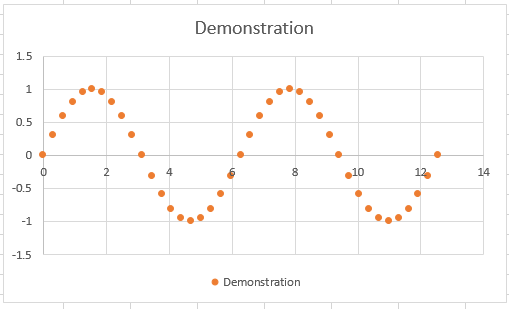
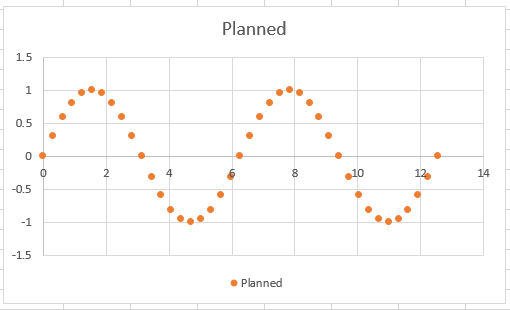
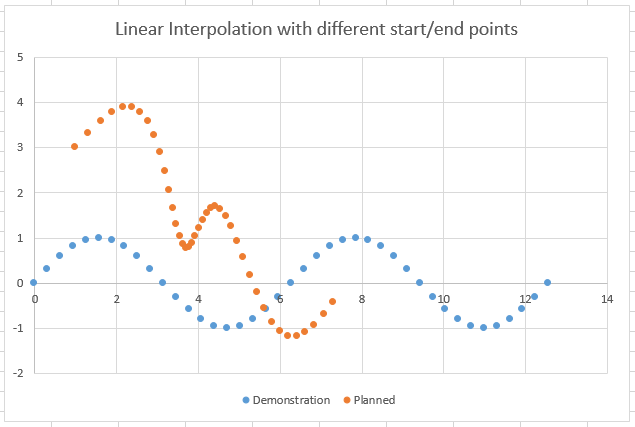
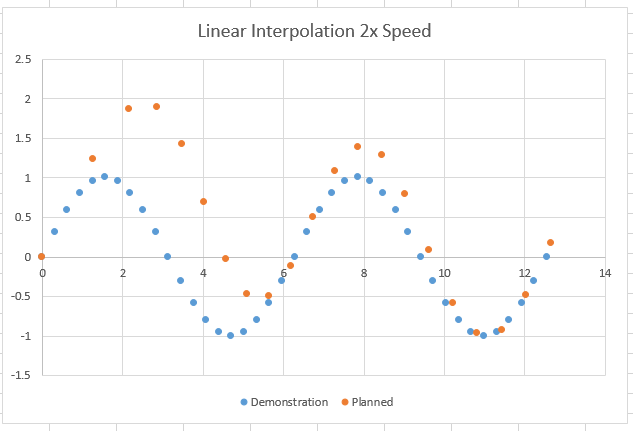
Project 1: Dynamic Movement Primitives

1. Single Noiseless demonstration
2. Single Noiseless demonstration plan

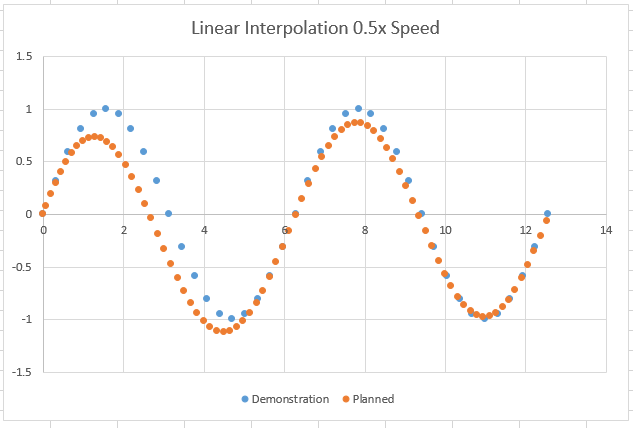


1. Single demo changed start and end goal
2. (a) 2x Faster Trajectory

The speeds are changed by changing the T value used in planning from the original 4 seconds to 8 seconds to slow it down by a factor of 2 and 2 seconds to speed it up by a factor of 2.

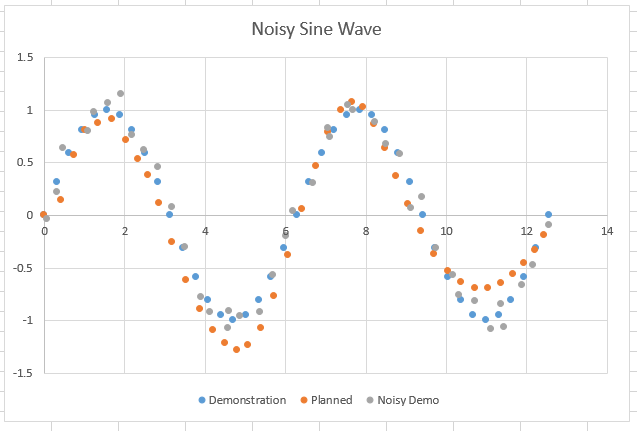


(b) 0.5x Speed Trajectory



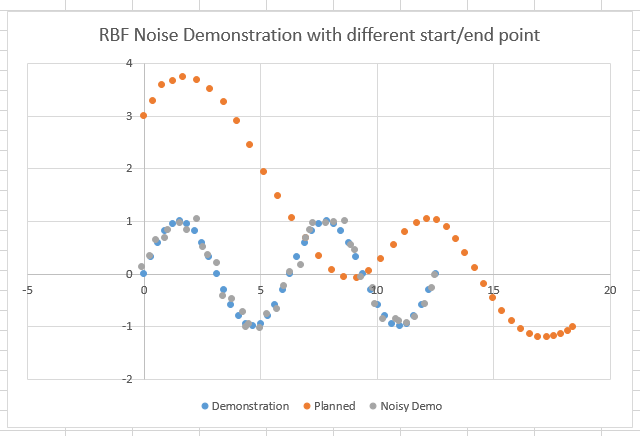
1. Multiple Noisy Demos and Planned Trajectory (timing data included in excel file)

I chose 42 functions to approximate the solution (1 centered on each data point). I used K = 70 and alpha = -1\*math.log(0.01). The spacing of the functions was done by setting a center for each of the data points which naturally made it so that the functions were more densely packed when s was near 0 which is important since the function spends most of its time in this area. The width were started at 1.0 at the s=1 centered function and then decreased by a factor of 1.06 each subsequent center for a final width of around .002. K was chosen by first doing a large sweep and then slowly converging onto a value of 70 where the plan matched the demo the best. The reproduction is quite accurate as can be seen below.



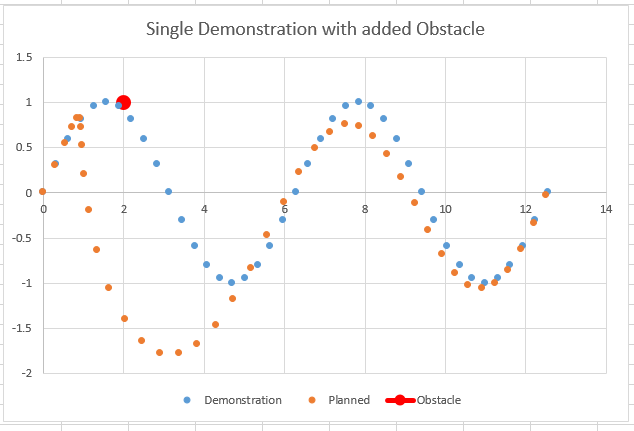
1. Multiple Noisy Demos and Planned Trajectory at different start and end point

As can be seen below, the solution generalized quite well to different endpoints and startpoints since the original sine wave can still be seen in this new path easily.

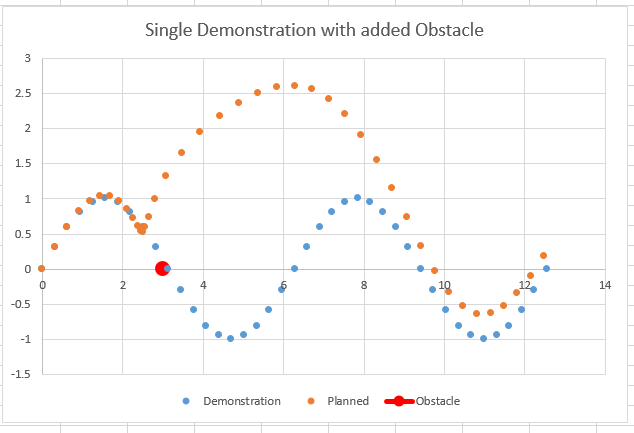


1. Added Coupling term

Obstacle at (2,1) to show an indirect hit and how the function reacts. The trajectory feels a force that weakens as a Gaussian depending on distance to the obstacle and the direction of the obstacle compared to the trajectory. As you can see it avoided below since it saw the obstacle from below and felt a downward force, enough to push it below the normal trajectory.



Obstacle at (3,0) to show how a head on collision would react. In this, the trajectory went towards the obstacle head on and did a direct U turn and went over since the net force was in the up direction.



1. Questions
   1. The DMP seemed to generalize properly when the start and end goals were relatively close to the trained points and when there were more data points in the path (such as in the slower trajectory). Making trajectories faster didn’t seem to work as well. I think this is due to the fact that the data points are sampled further from each other since the time step remains the same at dt = 0.1 and when that happens maybe the approximated values for each s aren’t as good as when they are closer to the initial ones. Either way even if the replication wasn’t perfect, the general style of the demonstration was just about always present.
   2. To make a system that can change on the fly, you would have to have the ability to detect and find the coordinates of obstacles in relation to its end effector. The protocol that you could apply would be whenever either the goal changed or an obstacle was detected, pause the current trajectory, recalculate a new trajectory based on the new locations of the goal and obstacle, and finally reboot the trajectory to finish the motion. By doing this whenever changes are made, we can be sure that the system is able to react to perturbations in both the environment and the desired trajectory itself. The magnitude of K and D would be very important here since they determine the force felt when the trajectory is off the desired path. Very high values of K and D would lead to a more forceful return to the desired trajectory while lower values might wander a bit further.
   3. Since policy search depends on the fact that cost needs to be a smooth function in respect to the parameter being changed, I would change K (and therefore D) and also the widths of the Gaussians. I would choose these because in my experimentation changing K seemed to have a consistent effect on the results. By this I mean that if my results weren’t good, increasing K made them incrementally better until a max was found and then the results got worse (note that I was changing D simultaneously to keep the system critically damp). So since the results kept getting monotonically better with respect to K until a max was reached, this is a great candidate for policy search. I believe the widths of the Gaussians would behave in the same way. I think this because intuitively these Gaussian basis functions work best when there is small overlap in between points and that is hard to do by hand for each of the many function centers especially as they get closer and closer together. As the overlap becomes better and better, the results should be better monotonically I believe. This would also be ideal to change in a policy search manner.

**Code**

**DMP.py**

import math

import numpy as np

from functionApproximators import interpolate\_function

from functionApproximators import GBF

# Parameters

ALPHA = -1\*math.log(0.01) # Alpha that leads to 99% convergence

T = 4

K = 70

D = 2\*math.sqrt(K)

numPoints = 40

numFuncs = 40\*2+2

centers = []

widths = []

curr = 1

width = 1.0

for i in range(numFuncs):

centers.append(curr)

curr = curr/float(1.258925412)

widths.append(width)

width = width/float(1.06)

# obstacle

OBSTACLE = [3,0]

# Returns DMP parameters

# trajectory --> n rows (num points) and m columns (dimension of data)

# K and D --> parameters

def DMPLearning(trajectory, K, D):

# Error Checking

num\_demos = len(trajectory)

if(num\_demos == 0):

return None

# Initialize our retunr value

dimensions = len(trajectory[0][0]['coord'])

ans = []

# Need to make one differential equation per dimension

for dim in range(dimensions):

elem = {

'num\_demos': num\_demos,

'f': None

}

f\_target = []

for traj in trajectory:

x = [ i['coord'][dim] for i in traj ]

v = [0]\*len(x)

v\_dot = [0]\*len(x)

for i in range(1,len(x)):

v[i] = T\*((x[i] - x[i-1])/float(traj[i]['t'] - traj[i-1]['t']))

v\_dot[i-1] = (v[i] - v[i-1])/float(traj[i]['t'] - traj[i-1]['t'])

v\_dot[len(x)-1] = 0

v[len(x)-1] = 0

#print(v)

# print(v\_dot)

# Solution to canonical system is s(t) = math.exp(-t\*alpha/T)

g = x[len(x)-1]

x\_0 = x[0]

for i in range(len(x)):

t = traj[i]['t']

entry = {}

entry["s"] = math.exp(-t\*ALPHA/T)

entry["value"] = (T\*v\_dot[i] + D\*v[i])/float(K) - (g - x[i]) + (g - x\_0)\*entry["s"]

f\_target.append(entry)

if(num\_demos > 1):

inputs = [entry["s"] for entry in f\_target]

for i in range(len(inputs)):

centers[i] = inputs[i]

outputs = [entry["value"] for entry in f\_target]

# if(dim == 0):

# print("dim " + str(dim))

# for i in range(len(inputs)):

# print(str(inputs[i]) + " " + str(outputs[i]))

gbf = GBF(numFuncs, centers, widths)

gbf.train(inputs, outputs)

# if(dim == 0):

# print("====")

# for i in range(len(inputs)):

# print(str(inputs[i]) + " " + str(gbf.predict(inputs[i])))

elem["f"] = gbf

else:

elem["f"] = f\_target

ans.append(elem)

return ans

def DMPPlanning(parameters, startPos, startVel, endPos, T\_new, dt):

num\_demos = parameters[0]['num\_demos']

plan = []

velocities = []

plan.append({'coord':[float(i) for i in startPos], 't': 0})

velocities.append([float(i) for i in startVel])

dimensions = len(startPos)

for i in range(1, math.ceil(T\_new/float(dt)+1)):

S = math.exp(-dt\*(i-1)\*ALPHA/T\_new)

point = [0]\*dimensions

velocity = [0]\*dimensions

couplingTerm = getAccelerationVector(getDist(plan[i-1]['coord'],OBSTACLE),plan[i-1]['coord'],OBSTACLE)

# print(couplingTerm)

for dim in range(dimensions):

v\_dot = 0

if(num\_demos == 1):

v\_dot = (K\*(endPos[dim] - plan[i-1]['coord'][dim]) - D\*velocities[i-1][dim] - K\*(endPos[dim] - plan[0]['coord'][dim])\*S + K\*interpolate\_function(S, parameters[dim]) + couplingTerm[dim])/float(T\_new)

else:

v\_dot = (K\*(endPos[dim] - plan[i-1]['coord'][dim]) - D\*velocities[i-1][dim] - K\*(endPos[dim] - plan[0]['coord'][dim])\*S + K\*parameters[dim]['f'].predict(S))/float(T\_new)

velocity[dim] = velocities[i-1][dim] + dt\*v\_dot

# print(v\_dot)

x\_dot = velocity[dim]/float(T\_new)

point[dim] = plan[i-1]['coord'][dim] + dt\*x\_dot

plan.append({'coord':point, 't': dt\*i})

velocities.append(velocity)

return plan

##############################################################

# Helper functions to turn any function into points

##############################################################

# Get Euclidean distance

def getDist(x,y):

return math.sqrt(math.pow(x[0]-y[0],2) + math.pow(x[1]-y[1],2))

def getAccelerationVector(dist,point,obstacle):

forceMultiplier = 500

stdv = 1

acceleration = forceMultiplier\*math.exp(-math.pow(dist,2)/(2\*math.pow(stdv,2)))

displacement = [(point[0] - obstacle[0])\*acceleration/dist, (point[1] - obstacle[1])\*acceleration/dist]

return displacement

# Function you are trying to approximate

def f(x):

return math.sin(x)

def addNoise(x):

return np.random.normal(scale=0.1) + x

def getCoordinates(func, startx, endx, numPoints, noise):

dx = (endx - startx)/float(numPoints)

coords = [0]\*(numPoints+1)

points = 0

x = startx

while points <= numPoints:

coord = {}

if(noise):

coord = {"coord" : [addNoise(x),addNoise(func(x))] ,"t": T\*points/float(numPoints)}

else:

coord = {"coord" : [x,func(x)] ,"t": T\*points/float(numPoints)}

coords[points] = coord

x = x + dx

points = points + 1

return coords

def combineTrajectories(trajectories):

finalTrajectory = []

for trajectory in trajectories:

for item in trajectory:

finalTrajectory.append(item)

return finalTrajectory

###############################################################

# gbf = GBF(3, [1,.5,0],[.5,.5,.5])

# input = [1, .5, 0.1]

# output = [1, 1, 1]

# gbf.train(input,output)

# exit()

traj1 = getCoordinates(f, 0, 4\*math.pi, numPoints, False)

#traj2 = getCoordinates(f, 0, 4\*math.pi, numPoints, True)

#traj = [traj1, traj2]

traj = [traj1]

model = DMPLearning(traj, K, D)

# for item in model:

# for elem in item['f']:

# print(elem)

plan = DMPPlanning(model, [0,0], [0,0], [4\*math.pi,0], T, 0.1)

print("==========GIVEN TRAJECTORY==========")

print('{:>5}\t{}'.format('t','coord'))

print("-"\*20)

for i in traj[0]:

print('{:>5.2f}\t{}'.format(i["t"],' '.join(['{:3.3f}'.format(elem) for elem in i["coord"]])))

# for i in traj[1]:

# print('{:>5.2f}\t{}'.format(i["t"],' '.join(['{:3.3f}'.format(elem) for elem in i["coord"]])))

print("\n")

print("=" \* 10)

print("\n")

print("==========PLANNED TRAJECTORY==========")

print('{:>5}\t{}'.format('t','coord'))

print("-"\*20)

for i in plan:

print('{:>5.2f}\t{}'.format(i["t"],' '.join(['{:3.3f}'.format(elem) for elem in i["coord"]])))

**FunctionApproximators.py**

import math

import numpy as np

#############################################################

# Function interpolation

#############################################################

def interpolate\_function(S, parameters):

f = parameters["f"]

i = 0

while(i < len(f) and f[i]["s"] > S):

i = i + 1

if(i == len(f)):

return f[i-1]["value"]

elif(i == 0):

return f[i]["value"]

else:

return f[i-1]["value"] + (S - f[i-1]["s"])\*(f[i]["value"]-f[i-1]["value"])/(f[i]["s"]-f[i-1]["s"])

##############################################################

# Implementing Gaussian Basis Functions

##############################################################

class GBF:

def \_\_init\_\_(self, numFuncs, centers, widths):

#Test if number of centers and widths match numFuncs

if(len(centers) != numFuncs or len(widths) != numFuncs):

raise Exception("Number of centers("+str(len(centers))+")/widths"+str(len(widths))+" provided doesn't match number of functions"+str(numFuncs)+".")

self.w = np.empty([numFuncs])

self.centers = centers

self.widths = widths

# Given funciton index i and s value returns f\_i(s)

def eval\_function(self, i, s):

return s\*math.exp(-1\*self.widths[i]\*math.pow((s - self.centers[i]),2))

# Arguments are 2 arrays of length M of the M input and output values

def train(self, input, output):

if(len(input) != len(output)):

raise Exception("Number of inputs("+len(input)+") and outputs("+len(output)+") don't match")

transformedInput = np.empty([len(input),len(self.w)])

for i in range(len(input)):

transformed = [self.eval\_function(func,input[i]) for func in range(len(self.w))]

transformedInput[i] = transformed

self.w = np.linalg.lstsq(transformedInput, output)[0]

def predict(self, input):

transformedInput = np.asarray([self.eval\_function(func,input) for func in range(len(self.w))])

prediction = transformedInput.dot(self.w)

return prediction