15-780: Problem Set #4

April 21, 2014

1. **Image convolution [10 pts]** In this question you will examine a basic property of discrete image convolution. Recall that convolving an $m \times n$ image $J \in \mathbb{R}^{m \times n}$ with a $p \times r$ filters $G \in \mathbb{R}^{p \times r}$ (for simplicity, we'll assume p and r are both odd) results in a new image $(J * G) \in \mathbb{R}^{m-p+1 \times n-r+1}$ where

$$(J * G)_{i,j} = \sum_{k=1}^{p} \sum_{\ell=1}^{r} J_{i+k-1,j+\ell-1} G_{k,\ell}.$$

For all the questions below, you can assume that J is large enough so that the (potentially multiple) convolutions all result in a valid return image (e.g., you can assume $m > p$ and $n > r$ anytime we perform a convolution)

(a) Is discrete image convolution a commutative operation? That is, if we have two filters $G_1 \in \mathbb{R}^{p_1 \times r_1}$ and $G_2 \in \mathbb{R}^{p_2 \times r_2}$, does it hold that, using the definition above

$$(J * G_1) * G_2 = (J * G_2) * G_1?$$

Prove or give a counterexample.

(b) For image processing, one of the less-than-ideal properties of the convolution as defined above is that the resulting images are smaller than the original image. One of the ways to solve this is by "zero-padding" the original image J with $(p-1)/2$ zeros above and below, and $(r-1)/2$ zeros on the sides. So for example, to convolve and image $J \in \mathbb{R}^{m \times n}$ with a filter $G \in \mathbb{R}^{3 \times 5}$, we would first form the matrix

$$\begin{bmatrix} 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & J_{11} & \cdots & J_{1n} & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & J_{m1} & \cdots & J_{mn} & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 0 \end{bmatrix}$$

and then convolve this image with G. This results in a new image that is the same size as the original J. Denote this procedure (zero-padding the left-hand matrix, then convolving by $\bar{*}$), is it the case that this operation is commutative? That is, does it hold that

$$(J \bar{*} G_1) \bar{*} G_2 = (J \bar{*} G_2) \bar{*} G_1$$

for any $J \in \mathbb{R}^{m \times n}$, $G_1 \in \mathbb{R}^{p_1 \times r_1}$, and $G_2 \in \mathbb{R}^{p_2 \times r_2}$? Prove or give a counterexample.

2. **Simple Temporal Network [20 pts]** In this written assignment, you are to answer the following questions about a simple temporal network (STN):

   (a) Draw an STN using the notation on slide 18 of the scheduling lecture for the following set of tasks and constraints:
   - There are three tasks: Task A, Task B, and Task C.
   - Task A must start at tick 5 or later and has a duration of 10 ticks.
   - Task B must start at least 15 ticks after Task A completes and has a duration of 20 ticks.
   - Task C must start immediately after Task B ends, must complete before tick 75, and has a duration of 10 ticks.

   (b) In the above example, what are the earliest start time, latest start time, earliest finish time, and latest finish time for the three tasks?

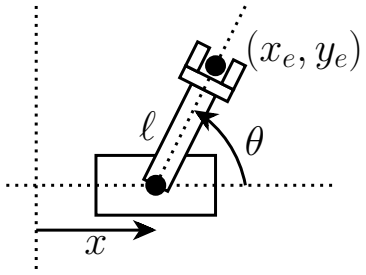
   (c) How would you update the STN in the example above to reflect that the duration of Task B has changed to 50 ticks? What would happen when you make that change and why?

   (d) At a conceptual level, how is an all-pair-shortest-paths algorithm used to compute the lower bound for a time point? How is it used to compute the upper bound?

   (e) What type of schedule is supported by an STN? (Hint: note the type of schedule that provides all the bounds you provided in Question 2b for each constituent task)

   (f) Suppose that a timeline for resource R is represented in an STN as two tasks, A and B, with there being a <0,∞> constraint from the endpoint of A to the start point of B to represent the sequencing of the tasks resulting from using the same resource. How could the STN be used to check if a new task, C, can be inserted on the timeline for R between tasks A and B?

3. **Planar revolute robot [20 pts]** This questions will consider the the following robot



   which has a state represented by the pair (x, θ) where x denotes the posit on of the robot's base along the x axis, and θ denotes the angle of the arm joint with respect to the x axis. The parameters describing the robot are ℓ, m_b, m_e, where ℓ denotes the length of the arm, m_1 denotes the mass of the robot base represented as a point mass at $(x, 0)$, and m_2 denotes the mass of the end effector represented as a point mass at the end effector location.

(a) **Forward kinematics** Denote the location of the end-effector by (x_e, y_e). Write the equations for x_e and y_e as a function of the robot state (x, θ) and parameter ℓ.

(b) **Jacobian** Write an expression for instantaneous velocity of the end-effector $(\dot{x}_e, \dot{y}_e)$ as a function of state velocities $(\dot{x}, \dot{\theta})$. Also derive the Jacobian of the system, and write these equations in terms of this Jacobian.

(c) **Inverse kinematics** Given a desired end-effector location $(\dot{x}_e, \dot{y}_e)$, determine an equation for the state variables (x, θ) to achieve this position. Discuss the scenarios where there may be zero, one, or multiple solutions to these equations.

(d) **Lagrangian** Write down the the Lagrangian of the system $L = T - U$ in terms of the states and velocities of the system $(x, \theta, \dot{x}, \dot{\theta})$.

(e) **Dynamics** Use the Euler-Lagrange equations to write down the dynamics of the system in the form of the manipulator equations

$$H(x, \theta) \begin{bmatrix} \ddot{x} \\ \ddot{\theta} \end{bmatrix} + C(x, \theta, \dot{x}, \dot{\theta}) + G(x, \theta) = \begin{bmatrix} F \\ \tau \end{bmatrix}$$

where F denotes the force applied to the base along the x axis, and τ denotes the torque applied to the robot's joint. Note that not every function in the manipulator equations *will* depend on all the inputs listed (so for example $C(x, \theta, \dot{x}, \dot{\theta})$ need not be a function of all of those variables), but these indicate the variables that each element *could* depend on.

4. **Scheduling with Set-Up Dependencies [25 pts]** In this programming assignment, you will develop a constraints-based scheduler for a modified version of the taxi-assignment problem, where a dispatcher assigns taxis to pick up fares at a particular time. In this version of the problem, all of the requests are known at the start of the day, and the job of the scheduler is to assign the cabs to service those requests. You will write the following function:

```
def scheduler(duration_matrix,cabs,requests)
```

where

duration_matrix is a list of lists providing the durations it takes to drive from any location in the input problem to any other location. Locations' IDs are consecutive integers from 0 to one less than the number of locations. For example, if there are 3 locations, they are named 0, 1, and 2. The duration matrix is a list of entries for each location, where each entry is a list of durations from the location with the name of the index of the entry to all of the locations. For example, for 3 locations, the duration matrix might look like the following: [[0,2,3],[2,0,5],[3,5,0]]. In this example, the duration from location 0 to location 1 is 2, from location 0 to location 2 is 3, and from location 1 to location 2 is 5. In this example, the reverse durations are the same, but they need not be (think one-way streets).

cabs is a dict of cabs where the key is the name of the cab and the value is a positive integer representing the number of passengers the cab can carry. An example is {'C1' : 3, 'C2' : 2, 'C3' : 4}. By definition, all cabs start at location 0.

requests is a dict where the key is the ID of the request and the value is a list with the following form: [<number-of-passengers><pickup-time>,<pickup-location>,<dropoff-location>]. <number-of-passenger> is a positive integer. <pickup-time> is a non-negative integer. The start of the day is assumed to begin at time 0. The locations are, as noted above, non-negative integers.

Your scheduler should return a dict that contains entries for all of the cabs. Each entry represents an itinerary for the cab and should have the form: [[<order-ID>,<pickup-time>], ...]. An example return value is {'C1' : [['a',10],['b',15]], 'C2' : [['d',30],['e',40]]}. In this example, C1 picks up request 'a' at time 10 and then request 'b' at time 15, and C2 picks up request 'd' at time 30 and request 'e' at time 40.

Your scheduler should enforce the following constraints:

- A cab servicing a request must be able to carry the number of passengers in the request.
- A cab must be at the pick-up location of a request to service a pick-up and at the drop-off location of the request to service the drop-off. While servicing a request, a cab must go directly from the pick-up location to the drop-off location and cannot service any other requests during that transit.
- All pickups and dropoffs have a duration of 5.
- A cab must be able to get to the pick-up location of each of its requests on its itinerary by their pick-up times.

Your scheduler should find a solution by incrementally scheduling individual requests. Each request should be scheduled by searching the existing itineraries (or timelines) of the cabs to find the time slot that introduces the least amount of travel time. To find that slot, your scheduler should map down each timeline and examine the time slots where the cab is not directly servicing a request. For each candidate time slot, your scheduler should test whether or not the new request can be inserted without violating any of the pick-up times of the downstream requests. If it can be inserted, then the slot should be saved with the additional travel time added to the timeline. Your scheduler should assign the request to the time-slot that introduces the least additional travel.

Your scheduler will be tested on a variety of input in which all requests should be feasible, i.e., schedulable. Your schedules will be checked to validate that they are consistent with the constraints.

Rapidly-exploring Random Trees [25 pts] In this problem you will implement an RRT for a n-link manipulator. In particular, you will need to implement the function

```
def rrt(theta_start, theta_goal, length, obs, epsilon):
```

This function takes as input three length-n lists, `theta_start`, `theta_goal`, and `length` (these lists specify the joint configuration of the of the start and goal locations, plus the length of each link respectively), a list of length-3 lists `obs` (each 3-element list describes the 2D location and radius of an obstacles in the form `[x,y,r]`), and scalar value epsilon. The function needs to return a list of length-n lists specifying a sequence of configurations for the manipulation such that 1) the first element of the path is equal to the start state and the end is equal to the goal; 2) no element of the path is colliding with any of the obstacles (we provide routines for collision detection); and 3) no two elements in the list are more than `epsilon` apart, i.e.,

```
np.linalg.norm(np.array(path[i]) - np.array(path[i+1])) <= epsilon
```

Such a path represents a valid (collision free) trajectory from the start to the goal state.

To implement this method, you should use a utility function included in the rrt.py template file:

```
def check_collisions(theta, length, obs):
    # returns true if the manipulator defined by theta and length
    # (both lists of n elements) collides with any of the obstances
    # in obs (defined as an list of lists [x,y,r])
```

This function will determine if `theta` is colliding with any of the obstacles or not.

You'll want to implement an RRT as described in the slides, where you'll additionally need to check that each node you add to the tree is non-colliding. Furthermore, to ensure that the tree actually reaches the goal, with some probably (say, 0.05) you will want to choose the goal state as the state to expand towards (x_{rand} in the notation of the slides). You can terminate the RRT algorithm when you actually reach this goal at a node in the tree. Then, you'll need to backtrack through the tree you constructed to find the actual path from the start state to the goal state.

You can represent the tree in many ways, but a simple way to do this is to maintain a list of tuples `(theta,parent)` where `parent` denotes the element in the list that the parent of this node; the initial (root) node can have a parent of -1. Since we ultimately just want to be able to trace back a path from the final element of the tree (which will be the goal), this simple list structure contains everything you need.

Also included with the code is a routine `draw_scene` that, given a set of configurations, will draw an animation of these configurations and the obstacle. This routine uses the matplotlib plotting library for Python, and it is not required for the assignment, but it can be helpful for debugging and for visualizing the results of your planning. Since the results of an RRT are inherently random, we can't provide "correct" paths returned from the tree, but the included `test_rrt.py` file contains code that will validate whether or not a given path is valid (as mentioned before, this just means that it has to go from the start to the goal, have distances less than `epsilon` between successive points, and have no collisions).