The Insubordinate Satellite: Soft Actor-Critic Regulator Exploring the Effect of Reward Shaping

Matthew Grewe, Esther Revenga Villagra, and Kaylie Rick
University of Colorado, Boulder, Smead Aerospace Department

With the advancement of reinforcement learning, spacecraft can be tasked to autonomously control their movement. This allows for increased autonomy and optimization while on orbit along with the ability to autonomously solve issues that arise. The team used soft-actor critic use a spacecraft's reaction wheels to control a spacecraft's angular momentum. The team is able to control the spacecraft's angular velocity but not the attitude since reinforcement learning is not well suited for this problem. The team tested different mission scenarios and different reward functions in order to get the most optimal results out of the soft actor critic algorithm.

I. Introduction

Three levels of objectives are set for this project. Originally Level 1 was to point and regulate the attitude of a spacecraft (SC) using multiple variable speed control moment gyroscopes (VSCMGs). However, attitude pointing is extremely difficult using Reinforcement Learning (RL) and the team was advised to change scope by PhD student Mark Stephenson and Professor Zachary Sunberg. The newly revamped Level 1 for this project is to use RL algorithms to regulate a SC for three different mission scenarios (initial conditions) using Reaction Wheels (RWs). Regulation consists of driving the angular velocity of the SC to zero, in other words, stops it from spinning. Level 2 is now updated to perform reward shaping and attempt to improve the performance of the RL algorithm. Lastly, Level 3 is to achieve tracking with RWs (pointing and regulating simultaneously).

II. Background and Related Work

In order to understand this project, some basic knowledge of attitude dynamics terms is needed. Attitude is the orientation of the SC in space. In this project, the orientation is of the SC's body frame relative to an inertial frame. The attitude of a SC can be described in a lot of different ways. In this project Modified Rodriguez Parameters (MRPs) will be used. This is denoted as σ. An important note about MRPs is that they only have a singularity at a rotation angle of 360° (not at zero though), so in order to avoid said singularity, a switch to their shadow set is performed. This switch can happen at any time, but it is typically convenient to do it when the norm of the MRP set becomes greater than 1, meaning the rotation is over 180°. These switches will show up later in plots where it will look like the attitude suddenly changes even though it does not. The attitude rate is denoted as ω, and it is the rate of change of the body in each of it's principal body axes. Therefore it is defined in the body frame.

A. Momentum Exchange Devices

The attitude of a SC can be controlled by applying torques (changes of angular momentum) to momentum exchange devices. Since no external torque is present (simplifying assumption for this project) the total angular momentum of the system must be conserved, so an equal an opposite change in angular momentum (torque) happens to the SC. One of theses momentum exchange devices is reaction wheels (RWs) that change the speed of a wheel in order to increase/decrease the angular momentum along the axis it spins about. For example: if a RW is lined up with the angular momentum vector about the X axis of the SC and a torque is applied to it that makes it accelerate in the clockwise direction, then the SC will respond by spinning in the counterclockwise direction. However, since the inertia of the SC is much larger than the inertia of the RW, the SC will rotate at a much slower speed than the RW. In previous work (homework and projects in the "ASEN 6010 Advanced Attitude Dynamics and Control" class), the SC was regulated with Lyapunov-derived feedback control. This form of control uses an energy like function and its derivative to define a control law.

III. Problem Formulation

This problem is formulated as a Markov Decision Process (MDP). In this formulation, the state space is defined to be continuous as it includes a combination of the attitude MRP and angular rates of the body with respect to the inertial frame. These were set to have an upper and lower bound. In the case of the MRPs, the allowed values were between -1 and 1 for each component because the switch to the shadow set occurs at norm 1. On the other hand, the angular rates were set to have a maximum and minimum values of -10 and 10 rad/s since anything above that would be too fast of a rotation anyway in a typical mission scenario. The SC is equipped with 4 RWs in a pyramid configuration, so the action space is also continuous and is made up by a four-dimensional vectors representing the torque applied to each RW. The lower and upper bounds on the allowed torques are -2 to 2 Nm, respectively. The transition is deterministic using the equations of motion derived from the most simple equation: torque is equal to the change in angular momentum $\dot{H} = L$. In summary, the transitions are defined by the attitude dynamics of the SC which include no process noise.

Finally, after spending a considerable amount of time determining an appropriate reward function for the system, the team consulted [1] and concluded that there was not an appropriate way to shape the reward to achieve tracking (control both the attitude pointing and the rotation speed of the SC). Therefore, the team focused on regulating the SC. Various reward functions that the team attempted throughout the project include (but are not limited to):

- $R = -\sum_{i=1}^{3} |\omega_i| - \sum_{j=1}^{4} |a_j| \longrightarrow$ (Used to verify Level 1 requirements)
- $R = -10 \sum_{i=1}^{3} |\omega_i| - log(\sum_{j=1}^{4} |a_j|)$
- $R = -\sum_{i=1}^{3} |\omega_i| \sum_{j=1}^{4} |a_j|$

where ω is the angular velocity of the SC body and a is the action vector of the torque applied to each RW. Note that these might not be the final reward functions used for the reward shaping of Level 2. More discussion on those will come later. However, in all cases, a reward of +100 was always awarded for reaching the terminal state (less 0.001 rad/s spin about all axes). It may be noticed that all the reward functions used have a $\sum_{j=1}^{4} |a_j|$ term. This is because one of the goals of the project is to try to minimize the amount of power consumed by the SC to regulate itself. Under the assumption that power will be directly proportional to the amount of torque applied, all the reward functions are designed to punish the use of high torques.

IV. Solution Approach

A. Reaction Wheel (RW) Dynamics - State Transition Function

The dynamics of the SC are used (and defined) as the state transition function of the MDP. A function was created to take in an input action (RW torque) from the RL agent and use a Runge Kutta forth order integration (RK4) to integrate the complex coupled dynamics by braking them up with a technique detailed in [2]. This function was developed from scratch by the team last semester and can be tuned to use reaction wheels, CMGs, or VSCMGs of any number, size, and orientation within the SC. For the purpose of this project, none of the 4 RWs are aligned with any of the 3 principal axes of the SC.

B. Reinforcement Learning Algorithm

To solve this problem, given the continuous state space and action space, the authors developed a Soft Actor-Critic (SAC) RL agent. SAC is an off-policy deep reinforcement learning algorithm that is capable of dealing with continuous state and action spaces. The team decided to use MATLAB's Reinforcement Learning Toolbox to implement and train the SAC agent. This seemed like the obvious choice since the state transition function had already been defined in MATLAB for the attitude dynamics of a SC from the ASEN 6010 course homework and project.

The SAC agent can be trained in a custom environment that was designed to represent the MDP definition of the problem. Said environment includes the custom definition of the continuous state and action spaces, which were bounded to represent the physical limitations of the problem (maximum and minimum torque available to actuators, MRP norm bounds, etc.). Both the state and action spaces were defined using MATLAB's `rlNumericSpec` function, which takes in the dimension and bounds of each of the spaces. The environment was then defined using the `rlFunctionEnv` function, that takes in those spaces as well as the state transition function and the `reset` function, which was derived from scratch for this problem.

The state transition function was reused from ASEN 6010's second homework assignment (developed following the guidance in [2]), which had the option to apply torques to a set of momentum exchange devices (at the time it used VSCMGs). The transition from VSCMGs to RWs was fairly straightforward, the wheel was just kept in place so that it

could not gimbal and create the huge torques that VSCMGs/CMGs produce. The reset function was defined to take the environment back to the initial state defined for each mission scenario.

Once the environment is defined and available, MATLAB provides a relatively simple way to define the actor and critic neural networks, as well as the training loop. It requires the definition of the RL agent, which then gets trained using the `train` function included in the RL toolbox. The agent is defined with the function `rlSACAgent`, which can either take in the state and action spaces and use MATLAB's default actor and critics networks, or (and this is what the authors implemented) it can take in custom actor and critic(s) neural networks.

Below there is a graphical representation of the actor and critic neural networks defined for this project (Figure 1). The actor network takes in the state and outputs the action, while the critic network takes in the state and action and outputs the Q-value. It is worth noticing that the structure of these networks was largely inspired by MathWorks' documentation on the subject [3]. The figure below shows the layers of the network structures (note that each fully connected layer "FC" contains 24 neurons), as well as the activation functions for each layer (ReLU for all layers except the actor's output layer, which uses a tanh activation function - once again, recommended by MathWorks [4]).

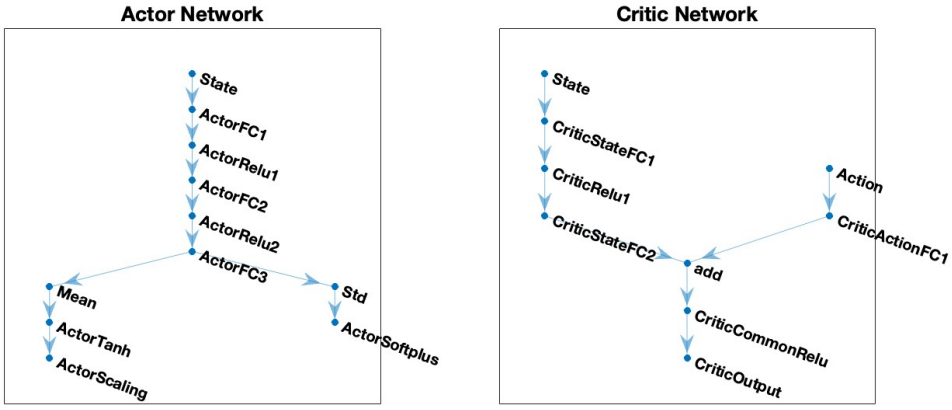


Fig. 1 SAC Actor and Critic Neural Networks

The reward function is defined within the step transfer function, as well as an analogue of an `isterminal` function. The reward function is a big part of this project, which focuses on the influence of its shape on the overall performance of the system. Given that it is defined in terms of the current state and action, one can choose to punish one more over the other (ex. punish high torques if you are interested in minimizing the amount of power consumption of a maneuver). Note that the step transfer function is required to output the next state (s'), the reward, a boolean determining if the state is terminal, and the "updated information" of the environment. There is a struct that can be passed in and out of functions containing relevant information for the environment that goes beyond the state. This includes the current speed of the wheels, the constants used in the problem, the current terminal state (in case that changes over time), etc.

Finally, both the `train` and `rlSACAgent`functions in MATLAB are accompanied by the `rlTrainingOptions` and `rlSACAgentOptions` functions, respectively. These functions allow for tuning of the training loop and agent as they allow for the modification of MATLAB's default properties. Some of the main fields that were edited/tuned for this project were the following: sample time, experience buffer length, mini-batch, discount factor, number of epochs (that it trains with the same data), target critic update frequency, number of warm-up steps, max training episodes, and max steps per training episode. Similarly, the learning rate and gradient threshold of the critic can also be adjusted with the `rlRepresentationOptions` function that works in conjunction with `rlQValueRepresentation` to define the critic. MATLAB also provides a `sim` function that takes in the trained agent, the environment, and the simulation options (defined through `rlSimulationOptions`). This function simulates an episode with the trained agent in the environment for however many steps are defined in the options (or until the terminal state is reached).

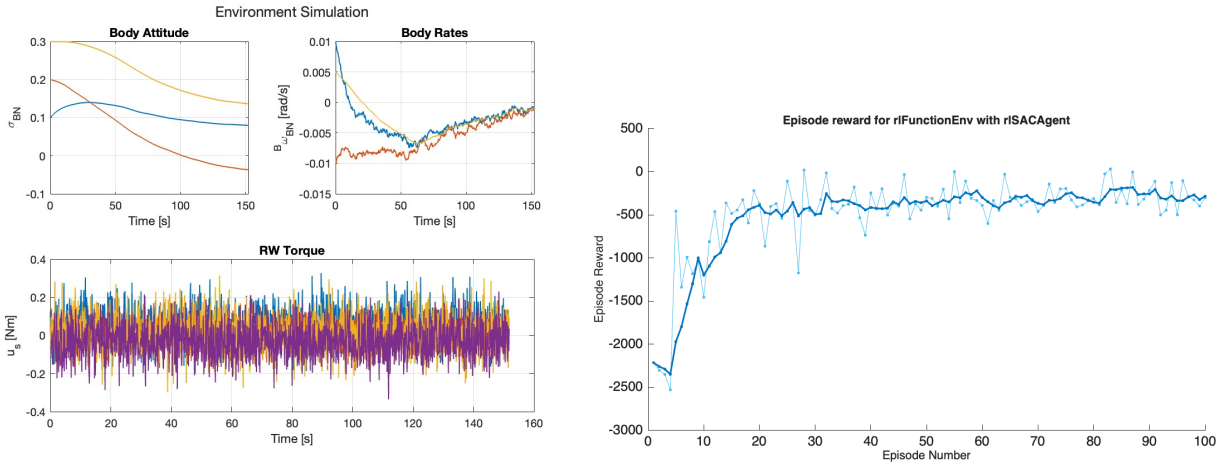
After fine tuning all these parameters, the authors arrived to a decent enough definition of the SAC agent and setup of the training loop. From there, the analysis of the reward shaping and actuator choice could begin, with the ultimate goal of attempting to point the SC in a predefined direction (and get it to maintain said pointing).

V. Results

A. Level 1

The level 1 requirement is to regulate a SC spinning in space assuming attitude position is known and the transition matrices are deterministic (i.e.: there is no process or measurement noise and the state can be measured directly). Three different initial conditions for the SC are used to compare the algorithms performance under different scenarios. The first initial condition is $\sigma_0 = [0.1, 0.2, 0.3]$ and $\omega_0 = [0.01, -0.01, 0.005]$ rad/s. This is a reasonable amount of spin: not too high, not too low. The second set has the same σ_0 but now $\omega_0 = [0.3, 0, 0]$ rad/s. This is a spin only about one axis. This should show that some RWs are favored over others because the RWs have different levels of control authority over different axes since they are in a pyramid configuration. The third set of initial conditions is a faster spin which represents a tumbling SC. This is a fairlry extreme situation that a satellite would be expected to face upon release from the cargo vehicle into its orbit. The initial condition in this case is set to be $\sigma_0 = [0, 0, 0]$ and $\omega_0 = [0.1, 0, -0.5]$ rad/s. Recall that the reward function used for this Level of the project was: $R = -\sum_{i=1}^{3} |\omega_i| - \sum_{i=1}^{4} |a_i|$

Under these three different initial conditions, the SAC agent was trained and then an episode was simulated with `sim`. This function outputs the evolution of the state along the episode and the applied actions. The training output can also be visualized as a learning curve with the `inspectTrainingResult` function. The results of the first set of initial conditions can be seen below in Figure 2. Figure 2a shows the attitude and attitude rate along with the torque applied to each RW. Since there is no reward for achieving any particular pointing, the "Body Attitude" plot is arbitrary for Level 1. The "Body Rates" plot shows what the agent was able to drive the angular speed to 0 (with a margin of 0.001 rad/s). That subplot also shows that the rates overshoot before reaching the desired value. Therefore, it can be concluded that the was a successful in this case. The bottom subplot in Figure 2a shows the action being taken in the form of torque on each RW. This plot looks pretty noisy, however, there are patterns that can be easily recognized: the average torque value applied to the "purple" RW is less than the average torque applied to the "yellow" or "blue" RWs. This indicated that the last two had more combined control authority over the spin axis than the purple one. Figure 2b is the learning curve for soft actor critic. This plot shows the episode's reward in light blue and average reward (5 episode moving window) in dark blue to see how quickly the SAC agent learns. It is also worth noticing the relatively low value of the torques applied, especially when compared to the next cases, as the initial spin was fairly low so the agent was happy applying small torques at the expense of keeping the low angular rate for longer time (took 150 s to reach terminal).

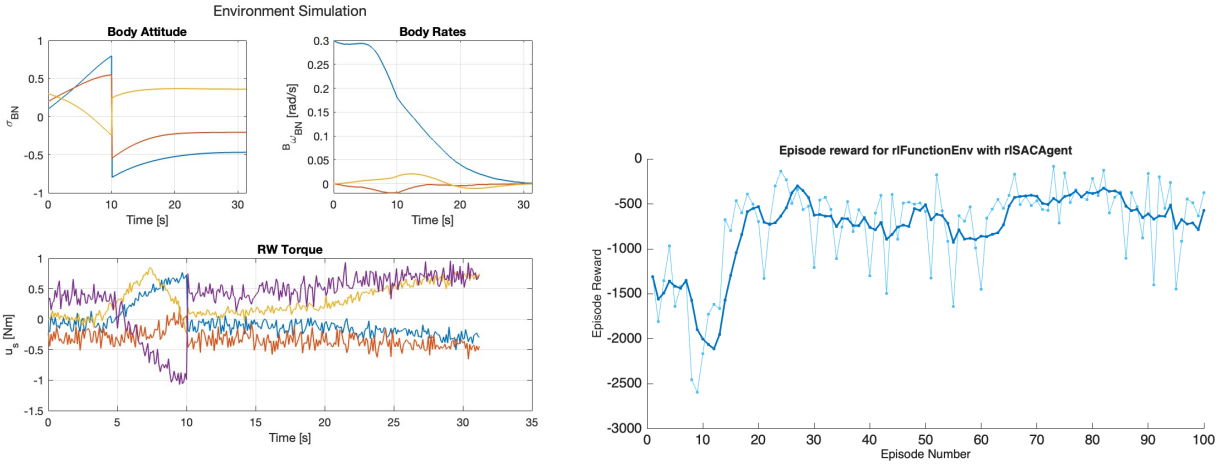


(a) Environment Simulation with trained agent **(b) SAC RL Learning Curve**

Fig. 2 IC1: Small initial spin about every axis

The next case starts with a rotation about only 1 axis. The result of the training is shown in Figure 3. There are some key differences between this run and the previous one. The input torque is more distinct from one another, showing a clear preference for some RW over others. This is due to the initial spin being about one axis, forcing the preference of the RWs that have higher control authority over that axis. It is also quite successful at keeping the angular rates that started at zero close to zero, and just decreasing lowering the speed about the axis of spin. Interestingly, this simulation shows one of the issues with RL in attitude control settings: it struggles to learn that the MRP will switch at norm 1

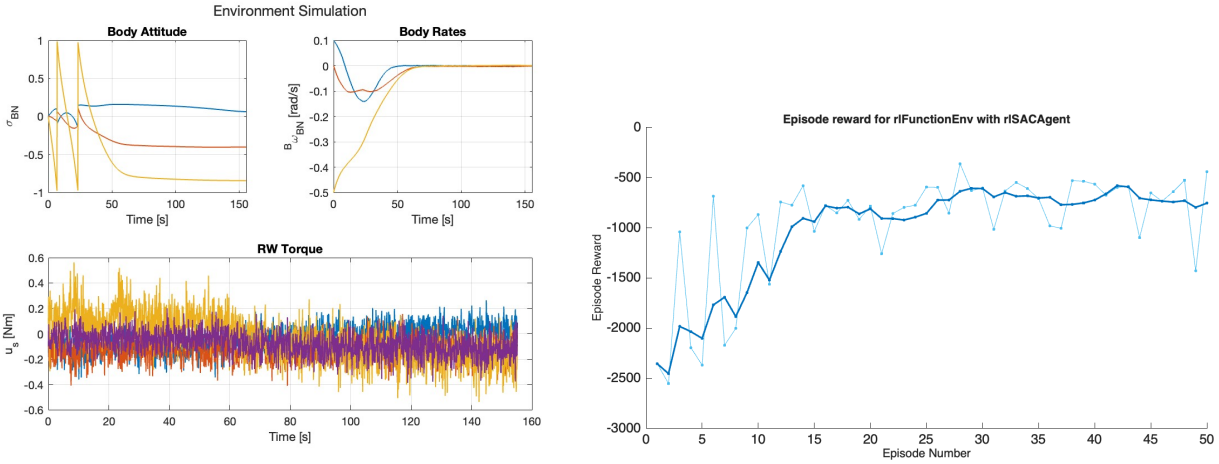
and instead it thinks that the state teleported from one place to another, causing the decision to switch the torques to other RWs. Despite this, it still manages to drive the angular speed to zero in a very efficient way. In fact, it takes less time, about 30 s, to achieve this goal when the previous one took about 150 s. This shows that in the trade-off between minimizing torque applied and time spent to get to the terminal state, the torque lost and the agent decided to apply larger torques to decrease the angular speed faster. This makes sense because the reward function punishes equally the amount of torque and the amount of spin, so it will want to minimize first the largest source of negative rewards (in this case the angular rate).



(a) Environment Simulation with trained agent **(b) SAC RL Learning Curve**

Fig. 3 IC2: Medium speed initial spin about one axis

Lastly, the third set of initial conditions represents a tumbling SC (larger values of angular rates). The torques applied (Figure 4a) are on average larger than in the 1st case but smaller than in the second. However, it maintains the higher torques for longer time. The takeaway of this is that initially it tries to bring the spin about the yellow axis to zero at the expense of overshooting with the blue one and unsettling the red one that was already at zero. It also seems like it was using mainly the yellow RW to do so. However, after t = 60 s it looks like the trade-off balanced and it attempted to use less torque as it became the main source of negative rewards. It is worth noticing that around this time the angular rate that it had was fairly similar to the initial condition for case 1, so it makes sense that it exhibits a similar behavior.



(a) Environment Simulation with trained agent **(b) SAC RL Learning Curve**

Fig. 4 IC3: Large speed initial spin

B. Level 2

In order to solve Level 2 requirements, the team looked at varying the reward function to see if it was possible to get better performance. Note that for every case moving forward, the initial conditions are the same as they were for the 1st case in Level 1. In Figure 5 one can see the reward shape that the team started with and used for Level 1. This reward has a shape that follows the function $R = -\sum_{i=1}^{3} |\omega_i| - \sum_{j=1}^{4} |a_j|$ is planar with a +100 reward when the motion of the SC has been arrested (within 0.001 rad/s). This solution seems to work fairly well, bringing the SC to a halt using quite a low amount of torque. As well, one can see in Figure 2b that the agent was able to train very well under all three initial conditions with fairly steep learning curves and then plateauing once it learned.

In an attempt to achieve a different performance, a new reward function was defined: $R = -\exp(3\sum|\omega_i|) - \sum|a_j|$. This function was intended to punish the angular rates much more than it does the torques, hopefully driving them to zero faster. It is clear from Figure 7a that it did achieve said goal, managing to drive the rates to less than 0.001 rad/s in 40 s. Compare this to Case 1 in level 1 when it took 150 s to do the same with roughly the same amount of average torque, and it would be clear that this reward shape is better suited and increases the performance of the controller. Furthermore, the way it decreases the angular speed is quite "asymptotic-looking" except for a minor overshoot in the angular rate about the "blue" axis, making this reward function arguably better than the previous one.

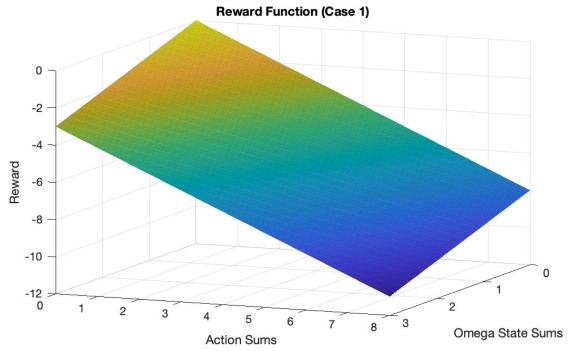


Fig. 5 Case 1: Shape of the reward function used in Level 1 (Figure 2)

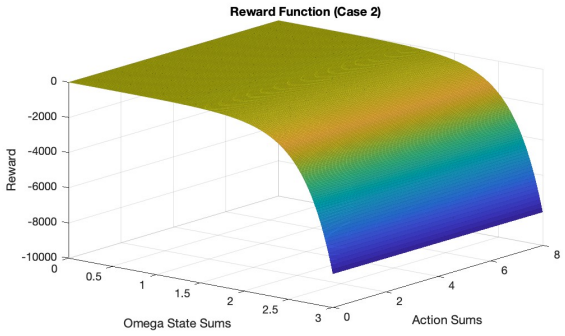
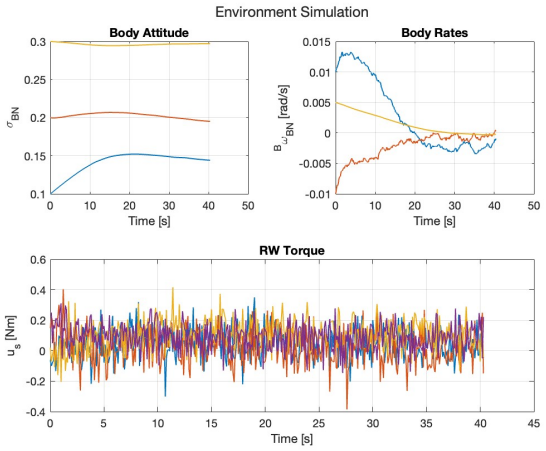
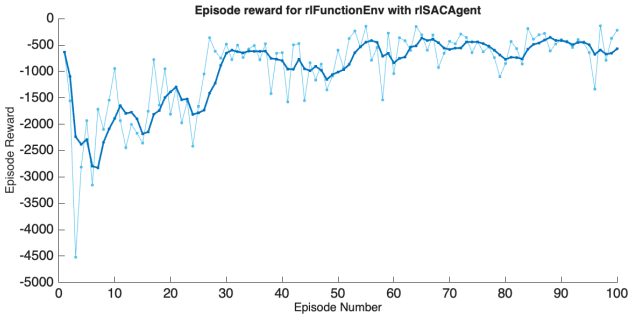


Fig. 6 Shape of the reward function for Case 2



(a) Environment Simulation with trained agent



(b) SAC RL Learning Curve

Fig. 7 CASE 2: $R = -\exp(3\sum|\omega_i|) - \sum|a_j|$

Finally, with the goal of finding a similar performance to Case 2 above but with another reward shape, the team devised a different reward function: $R = -10\sqrt{\sum|\omega_i|} - \sum|a_j|$. This function is less dramatic than Case 2 with a smoother curve that penalizes large angular rates. The main difference between the two is that this one is not an

asymptotic function, so it doesn't shoot off to negative infinity too fast. It is also a concave up function while the previous one was concave down. Judging from the results seen in Figure 9a, this function still manages to fulfill the goal of arresting the motion faster than the one used in Case 1 (it terminates in 50 s instead of 150 s). However, it is worse performing than the function used in Case 2. Not only it takes about 10 s longer to stop, but it also overshoots much more and uses a larger average torque. After designing 3 candidate reward functions, and many others that failed, the second reward function, was the best performing one.

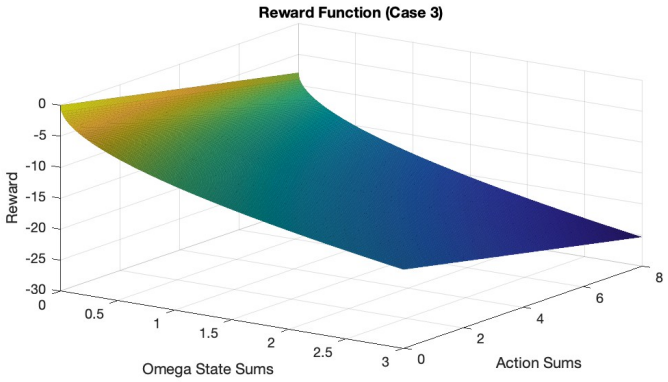
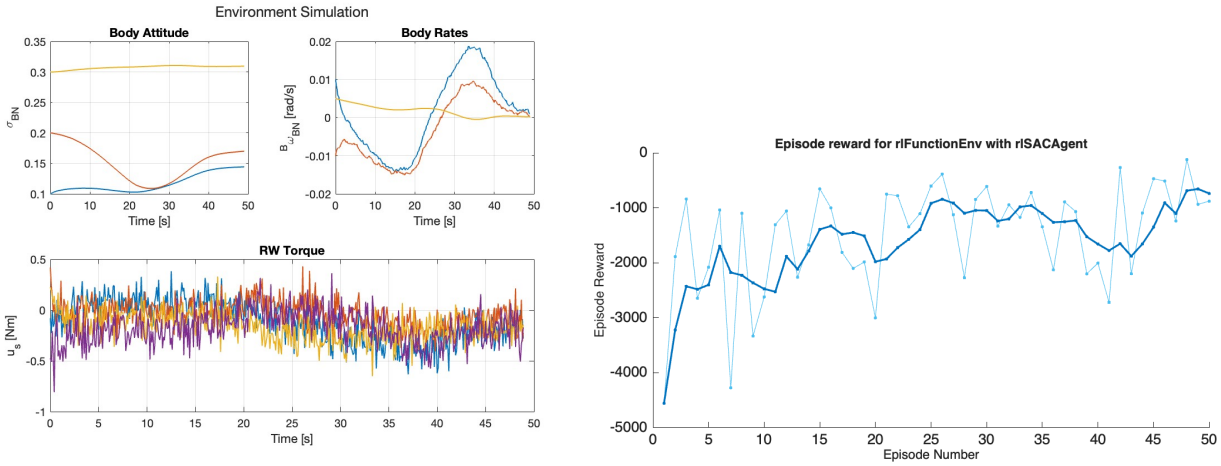


Fig. 8 Shape of the reward function for Case 3



(a) Environment Simulation with trained agent **(b) SAC RL Learning Curve**

Fig. 9 CASE 3: $R = -10\sqrt{\sum |\omega_i|} - \sum |a_j|$

C. Level 3

After many working hours and consulting [1], the team figured out that RL is not well suited for this problem. The soft actor critic method, the only method strong enough for this problem to deal with a continuous action and state space, could stop the SC from spinning or point it in a certain direction. When we tried to make the SC do both of these at once, the SAC agent was not able to learn a value function and would actually crash the simulation attempting to implement its learned policy (the accumulated reward that it achieved was so negative that MATLAB could not handle the number -1×10^{38}). Seemingly, SAC is so unstable/jittery when it comes to determining a policy that it makes pointing extremely hard for the agent to achieve while it attempts to stop. Apparently, reinforcement learning can be used for some simple problems in SC attitude control. However, as soon as the complexity increases, it becomes a really hard task to train an agent to perform anywhere close to well. At that point, it is not worth diving into these difficulties when the dynamics of the SC have been studied extensively and attitude control laws have been written efficiently to

achieve these goals in a far more optimal way than RL. Therefore, even though we tried for many hours to find the right reward function to solve this problem with RL, it will probably remain (for now) as a problem to be solved using traditional control methods.

VI. Conclusion and Possible Future Work

Using RL for SC attitude control seemed like a decently good and interesting idea. However, pointing a SC in a certain direction and halting the rotation of the SC are two different problems that can not be tackled simultaneously with RL. The SC always seemed to want to arrest its motion at a random attitude or spin as quickly as possible so that at some point it would reach the desired attitude in the simulation. The team worked to overcome these issues through numerous hours of reward shaping and consulting with people experienced in the field (such as Mark Stephenson [1] and [5]), but ultimately could not achieve the level 3 goal of arresting the motion of the SC at the desired attitude. Lucky for the team, the level 3 goals were stretch goals and while they ultimately were not able to achieve these goals, they were able to learn a lot more about what the potential benefits and draw backs of reinforcement learning were and the importance of selecting the right shape for the reward function. Additionally, the team was able to achieve their level 1 requirements of regulating the the angular velocity of the SC with the Soft Actor-Critic RL method, enabling the definition of a continuous state and action space. The team was also able to achieve the level 2 goal by varying the reward function and increasing the performance of the regulator. After starting with a reward function that was not particularly good (Figure 5), the team was able to modify the reward shape to improve the performance.

A very interesting spin-off from this project was discovered in MATLAB's documentation for RL: one can have more than one agent trained in the same environment. It might be possible to have two agents with two different goals: to regulate the SC and to point it in a particular direction. Ideally these two agents could be trained to perform together and attempt to achieve the level 3 objective that we never could. A possible drawback of this could be the chance to have the two agents fight each other instead of work together, but nonetheless, it would be interesting to see the dynamics of those interactions. Finally, if someone were able to get the double agent shenanigans to work, they could look at the computational efficiencies of using RL against traditional attitude control to determine if this could be a feasible product to implement on an autonomous SC on orbit some day.

VII. Contributions and Release

Esther wrote the SAC neural networks and the training loop, and experimented with different reward functions. Matthew wrote the attitude dynamics, and experimented with different reward functions. Kaylie wrote the attitude dynamics and experimented with different mission scenarios. Everyone helped write the report.

The authors grant permission for this report to be posted publicly so that it may help others in their quest of finishing this course without fighting their RL agent for hours, days, or weeks on end. The authors also recomend avoiding the delusional belief that an RL agent can do more than regulating a SC without spending an unearthly amount of time looking at a progress bars. Go study your attitude dynamics kids, there is a reason why RL has not been used in real attitude control thus far.

References

[1] Stephenson, M. A., "Practical Generalizable Methods for Learning Asymptotically Stabilizing Controllers," , 2023.

Mark's paper helped the team learn what they were doing wrong initially in the tracking problem and why they should turn it into a regulation problem. After reading Mark's report, the team was able to determine new appropriate requirements and some potential reward functions to use and how they could be shaped for better results.

[2] Schaub, H., and Junkins, J. L., *Analytical Mechanics of Space Systems*, 4th ed., AIAA, 2018.

This book was particularly helpful in creating the equations of motion for the spacecraft simulatuon. The team used this book to develope the equations of motion in their ASEN 5010 and ASEN 6010 classes where they learned the fundementals of attitude control and attitude dynamics.

[3] "MathWorks - Q Value Function," , 2024. URL https://www.mathworks.com/help/reinforcement-learning/ref/rl.function.rlqvaluefunction.html,

MathWorks' documentation has a lot of helpful insight into how to design the actor' and critics' neural networks, as well as a general example of how to connect all the different functions that are required to set up all the RL training algorithm.

[4] "MathWorks - Soft Actor-Critic Agent," , 2024. URL https://www.mathworks.com/help/reinforcement-learning/ug/sac-agents.html,

The team originally started by using MATLAB's built in Soft Actor Critic function. This gave them an initial idea as to how it worked and the requirements that went into it. After the team had gotten the soft actor critic to work by itself, the team looked into designing their own, hopefully more efficient agent that they then used for the rest of the project.

[5] Ito, K., and Yanao, T., "Adaptive Attitude Control of Spacecraft via Deep Reinforcement Learning with Lyapunov-Based Reward Design," *The 31th Workshop on JAXA Astrodynamics and Flight Mechanics*, 2021. URL https://jaxa.repo.nii.ac.jp/?action=repository_action_common_download&item_id=48318&item_no=1&attribute_id=31&file_no=1,

This paper gave the team another understanding about how reinforcement learning could be used on sppacecraft. While the paper did not explicitt state what algorithm or attitude set the researchers used, it was helpful in teaching the team some of the potential pitfalls that they may encouter while trying to figure out how to use reinforcement learning for spacecraft.

Appendix

```m
% ASEN 5264 Final Project
% Authors: Esther Revenga, Kaylie Rick, Matthew Grewe

clc; clear; close all; format compact;

% Add attitude functions folder to the MATLAB path
addpath(genpath('..'))

% Train and simulate the SAC agent in the environment
[tStates,simStates,tAct,simActions,trainResults] = CustomSAC_TrainAndSim();
save('GoodRuns/trainResults_level3.mat','trainResults')
inspectTrainingResult(trainResults) % Plot the training curve

% Plot the simulation
CMGplot(tStates,simStates,tAct,simActions)
savefig(gcf,'GoodRuns/SimPlot_level3.fig')
```

```m
function [tStates,simStates,tAct,simActions,trainResults] = CustomSAC_TrainAndSim()

    c = CMG_variableDef();

    % Define the environment
    env = createRLenv();
    obsInfo = getObservationInfo(env);
    actInfo = getActionInfo(env);

```

```
    % Define the critic
    statePath = [
        featureInputLayer(obsInfo.Dimension(1),'Normalization','none','Name','State') % must have
     dimension of state space
        fullyConnectedLayer(24,'Name','CriticStateFC1')
        reluLayer('Name','CriticRelu1')
        fullyConnectedLayer(24,'Name','CriticStateFC2')];
    actionPath = [
        featureInputLayer(actInfo.Dimension(1),'Normalization','none','Name','Action')
        fullyConnectedLayer(24,'Name','CriticActionFC1')];
    commonPath = [
        additionLayer(2,'Name','add')
        reluLayer('Name','CriticCommonRelu')
        fullyConnectedLayer(1,'Name','CriticOutput')];

    criticNetwork = layerGraph(statePath);
    criticNetwork = addLayers(criticNetwork, actionPath);
    criticNetwork = addLayers(criticNetwork, commonPath);
    criticNetwork = connectLayers(criticNetwork,'CriticStateFC2','add/in1');
    criticNetwork = connectLayers(criticNetwork,'CriticActionFC1','add/in2');

    criticOpts = rlRepresentationOptions('LearnRate',1e-03,'GradientThreshold',1);
    critic = rlQValueRepresentation(criticNetwork,obsInfo,actInfo,'Observation',{'State'},'Action
    ',{'Action'},criticOpts);

    % Define the actor
    actorNetwork = [
        featureInputLayer(obsInfo.Dimension(1),'Normalization','none','Name','State')
        fullyConnectedLayer(24,'Name','ActorFC1')
        reluLayer('Name','ActorRelu1')
        fullyConnectedLayer(24,'Name','ActorFC2')
        reluLayer('Name','ActorRelu2')
        fullyConnectedLayer(2*actInfo.Dimension(1),'Name','ActorFC3') % Output layer size is
    twice the action dimension
        ];

    % Separate the actor network outputs into mean and standard deviation
    meanPath = [
        fullyConnectedLayer(actInfo.Dimension(1),'Name','Mean')
        tanhLayer('Name','ActorTanh')
        scalingLayer('Name','ActorScaling','Scale',max(actInfo.UpperLimit))
        ];

    stdPath = [
        fullyConnectedLayer(actInfo.Dimension(1),'Name','Std')
        softplusLayer('Name','ActorSoftplus')
        ];

    actorNetwork = layerGraph(actorNetwork);
    meanPath = layerGraph(meanPath);
    stdPath = layerGraph(stdPath);

    actorNetwork = addLayers(actorNetwork, meanPath.Layers);
    actorNetwork = addLayers(actorNetwork, stdPath.Layers);

    actorNetwork = connectLayers(actorNetwork,'ActorFC3','Mean');
    actorNetwork = connectLayers(actorNetwork,'ActorFC3','Std');

    % % Plot the networks
    % figure()
    % subplot(1,2,2)
    % plot(criticNetwork)
    % title('Critic Network')
    % subplot(1,2,1)
    % plot(actorNetwork)
    % title('Actor Network')

    actor = rlContinuousGaussianActor(actorNetwork,obsInfo,actInfo, ...
```

```
        'ActionMeanOutputNames', 'ActorScaling', ... % Specify the names of the mean outputs
        'ActionStandardDeviationOutputNames', 'ActorSoftplus');

    % Define the agent
    agentOpts = rlSACAgentOptions(...
        'SampleTime',c.tStep,...
        'DiscountFactor',0.99,...
        'NumEpoch',10,...
        'TargetUpdateFrequency',1,...
        'NumWarmStartSteps',100,...
        'ExperienceBufferLength',1e6,...
        'MiniBatchSize',64);
    agent = rlSACAgent(actor,critic,agentOpts);

    % Train the agent
    trainOpts = rlTrainingOptions(...
        'MaxEpisodes',50,...
        'MaxStepsPerEpisode',1000,...
        'Verbose',true,...
        'Plots','None',...
        'StopTrainingCriteria','EpisodeCount');

    trainResults = train(agent,env,trainOpts);

    % Simulate trained agent in environment
    experience = sim(env,agent,rlSimulationOptions('MaxSteps',5000));
    tAct = experience.Action.act1.Time;
    simActions = reshape(experience.Action.act1.Data,actInfo.Dimension(1),[]);
    tStates = experience.Observation.obs1.Time;
    simStates = reshape(experience.Observation.obs1.Data,6,[]);

end
```

```
function [c] = CMG_variableDef()

    % ====== EDIT WHATEVER YOU NEED ================================

    c.numAct  = 4; % [~] Number of Actuators (RW/CMG/VSCMG)

    % Inertias (applies to all)
    c.Is  = diag([86 85 113]); % [kg m2] Inertia matrix of the s/c (fixed)
    c.J_G = diag([0.13 0.04 0.03]); % [kg m2] Variable Inertia
    c.Js = c.J_G(1,1); c.Jt = c.J_G(2,2); c.Jg = c.J_G(3,3);
    c.Iws = 0.1; % [kg m2] RW spin axis inertias

    % Initial Attitude
    c.sigmaBN_0  = [0.1; 0.2; 0.3];       % [~] Initial S/C attitude
    c.wBN_B_0    = [0.01; -0.01; 0.005];  % [rad/s] Initial angular rate of s/c

    % Desired final attitude
    c.desiredTerminalState = zeros(6,1);

    % Define Simulation time parameters
    c.tStep  = 0.1; % [s] Time step
    c.tFinal = 30;  % [s] Simulation End Time

    % CMG States
    c.Omega_0    = 14.4 * ones(c.numAct,1); % [rad/s] Initial wheel speeds
    c.gamma_0    = deg2rad([0;0;90;-90]); % [rad] Initial Angles of Gimbals
    c.gammaDot_0 = zeros(c.numAct,1); % [rad/s] Initial gimbal Angular rates

    % Gimbal Frames Definition
    theta       = deg2rad(54.75); % [rad]

    gg1Hat_B_0  = [cos(theta);0;sin(theta)];
    gs1Hat_B_0  = [0;1;0];
    gt1Hat_B_0  = cross(gg1Hat_B_0,gs1Hat_B_0);

```

```
    gg2Hat_B_0  = [-cos(theta);0;sin(theta)];
    gs2Hat_B_0  = [0;-1;0];
    gt2Hat_B_0  = cross(gg2Hat_B_0,gs2Hat_B_0);

    gg3Hat_B_0  = [0;cos(theta);sin(theta)];
    gs3Hat_B_0  = [1;0;0];
    gt3Hat_B_0  = cross(gg3Hat_B_0,gs3Hat_B_0);

    gg4Hat_B_0  = [0;-cos(theta);sin(theta)];
    gs4Hat_B_0  = [-1;0;0];
    gt4Hat_B_0  = cross(gg4Hat_B_0,gs4Hat_B_0);

    c.Gs_B      = [gs1Hat_B_0,gs2Hat_B_0,gs3Hat_B_0,gs4Hat_B_0];
    c.Gt_B      = [gt1Hat_B_0,gt2Hat_B_0,gt3Hat_B_0,gt4Hat_B_0];
    c.Gg_B      = [gg1Hat_B_0,gg2Hat_B_0,gg3Hat_B_0,gg4Hat_B_0];

    % Torques on system
    c.L_N   = [0;0;0]; % [Nm] External Torque in N-frame
    c.L_B   = mrp2dcm(c.sigmaBN_0) * c.L_N; % [Nm] External Torque in B-frame
    c.ug    = zeros(c.numAct,1); % [Nm] CMG motor torque
    c.us    = zeros(c.numAct,1); % [Nm] RW motor torque

    % Establish max & min torques of actuator
    c.maxU  = 2; % [Nm] Maximum value of the torque applied by the chosen actuator
    c.minU  = -2; % [Nm] Maximum value of the torque applied by the chosen actuator

    % Establish state limits
    c.maxState = [1; 1; 1; 10; 10; 10]; % [MRP upper limit; wBR_B upper limit]
    c.minState = -c.maxState;
end
```

```
function [env] = createRLenv()
    % [env] = createRLenv(initialState)
    % Creates an RL environment. Defines a continuous state space and action space.
    % Uses CMG_DynamicProp.m to define the transfer and reward functions between states.
    % Resetting the environment takes the current state back to "initialState".

    % State Space = dimension (6x1) --> Gimbal angle information is passed to the
    %                                   transition fucntion as part of the "info" variable.
    % Action Space = dimension (3x1) --> Encodes the torque applied to each of the 3 CMG gimbal
    motors

    % Inputs:   Nothing. It'll obtain the current initial state in the "resetFcn" call.
    % Output:   env = RL environment object

    % -------------------------------------------------------------

    c = CMG_variableDef();

    % Define state space (continuous)
    ssDimension = [6, 1]; % [MRP; w] - just keep track of gimbal
    stateSpace = rlNumericSpec(ssDimension,UpperLimit = c.maxState,LowerLimit = c.minState);

    % Define action space (continuous)
    asDimension = [c.numAct, 1]; % [number of actuators]
    actionSpace = rlNumericSpec(asDimension,LowerLimit = c.minU, UpperLimit = c.maxU);

    % Create environment
    env = rlFunctionEnv(stateSpace,actionSpace,"CMG_DynamicProp","resetFcn");
end
```

```
function [statePrime,reward,done,updatedInfo] = CMG_DynamicProp(action,info)

    % ========= EDIT THIS BLOCK ONLY ====================================

    s.us = action; % set to 'info.c.us' if CMG; set to 'action' if RW
    s.ug = info.c.ug; % set to 'info.c.ug' if RW; set to 'action' if CMG

```

```
    % ========= DON'T EDIT BELOW !! ======================================

    % Find number of actuators
    n = info.c.numAct;

    % extract info struct
    c = info.c;
    x0 = reshape(info.state,6,1);
    omega = reshape(info.omega,n,1);
    % omega = info.c.Omega_0;
    % gamma = reshape(info.gamma,n,1);
    gamma = info.c.gamma_0;
    % gammaDot = reshape(info.gammaDot,n,1);
    gammaDot = info.c.gammaDot_0;
    terminalState = reshape(info.terminalState,6,1);

    % Merge state with gimbal state
    x0 = [x0; omega; gamma; gammaDot];

    % INTEGRATION: Runge-Kutta 4 ------------------------------------
    k1 = c.tStep * mmatrixRate(x0, c, s);
    k2 = c.tStep * mmatrixRate(x0 + k1/2, c, s);
    k3 = c.tStep * mmatrixRate(x0 + k2/2, c, s);
    k4 = c.tStep * mmatrixRate(x0 + k3, c, s);
    x = x0 + (1/6) * (k1 + 2*k2 + 2*k3 + k4);

    % MRP SWITCHING -------------------------------------------------
    if norm(x(1:3)) > 1 % Switch to shadow set at 180 deg
        x(1:3) = -x(1:3)/(norm(x(1:3))^2); % Map sigmaBN to the shadow set
    end

    % Output updated info struct
    updatedInfo.state = x(1:6);
    updatedInfo.c = c;
    updatedInfo.omega = x(7:7+n-1);
    updatedInfo.gamma = x(7+n:7+2*n-1);
    updatedInfo.gammaDot = x(7+2*n:7+3*n-1);
    updatedInfo.terminalState = terminalState;

    % Define output state
    statePrime = x(1:6);

    % Check if the state is terminal
    done = isTerminalState(statePrime);

    % Define reward function
    reward = rewardFcn(statePrime,action);

end
```

```
function [xdot] = mmatrixRate(x,c,s)

    % Obtain the number of actuators and intertia
    n = c.numAct;
    Iws = c.Iws;

    % unpack state vector
    sigmaBN = x(1:3);
    wBN_B = x(4:6);
    Omega = x(7:7+n-1);
    gamma = x(7+n:7+2*n-1);
    gammaDot = x(7+2*n:7+3*n-1);

    L_B = c.L_B;

    Js = c.J_G(1,1);
    Jt = c.J_G(2,2);
    Jg = c.J_G(3,3);
```

```
19
20      % Preallocate
21      f_Omega = zeros(n,1); f_gammaDot = zeros(n,1);
22
23      f_wBN_B = L_B; % initialize
24
25      % Find MRP rate function of MASS MATRIX method
26      f_sigmaBN = 0.25 * BmatMRP(sigmaBN) * wBN_B;
27      f_gamma = gammaDot;
28
29      % Find current Gimbal Frame definition
30      [s.Gs_B,s.Gt_B,s.Gg_B] = now_gimbalFrame(gamma,c);
31
32      % Find current projections of wBN_B onto gimbal frame
33      [ws,wt,wg] = now_wProjections(wBN_B,c,s);
34
35      % Find current system's intertia tensor
36      I_B = now_I_B(s,c);
37
38      for i = 1:n
39
40          % Find angular acceleration of RW
41          f_Omega(i) = s.us(i) - Iws*gammaDot(i)*wt(i);
42
43          % Find angular acceleration of CMG
44          f_gammaDot(i) = s.ug(i) + (Js-Jt)*ws(i)*wt(i) + Iws*Omega(i)*wt(i);
45
46          f_wBN_B = f_wBN_B - (s.Gs_B(:,i)*(Js*gammaDot(i)*wt(i)-(Jt-Jg)*wt(i)*gammaDot(i))+...
47              s.Gt_B(:,i)*((Js*ws(i)+Iws*Omega(i))*gammaDot(i)-(Jt+Jg)*ws(i)*gammaDot(i)+Iws*Omega(
    i)*wg(i))-...
48              s.Gg_B(:,i)*(Iws*Omega(i)*wt(i)));
49      end
50
51      f_wBN_B = f_wBN_B - cross(wBN_B,I_B*wBN_B);
52
53      % MASS MATRIX method
54      M = [eye(3),zeros(3),zeros(3,n),zeros(3,n),zeros(3,n);...
55          zeros(3),I_B,zeros(3,n),Jg*s.Gg_B,Iws*s.Gs_B;...
56          zeros(n,3),zeros(n,3),eye(n),zeros(n),zeros(n);...
57          zeros(n,3),Jg*s.Gg_B',zeros(n),Jg*eye(n),zeros(n);...
58          zeros(n,3),Iws*s.Gs_B',zeros(n),zeros(n),Iws*eye(n)];
59
60      fvec = [f_sigmaBN;f_wBN_B;f_gamma;f_gammaDot;f_Omega];
61      stateDot = M \ fvec;
62
63      % Compile into state rate vector
64      xdot = [stateDot(1:3);... % sigmaBN
65              stateDot(4:6);... % wBN_B
66              stateDot(7+2*n:7+3*n-1);... % OmegaDot
67              stateDot(7:7+n-1);... % gammaDot
68              stateDot(7+n:7+2*n-1)]; % gammaDotDot
69  end
```

```
1   function [Gs_B,Gt_B,Gg_B] = now_gimbalFrame(gamma,c)
2   %nVSCMG_CurrentG Finds the current Gimbal frame definition
3   %   Inputs: n = number of VSCMG (single)
4   %           gamma = current gamma (4x1 vector)
5   %           c = constants struct (contains initial frame def.)
6
7
8   n = c.numAct;
9   Gs_B = zeros(3,n); Gt_B = Gs_B; Gg_B = Gs_B;
10  gamma0 = c.gamma_0;
11
12  for i = 1:n
13      gsHat_B_0 = c.Gs_B(:,i);
14      gtHat_B_0 = c.Gt_B(:,i);
15      ggHat_B_0 = c.Gg_B(:,i);
```

```
    % Define gimbal frame
    Gs_B(:,i) = cos(gamma(i)-gamma0(i))*gsHat_B_0 + sin(gamma(i)-gamma0(i))*gtHat_B_0;
    Gt_B(:,i) = -sin(gamma(i)-gamma0(i))*gsHat_B_0 + cos(gamma(i)-gamma0(i))*gtHat_B_0;
    Gg_B(:,i) = ggHat_B_0;
end
end
```

```
function [I_B] = now_I_B(s,c)
%now_I_B Finds the current system's inerta tensor
%   Inputs: n = number of VSCMG (single)
%           s = state struct (contains current frame def.)
%           c = constant struct (contains initial frame def.)

I_B = c.Is; n = c.numAct;
for i = 1:n
    BG = [s.Gs_B(:,i),s.Gt_B(:,i),s.Gg_B(:,i)];
    GB = transpose(BG);
    J_B = BG * c.J_G * GB;
    I_B = I_B + J_B;
end
end
```

```
function [ws,wt,wg] = now_wProjections(wBN_B,c,s)
%now_wProjections Finds the current projections of wBN_B onto Gimbal Frame
%   Inputs: n = number of VSCMG (single)
%           wBN_B = current wBN_B (3x1 vector)
%           s = state struct (contains current frame def.)

n = c.numAct;
ws = zeros(n,1); wt = ws; wg = ws;

for i = 1:n
    ws(i) = dot(s.Gs_B(:,i),wBN_B);
    wt(i) = dot(s.Gt_B(:,i),wBN_B);
    wg(i) = dot(s.Gg_B(:,i),wBN_B);
end
end
```

```
function [done] = isTerminalState(state)
    % Determine if the state is terminal
    % Outputs 'true' if the input state is terminal
    % Inputs:   state = state you want to check
    % Output:   done = boolean representing terminal vs non-terminal
    % ---------------------------------------------------------------

    if all(abs(state(4:6)) < 0.001)
        done = true;
    else
        done = false;
    end

end
```

```
function [reward] = rewardFcn(state,action)
    % [reward] = rewardFcn(sigmaBR,wBR_B,action)
    % Defines the reward function based on the current state-action pair
    % Inputs:   sigmaBR = current body to reference MRP
    %           wBR_B = current angular rate wrt reference (in body frame)
    %           action = current applied torque
    % Output:   reward = payoff of the state-action pair
    % ---------------------------------------------------------------------

    if all(abs(state(4:6)) < 0.001)
        reward = 100; % Reward for close to zero angular rates
    else
        reward = - sum(abs(state(4:6))) - sum(abs(action));
```

```
    end

end
```

```
function [initial,info] = resetFcn()
    % [initial,info] = resetFcn()
    % Defines the state that the environment returns to after a reset
    % CANNOT HAVE ANY INPUTS !!
    % -------------------------------------------------------------------

    % Obtain constants' struct
    c = CMG_variableDef();

    % Define the initial state
    initial = [reshape(c.sigmaBN_0,3,1); reshape(c.wBN_B_0,3,1)];

    % Define info output to pass into transition function
    info.c = c;
    info.state = initial;
    info.omega = info.c.Omega_0;
    info.gamma = info.c.gamma_0;
    info.gammaDot = info.c.gammaDot_0;
    info.terminalState = c.desiredTerminalState;
end
```

```
function [] = rewardShaping()

    clc; close all;

    addpath(genpath('..'))

    c = CMG_variableDef();

    % Define bounds
    actionMax = c.maxU;

    omegaStateSums = 0:0.01:3;
    actionSums = 0:0.01:4*actionMax;

    [OmegaStateSums, ActionSums] = meshgrid(omegaStateSums, actionSums);

    % REWARD FOR CASE 1
    r1 = zeros(size(OmegaStateSums));
    for i = 1:numel(omegaStateSums)
        for j = 1:numel(actionSums)
            r1(j,i) = - omegaStateSums(i) - actionSums(j);
        end
    end

    figure(1)
    surf(OmegaStateSums, ActionSums, r1, "EdgeAlpha",0.1)
    xlabel('Omega State Sums')
    ylabel('Action Sums')
    zlabel('Reward')
    title('Reward Function (Case 1)')
    savefig("GoodRuns/reward1shape.fig")

    % REWARD FOR CASE 2
    r2 = zeros(size(OmegaStateSums));
    for i = 1:numel(omegaStateSums)
        for j = 1:numel(actionSums)
            r2(j,i) = - exp(3*omegaStateSums(i)) - actionSums(j);
        end
    end

    figure(2)
    surf(OmegaStateSums, ActionSums, r2, "EdgeAlpha",0.1)
    xlabel('Omega State Sums')
```

```
    ylabel('Action Sums')
    zlabel('Reward')
    title('Reward Function (Case 2)')
    savefig("GoodRuns/reward2shape.fig")

    % REWARD FOR CASE 3
    r3 = zeros(size(OmegaStateSums));
    for i = 1:numel(omegaStateSums)
        for j = 1:numel(actionSums)
            r3(j,i) = - sqrt(100*omegaStateSums(i)) - actionSums(j);
        end
    end

    figure(3)
    surf(OmegaStateSums, ActionSums, r3, "EdgeAlpha",0.1)
    xlabel('Omega State Sums')
    ylabel('Action Sums')
    zlabel('Reward')
    title('Reward Function (Case 3)')
    savefig("GoodRuns/reward3shape.fig")

end
```