pmatrix} 0.25760434 & -0.96625049 & 0.97111483 \\ 0.96625049 & 0.25760434 & 0.37787368 \\ 0 & 0 & 1 \end{pmatrix}$$

check_SEn() outputs this matrix as True.

- 3x3 Test Case for *correct_SEn()*:

$$\begin{pmatrix} 0.69942428 & 0.57542726 & 0.83226602 & 0.26638623 \\ 0.51703353 & 0.77526382 & 0.78676897 & 0.66231425 \\ 0.99073504 & 0.09942187 & 0.47534292 & 0.37379139 \\ 0.80602158 & 0.33608584 & 0.27536248 & 0.50782669 \end{pmatrix}$$

check_SEn() outputs the matrix as False. Run *correct_SEn()*. This corrects the matrix to:

$$\begin{pmatrix} 0.05897287 & -0.08140945 & -0.99493452 & 0.26638623 \\ 0.18321096 & 0.98062241 & -0.06937889 & 0.66231425 \\ 0.98130319 & -0.17819144 & 0.07274522 & 0.37379139 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

check_SEn() outputs this matrix as True.

2 Uniform Random Rotations

1. *random_rotation_matrix(naive: bool)*: takes in a boolean value. If naive is true, generate random euler angles: $alpha = random(0, 2pi)$, $beta = random(0, pi)$, $gamma = random(0, 2pi)$. Then, convert them to rotation matrices:

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos(\gamma) & -\sin(\gamma) & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Next, compute the matrix multiplication in the order of x, y, and z.

$$R = R_z \cdot R_y \cdot R_x$$

Finally, return the resultant matrix R at the end of the if-statement. For the else branch of the statement, this means the naive boolean value is false, so the method should implement the function in Reference 1 of the assignment write-up (source 6). Generate three random x1, y1, and z1 variables between 0 and 1 using *random.uniform()*. Next, using these variables, pick a rotation about the pole, a direction to deflect the pole, and the amount of pole deflection:

$$\theta = 2\pi x_1$$

$$\phi = 2\pi x_2$$

$$z = x_3$$

Construct a vector for performing the reflection:

$$V = \begin{bmatrix} \cos(\phi)\sqrt{z} \\ \sin(\phi)\sqrt{z} \\ \sqrt{1-z} \end{bmatrix}$$

Then, construct the rotation matrix by rotating about the z-axis first:

$$R_z = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and rotate the z-axis about a random orientation:

$$M = \left(2V \cdot V^\top - I_3\right) \cdot R_z$$

Finally, return the matrix M at the end of the method.

Included in our folder submission are visualizations of the distribution of 1000 random rotations on a sphere, using the algorithm given in the assignment. One visualization portrays the distribution of rotations with the naive solution, and the other visualization portrays the distribution of rotations with the non-naive solution. The rotation distribution generated by the naive solution is visibly not uniform random (there is clumping in some areas and sparseness in other areas). The rotation distribution generated by the non-naive solution is visibly more uniform random.

2.1 Extra

The method *random_quaternion()* that takes in a boolean value. If it is true, it generates random euler angles and converts to a rotation matrix. If it is false, it implements the algorithm defined in the second reference provided (cited at number 8 of the Sources section). If true, random yaw, pitch, and roll angles are generated. Then, three rotation matrices are generated to represent the rotations around each axis by their corresponding angle. Finally, the naive solution is returned as the combined rotation matrix of these three matrices to describe the orientation in 3D. If it is false, it implements the pseudocode from source 8, where two random angles are generated and sigma values are calculated using a random number s generated between 0 and 1. Then, the quaternion components are calculated as follows:

$$w = \cos(\theta_2) \cdot \sigma_2$$

$$x = \sin(\theta_1) \cdot \sigma_1$$

$$y = \cos(\theta_1) \cdot \sigma_1$$

$$z = \sin(\theta_2) \cdot \sigma_2$$

The result is returned as an array (w, x, y, z).

3 Rigid Body in Motion

1. *interpolate_rigid_body()*: uses the given start pose and goal pose to output a sequence of poses that start at the first position and end at the final position. I had debated between using linear interpolation and cubic interpolation (polynomial time scaling), and ultimately chose to implement cubic interpolation to generate a smoother trajectory of the path the rigid body would take. Essentially, you utilize the formula:

$$s(t) = a_0 + a_1 * t + a_2 * t^2 + a_3 * t^3$$

   Using the numpy library method *linspace()* to get evenly spaced intervals of t, use t and apply this formula to x, y, and θ. The difference in angles between θ_0 and θ_1 should be normalized before this. The result is stacked into an array and transposed so it is formatted correctly. This path is returned at the end of the method.

2. *forward_propagate_rigid_body()*: uses the given start pose and a plan of velocities and durations to propagate the rigid body. The path is initialized as starting with the start pose. For each velocity and duration in the plan, extract the velocity components: V_x, V_y, and V_θ. Let dt represent the change in time. For every t in the range of 0 and duration, stepping at every $dt = 0.1$, compute the rotation matrix and translation vector using the extracted x, y, and θ values and multiply them together. This computation is multiplied by the change in time to represent the change in x and y (dx and dy). The extracted V_θ is multiplied by dt and added to the initial θ. The θ should be normalized again. In essence, these two loops capture the change in x, y, and θ at each change in time and modify the pose of the previous x, y, and θ and save those modifications, or poses, in a path array. This path array is returned at the end of the method.

3. *visualize_path()* visualizes a given path generated by either of the two previous methods. It uses the FuncAnimation library from matplotlib to animate the frames. The 2D planar environment is created by creating a figure with an ax. The ax x-limit and y-limit are set to [-10, 10] using the methods *set_xlim()* and *set_ylim()*. The data of each x, y, and θ are extracted from the path. For example, to extract all the x-values, you want to access all the rows of the first column (i.e. array[:,0]). The path is plotted on the ax using the x and y values with a blue dashed line. The body of the robot is declared with an outline in red. No data is set to it yet because that is done in the initialize function right after. The second function within the method animates the frames. The length and width of the robot is set in this function. Of each frame, the x, y, and θ values are extracted. The robot is drawn at the origin using the length and width as x and y coordinates, with an array called *robot* to store these coordinates. There are four corners, but five elements are in the array to ensure the robot is fully drawn. The array is transposed so the first row is all the

length coordinates and the second row is all the width coordinates. Next, the rotation matrix is computed using θ:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

The robot is rotated by computing the matrix multiplication between the rotation matrix R and the array *robot*, representing the coordinates of the robot's body. Then, by first creating the array of the x and y values (i.e. np.array([[x],[y]])), the robot is translated by this numpy array added with the product of R and the *robot* array. The robot body's data is set to this data of the translated and rotated robot, and the robot body is returned at the end of the animate function. Finally, all the functions are called to animate the plots like a slideshow, iterating through the frames and generating the animation this way. The last two lines of code show the plot and save it to the file named *component_3_test.gif*. Two visualizations are saved in our submission titled *component_3a.gif* and *component_3b.gif*, where 3a represents the visualization of the interpolated path and 3b represents the visualization of the propagated path.

4 Movement of an Arm

1. *interpolate_arm(start, goal)*: This algorithm is given a vector containing the start position of the arm, and the goal position of the arm. The number of intermediate steps is set to 100. The algorithm simply calculates the intermediary positions between the start and goal position, based on the number of steps (in this case, 100). The intermediary positions are calculated through this equation:

$$\theta_t = \theta_{\text{start}} + t * (\theta_{\text{end}} - \theta_{\text{start}})$$

where t is a number between 0 and 1. There are *steps* intermediary poses for each joint to calculate, evenly spaced between t = 0 and t = 1. The function calculates all the intermediary angles for each joint through the above equation, compiles them into a path (a list of tuples), and returns the path.

Contained in our folder submission is a visualization of *interpolate_arm()*, showing the robotic arm moving between start pose (0,0) and end pose $(\pi, \pi/2)$ over 100 steps.

2. *forward_propagate_arm(start_pose, plan)*: This method also calculates a path for the robotic arm, albeit in a different way. Each entry in the plan contains a velocity corresponding to a joint in the arm, and a duration. For each entry in the plan, these angular velocities are applied to the two joints, for the set duration. This allows us to see where the robotic arm would move if we applied velocities to the arm over periods of time, rather

than calculating evenly spaced out intermediary poses. The equation used to calculate a new orientation for a joint is as follows:

$$\theta = v * duration$$

We record all of the intermediary positions for the two joints based on the given plan, compile them into a path, and return the path.

Included in our submission is a simple visualization of *forward_propagate_arm()*, showing how we apply velocities of $\pi/4$ radians/sec in 1 second durations towards the first link individually, then the second link individually, and finally both links simultaneously. This method is of course capable of calculating paths based on all kinds of plans with different velocity and duration parameters.

3. *visualize_path(path)*: Based on the poses in the input path, this method reconstructs the configuration of the robot arm using forward kinematics and visualizes the input path in a GIF. Since we know the lengths of each link, and we know the joint angles of the robot based on the path, we can reconstruct the configuration of the robot arm using transformation matrices. First, we find the positions of the joints and links relative to their preceding robotic part and create transformation matrices describing that position. Then, to find the position of the joints and links relative to the base, we multiply the transformation matrices together starting from the base in sequence up to the robot part we are searching for. We used FuncAnimation included in matplotlib to animate the calculated robot configurations in each position. Two visualizations are saved in our submission illustrating the interpolated path and the propagated path of this 2-link planar manipulator in files titled *component_4a.gif* and *component_4b.gif*, respectively.

5 Sources

1. Modern Robotics: Mechanics, Planning, and Control (Chapter 3, pgs. 68-70)
2. Modern Robotics: Mechanics, Planning, and Control (Appendix B, pgs. 581-582)
3. Modern Robotics: Mechanics, Planning, and Control (Chapter 3, pgs. 89-90)
4. Modern Robotics, Chapter 3.3.1: Homogeneous Transformation Matrices
5. 3.1 Transformation Matrices
6. Fast Random Rotation Matrices in Graphics gems III (IBM version)
7. Effective Sampling and Distance Metrics for 3D Rigid Body Path Planning (pg. 2)
8. Correct Explanation of Yaw, Pitch, and Roll Euler Angles with Rotation Matrices and Python Code
9. Effective Sampling and Distance Metrics for 3D Rigid Body Path Planning (pg. 3)

10. Modern Robotics: Mechanics, Planning, and Control (Chapter 9, pgs. 329-330)
11. Modern Robotics, Chapter 9.3: Polynomial Via Point Trajectories
12. Modern Robotics: Mechanics, Planning, and Control (Chapter 3, pgs. 64-65)
13. Animations using Matplotlib