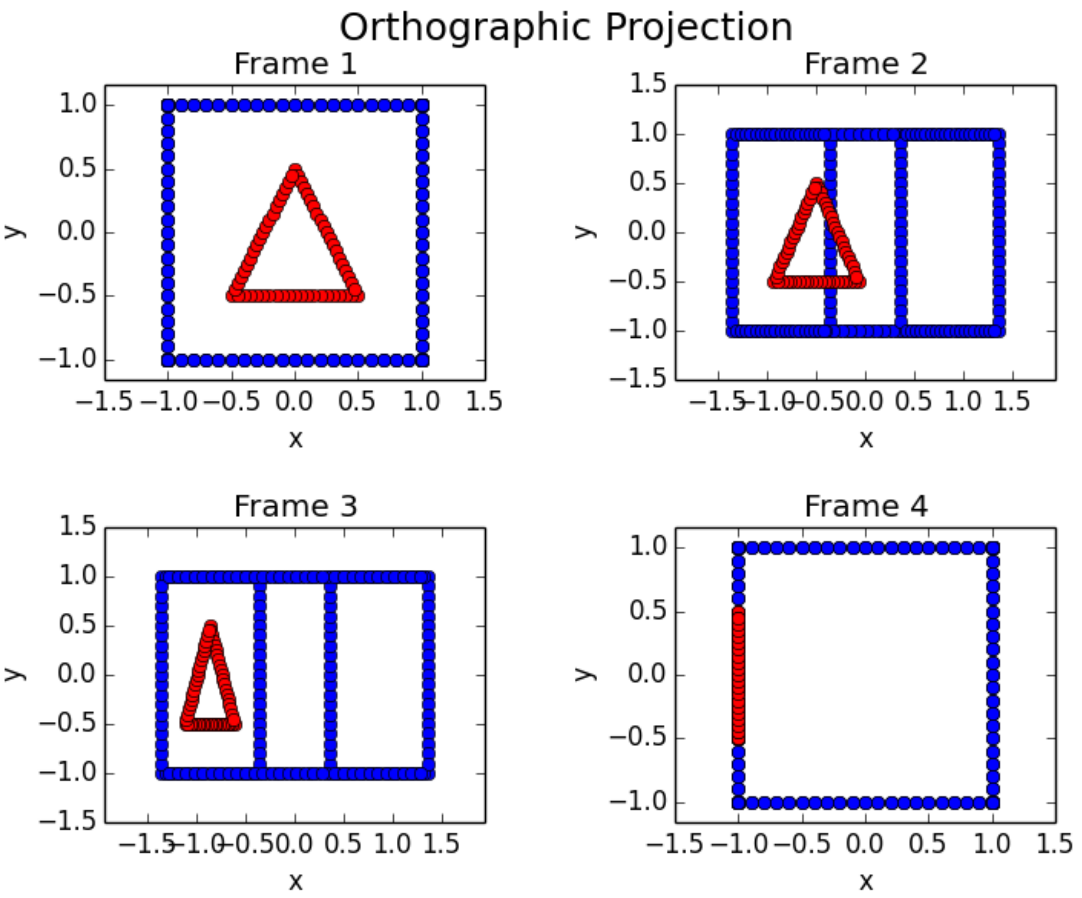
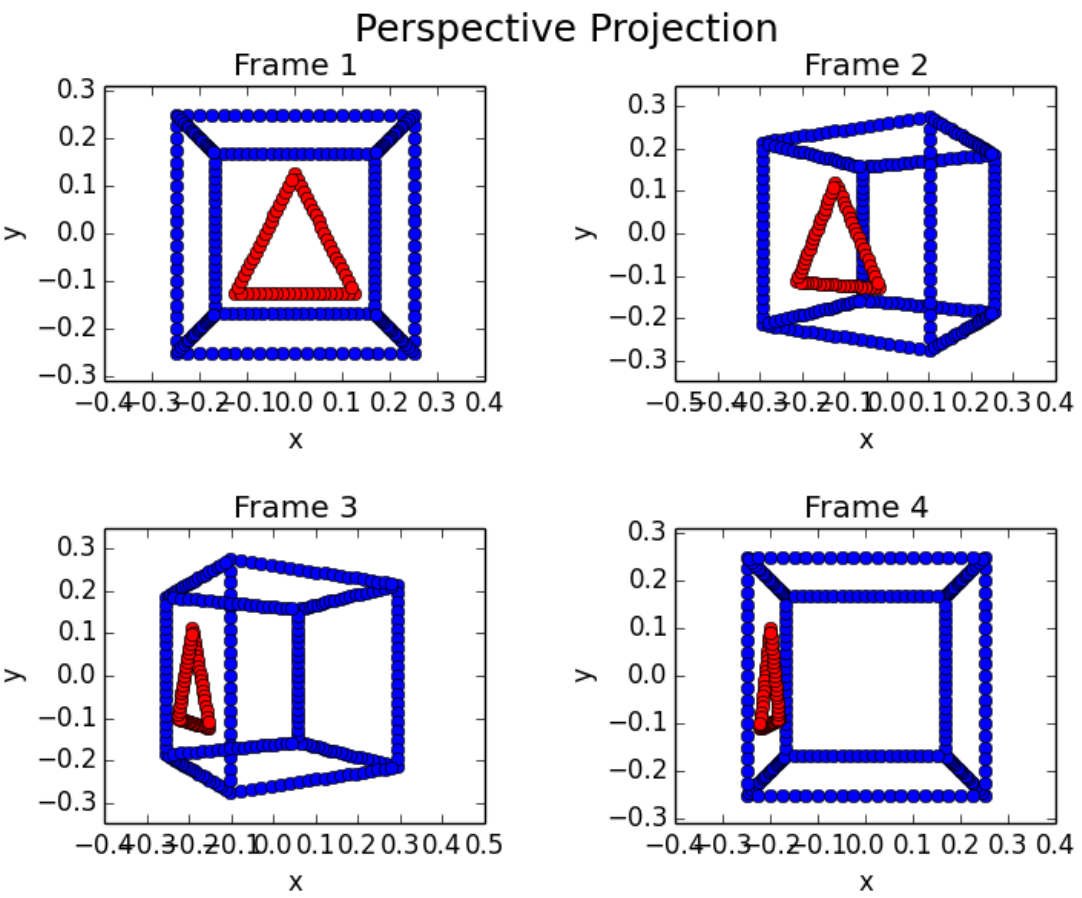
```
# CS4243 Assignment 3
# ==================
# Name: Tay Yang Shun
# Matric: A0073063M

import numpy as np
import matplotlib.pyplot as plt
from math import *

# Camera Intrinsic Parameters
u_0 = 0
v_0 = 0
B_u = 1
B_v = 1
k_u = 1
k_v = 1
f = 1

# Figure Plotting Constants
PLOT_HORIZONTAL_COUNT = 2
PLOT_VERTICAL_COUNT = 2
PLOT_PADDING = 0.5
PLOT_MARGIN = 0.2
PLOT_FONT_SIZE = 18
NUMBER_OF_POINTS_CUBE = 8

############
# Part 1.1 #
############

def pts_set_1():
  pts = np.zeros([11, 3])
  pts[0,:] = [-1, -1, -1]
  pts[1,:] = [1, -1, -1]
  pts[2,:] = [1, 1, -1]
  pts[3,:] = [-1, 1, -1]
  pts[4,:] = [-1, -1, 1]
  pts[5,:] = [1, -1, 1]
  pts[6,:] = [1, 1, 1]
  pts[7,:] = [-1, 1, 1]
  pts[8,:] = [-0.5, -0.5, -1]
  pts[9,:] = [0.5, -0.5, -1]
  pts[10,:] = [0, 0.5, -1]
  return pts

def pts_set_2():
  def create_intermediate_points(pt1, pt2, granularity):
    new_pts = []
    vector = np.array([x[0] - x[1] for x in zip(pt1, pt2)])
    return [np.array(pt2) + (vector * (float(i)/granularity)) for i in range(1,
granularity)]

  pts = []
  granularity = 20

  # Create cube wireframe
  pts.extend([[-1, -1, -1], [1, -1, -1], [1, 1, -1], [-1, 1, -1], \
              [-1, -1, 1], [1, -1, 1], [1, 1, 1], [-1, 1, 1]])

  pts.extend(create_intermediate_points([-1, -1, 1], [1, -1, 1], granularity))
  pts.extend(create_intermediate_points([1, -1, 1], [1, 1, 1], granularity))
  pts.extend(create_intermediate_points([1, 1, 1], [-1, 1, 1], granularity))
  pts.extend(create_intermediate_points([-1, 1, 1], [-1, -1, 1], granularity))

  pts.extend(create_intermediate_points([-1, -1, -1], [1, -1, -1], granularity))
  pts.extend(create_intermediate_points([1, -1, -1], [1, 1, -1], granularity))
  pts.extend(create_intermediate_points([1, 1, -1], [-1, 1, -1], granularity))
  pts.extend(create_intermediate_points([-1, 1, -1], [-1, -1, -1], granularity))

  pts.extend(create_intermediate_points([1, 1, 1], [1, 1, -1], granularity))
```

```
  pts.extend(create_intermediate_points([1, -1, 1], [1, -1, -1], granularity))
  pts.extend(create_intermediate_points([-1, -1, 1], [-1, -1, -1], granularity))
  pts.extend(create_intermediate_points([-1, 1, 1], [-1, 1, -1], granularity))

  # Create triangle wireframe
  pts.extend([[-0.5, -0.5, -1], [0.5, -0.5, -1], [0, 0.5, -1]])
  pts.extend(create_intermediate_points([-0.5, -0.5, -1], [0.5, -0.5, -1],
granularity))
  pts.extend(create_intermediate_points([0.5, -0.5, -1], [0, 0.5, -1], granularity))
  pts.extend(create_intermediate_points([0, 0.5, -1], [-0.5, -0.5, -1], granularity))

  return np.array(pts)

pts = pts_set_1()

def deg_to_rad(deg):
  # Converts an angle from degrees to radians
  return float(deg)/180 * pi

def conjugate(quat):
  # Calculates the conjugate of a quaternion
  return quat[0:1] + np.negative(quat[1:]).tolist()

def approx(value):
  # Approximate a floating point value
  val = round(value, 6)
  return 0.0 if val == 0.0 else val

def approx_mat(mat):
  # Approximate every value in a matrix
  for pt in np.nditer(mat, op_flags=['readwrite']):
    pt[...] = approx(pt)
  return mat

############
# Part 1.2 #
############

def quatmult(p, q):
  # Performs the multiplication of two quaternions
  s_p, v_p = p[0], np.array(p[1:])
  s_q, v_q = q[0], np.array(q[1:])
  s_pq = s_q * s_p - np.dot(v_q, v_p)
  v_pq = np.cross(v_p, v_q) + s_q * v_p + s_p * v_q
  out = [s_pq]
  out.extend(v_pq)
  return out

def quatmult_2(p, q):
  # Alternative version of quaternion multiplication just for kicks
  out = [0] * 4
  out[0] = p[0]*q[0] - p[1]*q[1] - p[2]*q[2] - p[3]*q[3]
  out[1] = p[0]*q[1] + p[1]*q[0] + p[2]*q[3] - p[3]*q[2]
  out[2] = p[0]*q[2] - p[1]*q[3] + p[2]*q[0] + p[3]*q[1]
  out[3] = p[0]*q[3] + p[1]*q[2] - p[2]*q[1] + p[3]*q[0]
  return out

def quatrot(p, q):
  # Performs rotation of two quaternions
  # p is the point to be rotated and q is the rotation quaternion
  return [approx(x) for x in quatmult(quatmult(q, p), conjugate(q))]
```

```
############
# Part 1.3 #
############

def quat2rot(q):
  # Returns a 3x3 rotation matrix parameterized with
  # the elements of an input quaternion
  q_0, q_1, q_2, q_3 = q
  return np.matrix([[q_0**2 + q_1**2 - q_2**2 - q_3**2, 2*(q_1*q_2 - q_0*q_3),
2*(q_1*q_3 + q_0*q_2)],
                    [2*(q_1*q_2 + q_0*q_3), q_0**2 + q_2**2 - q_1**2 - q_3**2,
2*(q_2*q_3 - q_0*q_1)],
                    [2*(q_1*q_3 - q_0*q_2), 2*(q_2*q_3 + q_0*q_1), q_0**2 + q_3**2 -
q_1**2 - q_2**2]])

# Calculating camera positions for each frame
initial_pos = [0, 0, 0, -5]
camera_pos = [initial_pos]
camera_rot_quat = [cos(deg_to_rad(-15)), 0, sin(deg_to_rad(-15)), 0]
pos_new = initial_pos

for i in range(3):
  pos_new = quatrot(pos_new, camera_rot_quat)
  camera_pos.append(pos_new)

# Camera positions for each frame
pos_1, pos_2, pos_3, pos_4 = camera_pos

# Calculating camera orientation for each frame
initial_orntn = np.identity(3)
camera_orntns = [initial_orntn]
camera_rot_mat = quat2rot([cos(deg_to_rad(15)), 0, sin(deg_to_rad(15)), 0])
orntn_new = initial_orntn

for i in range(3):
  orntn_new = camera_rot_mat * orntn_new
  camera_orntns.append(orntn_new)

camera_orntns = [np.array(approx_mat(m)) for m in camera_orntns]
# Camera orientations for each frame
quatmat_1, quatmat_2, quatmat_3, quatmat_4 = camera_orntns

##########
# Part 2 #
##########

def perspective_proj(s_p, t_f, i_f, j_f, k_f):
  # Calculate point after perspective projection
  sptf = s_p - t_f
  u_fp = f * float(np.dot(sptf, i_f)) / np.dot(sptf, k_f) * B_u + u_0
  v_fp = f * float(np.dot(sptf, j_f)) / np.dot(sptf, k_f) * B_v + v_0
  return [approx(p) for p in (u_fp, v_fp)]

def orthographic_proj(s_p, t_f, i_f, j_f, k_f):
  # Calculate point after orthographic projection
  sptf = s_p - t_f
  u_fp = float(np.dot(sptf, i_f)) * B_u + u_0
  v_fp = float(np.dot(sptf, j_f)) * B_v + v_0
  return [approx(p) for p in (u_fp, v_fp)]

def generate_projection_plots(pts, proj_fn, proj_name):
  fig = plt.figure()
  plt.subplots_adjust(hspace=PLOT_PADDING, wspace=PLOT_PADDING)
  for i in range(4):
    projected_pts = [proj_fn(pt, np.array(camera_pos[i][1:]), camera_orntns[i][0], \
                      camera_orntns[i][1], camera_orntns[i][2]) for pt in pts]
    plt.subplot(PLOT_HORIZONTAL_COUNT, PLOT_VERTICAL_COUNT, i+1)
    plt.margins(PLOT_MARGIN)
    plt.title('Frame ' + str(i+1))
    plt.xlabel('x')
```

```
    plt.ylabel('y')
    plt.axis('equal')
    for index, pt in list(enumerate(projected_pts)):
      # Use red colour for triangle points for easy identification
      plt.plot(pt[0], pt[1], 'bo' if index < 8+(20 - 1)*12 else 'ro')

  plt.suptitle(proj_name, fontsize=PLOT_FONT_SIZE)
  fig.savefig(proj_name + '.png')

generate_projection_plots(pts, perspective_proj, 'Perspective Projection')
generate_projection_plots(pts, orthographic_proj, 'Orthographic Projection')
```


Orthographic Projection
Frame 1
Frame 2
Frame 3
Frame 4
x
y


Perspective Projection
Frame 1
Frame 2
Frame 3
Frame 4
x
y