Stabilizing an Orbiting Spacecraft

Philip Miceli, Sayan Chatterjee

I. Introduction

The following paper explores possible methods to stabilize an orbiting spacecraft. Satellites are becoming more important in our daily lives. Whether it be using GPS for daily navigation or using the new Starlink satellites for global internet access. While the future is bright, the Earth Orbital Plane is becoming more congested with new satellites being launched every month. Just in the last two years, the number of LEO satellites has increased by fifty percent [1]. SpaceX already has over one thousand satellites in the LEO regime, with other companies following close pursuit. The International Space Station also inhabits the LEO regime. Recently, the ISS had to perform a 310 meter jump to dodge possible debris produced by Russian anti-satellite testing [2]. With the number of satellites being used, it will be almost impossible to control all these satellites to ensure non collision operations; hence, the use of autonomy is becoming a necessity in the LEO regime.

The outcome of this project is to determine the optimal controller for an orbiting spacecraft. The controller would stabilize the spacecraft after a quick thrust maneuver to dodge potential objects in the LEO regime. To achieve the goals of the project, two different themes were used to find the optimal control law. One involved using Policy Search applied through Local Search and Cross Entropy. The other is a more traditional approach with policy iteration via a linear quadratic regulator (LQR).

II. Background and Related Work

Designing a controller for a state space model is nothing new. Controller design is covered extensively in Classical or Modern Control Design. Often, controller design is done through conventional PID controllers from classical control or pole place placement through modern control. In the past, these approaches were acceptable, but required a great degree of effort to a suitable set of gains for the controller. Finding the gain of the controller involves trial and error of different parameters, and then seeing how it impacts the stability of the system (via Root Locus plot) and/or finding the transient response. For this project, another parameter was the actuator response. In the context of the problem formulation, this would represent the thrust the spacecraft applied to stabilize.

Manually tuning controllers for an orbiting spacecraft would represent a big undertaking. Especially since the system is constantly evolving as it deals with an unpredictable environment. For this reason, it would be very beneficial to have a system that is capable of tuning itself as new stimulus are applied to the environment. As mentioned before, there are many satellites in LEO, and having an autonomous system is becoming more of an necessity to avoid collision.

III. Problem Formulation

This problem deals with simplified non-linear dynamics model for an orbiting satellite described by the following equations of motion:

$$\ddot{r} = r\dot{\theta}^2 - \frac{\mu}{r^2} + u_1$$

$$\ddot{\theta} = -\frac{2\dot{\theta}\dot{r}}{r} + \frac{1}{r}u_2$$

$$\mu = 398600\ [\mathrm{km}^3/s^2]$$

$$r_0 = 6678\ [\mathrm{km}]$$

Given these non-linear equations of motion, we were able to linearize this system equations about a nominal orbital trajectory so that linear control design methods could be applied to this system. The open-loop statespace model for this problem takes the following form:

$$\dot{\vec{x}} = A\vec{x} + B\vec{u}$$

$$\vec{y} = C\vec{x} + D\vec{u}$$

The linearized matrices for this problem are as such:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ \frac{3\mu}{r_0^3} & 0 & 0 & 2\sqrt{\frac{\mu}{r_0}} \\ 0 & 0 & 0 & 1 \\ 0 & -2\sqrt{\frac{\mu}{r_0^5}} & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & \frac{1}{r_0} \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, D = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

One solution method for this problem is by defining this as an optimization function. The decision variables for this optimization problem are closed-loop poles, which are to be iterated through developing algorithms to minimize this optimization problem. The cost function in which we want to minimize can be described by the following pseudo-code:

Algorithm 1 Cost function f(a,b,c,d)

$poles \leftarrow [a, b, c, d]$
$K \leftarrow place(A, B, poles)$
$sys \leftarrow closedLoopSystem(A, B, C, D, K)$
$u1, u2 \leftarrow simulate(sys)$
$cost \leftarrow sum(|u1|) + sum(|u2|)$
$return \leftarrow cost$

This optimization problem is constrained such that the decision variables (poles), a, b, c, and d, must be less than zero (constrained to negative values). It should be noted, that some control design methods have been omitted from this report due to the scope of the paper. With that being said, the nonlinear model is fully controllable and observable, enabling controller design. When conducting the analyze, the disturbance was modelled as a sinusoidal bump. This bump represents the satellite firing its thrusters to avoid a possible collision.

IV. Solution Approach

This project went in two main directions while attempting to find an optimal controller to stabilize a satellite. The first direction this project took was to implement policy search algorithms to find an optimal closed-loop pole placement for the linearized satellite system and minimize the fuel consumption in a stabilizing maneuver. Two different policy search algorithm's were employed in an attempt to find the optimal controller for the given problem, Hooke-Jeeves policy search method and the Cross Entropy policy search method. This project also explored policy iteration approaches to design an optimal controller to stabilize the satellite. Each of these solution approaches will be further detailed in this section.

A. Hooke-Jeeves Policy Search

The first policy search algorithm implemented to optimally place closed-loop poles for this problem was a local search algorithm called the *Hooke-Jeeves method*. Hooke-Jeeves was implemented by discretizing our search space (closed-loop pole locations, a, b, c, and d) with all poles assumed to be only real and confined to negative real values [3]. The implementation of *Hooke-Jeeves* takes a step α in a negative and positive direction (within a vector containing a set of closed loop poles). The given poles at $\pm\alpha$ are then evaluated to determine if either set of poles minimize the cost function of this optimization function. In the case where the poles at $\pm\alpha$ better minimize the cost function, the algorithm updates the search space such that it is centered around this set of poles. Otherwise, α is decreased by some step size [3].

In the context of this problem, some simplifications were done to apply the algorithm. Due to four states being in the state space model, there needs to be a combination of four poles. Each of these poles must be on the left side of the real-imaginary plane to represent a stable system. With this in mind, a list of poles were generated ahead of time, ranging from 0 to -0.5 in value. With that constraint, there were approximately 1296 different pole combinations, each with a unique index. The α value was set to 100, and the ϵ value being set to 10. While α was less than the ϵ

value, a random pole was selected initially from the list. With that associated pole, the gains were calculated, and then a new stabilized system was formed. From there, the amount of input or thrust was calculated, which weighed into a generalized scoring. In the context of this problem, there are two thrust values over time for radial and in-track directions. Regardless of negative or positive values, the absolute value of the thrust amount was taken at a given time, and then summed to represent the total thrust amount. A lower thrust amount represented a more ideal system. With the scoring out of the way, the same process was done with the upper and lower bounds to find their thrust amounts. Upper and lower bounds were found by taking the current index, and then adding or subtracting α. If a better score was found, the index would move to that bound index. Otherwise, the α value would be halved, and new upper and lower bounds would be found. Again, the bounds will be evaluated until α is less than ϵ. Once the loop terminates the current pole index represents the optimized poles/gains of the system.

B. Cross Entropy Policy Search

After implementing a local policy search algorithm, the *Cross Entropy method* was implemented to try to find a better optimization for the given problem. One immediate benefit of implementing this algorithm was that it is capable of searching a continuous space, so we didn't need to discretize a specific search space. Instead, the cross entropy method starts with an initial probability distribution that describes a prior belief to where the best closed-loop exists and samples m sets of poles (a, b, c and d). For each sample, the set of poles are evaluated in the cost function. After all m samples have been evaluated, the samples are sorted in ascending order according to their associated cost. Then, m_{elite} of the best samples are chosen to re-fit a probability distribution corresponding with the best performing samples[3]. This process is continued until a convergence criteria is met, or the algorithm has taken the maximum allowed iterations.

While this algorithm consistently outperformed the *Hooke-Jeeves method*, multiple improvements were made to this algorithm to address the local minima issue. The local minima would occur when the covariance bounds of the post-fit distribution would become very small and would sample poles from a very tightly-bound distribution. As a result, the *Cross Entropy* algorithm became over-confident that it found the best distribution to describe the optimal pole-placement, and no other pole locations were considered. To deal with this issue, the algorithm was modified such that if the matrix-norm of the distribution's covariance became smaller than some threshold ϵ, the re-fit covariance distribution would be artificially inflated, which forced the algorithm to begin sampling from a larger distribution, thus eliminating complications from finding local minima.

A large part of solving this problem required extensive tuning of the parameters passed into the algorithm. The distribution type that was used to solve this problem was a multivariate normal distribution with an initial mean and covariance [3]:

$$D \sim \mathcal{N}(\mu, \sigma^2)$$

$$\mu = \begin{bmatrix} -5 \\ -5 \\ -5 \\ -5 \end{bmatrix}, \sigma^2 = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 10 \end{bmatrix}$$

Through adjusting the tuning parameters, we found that having a sample size of $m = 100$ provided a sufficient number of samples to search the environment thoroughly. Likewise, we set the elite number of samples $m_{elite} = 25$ to obtain our baseline results. In the tuning process, we experimented with a much larger initial covariance; however, the algorithm returned a final distribution well within the initial covariance bounds so we decided a very large initial covariance was unnecessary. We also increased the sample size, but found that after m was increased over 100, there was no measurable difference in algorithm performance. Also, the ratio of elite samples to total samples was also experimented with and we found that having a ratio between 20-30% provided the best results. While we could have budgeted even more time to tuning these parameters, we concluded that the effort would be better spent towards other portions of this project.

C. Policy Iteration on a Problem Linear-Quadratic Rewards

Moving forward, LQR is form form of a Policy Iteration that is very popular in the field of robotics and control system designs. The basic problem is to identify a mapping from states to controls that minimizes the quadratic cost of a linear system. The quadratic cost has the form:

$$c(x, u) = x^T Q x + u^T R u$$

Where $x \to \mathbb{R}^n$ is the state of the system and $u \to \mathbb{R}^k$ is the control. Q represents the cost function, that should be symmetric positive semi-definite. The R matrix represents the rewards matrix that is strictly positive definite. To solve such a problem, let this MDP be a finite-horizon problem repressed below.

$$J^{\pi}(x_t, t) = \sum_{t'=t}^{T-1} c(x_t, \pi(x_t, t'))$$

With help of Bellman's equations, the problem simplifies further to the expressions below.

$$K_t = -(R + B^T V_{t+1} B)^{-1} B^T V_{t+1} A$$

$$V_t = Q + K_t^T R K_t + (A + B K_t)^T V_{t+1} (A + B K_t)$$

$$J^*(x_t, t) = x_t^T V_t x_t$$

The following algorithm represents the best implementation of LQR through policy iteration [4].

Algorithm 2 OptimalValue(A,B,Q,R,t,T)

1: **if** t = T - 1 **then**
2: return Q
3: **else**
4: $V_{t+1} = OptimalValue(A, B, Q, R, t + 1, T)$
5: $K_t = -(R + B^T V_{t+1} B)^{-1} B^T V_{t+1} A$
6: return $V_t = Q + K_t^T R K_t + (A + B K_t)^T V_{t+1} (A + B K_t)$
7: **end if**

It is important to note the following algorithm can only be applied to discrete systems. Currently, the linearized system derived from above is a continuous system. To use the LQR above, the matrix exponential was taken to create a discrete system. Unfortunately, the combination of linearization and matrix exponential caused a lot of assumptions to be made of the system. LQR was unable to stabilize the system.

For this reason, it was decided to switch to the infinite horizon LQR problem. Where the cost can be written in the equation displayed below [5].

$$J = \int_0^{\infty} (x^T Q x + u^T R u + 2 x^T N u) dt$$

Where the associated gain will be associated with the equation below:

$$K = R^{-1}(B^T P + N^T)$$

However, there is another problem coming from this new form. The value of P is unknown, but can be found by solving the algebraic Riccati Equation:

$$A^T P + P A = P B R^{-1} B^T P + Q = 0$$

It should be noted that algebraic Riccati equation can be solved by using the principles of optimal control via the Hamiltonian Matrix. To simplify the process in this project, in built functions in Python and Matlab were used to find the solution to the algebraic Riccati Equation.

V. Results

A. Hooke-Jeeves Policy Search

Through running and implementing Hooke-Jeeve's policy search algorithm on our problem, the optimal set of closed-loop poles that minimize the cost function are:

$$p_{a,b,c,d} = [-0.2 - 0.5 - 0.4 - 0.2]$$

When these poles are turned into a state-feedback control law, we get the following control input history plots:

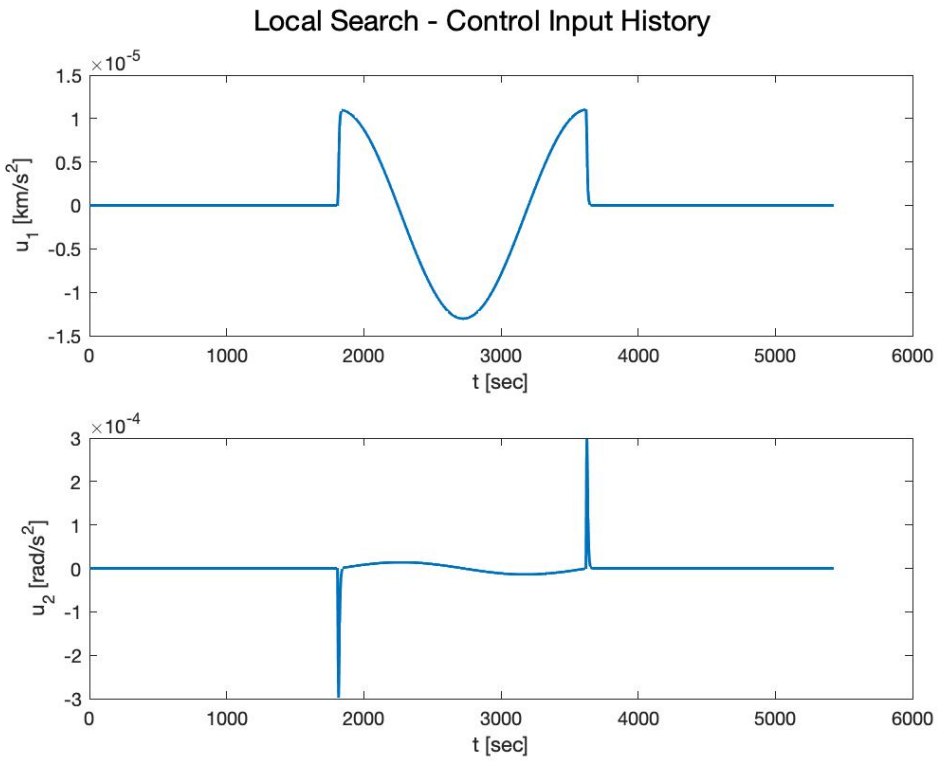

Fig. 1 Control input history with Hooke-Jeeve's determined controller

Through evaluating these poles in our defined cost function, the Hooke-Jeeve's selected poles received an overall cost of 1.2255. This score will be used as a baseline to compare the other optimal control design methods.

B. Cross-Entropy Policy Search

Through implementing a Cross-Entropy search algorithm on our problem, we found that the optimal set of closed-loop poles that minimize the cost function are:

$$p_{a,b,c,d} = [-1.8562 - 3.3537 - 2.0980 - 3.3267]$$

When these poles are turned into a state-feedback control law, we see the following control input history plots:

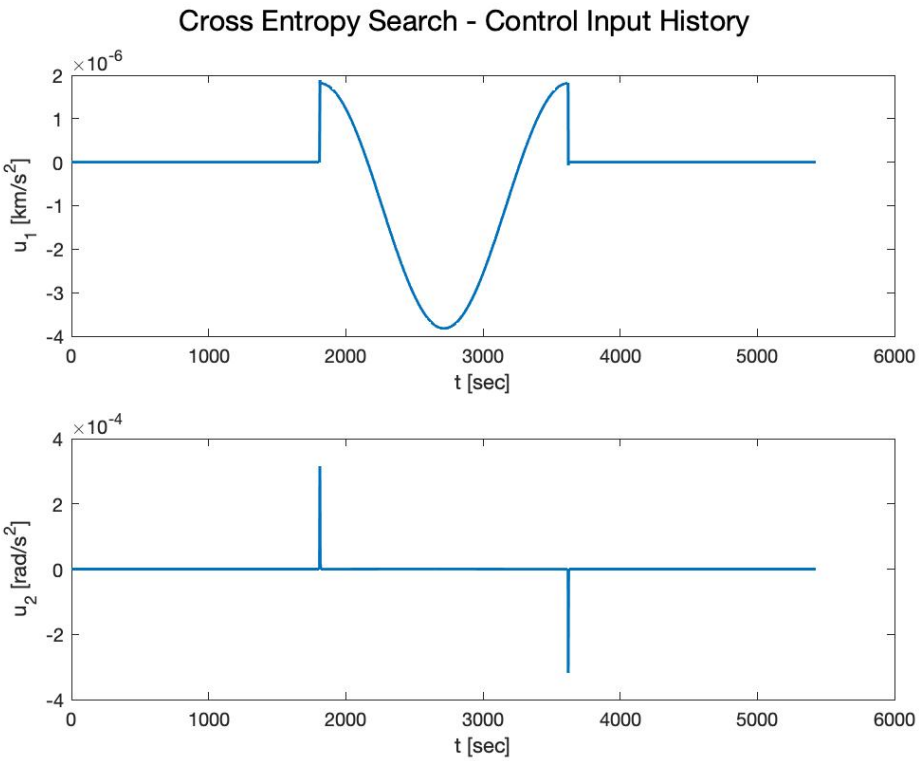

Fig. 2 Control input history with Cross-Entropy method determined controller

Through evaluating the Cross-Entropy method determined poles in our cost function, the selected poles received an overall cost of 0.1694. Receiving this score is a significant improvement to the Hooke-Jeeve's method as the designed set of closed-loop poles will require only 13.82% of the required fuel that the Hooke-Jeeve's selected poles require.

C. LQR: Policy Iteration

Before discussing the results of LQR, it is important to note the R and Q matrices. These matrices were the given reward and cost respectively of the system. The parameters were picked to be as realistic as possible for a spacecraft.

$$Q = \begin{bmatrix} 25 & 0 & 0 & 0 \\ 0 & 25 & 0 & 0 \\ 0 & 0 & 25 & 0 \\ 0 & 0 & 0 & 25 \end{bmatrix}, R = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

With that in mind, these were the associated poles and gains returned from the LQR approach.

$$p_{a,b,c,d} = [-3.37 - 1.05 - 0.013 + 0.013j - 0.013 - 0.013j]$$

$$K = \begin{bmatrix} 3.54e+00 & 4.424e+00 & -4.05e-05 & 1.54e+01 \\ 4.047e-05 & 1.55e-03 & 3.54e+00 & 2.66e+02 \end{bmatrix}$$

With said control law, the following plot shows the control input history with the LQR determined controller.

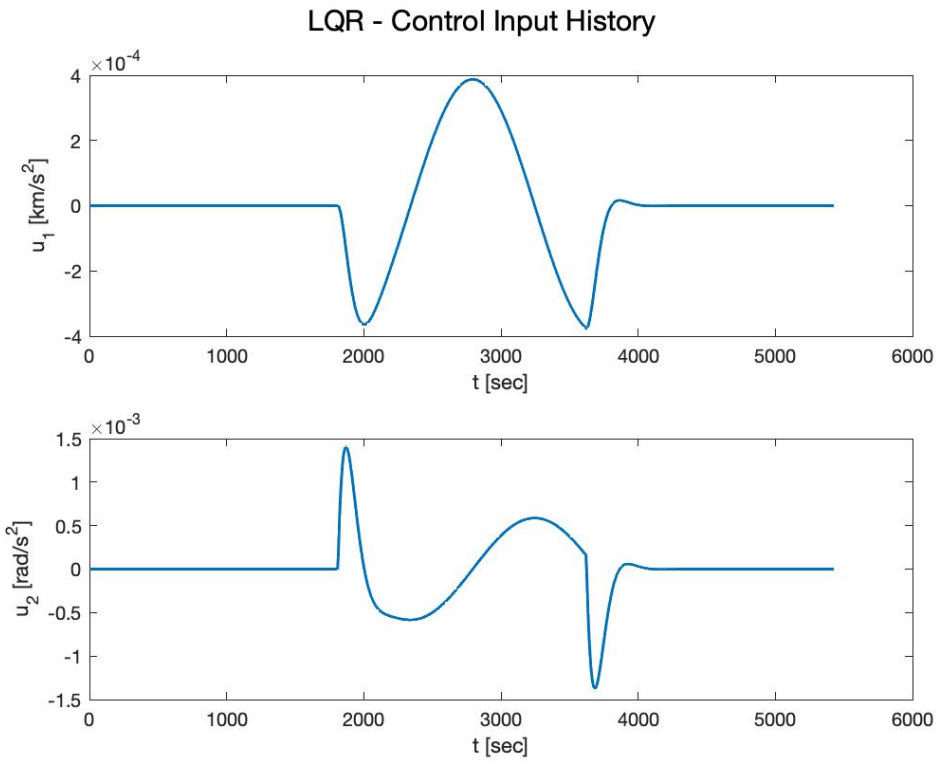

Fig. 3 Control input history with Cross-Entropy method determined controller

Through analyzing the control input plots for the LQR controller, we notice that there is a control input that is sustained for a much greater amount of time in comparison to the Hooke-Jeeve's and Cross Entropy policy search methods. Through evaluating the LQR chosen closed-loop poles with our cost function, the poles receive a score 47.99, which is a much more costly score than the closed-loop poles chosen from either policy search method. To try and improve the performance of LQR, we scaled the Q and R matrices such that matrix norm of Q was one million times greater than the matrix norm of R. This tells LQR to prioritize minimizing the control effort as opposed to prioritizing performance. Even in this case where Q and R matrices are scaled for this case, the selected closed-loop poles from LQR received a score of 9.4245 from our cost function. We can conclude from this result that policy search methods outperform LQR in the case of minimizing fuel consumption to perform a stabilizing maneuver.

VI. Conclusion

Through exploring both policy search and policy iteration to determine an optimal controller that minimizes fuel burned by spacecraft thrusters, we learned a lot of valuable information regarding defining and solving optimization problems, as well as how to solve MDP's with linear-quadratic rewards. Through exploring multiple types of optimization routines, including the Hooke-Jeeves method as well as the Cross-Entropy method, we explored. We found that the Hooke-Jeeves method had a quicker run-time due to the nature that we were searching a small discrete space. The Cross-Entropy method yielded better results but took a significantly longer time to run. This is mainly a result of evaluating a high number of samples in our cost function. The cost function builds a state-space system with given closed loop poles and simulates the system response over one orbital period. As a result, running many simulations for different sampled poles results in a high run-time. In the future, it would be helpful to come up with a quicker way to evaluate the cost function so that the search for the optimal poles would be more efficient. Another future improvement that we would like to make to the policy search approach would be increasing the amount of constraints placed on control system as well as increasing the complexity of the cost function. This would allow us to reward certain behaviors and design a controller with more specific applications (as opposed to just minimizing control input). When moving onto using LQR through a policy iteration approach, it produced some decent results. While not as quick as the Hooke-Jeeves

method, LQR was able to arrive at a solution within 5 seconds of execution. The only issue with LQR came down to the tuning of the Q and R matrices. These matrices were decided ahead of time, but introduced a degree of trial and error for tuning. To have a completely autonomous system, LQR would need some sort of intervention to have proper reward and cost matrices. Especially since these matrices could change due to the specific requirements. Through comparing the LQR approach to the policy search approach, we found that both policy search methods outperformed LQR, especially the Cross-Entropy method. This was a surprising result as LQR traditionally is used to design optimal controllers while specifying the cost of performance vs. the cost of control effort. To conclude, this project gave fantastic insight on how to use policy search and policy iteration methods to determine an optimal controller that stabilizes a satellite.

VII. Contributions and Release

The authors grant permission for this report to be posted publicly.

1) **Phil Miceli** was responsible for creating the Cross Entropy Algorithm. In addition to overall document formatting and explanation, he was responsible for the result and conclusion section.
2) **Sayan Chatterjee**: was responsible for the LQR and Local Search algorithm for the project. Also, he held responsibility for formatting the paper, including writing out equations and the reference section.

References

[1] Boley, A. C., and Byers, M., "Satellite mega-constellations create risks in Low Earth Orbit, the atmosphere and on Earth," *Scientific Reports*, Vol. 11, No. 1, 2021, pp. 1–8. >The following resource is a paper discussing the risk of LEO satellites. Specifically, the paper outlines the growing number of satellites coming into play, which causes an increase amount of collision in the future. Some talking points include a better catalog of satellites in the orbit, or implementing a collision avoidance algorithm to help regulate the LEO orbit.

[2] William, W., "Space station's orbit adjusted to dodge debris from old U.S. rocket," , Dec 2021. URL https://www.cbsnews.com/news/international-space-station-maneuver-avoid-us-space-debris-old-rocket/, > The news article penned discusses the recent event of the Russian ASAT launch. In reaction the launc of the ASAT, the ISS had to perform a rapid maneuver in order to avoid any possible debris. Astronauts were on standby to evacuate the ISS if needed. The news article outlines the possibility of LEO becoming very dangerous with orbital debris if the satellite numbers keep increasing.

[3] Kochen, T. A., M J; Wheeler, and Wray, K., "Algorithms for Decision Making," , 2022. > The following is the main textbook used for principles of decision making. Specifically for this paper, the resource provided the pseudo code for local search and cross entropy. The textbook was primarily used to describe pattern search in the paper.

[4] "LQR: The Analytic MDP," , ???? URL https://homes.cs.washington.edu/~bboots/RL-Fall2020/Lectures/LQR_notes.pdf/, > The following is another textbook provided by the University of Washington. Mostly used to describe the principles of LQR. In addition, the pseudo code for discrete LQR in the finite horizon problem was taken from here.

[5] "Lecture 4 Continuous time linear quadratic regulator," , ???? URL https://stanford.edu/class/ee363/lectures/clqr.pdf, > The following resource was used to write out the LQR equations. More specifically, the LQR for the infinite horizon problem was referenced from here.

Appendix

```
% ASEN 5519 - Decision Making Under Uncertainty - Final Project
% Philip Miceli and Sayan Chatterjee
% April 27 2022

% Housekeeping
clc; clear; close all;

%% Cross-entropy optimization method

% Define initial distribution of poles a,b,c, d
mu_0 = [-5;-5;-5;-5]; % Initial distribution mean
P_0 = 10*eye(4);

% Define Cross Entropy Policy Search parameters
p.mu = mu_0;
p.P = P_0;
p.m = 500;
p.m_elite = 200;
p.k_max = 100;
p.eps = 1e-4;

% Run optimizer
p_final = crossEntropyOptimizer(p);

%% Overview of results


%%% Local Search
poles_local = [-0.2000   -0.5000   -0.4000   -0.2000];
[t,x,u] = gatherResults(poles_local);

figure
sgtitle('Local Search - State Perturbation History')

subplot(4,1,1)
plot(t,x(:,1),'LineWidth',1.5)
xlabel('t [sec]')
ylabel('r [km]')

subplot(4,1,2)
plot(t,x(:,2),'LineWidth',1.5)
xlabel('t [sec]')
ylabel('dr/dt [km/s]')

subplot(4,1,3)
plot(t,x(:,3),'LineWidth',1.5)
xlabel('t [sec]')
ylabel('\theta [rad]')

subplot(4,1,4)
plot(t,x(:,2),'LineWidth',1.5)
xlabel('t [sec]')
ylabel('d\theta/dt [rad/s]')

figure
sgtitle('Local Search - Control Input History')

subplot(2,1,1)
plot(t,u(1,:),'LineWidth',1.5)
xlabel('t [sec]')
ylabel('u_1 [km/s^2]')

subplot(2,1,2)
plot(t,u(2,:),'LineWidth',1.5)
xlabel('t [sec]')
```

```
ylabel('u_2 [rad/s^2]')



%%% Cross Entropy Search
poles_cross = p_final.best_pole(57,:);
[t,x,u] = gatherResults(poles_cross);

figure
sgtitle('Cross Entropy Search - State Perturbation History')

subplot(4,1,1)
plot(t,x(:,1),'LineWidth',1.5)
xlabel('t [sec]')
ylabel('r [km]')

subplot(4,1,2)
plot(t,x(:,2),'LineWidth',1.5)
xlabel('t [sec]')
ylabel('dr/dt [km/s]')

subplot(4,1,3)
plot(t,x(:,3),'LineWidth',1.5)
xlabel('t [sec]')
ylabel('\theta [rad]')

subplot(4,1,4)
plot(t,x(:,2),'LineWidth',1.5)
xlabel('t [sec]')
ylabel('d\theta/dt [rad/s]')

figure
sgtitle('Cross Entropy Search - Control Input History')

subplot(2,1,1)
plot(t,u(1,:),'LineWidth',1.5)
xlabel('t [sec]')
ylabel('u_1 [km/s^2]')

subplot(2,1,2)
plot(t,u(2,:),'LineWidth',1.5)
xlabel('t [sec]')
ylabel('u_2 [rad/s^2]')

%%% LQR Poles
poles_lqr= [-3.37 -1.05 -0.013+0.013j -0.013-0.013j];
[t,x,u] = gatherResults(poles_lqr);

figure
sgtitle('LQR - State Perturbation History')

subplot(4,1,1)
plot(t,x(:,1),'LineWidth',1.5)
xlabel('t [sec]')
ylabel('r [km]')

subplot(4,1,2)
plot(t,x(:,2),'LineWidth',1.5)
xlabel('t [sec]')
ylabel('dr/dt [km/s]')

subplot(4,1,3)
plot(t,x(:,3),'LineWidth',1.5)
xlabel('t [sec]')
ylabel('\theta [rad]')

subplot(4,1,4)
plot(t,x(:,2),'LineWidth',1.5)
```

```
xlabel('t [sec]')
ylabel('d\theta/dt [rad/s]')

figure
sgtitle('LQR - Control Input History')

subplot(2,1,1)
plot(t,u(1,:),'LineWidth',1.5)
xlabel('t [sec]')
ylabel('u_1 [km/s^2]')

subplot(2,1,2)
plot(t,u(2,:),'LineWidth',1.5)
xlabel('t [sec]')
ylabel('u_2 [rad/s^2]')


function p_final = crossEntropyOptimizer(p)
%CROSSENTROPYOPTIMIZER A function to optimize a policy using the cross
%entropy method detailed in Algorithm 10.4 of the class textbook

mu = p.mu;
P = p.P;
m = p.m;
m_elite = p.m_elite;
k_max = p.k_max;
eps = p.eps;

for k = 1:k_max

    % Sample m values for poles a,b,c,d
    theta_s = mvnrnd(mu,P,m);

    for i = 1:m

        % Grab sampled poles to evaluate in cost fxn
        eval_poles = theta_s(i,:);

        % Evaluate cost function with set of poles
        U_s(i) = evaluatePoles(eval_poles);

    end

    % Sort samples of theta_s in order of ascending cost
    [~,sort_idx] = sort(U_s');
    theta_s = theta_s(sort_idx,:);

    % Select m_elite samples to fit distribution
    theta_elite = theta_s(1:m_elite,:);

    % Find new distribution to fit data
    mu = mean(theta_elite);
    P = cov(theta_elite);

    % Store best results for timestep as metric to measure performance
    best_U(k) = min(U_s);
    best_U_idx = find(U_s == best_U(k));
    best_theta(k,:) = theta_s(best_U_idx,:);

    % When covariance gets to small, start cross entropy process over.
    if(norm(P) < eps)
        P = p.P;
    end

end

p_final.mu = mu;
p_final.P = P;
```

```
p_final.best_U = best_U;
p_final.best_pole = best_theta;
end


function cost = evaluatePoles(p)
%EVALUATEPOLES A function that evaluates a set of poles a,b,c,d

%% System Creation
mu = 398600; % km^3/s^2
r_0 = 6678; % km
theta_dot_0 = sqrt(mu/(r_0^3));
A = [0, 1, 0, 0;
    3*mu/(r_0^3), 0, 0, 2*sqrt(mu/r_0);
    0, 0, 0, 1;
    0, -2*sqrt(mu/(r_0^5)), 0, 0];
B = [0, 0;
    1, 0;
    0, 0;
    0, 1/r_0];
C = [1, 0, 0, 0;
    0, 0, 1, 0];
D = [0, 0;
    0, 0];

% Define the given initial condition
x_0 = [r_0; 0; 0; (theta_dot_0)];


% Calculate the orbital period
T = 2*pi*sqrt((r_0^3)/mu);
t_vec = linspace(0,T,180000);

% Create the sinusoidal disturbance
sinbump = @(L,A) A*(0.5+0.5*(sin(linspace(-pi/2,pi*3/2,length(t_vec)/3*L))));
B1 = sinbump(1, 0.5)';

radius_ref = [zeros(length(t_vec)/3,1);B1;zeros(length(t_vec)/3,1)];
angle_ref = zeros(length(t_vec),1);

rhistvec = [radius_ref,angle_ref];

%% Evaluate cost of poles

try
    K = place(A,B,p);
catch
    cost = Inf;
    return;
end

F = inv(C*(inv(-A+B*K))*B);

% Do a Check if F is applicable
if isnan(F)
    cost = Inf; % Make the Score Infinite to Discourage Picking
    return
end

A_cl = A-B*K;
B_cl = B*F;

CLsys = ss(A_cl, B_cl, C, D);
[~,~,X_CL1] = lsim(CLsys,rhistvec,t_vec);
U_CL1 = -K*X_CL1' + F*rhistvec';

Thruster1 = sum(abs(U_CL1(1,:)));
Thruster2 = sum(abs(U_CL1(2,:)));
```

```
cost = Thruster1 + Thruster2;

end


function [t_vec,X_CL1,U_CL1] = gatherResults(p)
%PLOTRESULTS Generates system response  for given set of poles

mu = 398600; % km^3/s^2
r_0 = 6678; % km
theta_dot_0 = sqrt(mu/(r_0^3));
A = [0, 1, 0, 0;
    3*mu/(r_0^3), 0, 0, 2*sqrt(mu/r_0);
    0, 0, 0, 1;
    0, -2*sqrt(mu/(r_0^5)), 0, 0];
B = [0, 0;
    1, 0;
    0, 0;
    0, 1/r_0];
C = [1, 0, 0, 0;
    0, 0, 1, 0];
D = [0, 0;
    0, 0];

% Define the given initial condition
x_0 = [r_0; 0; 0; (theta_dot_0)];


% Calculate the orbital period
T = 2*pi*sqrt((r_0^3)/mu);
t_vec = linspace(0,T,180000);

% Create the sinusoidal disturbance
sinbump = @(L,A) A*(0.5+0.5*(sin(linspace(-pi/2,pi*3/2,length(t_vec)/3*L))));
B1 = sinbump(1, 0.5)';

radius_ref = [zeros(length(t_vec)/3,1);B1;zeros(length(t_vec)/3,1)];
angle_ref = zeros(length(t_vec),1);

rhistvec = [radius_ref,angle_ref];

K = place(A,B,p);
F = inv(C*(inv(-A+B*K))*B);

A_cl = A-B*K;
B_cl = B*F;

CLsys = ss(A_cl, B_cl, C, D);
[~,~,X_CL1] = lsim(CLsys,rhistvec,t_vec);
U_CL1 = -K*X_CL1' + F*rhistvec';

end



% Author: Sayan Chatterjee
% Date: April 26th, 2022
% Purpose: Testbed script for ASEN5519 final project
clc; clear; close all;tic;

% Load the Matrices from other Programme
load('poles.mat');

% Define some Initial Criteria
alpha = 100;
epsilon = 10;
score = NaN(length(poles),1);
```

```
%% System Creation
mu = 398600; % km^3/s^2
r_0 = 6678; % km
theta_dot_0 = sqrt(mu/(r_0^3));
A = [0, 1, 0, 0;
    3*mu/(r_0^3), 0, 0, 2*sqrt(mu/r_0);
    0, 0, 0, 1;
    0, -2*sqrt(mu/(r_0^5)), 0, 0];
B = [0, 0;
    1, 0;
    0, 0;
    0, 1/r_0];
C = [1, 0, 0, 0;
    0, 0, 1, 0];
D = [0, 0;
    0, 0];
OLsys = ss(A,B,C,D);
% Define the given initial condition
x_0 = [r_0; 0; 0; (theta_dot_0)];


% Calculate the orbital period
T = 2*pi*sqrt((r_0^3)/mu);
t_vec = linspace(0,T,180000);

% Create the sinusoidal disturbance
sinbump = @(L,A) A*(0.5+0.5*(sin(linspace(-pi/2,pi*3/2,length(t_vec)/3*L))));
B1 = sinbump(1, 0.5)';


radius_ref = [zeros(length(t_vec)/3,1);B1;zeros(length(t_vec)/3,1)];
angle_ref = zeros(length(t_vec),1);

rhistvec = [radius_ref,angle_ref];


%% Iteration Step: Local Step
index = randi([1,length(poles)],1,1);
while alpha > epsilon
    p = poles(index,:);
    try
        K = place(A,B,p);
    catch
        score(index) = Inf;
        index = randi([1,length(poles)],1,1);
        continue
    end
    F = inv(C*(inv(-A+B*K))*B);

    % Do a Check if F is applicable
    if isnan(F)
        score(index) = Inf; % Make the Score Infinite to Discourage Picking
        index = randi([1,length(poles)],1,1);
        continue
    end

    A_cl = A-B*K;
    B_cl = B*F;

    CLsys = ss(A_cl, B_cl, C, D);
    [~,~,X_CL1] = lsim(CLsys,rhistvec,t_vec);
    U_CL1 = -K*X_CL1' + F*rhistvec';

    Thruster1 = sum(abs(U_CL1(1,:)));
    Thruster2 = sum(abs(U_CL1(2,:)));
    score(index) = abs(Thruster2)+abs(Thruster1);
```

```
% Now Check the Boundaries alpha+index or index-alpha
% Upper
if (index+alpha) > length(score)
    scoreUpperIndex = length(score);
else
    scoreUpperIndex = index+alpha;
end
pUpper = poles(scoreUpperIndex,:);
try
    KUpper = place(A,B,pUpper);
catch
    score(index) = Inf;
    index = randi([1,length(poles)],1,1);
    continue
end
FUpper = inv(C*(inv(-A+B*KUpper))*B);
% Do a Check if F is applicable
if isnan(FUpper)
    score(index) = Inf; % Make the Score Infinite to Discourage Picking
    index = randi([1,length(poles)],1,1);
end
A_clUpper = A-B*KUpper;
B_clUpper = B*FUpper;
CLsysUpper = ss(A_clUpper, B_clUpper, C, D);
[~,~,X_CL1Upper] = lsim(CLsysUpper,rhistvec,t_vec);
U_CL1Upper = -KUpper*X_CL1Upper' + FUpper*rhistvec';
Thruster1 = sum(abs(U_CL1Upper(1,:)));
Thruster2 = sum(abs(U_CL1Upper(2,:)));
score(scoreUpperIndex) = abs(Thruster2)+abs(Thruster1);
scoreUpper = score(scoreUpperIndex);

% Lower
if (index-alpha) < 1
    scoreUpperIndex = 1;
else
    scoreLowerIndex = index-alpha;
end
pLower = poles(scoreLowerIndex,:);
try
    KLower = place(A,B,pLower);
catch
    score(index) = Inf;
    index = randi([1,length(poles)],1,1);
    continue
end
FLower = inv(C*(inv(-A+B*KLower))*B);
% Do a Check if F is applicable
if isnan(FLower)
    score(index) = Inf; % Make the Score Infinite to Discourage Picking
    index = randi([1,length(poles)],1,1);
end
A_clLower = A-B*KLower;
B_clLower = B*FLower;
CLsysLower = ss(A_clLower, B_clLower, C, D);
[~,~,X_CL1Lower] = lsim(CLsysLower,rhistvec,t_vec);
U_CL1Lower = -KLower*X_CL1Lower' + FLower*rhistvec';
Thruster1 = sum(abs(U_CL1Lower(1,:)));
Thruster2 = sum(abs(U_CL1Lower(2,:)));
score(scoreLowerIndex) = abs(Thruster2)+abs(Thruster1);
scoreLower = score(scoreLowerIndex);


% Check to See which has better Performance. In this case, anything
% lower is better. Less thruster use.
if(scoreLower < score(index))
    index = scoreLowerIndex;
elseif(scoreUpper < score(index))
```

```
        index = scoreUpperIndex;
    else
        alpha = round(alpha/2);
    end

end
fprintf("The System took %f seconds to run everything\n",toc);
```

```
# -*- coding: utf-8 -*-
"""
Created on Sat Apr 30 14:12:49 2022

@author: sayan & phil
@source: http://www.mwm.im/lqr-controllers-with-python/
"""

import numpy as np
import scipy.linalg
def lqr(A,B,Q,R):
    """Solve the continuous time lqr controller.

    dx/dt = A x + B u

    cost = integral x.T*Q*x + u.T*R*u
    """
    X = np.matrix(scipy.linalg.solve_continuous_are(A, B, Q, R))

    #compute the LQR gain
    K = np.matrix(scipy.linalg.inv(R)*(B.T*X))

    eigVals, eigVecs = scipy.linalg.eig(A-B*K)

    return K, X, eigVals


Q = np.array([[25,0,0,0],[0,25,0,0],[0,0,25,0],[0,0,0,25]])
R = np.array([[2,0],[0,2]])
B = np.array([[0,0],[1,0],[0,0],[0,0.0001]])
A = np.array([[0,1,0,0],[0,0,0,15.4517],[0,0,0,1],[0,0,0,0]])

K,X,eigVals = lqr(A,B,Q,R)
```