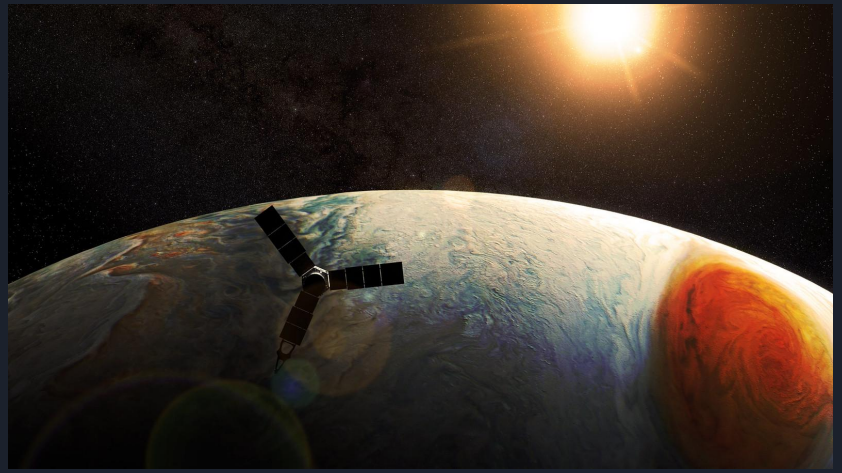
Reinforcement Learning for Spacecraft Orbits

Christopher Figueroa, Sai Shreya Jillella,
Damini Singh Thakur, Aishwarya Uyyala

Abstract

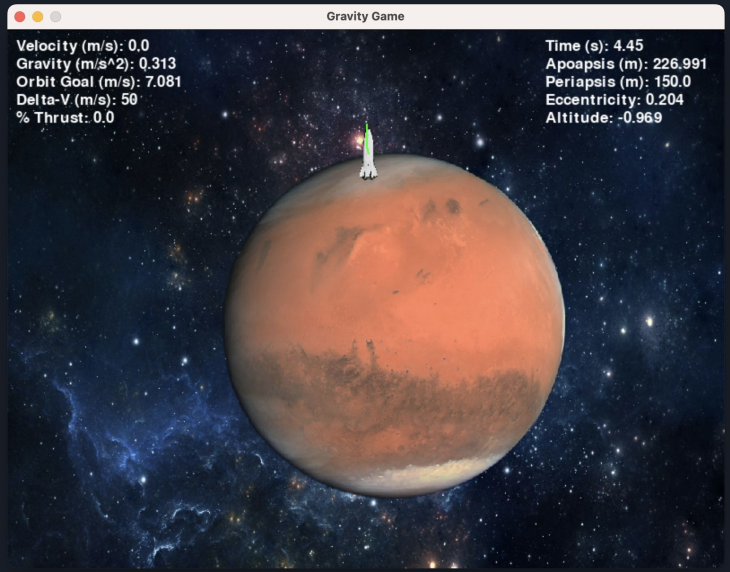
- Reinforcement learning trains spacecraft to orbit planets
- Randomized planet size and mass
- Adaptable to varying conditions
- 5 Essential equations:
  - Gravity: $G * (M / R^2)$
  - Orbital velocity: $\sqrt{((M*G) / R)}$
  - (A) Apoapsis: highest point in orbit
  - (P) Periapsis: lowest point in orbit
  - Eccentricity: (A-P) / (A+P)



Juno spacecraft in orbit around Jupiter

Motivation

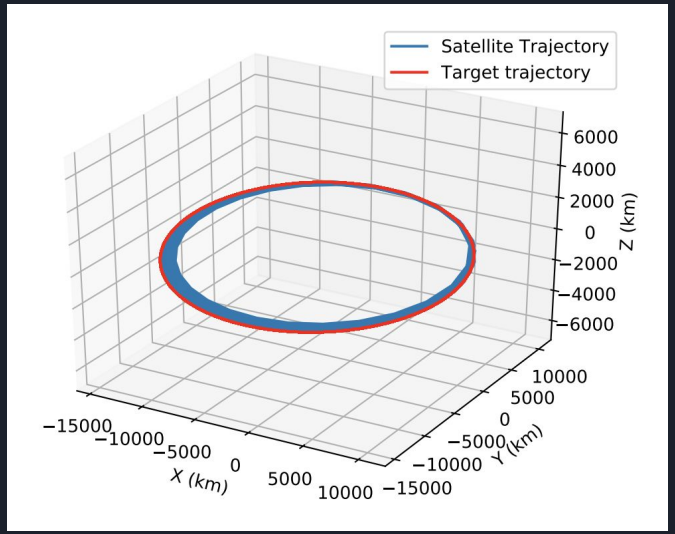
- Uses simplified game-optimized physics with relevant parameters
- Less computationally intense than other similar algorithms
- Randomized planet parameters:
  - Radius
  - Mass
  - Gravity
  - Orbital velocity
- Eliminates need for unnecessary and complex parameters
- Western Michigan University research is much more complex



Project game randomized planet Mars

Additional Research

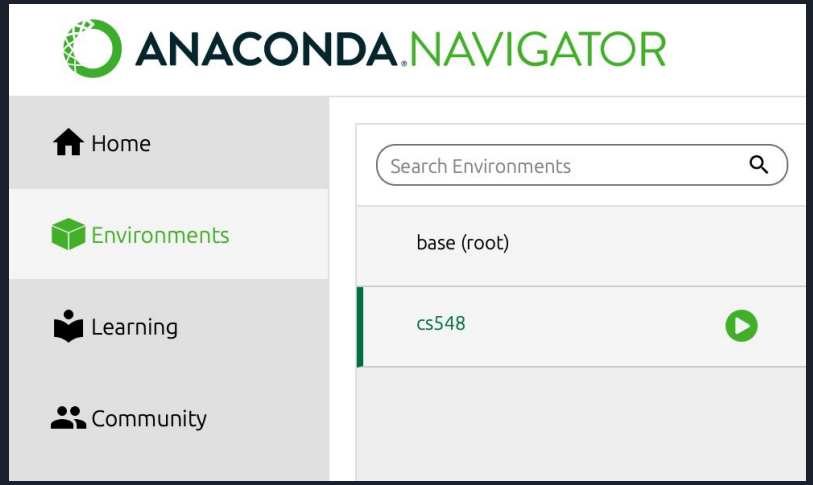
- Reinforcement learning research at Western Michigan University
- Research at Carleton University
- Realistic complex physics calculations
- Set on a real-life scale of an orbit around Earth
- Learning models:
  - Markov decision process
  - Q-Learning
  - Policy gradient
  - Reward function
  - Experience replay



Western Michigan University Earth Orbit[1]

Technical Description (Libraries Used)

- 2D Python orbital mechanics game
- Python libraries:
  - Pygame
  - Math
  - Numpy
- Reinforcement learning:
  - Torch
  - Matplotlib
  - Scikit-learn
- Anaconda environment
- Jupyter-notebook file

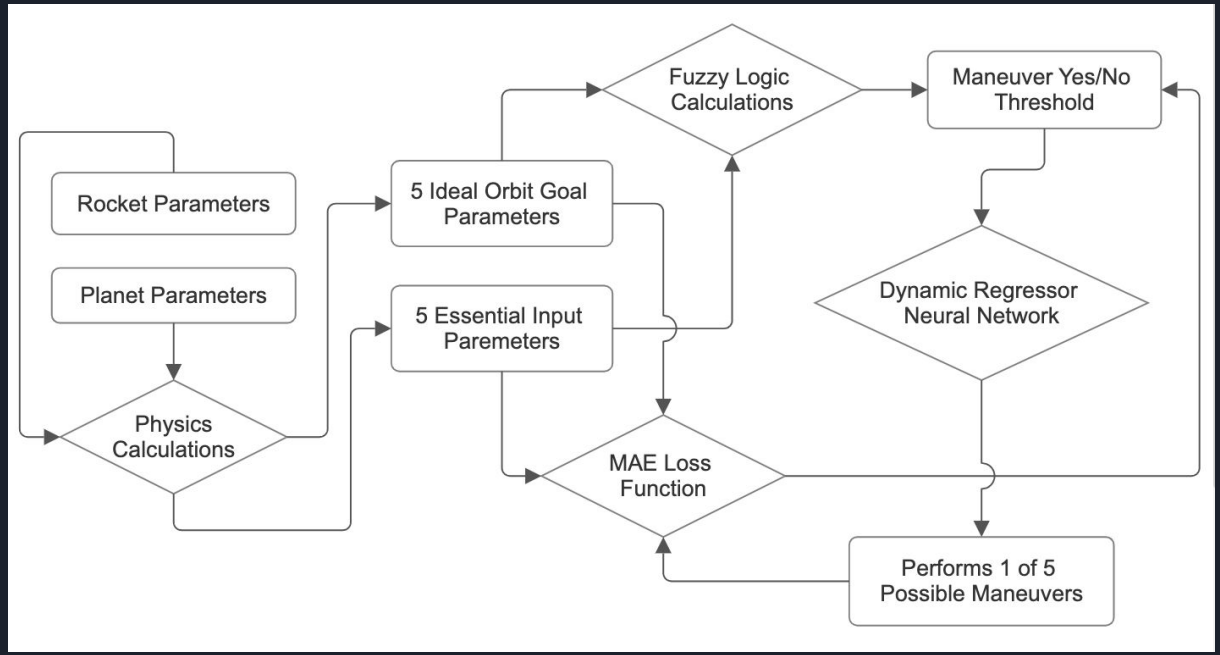


Python Anaconda Navigator Mac OS

Reinforcement Learning

- Learning models used:
  - Dynamic Regressor Net: allows for varying neural network sizes
  - Mean Absolute Error: loss function, part of Torch library
  - Sigmoid Activation: restricts action variable to a range of 0 to 5
- Action functions:
  - State: current gravity, speed, apoapsis, periapsis, eccentricity
  - Action: calls one of 5 possible spacecraft moves
  - Reward: depends on how close state is to ideal
  - Ideal: when spacecraft is in orbit
- 5 possible moves:
  - Rotate right, Rotate left, Throttle up, Throttle down, Apply throttle

Functions Diagram



Rocket/Planet Parameters

- G: gravitational constant, D: distance from planet, M: planet mass
- 5 Physics equations:
  - Gravity: $G * (M / D^2)$
  - Velocity: rocket movement in meters/sec
  - (A) Apoapsis: highest point in orbit
  - (P) Periapsis: lowest point in orbit
  - Eccentricity: (A-P) / (A+P)
- The 5 physics inputs used for fuzzy logic
- Ideal values:
  - Gravity around orbit altitude: planet radius + 50 meters
  - Velocity at orbit: $\sqrt{((M*G) / R + 50)}$
  - Apoapsis ≈Periapsis
  - Eccentricity ≈ 0

Fuzzy Logic Crisp Values

- Crisp: range of possible values for a given parameter
- 3 categorizations for each crisp range:
  - Low, Medium, High
  - The 5 parameters: gravity, speed, apoapsis, periapsis, eccentricity
- Assigned to rules that correspond with the values
- Rules determine is spacecraft should perform maneuver or not

```
# the periapsis represents the lowest point in an orbit (M)
periapsis['low'] = fuzz.trapmf(periapsis.universe, [0, 0, 70, 140])
periapsis['medium'] = fuzz.trapmf(periapsis.universe, [120, 170, 230, 280])
periapsis['high'] = fuzz.trapmf(periapsis.universe, [220, 260, 300, 300])

# eccentricity represents the difference between apoapsis and periapsis as a ratio of 1
eccentricity['significant'] = fuzz.trapmf(eccentricity.universe, [0, 0, 0.1, 0.2])
eccentricity['moderate'] = fuzz.trapmf(eccentricity.universe, [0.2, 0.2, 0.4, 0.8])
eccentricity['minimal'] = fuzz.trapmf(eccentricity.universe, [0.5, 0.6, 1.0, 1.0])

# defines the output variables
maneuver['no'] = fuzz.trapmf(maneuver.universe, [0, 0, 45, 55])
maneuver['yes'] = fuzz.trapmf(maneuver.universe, [45, 55, 100, 100])
```

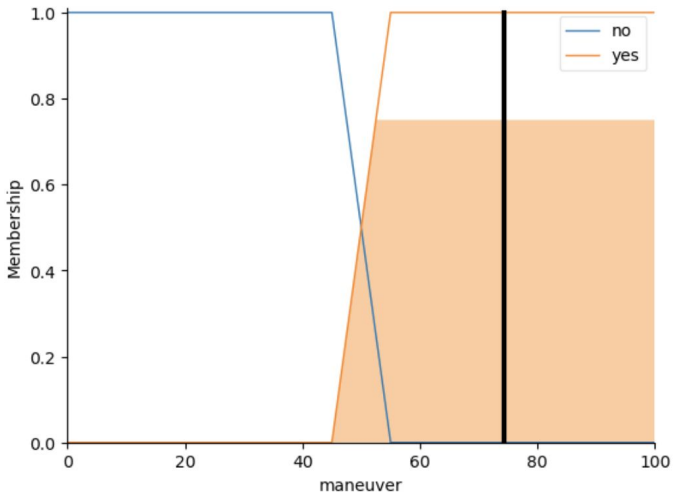
Fuzzy Logic Rules

- Low, Medium, and High crisp values are assigned to rules
- 2 rules for Fuzzy logic:
  - Upper/lower extremes: low/high values
  - No maneuver needed: medium values
- Medium values signify spacecraft has reached orbit
- Threshold value rule outcomes:
  - Score 0-50: spacecraft needs to maneuver
  - Score 51-100: ideal values for orbit

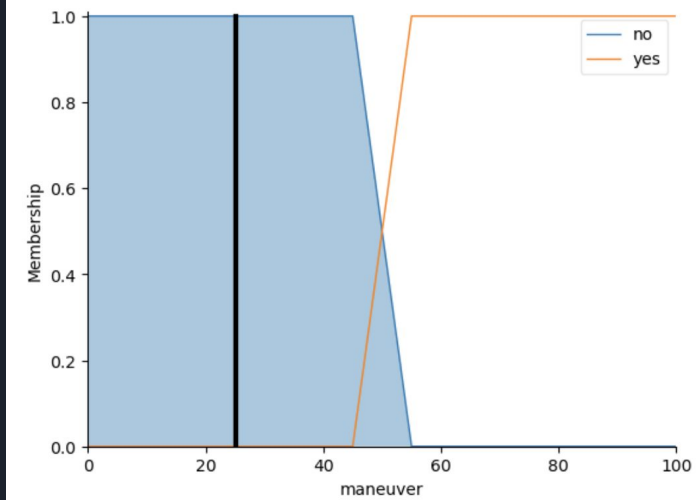
```
# the upper and lower extremes
rule1 = ctrl.Rule(gravity['strong'] | velocity['slow'] |
                  velocity['fast'] | periapsis['low'] | apoapsis['low'] |
                  apoapsis['high'] | periapsis['high'] |
                  eccentricity['moderate'] | gravity['weak'] |
                  eccentricity['significant'], maneuver['yes'])
# no maneuver needed
rule2 = ctrl.Rule(gravity['moderate'] & velocity['nominal'] &
                  apoapsis['medium'] & periapsis['medium'] &
                  eccentricity['minimal'], maneuver['no'])
```

Maneuver Threshold Graphs

Gravity: 1.1 SG
Velocity: 0.6 OF
Apoapsis: 180 M
Periapsis: 150 M
Eccentricity: 0.5
Maneuver Threshold: 74.32926829268293
Loss: 27.0



Gravity: 1.0 SG
Velocity: 1.0 OF
Apoapsis: 200 M
Periapsis: 200 M
Eccentricity: 1.0
Maneuver Threshold: 25.08333333333333
Loss: 0.0



Mean Absolute Error

- Takes in a list of current state values and ideal values
- Finds the absolute error between the state and ideal values
- Used to determine loss of current state
- Loss determines how close spacecraft is to achieving orbit

```
# initializes Mean Absolute Percentage Error loss function
def get_mape(invals, ideals):
    diff_grav = (abs(ideals[0]-invals[0]) / ideals[0]) * 100
    diff_vel = (abs(ideals[1]-invals[1]) / ideals[1]) * 100
    diff_ap = (abs(ideals[2]-invals[2]) / ideals[2]) * 100
    diff_per = (abs(ideals[3]-invals[3]) / ideals[3]) * 100
    diff_ecc = (abs(ideals[4]-invals[4]) / ideals[4]) * 100
    mape = (diff_grav+diff_vel+diff_ap+diff_per+diff_ecc) / 5
    return mape
```

Dynamic Regression Neural Network

- Dynamic Regressor allows for use of neural networks of varying sizes
- Non-dynamic Regressor yields incorrect shape error
- Features of network:
  - 5 features, one for each physics parameter
  - 5 hidden layers
  - 1 output, used for action variable

```
# initializes the Dymanic regressor net which allows for flexibility of size
class DynamicRegressorNet(torch.nn.Module):
    def __init__(self, n_hidden, n_output):
        super(DynamicRegressorNet, self).__init__()
        self.fc1 = torch.nn.Linear(in_features=5, out_features=n_hidden)
        self.fc2 = torch.nn.Linear(in_features=n_hidden, out_features=n_output)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

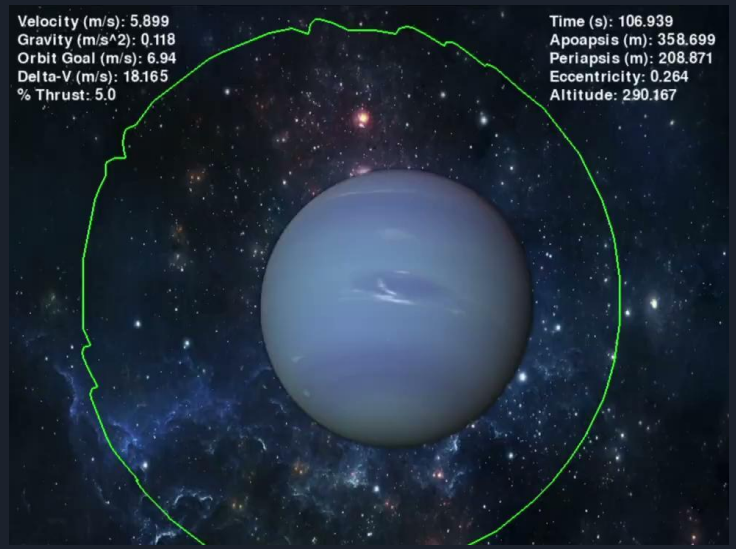
Reinforcement Learning

```
# runs reinforcement learning for 120 seconds
if gtime < 120:

    # gets the current state of the 5 crisp values
    state = [gravitational_force, speed, apoapsis, periapsis, (1-eccentricity)]
    crisp_grav, crisp_vel, crisp_ap, crisp_per, crisp_ecc = gravitational_force, speed, apoapsis, periapsis,

    # applies sigmoid activation function to restrict numerical range of action
    action = torch.sigmoid(net(torch.Tensor(state)))
    scaled_action = scale_and_shift(action, min_val=0, max_val=5)
    action_float = scaled_action.item()


    # applies an action depending on the action float value
    if action_float < 1:
        print("rotate right")
        rotate_ship(-10)
    if action_float > 1 and action_float < 2:
        print("rotate left")
        rotate_ship(10)
    if action_float > 2 and action_float < 3:
        print("throttle up")
        throttle = throttle + 0.025
    if action_float > 3 and action_float < 4:
        print("apply throttle")
        apply_throttle()
    if action_float > 4 and action_float < 4.9:
        print("throttle down")
        throttle = throttle - 0.025



    # compute the loss (reward) based on the updated crisp variables
    new_state = get_state()
    reward = get_mape(new_state, ideals)

    # update the neural network based on the reward
    loss = loss_func(net(torch.Tensor(new_state)), torch.Tensor([reward]))
    learner.zero_grad()
    loss.backward()
    learner.step()
```

- Action: determined by state net function
- State: the 5 physics input parameters
- Sigmoid: restricts action number to 0 to 5
- Action Number: determines what maneuver to perform
- Reward: uses MAE to determine loss
- Loss: determines what learner will do next
- Learner: determines next action value

Reinforcement Learning Demo

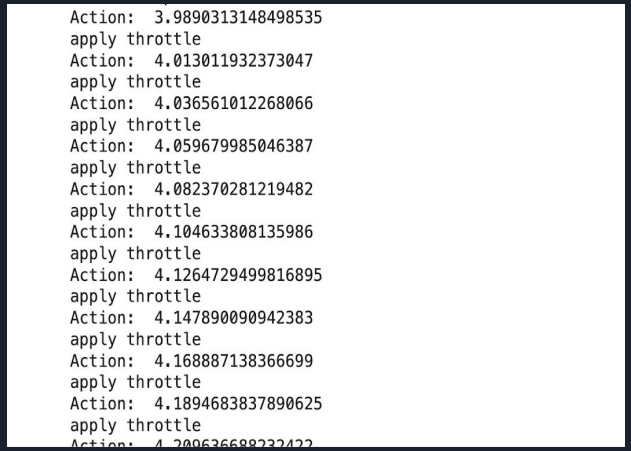
Beginning of learning loop



End of learning loop

Current Progress

- Fuzzy Logic and State comparison functions have no issues
- MAE functions and Dynamic Regressor Network work reliably
- Physics calculation progress:
  - Most of the parameters are accurately calculated
  - The periapsis value is lower than the expected value
- Reinforcement Learning progress:
  - Loss calculation is accurate
  - Learner steps are correctly scaled
  - Action selection occasionally gets stuck in the 5th action



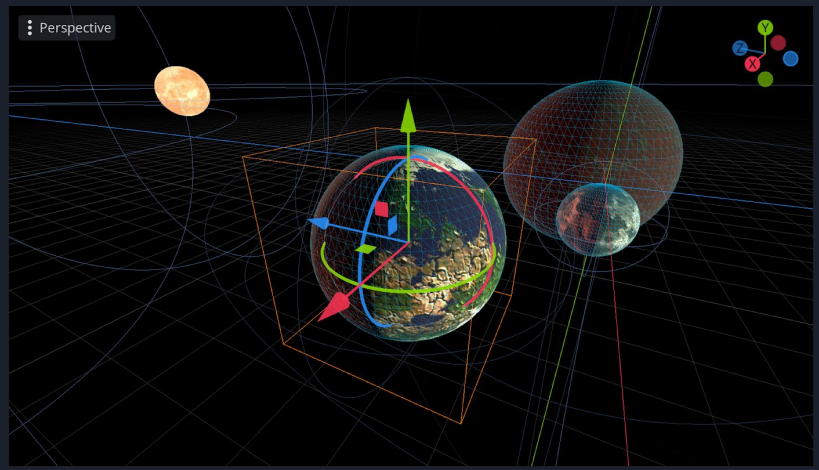
Action stuck on 5th apply throttle action

Conclusion

- Reinforcement Learning models can adapt to varying planets
- Planets with random parameters:
  - Radius, Mass, Gravity, Orbital velocity
- Spacecraft requires only 5 physics calculations:
  - Gravity, Velocity, Apoapsis, Periapsis, Eccentricity
- Models Used:
  - Fuzzy logic
  - MAE loss
  - DRN Network
- More efficient than more complex spacecraft learning models
- Suitable for 2D video games

Future Work

- Plans to expand to 3D game spacecraft training
- Expand into real-scale planets
- Similar physics parameters, but with z-axis added
- Godot game engine libraries
  - AI Godot Gym
  - Godot-Python
  - GD Extension



Godot 3D solar system

Video Code Explanation

Sources

- [1] Western Michigan University research: https://scholarworks.wmich.edu/cgi/viewcontent.cgi?article=4537&context=dissertations
- [2] Orbital parameters: https://courses.lumenlearning.com/suny-osuniversityphysics/chapter/13-5-keplers-laws-of-planetary-motion/
- [3] Gravity and momentum: https://oer.pressbooks.pub/lynnanegeorge/chapter/chapter-1/
- [4] Carleton University research: https://www.aiaa.org/docs/default-source/default-document-library/publications/ondeepreinforcementlearningforspacecraftguidance.pdf
- [5] Specific orbital energy: https://deutsch.physics.ucsc.edu/6A/book/gravity/node15.html
- [6] Apoapsis/Periapsis: https://faculty.fiu.edu/~vanhamme/ast3213/orbits.pdf