Intelligent Systems Engineering
ECE 449
Dr. Scott Dick
Fall 2023

KESSLER GAME© Development Guide

Table of Contents

PREFACE

This document is a user guide for the Kessler Game©, a Python-based implementation of the classic asteroid-smasher video game. Kessler Game was developed and is maintained as open-source software under the Apache 2.0 license by Thales North America, as a resource for the Explainable Fuzzy Challenge (XFC) competition, hosted by the Annual Meeting of the North American Fuzzy Information Processing Society (NAFIPS).

The purpose of this guide is principally to support students in the course ECE 449 Intelligent Systems Engineering, taught by Dr. Scott Dick at the University of Alberta. In particular, this document supports the class term project, in which teams of 3 students must build an intelligent agent to play the Kessler Game. This agent is required to be a fuzzy inferential system that at a minimum controls all three outputs of the action loop (thrust, turn rate, and firing a bullet); for higher marks, the agent must be optimized via a genetic algorithm (resulting in a genetic-fuzzy agent).

Optionally, the groups may participate in an intra-class tournament between their agents. Their agents are entered into head-to-head competition with each other two at a time; a round-robin phase is used to determine seedings for a final round of 16, which then proceeds as a typical single-elimination tournament with gold and bronze medal games.

ACKNOWLEDGEMENTS

Our thanks to Thales North America for their assistance in setting up the competition infrastructure used in ECE 449, and for conceptual guidance in developing related course materials.

1. KESSLER GAME INSTALLATION

1.1 Acquiring and Installing the Kessler Game Codebase

The Kessler Game (hereafter "the game") is distributed as a Python package. Two methods for installing the game to a local host are available; installing a prebuilt library (wheel) using pip install, or directly downloading a source tarball. The former is the simplest, and will be covered in this guide. We advise that only students with a good working knowledge of the Python ecosystem attempt to download and install the game from a source tarball.

Wheel files (*.whl) are a method of distributing Python code, with all of its dependencies resolved. The Kessler game project wheel is distributed via the Python Package Index, (PyPI for short). This is an open-source community repository for Python projects. In brief, there are two steps to installing a wheel file:

1. Download the *.whl file.
2. Issue the pip install command from your Python environment, with the pathname of the wheel file as the argument.

Each Python IDE has its own approach to executing a pip install command; it is usually not sufficient to simply perform a pip install form the command line outside of the IDE, as the IDEs create their own virtual environment. However, you should check the documentation for your specific IDE to confirm this.

1.2 Test Installation Using the Game Examples Provided

To verify correct installation of the game libraries, we found it most useful to download the example controller from the Kessler Game Github, and run the game using it. The example controller (test_controller.py) is the simplest possible control agent for the game: the ship does not move, but is ordered to spin at a constant rate, firing constantly, at every game step. It does, however, exercise the game control loop, and demonstrate that the game library is successfully working. (**Note:** Students in ECE 449 will be expected to create a genetic-fuzzy agent that can defeat this simple agent over the course of five games for full credit on the group project).

To begin with, after installing the game, create a new project in your Python IDE, and then download the files scenario_test.py, graphics_both.py, and test_controller.py into your project's source directory. Load all three into the IDE for editing. The source code for test_controller.py and scenario_test.py (the latter modified to also accept Dr. Dick's own controller) are copied in Figures 1 and 2 below for convenience.

```python
# -*- coding: utf-8 -*-
# Copyright © 2022 Thales. All Rights Reserved.
# NOTICE: This file is subject to the license agreement defined in file 'LICENSE', which is part of
# this source code package.

from kesslergame import KesslerController # In Eclipse, the name of the library is kesslergame, not src.kesslergame
from typing import Dict, Tuple
import skfuzzy as fuzz


class TestController(KesslerController):
    def __init__(self):
        self.eval_frames = 0

    def actions(self, ship_state: Dict, game_state: Dict) -> Tuple[float, float, bool]:
        """
        Method processed each time step by this controller.
        """

        thrust = 0
        turn_rate = 90
        fire = True
        self.eval_frames +=1

        return thrust, turn_rate, fire

    @property
    def name(self) -> str:
        return "Test Controller"
```

Figure 1: Source Code for test_controller.py

```python
# -*- coding: utf-8 -*-
```

```
# Copyright © 2022 Thales. All Rights Reserved.
# NOTICE: This file is subject to the license agreement defined in file 'LICENSE', which is part of
# this source code package.

import time
from kesslergame import Scenario, KesslerGame, GraphicsType
from test_controller import TestController
from scott_dick_controller import ScottDickController
from graphics_both import GraphicsBoth


my_test_scenario = Scenario(name='Test Scenario',
                            num_asteroids=5,
                            ship_states=[
                                {'position': (400, 400), 'angle': 90, 'lives': 3, 'team': 1},
                                {'position': (600, 400), 'angle': 90, 'lives': 3, 'team': 2},
                            ],
                            map_size=(1000, 800),
                            time_limit=60,
                            ammo_limit_multiplier=0,
                            stop_if_no_ammo=False)

game_settings = {'perf_tracker': True,
                 'graphics_type': GraphicsType.Tkinter,
                 'realtime_multiplier': 1,
                 'graphics_obj': None}
game = KesslerGame(settings=game_settings)  # Use this to visualize the game scenario
# game = TrainerEnvironment(settings=game_settings)  # Use this for max-speed, no-graphics simulation
pre = time.perf_counter()
score, perf_data = game.run(scenario=my_test_scenario, controllers = [TestController(), ScottDickController()])
print('Scenario eval time: '+str(time.perf_counter()-pre))
print(score.stop_reason)
print('Asteroids hit: ' + str([team.asteroids_hit for team in score.teams]))
print('Deaths: ' + str([team.deaths for team in score.teams]))
print('Accuracy: ' + str([team.accuracy for team in score.teams]))
print('Mean eval time: ' + str([team.mean_eval_time for team in score.teams]))
print('Evaluated frames: ' + str([controller.eval_frames for controller in score.final_controllers]))
```

Figure 2: Source Code for scenario_test.py

Let us being by examining test_controller.py. This is a Python class, meaning (for simplicity) it is a new data type that contains both data and methods that operate on the data. An outside program (in this case scenario_test.py) will be able to create a new instance object of this type, and access it like any other Python object. Our class name will be TestController, and it will be a subclass of KesslerController, meaning it will inherit all of the methods and *class and instance variables*[1] belonging to KesslerController.

This is the first time we are referencing something from the game library we installed, and thus we have to tell Python where to find this name. That is the purpose of the `from kesslergame import KesslerController` line. kesslergame is the name of the library we ended up with after pip install finished. However, note that this is not *exactly* what was written in test_controller.py when it was downloaded from Github. From that source, the from command was searching for src.kesslergame. In the Eclipse IDE at least, this didn't work; the IDE could not find src.kesslergame. Our solution was to delete the "src." prefix, which solved the problem.

There are three methods in TestController: init(), actions(), and name(). init() is the constructor method, defined for every class in Python; this method is automatically called whenever an object of this class in instantiated. Instance variables are defined an initialized in this method using the keyword *self*; this is simply a reference to the object. name() is an accessor method (a "getter"), whose job is simply to return the assigned name of the object.

The real action in TestController happens in the actions() method, which is called on every time step of the game loop. Video games may seem to a player like a continuous stream of actions and reactions, but the reality is different. Video games have a frame rate, which is the rate at which a new frame is drawn and presented on the screen; 30 frames per second (fps) is a minimum standard, most games run at 60 fps for smoothness, and the highest-end titles might go as high as 240 fps if the screen supports it (most don't until the monitor gets very expensive). Even then, however, that leaves over 4 milliseconds per frame of time for computation, and a modern computer can do a lot of work in that amount of time. So, before each frame is drawn and presented, there is time for a game agent to make decisions that will be reflected in that next frame. That's the purpose of the actions() method; the arguments to this method contain the complete game state at the end of the last frame, and actions() will decide what the next player action to take will be.

Consider the method signature (i.e. function prototype) for actions():

```
def actions(self, ship_state: Dict, game_state: Dict) -> Tuple[float, float, bool]:
```

As discussed, *self* is just a reference to the current object. ship_state is a data collection (specifically a Dictionary) that gives the current state of the player ship object. game_state, meanwhile, provides all the rest of the game state information as another Dictionary. This method returns two floating-point values and a Boolean value. These are, respectively, the *thrust* to be

[1] There is a distinction between class variables and instance variables in Python. A class variable is bound to the definition of the class itself; there is only one copy, that all of the various objects instantiated from the class could access. Instance variables are bound to a specific object of that class type (i.e. an instance of the class). No other instance of the class can access that instance variable; it is private to that single object.

applied (in m/sec^2), the *turn_rate* for the ship (in degrees per second), and whether or not to *fire* a bullet. It is the task of the actions() method to determine these three control values for the next frame of the game. The game library will interpret these values and execute the appropriate changes to the video frame and game state; you the developer need not worry about any of this. As can be seen in TestController.actions(), the basic controller always sets *thrust* to zero (the player ship does not move); *turn_rate* is a constant +90 degrees per second; and a bullet is fired at every time step. The resulting spray of bullets is comical, but somewhat effective in destroying asteroids. In theory, an intelligent controller should consistently defeat this *un*intelligent algorithm.

The other file that developers need to be concerned with is scenario_test.py. This file defines an initial game state (starting positions and velocities for whatever number of asteroids will be added to the game), instantiates and adds KesslerController objects (or objects from a subclass of KesslerController) to the game as players, launches the main game loop, and records & reports the performance of each player. Notice that this is not a Python class, but rather a script; this is the driver file that starts the game program (and thus the "main" method for running and debugging the game in your IDE). The two things you will need to do are:

- Import your controller class:
  ```
  from test_controller import TestController
  ```
  - Note the syntax: import the controller class name from the filename where it resides (drop the .py extension), which should be in the same project directory as scenario_test.py.
- Add the name of your controller class to the list of controllers passed to game.run().

The version of scenario_test.py above includes both TestController and ScottDickController, written by Dr. Dick. This will run Dr. Dick's controller head-to-head against the TestController.

Figure 3: scott_dick_controller.py

```
# ECE 449 Intelligent Systems Engineering
# Fall 2023
# Dr. Scott Dick

# Demonstration of a fuzzy tree-based controller for Kessler Game.
# Please see the Kessler Game Development Guide by Dr. Scott Dick for a
#   detailed discussion of this source code.

from kesslergame import KesslerController # In Eclipse, the name of the library is kesslergame, not src.kesslergame
from typing import Dict, Tuple
from cmath import sqrt
import skfuzzy as fuzz
from skfuzzy import control as ctrl
import math
import numpy as np
import matplotlib as plt



class ScottDickController(KesslerController):



    def __init__(self):
        self.eval_frames = 0 #What is this?

        # self.targeting_control is the targeting rulebase, which is static in this controller.
        # Declare variables
        bullet_time = ctrl.Antecedent(np.arange(0,1.0,0.002), 'bullet_time')
        theta_delta = ctrl.Antecedent(np.arange(-1*math.pi,math.pi,0.1), 'theta_delta') # Radians due to Python
        ship_turn = ctrl.Consequent(np.arange(-180,180,1), 'ship_turn') # Degrees due to Kessler
        ship_fire = ctrl.Consequent(np.arange(-1,1,0.1), 'ship_fire')

        #Declare fuzzy sets for bullet_time (how long it takes for the bullet to reach the intercept point)
        bullet_time['S'] = fuzz.trimf(bullet_time.universe,[0,0,0.05])
        bullet_time['M'] = fuzz.trimf(bullet_time.universe, [0,0.05,0.1])
        bullet_time['L'] = fuzz.smf(bullet_time.universe,0.0,0.1)

        #Declare fuzzy sets for theta_delta (degrees of turn needed to reach the calculated firing angle)
        theta_delta['NL'] = fuzz.zmf(theta_delta.universe, -1*math.pi/3,-1*math.pi/6)
        theta_delta['NS'] = fuzz.trimf(theta_delta.universe, [-1*math.pi/3,-1*math.pi/6,0])
        theta_delta['Z'] = fuzz.trimf(theta_delta.universe, [-1*math.pi/6,0,math.pi/6])
```

```
theta_delta['PS'] = fuzz.trimf(theta_delta.universe, [0,math.pi/6,math.pi/3])
theta_delta['PL'] = fuzz.smf(theta_delta.universe,math.pi/6,math.pi/3)

#Declare fuzzy sets for the ship_turn consequent; this will be returned as turn_rate.
ship_turn['NL'] = fuzz.trimf(ship_turn.universe, [-180,-180,-30])
ship_turn['NS'] = fuzz.trimf(ship_turn.universe, [-90,-30,0])
ship_turn['Z'] = fuzz.trimf(ship_turn.universe, [-30,0,30])
ship_turn['PS'] = fuzz.trimf(ship_turn.universe, [0,30,90])
ship_turn['PL'] = fuzz.trimf(ship_turn.universe, [30,180,180])

# Declare singleton fuzzy sets for the ship_fire consequent; -1 -> don't fire, +1 -> fire; this will be
#    thresholded and returned as the boolean 'fire'
ship_fire['N'] = fuzz.trimf(ship_fire.universe, [-1,-1,0.0])
ship_fire['Y'] = fuzz.trimf(ship_fire.universe, [0.0,1,1])

#Declare each fuzzy rule
rule1 = ctrl.Rule(bullet_time['L'] & theta_delta['NL'], (ship_turn['NL'], ship_fire['N']))
rule2 = ctrl.Rule(bullet_time['L'] & theta_delta['NS'], (ship_turn['NS'], ship_fire['Y']))
rule3 = ctrl.Rule(bullet_time['L'] & theta_delta['Z'], (ship_turn['Z'], ship_fire['Y']))
rule4 = ctrl.Rule(bullet_time['L'] & theta_delta['PS'], (ship_turn['PS'], ship_fire['Y']))
rule5 = ctrl.Rule(bullet_time['L'] & theta_delta['PL'], (ship_turn['PL'], ship_fire['N']))
rule6 = ctrl.Rule(bullet_time['M'] & theta_delta['NL'], (ship_turn['NL'], ship_fire['N']))
rule7 = ctrl.Rule(bullet_time['M'] & theta_delta['NS'], (ship_turn['NS'], ship_fire['Y']))
rule8 = ctrl.Rule(bullet_time['M'] & theta_delta['Z'], (ship_turn['Z'], ship_fire['Y']))
rule9 = ctrl.Rule(bullet_time['M'] & theta_delta['PS'], (ship_turn['PS'], ship_fire['Y']))
rule10 = ctrl.Rule(bullet_time['M'] & theta_delta['PL'], (ship_turn['PL'], ship_fire['N']))
rule11 = ctrl.Rule(bullet_time['S'] & theta_delta['NL'], (ship_turn['NL'], ship_fire['Y']))
rule12 = ctrl.Rule(bullet_time['S'] & theta_delta['NS'], (ship_turn['NS'], ship_fire['Y']))
rule13 = ctrl.Rule(bullet_time['S'] & theta_delta['Z'], (ship_turn['Z'], ship_fire['Y']))
rule14 = ctrl.Rule(bullet_time['S'] & theta_delta['PS'], (ship_turn['PS'], ship_fire['Y']))
rule15 = ctrl.Rule(bullet_time['S'] & theta_delta['PL'], (ship_turn['PL'], ship_fire['Y']))

#DEBUG
#bullet_time.view()
#theta_delta.view()
#ship_turn.view()
#ship_fire.view()



# Declare the fuzzy controller, add the rules
# This is an instance variable, and thus available for other methods in the same object. See notes.
```

```
        self.targeting_control = ctrl.ControlSystem()
        self.targeting_control.addrule(rule1)
        self.targeting_control.addrule(rule2)
        self.targeting_control.addrule(rule3)
        self.targeting_control.addrule(rule4)
        self.targeting_control.addrule(rule5)
        self.targeting_control.addrule(rule6)
        self.targeting_control.addrule(rule7)
        self.targeting_control.addrule(rule8)
        self.targeting_control.addrule(rule9)
        self.targeting_control.addrule(rule10)
        self.targeting_control.addrule(rule11)
        self.targeting_control.addrule(rule12)
        self.targeting_control.addrule(rule13)
        self.targeting_control.addrule(rule14)
        self.targeting_control.addrule(rule15)



    def actions(self, ship_state: Dict, game_state: Dict) -> Tuple[float, float, bool]:
        """
        Method processed each time step by this controller.
        """
        # These were the constant actions in the basic demo, just spinning and shooting.
        #thrust = 0 <- How do the values scale with asteroid velocity vector?
        #turn_rate = 90 <- How do the values scale with asteroid velocity vector?

        # Answers: Asteroid position and velocity are split into their x,y components in a Tuple each.
        # So are the ship position and velocity, and bullet position and velocity.
        # Units are meters, m/sec, m/sec^2 for thrust.
        # Everything happens in a time increment: delta_time, which appears to be 1/30 sec; this is hardcoded.
        # So, position is updated by multiplying velocity by delta_time, and adding that to position.
        # Ship velocity is updated by multiplying thrust by delta time.
        # Ship position for this time increment is updated after the thrust was applied.


        # My demonstration controller does not move the ship, only rotates it to shoot the nearest asteroid.
        # Goal: demonstrate processing of game state, fuzzy controller, intercept computation.
        # Intercept-point calculation derived from the Law of Cosines, see notes for details and citation.

        # Find the closest asteroid (disregards asteroid velocity)
        ship_pos_x = ship_state["position"][0]     # See src/kesslergame/ship.py in the KesslerGame Github
        ship_pos_y = ship_state["position"][1]
```

```
closest_asteroid = None

for a in game_state["asteroids"]:
    #Loop through all asteroids, find minimum Eudlidean distance
    curr_dist = math.sqrt((ship_pos_x - a["position"][0])**2 + (ship_pos_y - a["position"][1])**2)
    if closest_asteroid is None :
        # Does not yet exist, so initialize first asteroid as the minimum.
        closest_asteroid = dict(aster = a, dist = curr_dist)

    else:
        # closest_asteroid exists, and is thus initialized.
        if closest_asteroid["dist"] > curr_dist:
            # New minimum found
            closest_asteroid["aster"] = a
            closest_asteroid["dist"] = curr_dist

# closest_asteroid is now the nearest asteroid object.
# Calculate intercept time given ship & asteroid position, asteroid velocity vector, bullet speed (not
#   direction).
# Based on Law of Cosines calculation, see notes.

# Side D of the triangle is given by closest_asteroid.dist. Need to get the asteroid-ship direction
#    and the angle of the asteroid's current movement.
# REMEMBER TRIG FUNCTIONS ARE ALL IN RADAINS!!!


asteroid_ship_x = ship_pos_x - closest_asteroid["aster"]["position"][0]
asteroid_ship_y = ship_pos_y - closest_asteroid["aster"]["position"][1]

asteroid_ship_theta = math.atan2(asteroid_ship_y,asteroid_ship_x)

asteroid_direction = math.atan2(closest_asteroid["aster"]["velocity"][1],closest_asteroid["aster"]["velocity"][0])
 # Velocity is a 2-element array [vx,vy].
my_theta2 = asteroid_ship_theta - asteroid_direction
cos_my_theta2 = math.cos(my_theta2)
# Need the speeds of the asteroid and bullet. speed * time is distance to the intercept point
asteroid_vel = math.sqrt(closest_asteroid["aster"]["velocity"][0]**2 + closest_asteroid["aster"]["velocity"][1]**2)
bullet_speed = 800 # Hard-coded bullet speed from bullet.py

# Determinant of the quadratic formula b^2-4ac
targ_det = (-2*closest_asteroid["dist"]*asteroid_vel*cos_my_theta2)**2-(4*(asteroid_vel**2-bullet_speed**2)*
     closest_asteroid["dist"])
```

```
        # Combine the Law of Cosines with the quadratic formula for solve for intercept time. Remember, there are two values
produced.

intrcpt1=((2*closest_asteroid["dist"]*asteroid_vel*cos_my_theta2)+math.sqrt(targ_det))/(2*(asteroid_vel**2bullet_speed**2))
intrcpt2=((2 * closest_asteroid["dist"]*asteroid_vel*cos_my_theta2)-math.sqrt(targ_det))/(2 * (asteroid_vel**2-bullet_speed**2))

        # Take the smaller intercept time, as long as it is positive; if not, take the larger one.
        if intrcpt1 > intrcpt2:
            if intrcpt2 >= 0:
                bullet_t = intrcpt2
            else:
                bullet_t = intrcpt1
        else:
            if intrcpt1 >= 0:
                bullet_t = intrcpt1
            else:
                bullet_t = intrcpt2

        # Calculate the intercept point. The work backwards to find the ship's firing angle my_theta1.
        intrcpt_x = closest_asteroid["aster"]["position"][0] + closest_asteroid["aster"]["velocity"][0] * bullet_t
        intrcpt_y = closest_asteroid["aster"]["position"][1] + closest_asteroid["aster"]["velocity"][1] * bullet_t

        my_theta1 = math.atan2((intrcpt_y - ship_pos_y),(intrcpt_x - ship_pos_x))

        # Lastly, find the difference betwwen firing angle and the ship's current orientation. BUT THE SHIP HEADING IS IN DEGREES.
        shooting_theta = my_theta1 - ((math.pi/180)*ship_state["heading"])

        # Wrap all angles to (-pi, pi)
        shooting_theta = (shooting_theta + math.pi) % (2 * math.pi) - math.pi

        # Pass the inputs to the rulebase and fire it
        shooting = ctrl.ControlSystemSimulation(self.targeting_control,flush_after_run=1)

        shooting.input['bullet_time'] = bullet_t
        shooting.input['theta_delta'] = shooting_theta

        shooting.compute()

        # Get the defuzzified outputs
        turn_rate = shooting.output['ship_turn']

        if shooting.output['ship_fire'] >= 0:
            fire = True
```

```
        else:
            fire = False

        # And return your three outputs to the game simulation. Controller algorithm complete.
        thrust = 0.0

        self.eval_frames +=1

        #DEBUG
        print(thrust, bullet_t, shooting_theta, turn_rate, fire)

        return thrust, turn_rate, fire
```

2. FUZZY INFERENTIAL CONTROL OF TURN_RATE AND FIRING

2.1 Inputs to the Fuzzy Controller

Before discussing the construction of the fuzzy control algorithm in scott_dick_controller.py, it is necessary to define the inputs to the controller. These are the length of time it will take for a bullet fired in this time step to reach the nearest asteroid, and the angle (in degrees) from the ships current heading to the *correct* heading for the bullet to hit the asteroid. These need to be calculated by finding the closest asteroid (which is moving in the same 2D plane as the stationary ship), and calculating the intercept point of a bullet and the asteroid at some future time step. Obviously, the first step is to find *which* asteroid is the closest.

All asteroids, including their position, velocity, size and mass, form part of the game state that is passed to the actions() method when it is invoked. The entire collection of asteroids is passed as a Dictionary object, with each asteroid then being a Dictionary of its own. Dictionaries are Iterable objects, which means you can loop through all of their contents one by one. We do so with the command

```
for a in game_state["asteroids"]:
```

which iteratively retrieves each element in the Dictionary `game_state["asteroids"]`, and assigns it to the variable 'a', which is itself a Dictionary. The element `a["position"]` is the current position of asteroid a. However, this position is stored as a two-element Tuple, which can only be accessed by its numeric index. Thus, `a["position"][0]` is the x-coordinate of the asteroid's position, and `a["position"][1]` is the y-coordinate. Likewise, before starting the loop, we stored the x- and y-coordinate of the ship's position in `ship_pos_x` and `ship_pos_y`, for clarity in our code. We execute the usual algorithm for finding a minimum of a list, scanning through the list and keeping a running track of the closest asteroid to the ship's position found so far. At the end of the loop, the Dictionary object closest_asteroid records the closest asteroid to the ship, and what that closest distance is (we will need this in the next step).

The interception point between the closest asteroid and a bullet fired from the ship is a point somewhere along the future path of the asteroid, which is fully defined because the velocity of every asteroid is constant. We simply need to find the moment in time where that interception will happen; since our game happens in a flat 2D surface, we can do this using the Law of Cosines. Consider the diagram in Figure 4, showing the relationships between the ship, asteroid, and bullet to be fired.

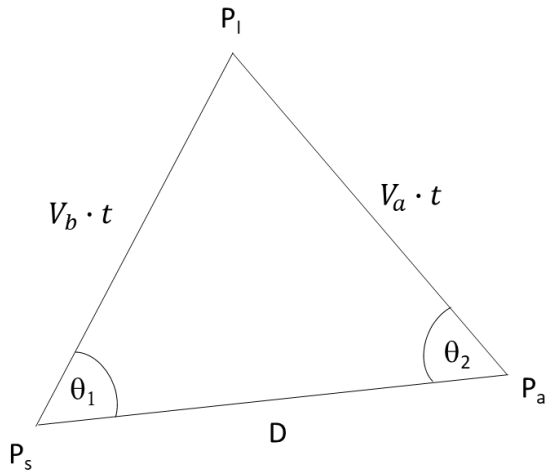

Figure 4: Bullet & Asteroid Intercept Problem

The bullet will start from the current position of the ship, P_s. When the bullet is fired, it immediately has a velocity V_b of 800 m/sec (hardcoded value), traveling in the same direction as the ship was oriented at that time step. Meanwhile, the asteroid is at position P_a, traveling in a known, constant direction at a known, constant speed, all

given by the velocity vector V_a. Because the positions P_s and P_a and the velocity vector V_a are known, we can immediately determine the length of side D, and the angle θ_2. What we do not know is the proper orientation of the ship to fire a bullet (represented by angle θ_1), nor the length of time t that the bullet and asteroid must remain on their separate courses in order to meet at the interception point P_I (as the intercept point itself is unknown). However, as this is a triangle, we can use the Law of Cosines to determine a quadratic expression for t, which we then solve using the quadratic formula. That value of t can then be used to determine the point P_I, as a point on the future position of the asteroid, given by

$$P_I = P_a + V_a \cdot t \tag{1}$$

The Law of Cosines tells us that length of side C of a triangle is given by

$$C^2 = A^2 + B^2 - 2 \cdot A \cdot B \cdot \cos(\theta_{AB}) \tag{2}$$

where A and B are the lengths of the other two sides, and θ_{AB} is the angle between A and B (and so opposite to side C). If in Figure 4 we assign $C = |V_b \cdot t|$, $A = |D|$, $B = |V_a \cdot t|$, and $\theta_2 = \theta_{AB}$, we have the equation

$$(V_b \cdot t)^2 = |D|^2 + (V_a \cdot t)^2 - 2 \cdot |D| \cdot (V_a \cdot t) \cdot \cos(\theta_2) \tag{3}$$

This is a quadratic equation, with a single unknown (t), and so we can directly solve it. We set the equation equal to zero and rearrange the terms into the canonical form $ax^2 + bx + c$. We then solve the equation using the quadratic formula

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{4}$$

We compute the determinant $b^2 - 4ac$ on one line, and then each of the solutions on a line of their own, producing intrcpt1 and intrcpt2. You may recall that the solutions to the quadratic formula might both be real-valued, or complex-valued (technically a pair of complex conjugates); in our case, complex values would indicate that no intercept is possible. However, since the player ship is stationary and the bullets are far faster than the asteroids, this won't happen. What *could* happen is that one of the values is negative, indicating that one possible intercept is a point the asteroid has already passed. Alternatively, both possible intercepts may yet lie in the future. We want the soonest possible intercept, so we take the smallest solution for t that is still positive, yielding bullet_t; this is one of the inputs to the fuzzy controller. Substitute this value for t in Eq. (1), and we have the intercept position. Finally, the line

```
my_theta1 = math.atan2((intrcpt_y - ship_pos_y),(intrcpt_x - ship_pos_x))
```

gives us θ_1, the direction we need to shoot in. We take the difference between the ship's current angle and θ_1, normalize it to $[-\pi,\pi]$, producing shooting_theta, the other input to our fuzzy controller.

2.2 Fuzzy Controller Design

The controller design code for scott_dick_controller.py is contained in the init() method, and thus runs once at object creation time. The code follows the control examples from the SKFUZZY documentation fairly closely; we first declare the universal sets for our two inputs (bullet_time and theta_delta) and two outputs (ship_turn and ship_fire), We then create our linguistic variables by defining fuzzy subsets of the universal sets, build our rules, and finally collect them into a ControlSystem object that we define as an instance variable (making it visible to the actions() method.

The ranges for the input and output variables are the entire unit circle for theta_delta and ship_turn; the former is in radians because that's what math.atan2() produces, while the latter is in degrees because that's what the game library requires. SKFUZZY does not actually have singleton fuzzy sets, so our Boolean firing output ship_fire has to be represented by fuzzy sets in [-1,1]. Finally, the range of values for bullet_time was determined by trial and error.

For control problems where I will have mostly large actions, but more refined ones when I am close to a target value, I prefer to define non-uniform fuzzy sets. I usually represent "Large" with a sigmoid function; skfuzzy.smf or skfuzzy.zmf, depending on which side should be open. I then represent "Medium" and/or "Small" with triangular functions (skfuzzy.trimf). Since bullet_time is exclusively positive, I only define the fuzzy sets L, M, and S for it. theta_delta and ship_turn can have positive and negative values, but they should be symmetric; I thus define NL (Negative Large), NS, Z (Zero), PS, and PL. Z is also a trimf, symmetric about 0. Ship_fire receives two trimf fuzzy sets, each with their maximum at the extremes, falling to 0 at 0.

For the rules, I turn the ship to reduce the value of theta_delta, at a rate proportional to the size of theta_delta. I mostly hold fire when theta_delta is Large, to improve accuracy; however, if bullet_time is Small, I will fire regardless of the value of theta_delta. (It's worth noting that these rules do have a problem; they often fire bullets right behind an asteroid moving tangentially to the ship. Correcting this problem is left as an exercise for the reader.)

2.3 Control Actions

Returning to the actions() method, once bullet_t and shooting_theta have been determined, I create a new ControlSystemSimulation object, and pass them in as inputs. The defuzzified rule outputs are available from the Dictionary object shooting.output. I directly pass `shooting.output['ship_turn']` as the turn_rate output, while I have to threshold the `shooting.output['ship_fire']` output to convert it to the Boolean fire. I set thrust to zero, and return all three as the outputs of the actions() method.