CS498 Intelligent Robotics, Spring 2020
Homework 1: Transformations and Grid Search

Due date: 2/5/2020

Instructions: This assignment is to be completed *individually*. You are allowed to discuss the problems with your classmates, but all work must be your own. You are not permitted to copy answers, take written notes, or look up online solutions.

All coding will be done on the Klamp't Jupyter system. Setup is as follows:

1. Login and open up a terminal under the `Run` tab.
2. If this is your first time, run `git clone https://github.com/krishauser/cs498ir_s2020.git; cs498ir_s2020/setup.sh`. If this is not your first time, run `cs498ir_s2020/update.sh` to get the updated homework code. Go ahead and close the terminal tab, and return to the file browser. The homework code is in `cs498ir_s2020/HW1`.
3. When you run the notebooks, check the Kernel menu to make sure you are using the `cs498ir-virtualenv` kernel.
4. Make sure to periodically backup your code to your local machine. To do this, select the notebook in the file browser and choose `Download`. (We don't expect to have the server die, but Engineering IT won't make any guarantees!)

You may implement your calculations manually using the elementary operations found in the Python math module, or you may use the functions in the klampt.vectorops or klampt.so2 module.

To submit, upload your written answers in the form of an image or PDF, and programming answers in the form of Jupyter notebook files (*.ipynb), onto the class http://learn.illinois.edu site.

Written Problem 1: 2D Rotations

Recall that a 2D rotation about the origin by angle θ can be represented as a matrix $R(\theta)$ given by

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

A. Using standard trigonometric identities, prove that $R(\theta)^{-1} = R(-\theta) = R(\theta)^T$

B. Using standard trigonometric identities, prove that the product of two rotation matrices is equal to the rotation matrix defined by the sum of the individual angles. Specifically, prove that $R(\theta_1)R(\theta_2) = R(\theta_1 + \theta_2)$. (Incidentally, this proves that rotation order does not matter in 2D.)

Written Problem 2: 2D Rigid Transforms

Recall that a rigid transform in 2D can be represented by a rotation about the origin by angle θ followed by translation by a 2D vector $\vec{t}$. Specifically, we will say a rigid transform $T \equiv T(\theta, \vec{t})$ is an operator $T: R^2 \to R^2$ defined by

$$T\vec{x} = R(\theta) \cdot \vec{x} + \vec{t}$$

A. Prove that all rigid transforms preserve distance, that is, $\|T\vec{x} - T\vec{y}\| = \|\vec{x} - \vec{y}\|$ for all rigid transforms $T \equiv T(\theta, \vec{t})$ and points $\vec{x}, \vec{y} \in R^2$.

B. Prove that the inverse of a rigid transform is also a rigid transform. Derive the parameters of the inverse transform. Specifically, show that $T(\theta, \vec{t})^{-1} = T(\theta', \vec{t}')$ for some values θ' and $\vec{t}'$, and give an expression of θ' and $\vec{t}'$ in terms of θ and $\vec{t}$. (As usual the inverse operator is defined such that $\vec{y} = T\vec{x}$ if and only if $\vec{x} = T^{-1}\vec{y}$)

C. Prove that the composition of two rigid transforms is also a rigid transform, and derive the parameters of their composition. Specifically, let $T_1 \equiv T_1(\theta_1, \vec{t}_1)$ and $T_2 \equiv T_2(\theta_2, \vec{t}_2)$ be two rigid transforms. The composition $T = T_1 \circ T_2$ is defined such that $T\vec{x} = (T_1 \circ T_2)\vec{x} = T_1(T_2\vec{x})$ for all $\vec{x}$. Find the orientation and translation parameters $(\theta, \vec{t})$ corresponding to T.

D. Suppose we wanted not to rotate about the origin, but about some rotation center $\vec{c}$. How can such a transform be represented in the form $R(\theta) \cdot \vec{x} + \vec{t}$?

Written Problem 3: Homogeneous coordinates

Thinking of rigid transforms as operators is somewhat clumsy. Homogeneous coordinates allow us to think of them simply as matrices. Let us define the "homogenizer" operator $\hat{\cdot}$ as the function that transforms a 2D vector $\vec{p} = [x, y]^T$ to homogeneous coordinates $\hat{p} = [x, y, 1]^T$. Let us also define the "de-homogenizer" operator $\check{\cdot}$ such that it transforms a 3D vector $\vec{p} = [x, y, w]^T$ to 2D coordinates by division by the homogeneous coordinate w: $\check{p} = \left[\frac{x}{w}, \frac{y}{w}\right]^T$.

A. Prove that the operator $T(\theta, \vec{t})$ can be represented by a linear transform in homogeneous coordinates via the matrix

$$\hat{T}(\theta, \vec{t}) = \begin{bmatrix} \cos\theta & -\sin\theta & t_x \\ \sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix},$$

meaning that if $\vec{y} = \hat{T}(\theta, \vec{t}) \cdot \hat{x}$, then $\check{y} = T(\theta, \vec{t})\vec{x}$.

B. Prove that matrix multiplication in homogeneous coordinates is equivalent to composition of transform operators, that is, $\hat{T}_1 \cdot \hat{T}_2 = (\widehat{T_1 \circ T_2})$

C. When applying a rigid transform to a *directional* quantity like a camera's view direction (as opposed to a *positional* quantity like the coordinates of a point), it is inappropriate to add the translational component. One way to represent this discrepancy would be to define a new operator that works on direction vectors (let's call it $\star$) such that $T(\theta, \vec{t}) \star \vec{d} = R(\theta) \cdot \vec{d}$. Another way to do this would be to define homogeneous coordinates of directional quantities such that the homogeneous coordinate is zero, i.e., if $\vec{d} = [x, y]^T$ is a direction, then $\hat{d} = [x, y, 0]^T$.

Prove that multiplication in homogeneous coordinates does the same thing as the $\star$ operator, and furthermore, it preserves the convention that directions have 0 as their homogeneous coordinates.

Written Problem 4: 3D Rotations

Rotations about the x, y, and z axes by angle θ can be represented as 3x3 matrices $R_x(\theta)$, $R_y(\theta)$, and $R_z(\theta)$ given by

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

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

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

A. Given an example to show that in general, rotation composition in 3D is not symmetric. I.e., find two 3D rotation matrices R_1 and R_2 such that $R_1 \cdot R_2 \neq R_2 \cdot R_1$.

B. A common *camera coordinates* convention is to represent 3D points in such that $+x$ is the "right" direction (positive to the right of the image center), $+y$ is the down direction (positive below the image center), and $+z$ is the "forward" direction (i.e., positive in front of the camera). The origin lies at the focal point.

   A common *egocentric coordinates* convention in mobile robotics represents $+x$ as the forward direction of the robot (from its point of view), $+y$ toward the left of the robot, and $+z$ pointing upward (from the ground toward the sky).

   Suppose our robot holds the camera pointing straight forward, that is, the camera is oriented such that:

   1. The camera's forward direction is level with the ground plane.
   2. The camera's forward direction points directly along the forward direction of the robot.
   3. The sky is up and the ground is down in the camera image.

   Give the rotation matrix R such that matrix multiplication $\vec{x}_e = R \cdot \vec{x}_c$ computes the egocentric coordinates $\vec{x}_e$ of the 3D point whose camera coordinates are $\vec{x}_c$.

C. The robot can freely rotate about its Z axis and translate on the plane, and the camera's origin is placed at coordinates (3cm,0cm,6cm) in the robot's local coordinate frame. Give a 3D rigid transform T, giving a transform of the robot in world coordinates, that the robot should move to so that 1) its camera is placed at (1.5m,1.2m,0.06m) and 2) is pointing perpendicularly toward a wall whose outward normal direction has heading 135°.

Programming Lab A: Direction and Magnitude

Lab 1a displays two endpoints of a line segment, which move according to the functions `source_motion(t)` and `target_motion(t)`. The on-screen display incorrectly calculates the length and angle of this segment, printing out 0 and 0, respectively. Your job is to properly calculate these values.

Complete the function `lab1a(point1,point2)` which should return a tuple `(length,angle)` describing the magnitude and heading of the vector from `point1` to `point2`.

Hint: run the `selfTest()` function to verify that you are performing the calculations properly.

Programming Lab B: Rotation of Points

Lab 1b displays a clock-like object with a center point and two peripheral points. They should be rotating simultaneously about the center point, but the display is broken.

Complete the function `lab1b(point,angle)` which should return a new point rotated about the origin by the given angle. Both input and output points are given by a length-2 tuple (or list) of floats.

Hint: run the `selfTest()` function to verify that you are performing the calculations properly.

Programming Lab C: Composition of Rigid Transformations

Lab 1c displays a car-like vehicle. By default the car will use automatic controls (chosen at random by default), but you can also drive it around using the arrow buttons.

The display of the car is broken! It does not properly rotate, and the front wheels do not properly indicate the steering angle.

1. Complete the function `get_rotation_matrix(xform)`. Here an xform is a tuple (tx,ty,angle) indicating the translation and rotation of a rigid body transformation. `get_rotation_matrix` should return a 2x2 matrix so that matrix-vector multiplication with a point should give the rotated point (i.e., duplicate the behavior of `lab1b()`). When this is correctly implemented, the car body should rotate as the car turns.

2. Complete `lab1c(xform1,xform2)`. This function composes two xforms together so that applying the result to a point is equivalent to first applying xform2, and then applying xform1. When this is correctly implemented, the front wheels should rotate to indicate steering angle.

To help you debug, you can change the automatic controls by editing the `control()` function. Return values 'up' and 'down' increase and decrease the velocity, and 'left' and 'right' increase and decrease the steering angle.

Hint: run the `selfTest()` function to verify that you are performing the calculations properly.

Programming Lab D: Grid Search

In this assignment you will implement a grid search motion planner for a point robot. A basic search on an integer 8-connected grid is provided for you, as well as functions for visualizing the planned path. Run the `selfTestX()` functions to verify that you are performing the calculations correctly.

1. *Discretization*. The first problem is that the point robot moves in continuous space, while grids operate in integer coordinates! Implement a grid search motion planner in the `grid_planner` function so that it searches over a box, with lower bound `bmin=(xmin,ymin)` to upper bound `bmax=(xmax,ymax)`. Perform the search between the nearest points on the continuous grid to the start and goal before searching, and don't forget to produce a path that connects to the start and goal!
2. *Collision testing*. The `in_obstacle(p)` function tests whether the point `p` is within one of the circular obstacles in `obstacle_list`. Modify the planner to avoid obstacles by *modifying graph that is searched*. Don't worry about collisions along edges.
3. *Optimal paths*. The `cost` argument to `dijkstras` isn't currently being used, which creates somewhat odd, suboptimal, diagonal artifacts around obstacles. Create a cost function `grid_cost((i1,j1),(i2,j2))` so that the value is the Euclidean distance between (continuous space) grid points.
4. *Efficiency using A* search*. Set bmin=(-10,-10), bmax=(10,10), N=100, M=100 and observe how long the search takes. We should be able to do better if the start and goal are close together using A* search on an implicit graph. Read the documentation of the `astar_implicit` function to determine the format of the `successors` argument. Implement `my_cost` and `successors` so that it dynamically performs collision detection in the midst of searching. Next, implement a Euclidean distance heuristic in `my_heuristic`, and make sure it improves the running time in `selfTest4`.
5. *Extra credit*. Implement a version of A* that implicitly constructs the search graph. It must take less than O(MN) time if the start and goal can be connected by a short path, and O(MN) time if the path is long.

The files graph.py, grid.py, and search.py provide utility functions for you to use, and are imported in the first cell. These modules define:

- `graph`: implements `AdjListGraph`, a directed graph data structure that uses an adjacency list for edge storage.
- `grid`: implements `grid_graph(M,N,diagonals=False)`, a function for constructing an `AdjListGraph` on integer coordinates `(i,j)`, with `i=0,…,M-1` and `j=0,…,N-1`.
- `search`: implements Dijkstra's algorithm on a graph, and A* search on a graph or an implicit graph.

Hint: write `help(THING)` in a cell to get the documentation for `THING` (module, class, or function).