# Abstract

A robotic arm was configured, and an end effector designed for the purpose of moving a payload of known weight and size from one specified location to another specified location while avoiding an obstacle of known radius and location.

The gripper design and robotic configuration were successful in meeting the workspace requirements. It was also demonstrably effective at picking up the payload even with a moderate amount of positional variation.

There were some software issues which prevented effective pathfinding in the final product which could be improved in the future with bug fixes resulting in faster movement paths.

We could dramatically improve execution time with closed-loop feedback as the algorithm had no way of knowing whether the robot arm had achieved commanded position and it was necessary to simply wait an amount of time in which the robot was likely to have made the move.

Link rigidity and improved gripper materials could improve movement precision and tolerate crash conditions more effectively.

The following sections detail the various scored aspects of the project and discuss the successes and drawbacks of each of the functional areas.

# Inverse kinematics

# https://lh4.googleusercontent.com/OPs5rslw6Z0y32bzlcT3YciGjnOGyLmDYKXDPdhig-tSE7RKQGHrQGmkFURKM0v3R57M07xdDT-0uDBso9oDu4QodBc0hqlsIrHAMTvR0UglZZC9-RecZBssG0179PHoFSAVq3yH

Figure : Inverse kinematics scratch

# Robot design

The robot design was set up so that the inverse kinematics were as simple as practically possible while maintaining the ability to leave the wrist in a vertical orientation throughout the required workspace. Though not truly dexterous, it was dexterous in the specific orientation required for our purposes.

# End effector design

## Iteration 1 – Parallel gripper fingers

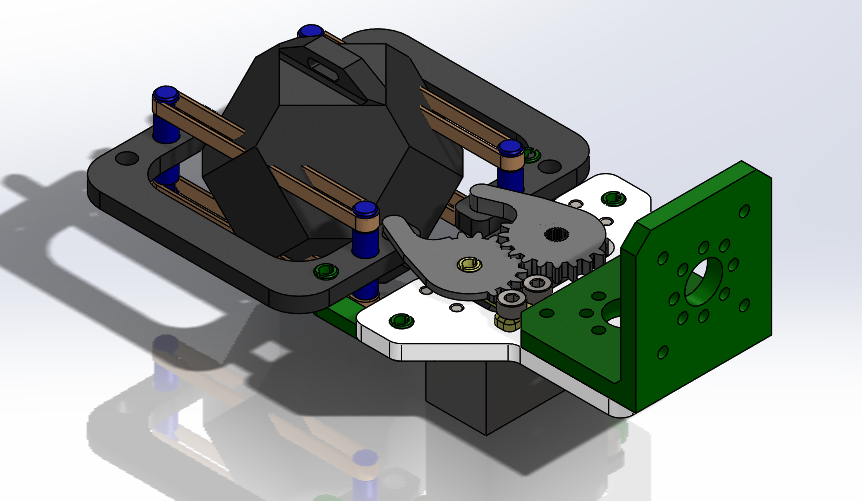


Figure 2: Gripper iteration 1

The idea behind this design was to provide a lot of positional error forgiveness. There is a lot of lash in the servo gears, and the light robot links have a substantial amount of flex and sag. All of this adds up to make very precise positioning difficult. Therefore, designing a gripper that can corral a payload from slightly off the mark beneficial.

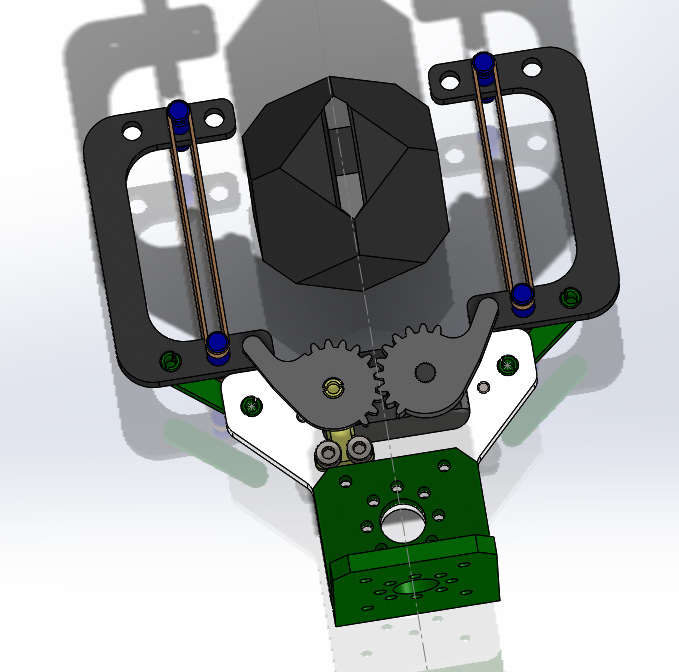


Figure 3: Open gripper jaws

Rubber bands were used as the contact surface of the gripper fingers. This design decision was made to add compliance to the gripper so that the servo could reach the commanded “closed” position without heating up or wasting battery power fighting against the rigidity of the fingers. The bands also conform around the ball in such a way as to make it very unlikely that the ball would be dropped from the gripper in a closed configuration. The parallel actuation of the grippers coupled with these parallel bands results in a very robust gripping solution for a robot with the specific draw-backs of this system.

## Iteration 2 – Parallel, vertically oriented fingers

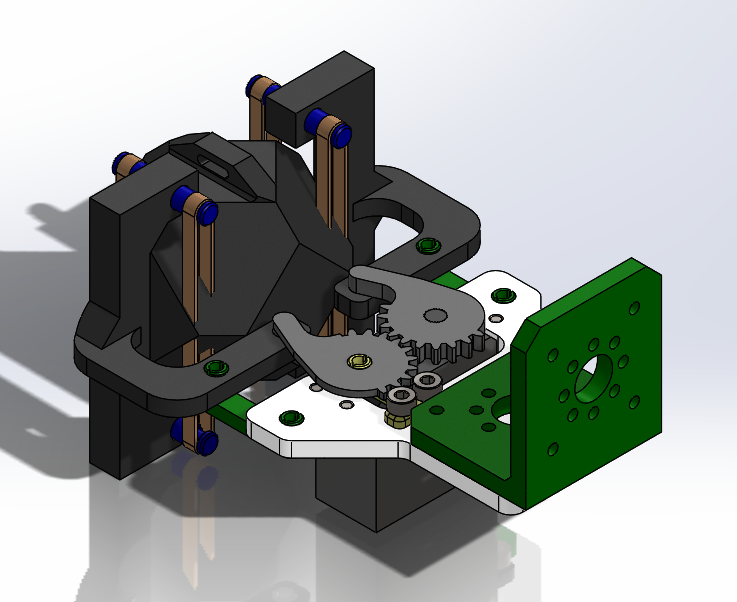


Figure 4: Gripper iteration 2

The second gripper iteration has the same teach-point as the first, but with the fingers rotated to a vertical configuration. This was necessary to achieve points in the workspace very close to the base of the robot.

# Performance

## Attempt 1

A little slow but on target. Probably due in part to sub-optimal path design, but primarily due to the lack of closed loop feedback. The program had no way of determining when the robot had actually reached a requested position and relied instead on delays.

## Attempt 2

Fresh batteries and disparate servo speeds doomed this round as the end-effector pushed the payload out of the pickup zone on its approach. The crash also resulted in substantial gripper finger damage which would have precluded any additional attempts if we had gotten them.

# Code

The original pathfinding algorithm that was developed was not used. It seemed that there was a version conflict or some other un-diagnosed bug when translating from one machine to another. However, it is included here to document the intended functionality of the project.

Subsequently, Sanitize inputs.py was included as the other dependency of our program. This was used in the final result for error checking and sanitizing input data from the user. It also assisted with proper numpy array syntax.

## Astar.py

import sanitize\_inputs as si

import numpy as np

class obstacle():

def \_\_init\_\_(self, location, radius, height=float('Inf')):

self.loc = location

self.r = radius

self.h = height

self.d = (self.loc.x\*\*2+self.loc.y\*\*2)\*\*0.5

#The following calculates the angular 'shadow' cast by the object

self.beta = np.arctan2(self.loc.y, self.loc.x)

self.theta = np.arcsin(self.r/self.d)

self.angle1 = self.beta + self.theta

self.angle2 = self.beta - self.theta

def collision\_detect(self, point):

'''This function accepts a column vector and checks if it collides with

the obstacle object.'''

d = ((self.loc.x-point.loc.x)\*\*2

+ (self.loc.y-point.loc.y)\*\*2)\*\*0.5

if ((self.loc.z <= point.loc.z <= self.loc.z + self.h)

and d <= self.r):

return(True)

else:

return(False)

def robot\_collision\_detect(self, point):

d = ((point.loc.x)\*\*2

+ ((point.loc.y))\*\*2)\*\*0.5

beta = np.arctan2(point.loc.y, point.loc.x)

if d >= self.d - self.r and self.angle2 < beta < self.angle1:

return(True)

else:

return(False)

class node():

def \_\_init\_\_(self, location):

self.loc = location

def open\_node(self, gcost, hcost, parent=None):

self.parent = parent

if parent != None:

self.parent\_g\_cost = parent.gcost

else:

self.parent\_g\_cost = 0

self.gcost = self.parent\_g\_cost + gcost

self.hcost = hcost

self.fcost = self.gcost + self.hcost

def print\_node(self):

print("X: ",self.loc.x,

"Y: ",self.loc.y,

"Z: ",self.loc.z,

"\nG-cost: ",self.gcost,

"\nH-cost: ",self.hcost,

"\nF-cost: ",self.fcost,sep='')

print("\n")

def set\_walkable(self, walk=True):

'''If walk is True, the node is reachable, and is not blocked by an

obstacle.'''

self.walk = walk

class work\_envelope():

def \_\_init\_\_(self, x\_dim, y\_dim, z\_dim, dx=1, dy=1, dz=1):

self.dx = dx

self.dy = dy

self.dz = dz

self.x\_dim = x\_dim

self.y\_dim = y\_dim

self.z\_dim = z\_dim

self.open\_nodes = []

self.closed\_nodes = []

self.obstacles = []

def dist(self,n1, n2):

d = (((n1.loc.x-n2.loc.x))\*\*2

+ ((n1.loc.y-n2.loc.y))\*\*2

+ ((n1.loc.z-n2.loc.z))\*\*2)\*\*0.5

return(d)

def check\_match(self, n1, n2):

'''This function checks whether the coordinates of two given nodes

match, and thereby whether one is a duplicate of the other.'''

epsilon = 0.01

if (abs(n1.loc.x - n2.loc.x) < epsilon

and abs(n1.loc.y - n2.loc.y) < epsilon

and abs(n1.loc.z - n2.loc.z) < epsilon):

match = True

else:

match = False

return(match)

def check\_existence(self, n, gcost):

'''This function checks for the existence of a given node, n,

in the open\_nodes and closed\_nodes lists.'''

exists = False

for elem in self.open\_nodes:

exists = self.check\_match(elem, n)

if exists:

elem.gcost = min(gcost, elem.gcost)

break

else:

pass

for elem in self.closed\_nodes:

exists = self.check\_match(elem, n)

if exists:

break

else:

pass

return exists

def check\_collision(self, node):

collides = False

for o in self.obstacles:

collides = o.robot\_collision\_detect(node)

if collides:

break

return(collides)

def new\_node(self, gcost, new, end, n):

exists = self.check\_existence(new, gcost)

if exists == False:

hcost = self.dist(new, end)

if (self.check\_collision(new)):

self.closed\_nodes.append(new)

else:

new.open\_node(gcost, hcost, n)

self.open\_nodes.append(new)

else:

pass

def close\_node(self, n, end):

# temporary variables for current node

x = n.loc.x

y = n.loc.y

z = n.loc.z

# Explore x dimension

gcost = self.dx

new = node(si.col\_vec([x-self.dx,y,z]))

self.new\_node(gcost, new, end, n)

new = node(si.col\_vec([x+self.dx,y,z]))

self.new\_node(gcost, new, end, n)

# Explore y dimension

gcost = self.dy

new = node(si.col\_vec([x,y-self.dy,z]))

self.new\_node(gcost, new, end, n)

new = node(si.col\_vec([x,y+self.dy,z]))

self.new\_node(gcost, new, end, n)

# Explore z dimension

gcost = self.dz

new = node(si.col\_vec([x,y,z-self.dz]))

self.new\_node(gcost, new, end, n)

new = node(si.col\_vec([x,y,z+self.dz]))

self.new\_node(gcost, new, end, n)

# Explore xy diagonals

gcost = (self.dx\*\*2 + self.dy\*\*2)\*\*0.5

new = node(si.col\_vec([x-self.dx,y-self.dy,z]))

self.new\_node(gcost, new, end, n)

new = node(si.col\_vec([x-self.dx,y+self.dy,z]))

self.new\_node(gcost, new, end, n)

new = node(si.col\_vec([x+self.dx,y-self.dy,z]))

self.new\_node(gcost, new, end, n)

new = node(si.col\_vec([x+self.dx,y+self.dy,z]))

self.new\_node(gcost, new, end, n)

# Explore yz diagonals

gcost = (self.dy\*\*2 + self.dz\*\*2)\*\*0.5

new = node(si.col\_vec([x,y-self.dy,z-self.dz]))

self.new\_node(gcost, new, end, n)

new = node(si.col\_vec([x,y+self.dy,z-self.dz]))

self.new\_node(gcost, new, end, n)

new = node(si.col\_vec([x,y-self.dy,z+self.dz]))

self.new\_node(gcost, new, end, n)

new = node(si.col\_vec([x,y+self.dy,z+self.dz]))

self.new\_node(gcost, new, end, n)

# Explore xz diagonals

gcost = (self.dx\*\*2 + self.dz\*\*2)\*\*0.5

new = node(si.col\_vec([x-self.dx,y,z+self.dz]))

self.new\_node(gcost, new, end, n)

new = node(si.col\_vec([x-self.dx,y,z-self.dz]))

self.new\_node(gcost, new, end, n)

new = node(si.col\_vec([x+self.dx,y,z+self.dz]))

self.new\_node(gcost, new, end, n)

new = node(si.col\_vec([x+self.dx,y,z-self.dz]))

self.new\_node(gcost, new, end, n)

# Explore corners

gcost = (self.dx\*\*2 + self.dy\*\*2 + self.dz\*\*2)\*\*0.5

new = node(si.col\_vec([x-self.dx,y-self.dy,z-self.dz]))

self.new\_node(gcost, new, end, n)

new = node(si.col\_vec([x-self.dx,y-self.dy,z+self.dz]))

self.new\_node(gcost, new, end, n)

new = node(si.col\_vec([x-self.dx,y+self.dy,z-self.dz]))

self.new\_node(gcost, new, end, n)

new = node(si.col\_vec([x-self.dx,y+self.dy,z+self.dz]))

self.new\_node(gcost, new, end, n)

new = node(si.col\_vec([x+self.dx,y-self.dy,z-self.dz]))

self.new\_node(gcost, new, end, n)

new = node(si.col\_vec([x+self.dx,y-self.dy,z+self.dz]))

self.new\_node(gcost, new, end, n)

new = node(si.col\_vec([x+self.dx,y+self.dy,z-self.dz]))

self.new\_node(gcost, new, end, n)

new = node(si.col\_vec([x+self.dx,y+self.dy,z+self.dz]))

self.new\_node(gcost, new, end, n)

self.closed\_nodes.append(self.open\_nodes.pop(0))

def sort\_nodes(self):

self.open\_nodes.sort(key=lambda x: x.fcost, reverse=False)

def generate\_obstacle(self, obstacle):

'''Append an obstacle to the work envelope's list of obstacles.'''

self.obstacles.append(obstacle)

def generate\_path(start, goal, n, \*obstacles):

'''This function generates a path given a starting location, a goal

location, and an arbitrary number of obstacles.'''

epsilon = 0.25

dx = abs(goal.loc.x-start.loc.x)/n

dy = abs(goal.loc.y-start.loc.y)/n

dz = abs(goal.loc.z-start.loc.z)/n

print("dx: ",dx,"dy: ",dy,"dz: ",dz,sep='')

w\_env = work\_envelope(1, 1, 1, dx, dy, dz)

path\_complete = False

for o in obstacles:

w\_env.generate\_obstacle(o)

start\_node = start

goal\_node = goal

start\_h\_cost = w\_env.dist(start\_node, goal\_node)

start\_node.open\_node(0, start\_h\_cost, parent=None)

w\_env.open\_nodes.append(start\_node)

try:

while path\_complete == False:

w\_env.sort\_nodes()

if w\_env.check\_match(w\_env.open\_nodes[0], goal\_node):

print("Found the end.")

path = [w\_env.open\_nodes[0]]

while (path[-1].parent != None):

path.append(path[-1].parent)

print("Finished the path")

print("Opened",len(w\_env.open\_nodes),"nodes.")

print("Closed",len(w\_env.closed\_nodes),"nodes.")

break

w\_env.close\_node(w\_env.open\_nodes[0], goal\_node)

## print("Open nodes: ", len(w\_env.open\_nodes))

## print("Closed nodes: ", len(w\_env.closed\_nodes))

except IndexError:

print("All nodes opened.")

path\_mat = np.array([goal\_node.loc.x, goal\_node.loc.y, goal\_node.loc.z])

for elem in path:

path\_mat = np.row\_stack([[elem.loc.x, elem.loc.y, elem.loc.z], path\_mat])

## elem.print\_node()

return(path\_mat)

## Sanitize inputs.py

import numpy as np

class col\_vec():

'''Retrieves a list of real number for x, y, and z from the user,

and constructs a numpy column vector.'''

def \_\_init\_\_(self, coords):

self.x = coords[0]

self.y = coords[1]

self.z = coords[2]

self.vec = np.array([[self.x],[self.y],[self.z]])

def get\_real\_number(prompt=None, upper=float('Inf'), lower=float('-Inf')):

'''Gets a real number from the user with an optional prompt. Positive and

negative limits can be set. If not set, the default values are 'Inf' and

'-Inf' respectively.'''

num\_flag = False

while(not num\_flag):

try:

number = float(input(prompt))

if lower < number < upper:

num\_flag = True

else:

print("Enter a real number between",lower,"and",upper)

except ValueError:

print("Enter a real number.")

num\_flag = False

return(number)

def get\_integer(prompt=None, upper=float('Inf'), lower=float('-Inf')):

'''Gets an integer from the user with an optional prompt. Positive and

negative limits can be set. If not set, the default values are 'Inf' and

'-Inf' respectively.'''

num\_flag = False

while(not num\_flag):

try:

number = int(input(prompt))

number += 0

# This will throw an exception if number is not an integer.

if lower < number < upper:

num\_flag = True

else:

print("Enter a real number between",lower,"and",upper)

except ValueError:

print("Enter an integer.")

num\_flag = False

return(number)

def get\_letter(prompt=None, accept=None):

# Gets a single alpha character that is included in the list 'accept'

# Optionally include a prompt to the user

# omitting the accept list allows all alpha characters.

flag = False

while(not flag):

letter = str(input(prompt))

if(letter.isalpha() and len(letter) == 1):

if accept != None:

for i in accept:

if letter == i or accept==None:

flag = True

break

else:

pass

else:

flag = True

else:

pass

return(letter)

def get\_coords(rows=3):

'''This function gets the coordinates for a point in 3D space from the user.

It includes the error checking logic required to ensure the point's

useability in subsequent functions.'''

P\_x = get\_real\_number("X >>> ")

P\_y = get\_real\_number("Y >>> ")

P\_z = get\_real\_number("Z >>> ")

point = col\_vec([P\_x,P\_y,P\_z])

if rows == 3:

return(point)

elif rows == 4:

point.vec = np.row\_stack([point.vec,[1]])

return(point)

else:

print("Invalid argument.")

return(None)

# Inverse kinematics

def KraussIK(Loc1):

   #Loc = Loc1 - Pgripper

   Loc = Loc1 + np.array([0,0,1,0])

   t1 = np.arctan2(Loc[1],Loc[0]) \*rtd

   r = np.sqrt(Loc[1]\*\*2 + Loc[0]\*\*2)

   L = np.sqrt(r\*\*2 - (C+D)\*\*2)

   phi = np.arctan2((C+D),L) \*rtd

   theta1 = t1 + phi

   T\_01 = robotics.DH(0,0,theta1,0)

   T\_10 = robotics.HTinv(T\_01)

   P\_tip = np.dot(T\_10,Loc)

   P\_tip = robotics.prettymat(P\_tip)

   P\_tip\_1 = P\_tip

   #print(P\_tip\_1)

   dz = P\_tip\_1[2] - A

   dr = P\_tip\_1[0] - B

   rs = dr\*\*2 + dz\*\*2

   cosalpha = (L2\*\*2 + L3\*\*2 - rs)/(2\*L2\*L3)

   sinalpha = np.sqrt(1-cosalpha\*\*2)

   alpha1 = np.arccos(cosalpha)

   alpha = np.arctan2(sinalpha,cosalpha)\*rtd

   theta3 = alpha-180

   beta = np.arctan2(dz,dr)

   phi = np.arctan2(L3\*np.sin(-theta3\*dtr),L2+L3\*np.cos(-theta3\*dtr))

   theta2 = (beta+phi)\*rtd

   theta4 = (90 - (-theta3)) + theta2

   return theta1,theta2,theta3,theta4