Optimal VTOL of SpaceX's Grasshopper

12/12/12

Brian Mahlstedt

Contents

Motivation

Grasshopper is SpaceX's prototypical first stage, a testbed for vertical takeoff and landing (VTOL) to be implemented on future generations of reusable Falcon 9 launch vehicles. Traditional optimal control algorithms such as LQR and LQG could achieve comparable results, but reinforcement learning provides the infrastructure to *adapt* to unmodeled scenarios. This capability to update and learn a new optimal policy makes RL an attractive solution for the unpredictable conditions of landing a rocket-based launch vehicle on Mars. I have already developed LQR and LQG controllers through prior experience, so direct comparisons between the robustness of the controller and the disturbance rejection of the closed-loop plant will be observed.

Results

I successfully derived a model for, simulated, trained, animated, and produced a fully functional and stable controller for a virtual Grasshopper vehicle. The controller performs wonderfully in the simulator, but was not tested on hardware. A processor rated at ~50GFLOPS might need around 10hrs to confidently learn an optimal policy from scratch. Such is not ideal for realtime architectures, but an MCU may deploy a pre-learned algorithm with reasonably power-budgeted adaptability. Building upon a previously optimal policy to accommodate disturbances is significantly less of a computationally intensive task, and *may* be performed onboard in near-realtime.

Reinforcement Learning Theory

A comprehensive theoretical discussion is not appropriate here, but I'll provide my staple equations:

$$V^*(s) = \max_{a \in A} \ R(s,a) + \gamma \sum_{s' \in S}^{|S|} P_{sa}(s')V^*(s)$$

$$\pi^*(s) = \underset{a \in A}{\operatorname{argmax}} \ R(s,a) + \gamma \sum_{s' \in S}^{|S|} P_{sa}(s')V^*(s)$$

A Markov Decision Process (MDP) tuple is logged at every timestep of the dynamical situation, updating its state transition probabilities P_{sa} and reward function R at every 'failure'. It then employs synchronous value iteration by solving Bellman's equations to find the optimal value function V^* and policy π^* for this new MDP model.

2DOF Inverted Pendulum

Before embarking upon the dynamical, computational, and theoretical complexity of the 6 degrees-of-freedom rocket, the logical initial testbed was a standard inverted pendulum. In machine learning, preliminary optimization before practical application often leads to convoluted code, so simplistic beginnings are always preferred. All innovate ideas, peripheral script calls, animation tweaks, parameter manipulations, *all* were vetted in this simple springboard before being implemented upon the full system. This led to streamlined debugging, better state space discretization, more efficient code structure, and many more benefits. Because of the parallelization of both simulators (the only difference being the equations of motion that propagate the dynamics), the following discussion of 2DOF conclusions is *identically applicable to the 6DOF launch vehicle.*

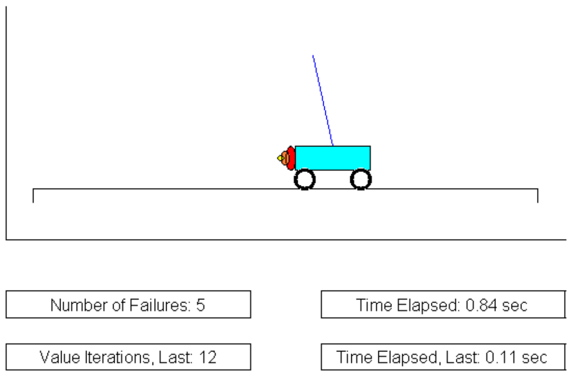
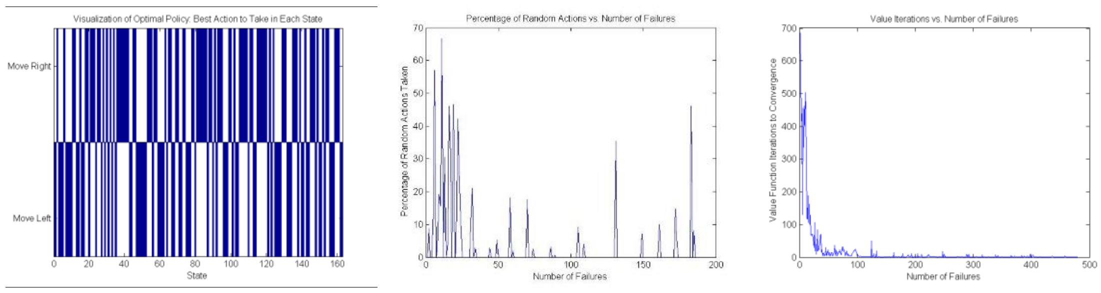


Figure 1: 2DOF Inverted Pendulum Simulator

Fig. 1 shows the simulator in action. The command window updates to display the number of failures, value iterations, and erection time. Fig. 2 shows the binary output of the optimal policy, which is updated at each failure. Figures 3 and 4 show convergence of the algorithm, with the number of value iterations decreasing as the policy learns, and the percentage of actions that are randomly chosen decreasing as the policy becomes more deterministic. Videos of various sequences of trials may be viewed at http://www.youtube.com/watch?v=Vj1dPPQIBaE and http://www.youtube.com/watch?v=Sku2IHwJgn0. These animations are extremely useful for building intuition for the strategies and capabilities of reinforcement learning.



Figures 2-4: Optimal Policy, Random Action Selection %, Value Iterations per Failure

RL Algorithm Construction

The following (Fig. 5) shows the ultimate output of the simulation, a learning curve of success time vs. number of failures. It exhibits initial learning and then plateaus, appearing to converge upon a local optimum because intuitively, the global optimum is a policy that gets the inverted pendulum to remain erect forever. This is a product of multiple sources of nonconvexity: (1) The simulator gets initial rewards for not failing, then associates those state/action pairs with reward, then repeats those patterns in a routine that encompasses only a small fraction of the state space. This can be mitigated with the introduction of randomness into the simulation, making it less 'greedy'. I have manifested this solution in my dynamics by adding noise to the forcing function, and providing a small probability of an action failing or the opposite of the intended action occurring. I have also expanded the range of *state reinitialization*, forcing the simulator to start in very poor states it might never find itself in. Both solutions provide a much more comprehensive final policy. (2) The second source of local optima is poor/coarse discretization of the state space. This can be mollified by logical evaluation of where accuracy is needed; for this simulation, the angle from vertical is more significant of a metric than the lateral position, so that should be discretized more finely. This, along with manipulation of the limits defining failure, provide the simulator encouragement to explore the entire state space intelligently with precision in the regions that need it. Using these techniques, I was able to tweak the simulation and get the pendulum to remain inverted forever.

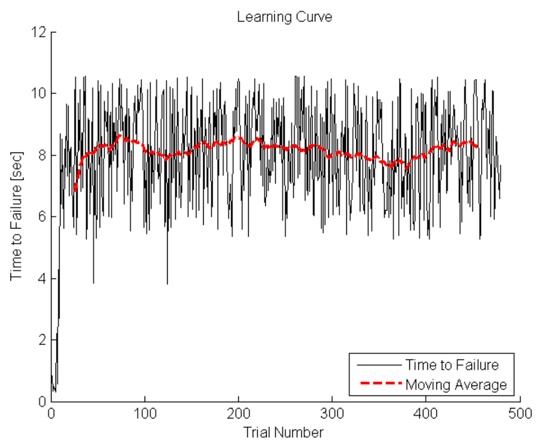
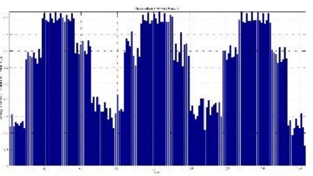
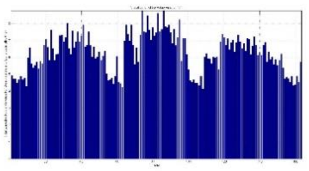
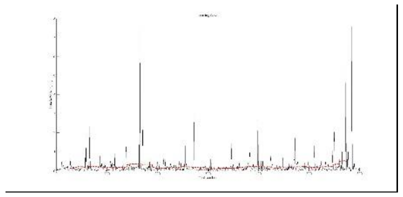
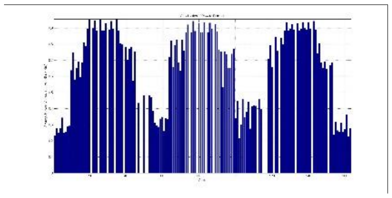
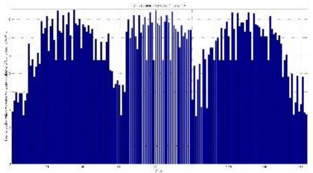
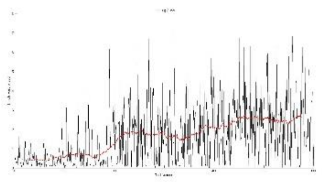


Figure 5: Learning Curve

The following plots illustrate the difference in selection of initial conditions for the state.

Figures 6-8: R, V, and Learning Curve for Uniform State Reinitialization

Figures 9-11: R, V, and Learning Curve for Gaussian State Reinitialization

Uniform state reinitialization entails picking a new state (upon failure) at random from the entire state space. Gaussian state reinitialization draws the state from a normal distribution, centered around each state element being zero ("good" conditions, $x \approx 0$ and $\theta \approx 0$). Figures 6 & 7 vs. 9 & 10, showing the optimal reward and value function, reveal the expected result that the *uniform reinitialization forces the algorithm to explore more of the state space*, yielding a better result less subject to local optima. But Fig. 8 vs. Fig. 11 shows the downside of the uniform distribution; the *Gaussian state reinitialization yields a much better learning curve*, because it generally starts in better conditions and can thusly remain erect for longer. In practice, Gaussian state reinitialization is a good default, and uniform state reinitialization should be used when urgency of learning is a factor. If modularity is an option, starting with uniform for initial learning and continuing with Gaussian once it has gained robustness is the best procedure.

Note also that the reward and value functions make logical sense. There are three main peaks representing the 3-range discretization of the angular velocity. Each of those has 6 peaks for the 6-range discretization of the angular position. And so on for the 3 linear velocity and 3 linear position subsequent peaks.

Here is an extensive list of the parameterized variables in my simulation, all of which I implemented, and the conclusions gleaned from their options:

1. **Synchronous Value Iteration:** Found to be *slightly* faster than asynchronous.
2. **Asynchronous Value Iteration:** Converges to same optimum as synchronous. All of these iteration schemes initialize from the previous value to encourage speed.
3. **Policy Iteration:** Wrote a separate function to solve Bellman's equations for the value functions - it often took fewer iterations to converge, but ran a little more slowly.
4. **State + Action Rewards:** Slowed the simulation down, but provided *great* results and flexibility.
5. **Do-Nothing Actions:** Expanded the action space. Have to be careful this isn't

rewarded too highly or it will always fail (should weight state more than action).

6. **ODE Solver**: Using a higher-order intelligent ode solver to propagate the state increases accuracy and convergence, but is not worth the extra computational time/complexity if the step size is small enough (which 0.01s is).
7. **Dynamic Parameters**: Earth's gravity is around 3x stronger than Mars'. The pendulum remains inverted for shorter periods of time.
8. **Tolerance for Value Iteration:** 0.01 is usually a good combination of time/accuracy. The lower this is, the longer the simulation will take to converge on a new value function after each failure.
9. **Threshold for Single-Convergence**: 20 is usually appropriate. Increase to force more learning stability.
10. **State Failure Limits:** Relax them a bit to allow the controller to move into a new regime of action/state pairs.
11. **Discount Factor:** 0.95 has been working well. Decrease this to make the algorithm weight quicker rewards more heavily.
12. **Dynamics Timestep:** 0.01s is fine for quick use, but use 0.001s for rigor. The finer the time step, the more accurate the physical simulation.
13. **Convergence Criteria:** Assert that value iteration has converged after the *mean* of all value function states has differed below some tolerance, rather than waiting for the worst case to converge. This accounts for those future trials where the number of iterations jumps very high after a seeming run of cumulative value iteration success.

The final parameter, which deserves its own section, is the reward function. Simulation behavior varies significantly with selection of different rewards. Initially, I employed the basic reward function: -1 for failure, and 0 otherwise. This kept the pendulum inverted, but did nothing for regulation of the state; i.e. all states that weren't the failure state were considered equal. A better reward function weighted both, where a higher reward was given for x or θ being zero, and at the limits of the acceptable ranges, the reward was zero. An even better reward function considered the action as well, rewarding for x or θ or F being zero. The reward function I ultimately used implemented all of these, where there were weights for both individual states *against each other*, as well as total state vs. action (ala LQR controller weighting). The reward function also didn't give 0 for the limits of the state space, it instead gave -1 to ensure failure was duly avoided. This reward function behaved spectacularly.

6DOF Grasshopper Problem Statement

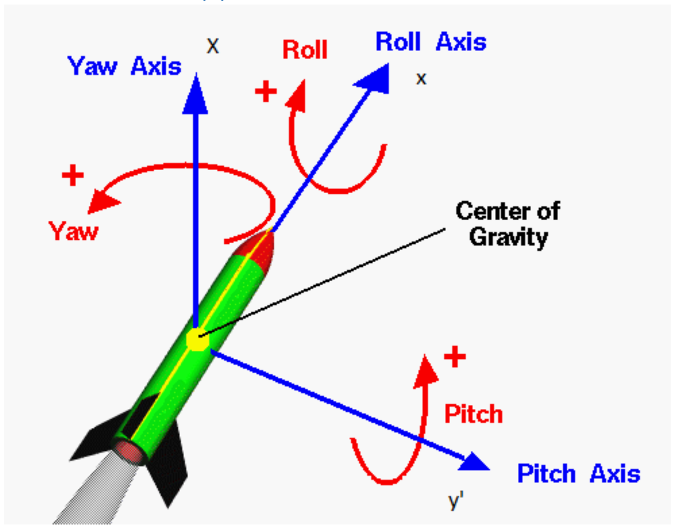


Figure 12: Configuration (credit: NASA Glenn)

I'm defining the altitude and lateral position of Grasshopper's center of mass with (X, Y, Z) and the rotational position with Euler Angles (ψ, θ, ϕ), which are respectively yaw, pitch and roll. The full derivation is in the appendix, but I'm *not* assuming small angles, I'm restricting my Euler angles to avoid singularities, I'm neglecting planetary rotation, gravity differentials, drag, mass loss, and I'm approximating the rocket as a cylinder. These are all extremely reasonable assumptions for a free rigid body in unconstrained space. Note that x is the altitude, and therefore the gravitational force is in the $-\hat{x}$ direction.

My state is therefore

$$\bar{x} = \left[X\ Y\ Z\ \dot{X}\ \dot{Y}\ \dot{Z}\ \psi\ \theta\ \phi\ \dot{\psi}\ \dot{\theta}\ \dot{\phi}\right]^T$$

And the resulting equations of motion are

$$\boxed{\begin{aligned}
\ddot{x} &= \frac{1}{m}\left(F_x c\psi c\theta + F_y(c\psi s\theta s\phi - s\psi c\phi) + F_z(s\psi s\phi + c\psi s\theta c\phi)\right) - g\\
\ddot{y} &= \frac{1}{m}\left(F_x s\psi c\theta + F_y(c\psi c\phi + s\psi s\theta s\phi) + F_z(s\psi s\theta c\phi - c\psi s\phi)\right)\\
\ddot{z} &= \frac{1}{m}\left(-F_x s\theta + F_y c\theta s\phi + F_z c\theta c\phi\right)\\
\\
\ddot{\phi} &= \frac{M_x}{I_a} + \dot{\psi}\dot{\theta}c\theta + \frac{s\theta}{I_t c\theta}\left(M_z c\phi + M_y s\phi + I_a(\dot{\phi}\dot{\theta} - \dot{\psi}\dot{\theta}s\theta) + 2I_t\dot{\psi}\dot{\theta}s\theta\right)\\
\ddot{\theta} &= \frac{1}{I_t}\left(0.5(I_a - I_t)\dot{\psi}^2 s2\theta - I_a\dot{\phi}\dot{\psi}c\theta + M_y c\phi - M_z s\phi\right)\\
\ddot{\psi} &= \frac{1}{I_t c\theta}\left(M_z c\phi + M_y s\phi + I_a(\dot{\phi}\dot{\theta} - \dot{\psi}\dot{\theta}s\theta) + 2I_t\dot{\psi}\dot{\theta}s\theta\right)
\end{aligned}}$$

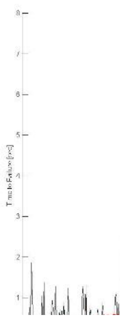
My state space limits yaw and pitch to 6°. Roll is unlimited. After intelligent discretization of these ranges, my state vector ultimately has 1728 elements (+1 for the failure state).

Regarding force capability, I've modeled Grasshopper with an axial Merlin 1D engine, theoretically capable of 147,000lbs of thrust. I've enforced that this main rocket is always firing. There are effectively 4 cold gas thrusters on Grasshopper for lateral stability; one in each quadrant, capable of firing in one direction. This yields a 9-dimensional action space - basically two orthogonal thrusters that can fire $(+/-/0)$. The axial M1D has thrust-vectoring capability of around 15°, which simply manifests itself by increasing the thrust of the lateral cold gas while decreasing the axial force. I've incorporated this.

An integral part of coding any simulator from scratch is the development of an accurate model. Since I have appropriate kinematic expertise, I chose to derive this explicit model by hand. Coupled with the deterministic action space, the simulator proved to be very robust.

6DOF Plots

Here are some plots from the output of my 6DOF Grasshopper simulations. I've had to tweak a few functions to get everything to run, but the inherent difference is almost literally reduced to substitution of the equations of motion. Therefore, my extensive analysis of the inverted pendulum RL applies equally here, so turn to that section for rigor.



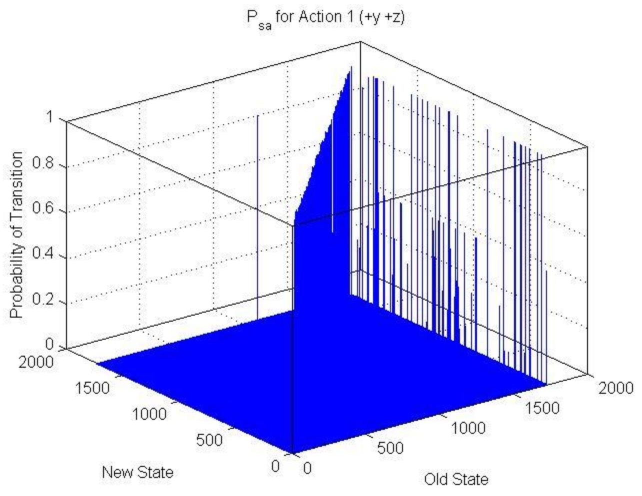


Figure 13: State Transition Probability Matrix

Fig. 13 shows P_{sa} for a specific station transition probability matrix. It remains nearly diagonal, as expected – you are most likely to transition to a similar state. It is extremely sparse, which expresses the relative benefit of other reinforcement learning schemes like Q-Learning. It order to get intuitively useful information from these plots, we'd have to visualize the data in higher dimensions because of the discretization of the state space.

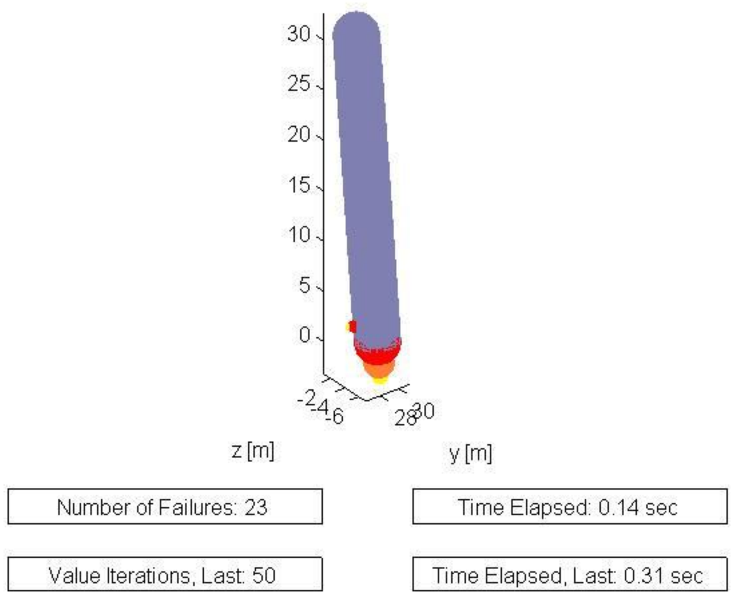


Figure 14: Grasshopper Simulator

Future Work

Here is a short list of the subsequent endeavors I'd pursue next with the simulator - given the fact that I single-handedly took an RL problem from scratch, derived a model by hand, simulated it, learned it, and implemented it in an animation within two months, I'm fairly satisfied with the length of this 'remaining work' section.

1. **Continuous State MDP**: Discretizing the state space while still accurately describing the pose well enough for a decent policy has been hard. I've had to be very crafty in distributing the state space. MATLAB runs out of memory while trying to store P_{sa} when it gets up to around $3000 \times 3000 \times 9$, which is extremely easy to surpass.
2. **Vertical Landing:** I'd have to remove the singularities by introducing a second, redundant set of Euler angles, then invert gravity upon stabilization.
3. **Time-Varying Dynamics / Finite Horizon MDP**: Inclusion of orbital rate, changing gravity, etc. The finite horizon doesn't have to be something chronological; I'd likely choose to step forward with respect to *altitude,* such that when $x = 0$, time's up.
4. **Solidworks Animation**: I've worked with importing 3D solid models and rendering them within MATLAB before, which would have resulted in nice animations (albeit much longer computations).

I'd like to acknowledge Dr. Andrew Ng and the entire teaching staff of CS229 for their dedicated commitments in profession of this course.

Appendix 1: Dynamics

Notes/Assumptions

- I'm *not* assuming small deviations from the nominal vertical position, which would result in 2 decoupled 1DOF inverted pendulums.
- My restoring forces are not in the inertial frame, as a pendulum on a cart. Mine are in the body frame, and must be projected back into the inertial frame.
- I am neglecting mass loss & working with the rocket's principle coordinate frame, which removes the dm/dt and dI/dt terms from Newton's and Euler's equations.
- Using Euler angles with a set sequence $(3-2-1)$ and set ranges of yaw $[0{,}360)$, pitch $(-90{,}90)$, and roll $[0{,}360)$, there is only one singularity where multiple Euler angle sets yield the same attitude. This is the singularity parallel to the z-axis. To circumvent this, I can:
    a. Restrict pitch from ever fully reaching -90 or 90, then orient the body frame so that this singularity is lateral, not along the important rocket axial dimension. This is the workaround that I have implemented.
    b. Define two regions of my state space, one using Euler angles with respect to one frame, the other using another set of Euler angles with respect to another frame. Depending upon the configuration, pick the Euler angles with no singularity.
- I'm neglecting the earth's rotation, although that would be very easy to introduce.
- I'll approximate the principal moments of inertia with a cylinder.
- Angular momentum, linear momentum, and energy are all quantities that are NOT conserved.

Derivation of the Equations of Motion:

My state is:

$$\bar{x} = \left[X\ Y\ Z\ \dot{X}\ \dot{Y}\ \dot{Z}\ \psi\ \theta\ \phi\ \dot{\psi}\ \dot{\theta}\ \dot{\phi}\right]^T$$

So the goal of EOM derivation is to get:

$$\left[\ddot{X}\ \ddot{Y}\ \ddot{Z}\ \ddot{\psi}\ \ddot{\theta}\ \ddot{\phi}\right]^T = f(\bar{x}, \vec{F}, \vec{M})$$

Which can subsequently be propagated to obtain the next state with a simple 1st - order Euler algorithm or a more complicated higher order solver native to Matlab.

Kinematic Relations

$$\vec{r}_{body} = R_\phi R_\theta R_\psi\, \vec{R}_{inertial}$$

$$\vec{r}_{body} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\phi & s\phi \\ 0 & -s\phi & c\phi \end{bmatrix} \begin{bmatrix} c\theta & 0 & -s\theta \\ 0 & 1 & 0 \\ s\theta & 0 & c\theta \end{bmatrix} \begin{bmatrix} c\psi & s\psi & 0 \\ -s\psi & c\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \vec{R}_{inertial}$$

$$\vec{r}_{body} = \begin{bmatrix} \boldsymbol{c\psi c\theta} & \boldsymbol{s\psi c\theta} & \boldsymbol{-s\theta} \\ \boldsymbol{-s\psi c\phi + c\psi s\theta s\phi} & \boldsymbol{c\psi c\phi + s\psi s\theta s\phi} & \boldsymbol{c\theta s\phi} \\ \boldsymbol{s\psi s\phi + c\psi s\theta c\phi} & \boldsymbol{-c\psi s\phi + s\psi s\theta c\phi} & \boldsymbol{c\theta c\phi} \end{bmatrix} \vec{R}_{inertial}$$

The inverse of this matrix yields a passive rotation matrix that represents a vector with body coordinates in the inertial frame. Because rotation matrices are orthogonal, the inverse is equal to the transpose.

$$\vec{R}_{inertial} = \begin{bmatrix} c\psi c\theta & -s\psi c\phi + c\psi s\theta s\phi & s\psi s\phi + c\psi s\theta c\phi \\ s\psi c\theta & c\psi c\phi + s\psi s\theta s\phi & -c\psi s\phi + s\psi s\theta c\phi \\ -s\theta & c\theta s\phi & c\theta c\phi \end{bmatrix} \vec{r}_{body}$$

$$yaw\!: \; 0 \le \psi < 2\pi \qquad pitch\!: \; -\frac{\pi}{2} < \theta < \frac{\pi}{2} \qquad roll\!: \; 0 \le \phi < 2\pi$$

$$\vec{\omega}_B = R_\phi R_\theta R_\psi \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} + R_\phi R_\theta \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + R_\phi \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix}$$

$$\vec{\omega}_B = \begin{bmatrix} \boldsymbol{p} \\ \boldsymbol{q} \\ \boldsymbol{r} \end{bmatrix}_{\boldsymbol{B}} = \begin{bmatrix} \boldsymbol{\dot{\phi} - \dot{\psi} s\theta} \\ \boldsymbol{\dot{\psi} c\theta s\phi + \dot{\theta} c\phi} \\ \boldsymbol{\dot{\psi} c\theta c\phi - \dot{\theta} s\phi} \end{bmatrix}_B = \begin{bmatrix} 1 & 0 & -s\theta \\ 0 & c\phi & c\theta s\phi \\ 0 & -s\phi & c\theta c\phi \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}$$

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \frac{1}{c\theta} \begin{bmatrix} c\theta & s\theta s\phi & s\theta c\phi \\ 0 & c\theta c\phi & -c\theta s\phi \\ 0 & s\phi & c\phi \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix}_B$$

Notice that there is an ‘overhead’ singularity at $\theta = 90^o$, where a perfectly vertical pitch would eliminate a degree of determinism from the configuration. In such an orientation, there is an infinite quantity of Euler angle sets that would describe that attitude of the rocket.

Aside: 3-1-3 Euler angle development would yield:

$$\vec{r}_{body} = R_\psi R_\theta R_\phi \, \vec{R}_{inertial}$$

$$\vec{r}_{body} = \begin{bmatrix} c\psi & s\psi & 0 \\ -s\psi & c\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\theta & s\theta \\ 0 & -s\theta & c\theta \end{bmatrix} \begin{bmatrix} c\phi & s\phi & 0 \\ -s\phi & c\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \vec{R}_{inertial}$$

$$\vec{r}_{body} = \begin{bmatrix} c\phi c\psi - c\theta s\psi s\phi & s\phi c\psi + s\psi c\theta c\phi & s\theta s\psi \\ -c\phi s\psi - c\theta c\psi s\phi & -s\phi s\psi + c\psi c\theta c\phi & s\theta c\psi \\ s\theta s\phi & -s\theta c\phi & c\theta \end{bmatrix} \vec{R}_{inertial}$$

$$\vec{R}_{inertial} = \begin{bmatrix} c\phi c\psi - c\theta s\psi s\phi & -c\phi s\psi - c\theta c\psi s\phi & s\theta s\phi \\ s\phi c\psi + s\psi c\theta c\phi & -s\phi s\psi + c\psi c\theta c\phi & -s\theta c\phi \\ s\theta s\psi & s\theta c\psi & c\theta \end{bmatrix} \vec{r}_{body}$$

$$rotation\text{:}\ \ 0 \le \psi < 2\pi \qquad nutation\text{:}\ \ 0 < \theta < \pi \qquad precession\text{:}\ \ 0 \le \phi < 2\pi$$

$$\vec{\omega}_B = R_\psi R_\theta R_\phi \begin{bmatrix} 0 \\ 0 \\ \dot{\phi} \end{bmatrix} + R_\psi R_\theta \begin{bmatrix} \dot{\theta} \\ 0 \\ 0 \end{bmatrix} + R_\psi \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix}$$

$$\vec{\omega}_B = \begin{bmatrix} p \\ q \\ r \end{bmatrix}_B = \begin{bmatrix} \dot{\phi} s\theta s\psi + \dot{\theta} c\psi \\ \dot{\phi} s\theta c\psi - \dot{\theta} s\psi \\ \dot{\phi} c\theta + \dot{\psi} \end{bmatrix}_B = \begin{bmatrix} s\theta s\psi & c\psi & 0 \\ s\theta c\psi & -s\psi & 0 \\ c\theta & 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}$$

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \frac{1}{s\theta} \begin{bmatrix} s\psi & c\psi & 0 \\ c\psi s\theta & -s\psi s\theta & 0 \\ -s\psi c\theta & -c\psi c\theta & s\theta \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix}_B$$

Notice that again there is a singularity along the z axis at $\theta = 0^o$, where an upright pose would eliminate a degree of determinism from the configuration. In such an orientation, there is an infinite quantity of Euler angle sets that would describe that attitude of the rocket.

Force Balance

Newton's Equations:

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix}_B = R_\phi R_\theta R_\psi \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix}_I \qquad \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix}_I = R_\psi^T R_\theta^T R_\phi^T \begin{bmatrix} u \\ v \\ w \end{bmatrix}_B$$

The thrusters exert forces in the body frame, so we must express these in inertial coordinates:

$$\vec{F}_B = \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix}_B \xrightarrow{now\ inertial} m \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix}_I = R_\psi^T R_\theta^T R_\phi^T \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix}_B + \begin{bmatrix} -mg \\ 0 \\ 0 \end{bmatrix}_I$$

$$\begin{bmatrix}\ddot{x}\\ \ddot{y}\\ \ddot{z}\end{bmatrix}_I = \frac{1}{m}\begin{bmatrix} c\psi c\theta & -s\psi c\phi + c\psi s\theta s\phi & s\psi s\phi + c\psi s\theta c\phi \\ s\psi c\theta & c\psi c\phi + s\psi s\theta s\phi & -c\psi s\phi + s\psi s\theta c\phi \\ -s\theta & c\theta s\phi & c\theta c\phi \end{bmatrix}\begin{bmatrix}F_x\\ F_y\\ F_z\end{bmatrix}_B + \begin{bmatrix}-g\\ 0\\ 0\end{bmatrix}_I$$

Euler's Equations:

In Body Frame: $\vec{M} = \left(\frac{d\vec{H}}{dt}\right)_I = \left(\frac{d\vec{H}}{dt}\right)_B + {}^I\vec{\omega}^B \times \vec{H}$ where:

$$\{I\} = \begin{bmatrix} I_a & 0 & 0 \\ 0 & I_t & 0 \\ 0 & 0 & I_t \end{bmatrix}$$

The rocket is approximated as an axisymmetric prolate cylinder, and the moments of inertia are with respect to a body fixed frame with coordinate axes aligned with the principle axes of the body.

$$\vec{\omega}_B = \begin{bmatrix}p\\ q\\ r\end{bmatrix}_B \qquad \vec{H}_B = \begin{bmatrix}I_a p\\ I_t q\\ I_t r\end{bmatrix}_B$$

$$\begin{bmatrix}M_x\\ M_y\\ M_z\end{bmatrix}_B = \begin{bmatrix}I_a\dot{p}\\ I_t\dot{q}\\ I_t\dot{r}\end{bmatrix}_B + \begin{vmatrix}\hat{x} & \hat{y} & \hat{z}\\ p & q & r\\ I_a p & I_t q & I_t r\end{vmatrix} = \begin{bmatrix} I_a\dot{p} \\ I_t\dot{q} - pr(I_t - I_a) \\ I_t\dot{r} + pq(I_t - I_a)\end{bmatrix}_B$$

Taking derivatives of our body rate - Euler angle rate relations:

$$\begin{aligned}
\dot{p} &= \ddot{\phi} - \ddot{\psi}s\theta - \dot{\psi}\dot{\theta}c\theta \\
\dot{q} &= \ddot{\theta}c\phi - \dot{\theta}\dot{\phi}s\phi + \ddot{\psi}c\theta s\phi + \dot{\psi}\left(\dot{\phi}c\phi c\theta - \dot{\theta}s\phi s\theta\right) \\
\dot{r} &= -\ddot{\theta}s\phi - \dot{\theta}\dot{\phi}c\phi + \ddot{\psi}c\theta c\phi + \dot{\psi}(-\dot{\theta}s\theta c\phi - \dot{\phi}s\phi c\theta)
\end{aligned}$$

$$\begin{bmatrix}M_x\\ M_y\\ M_z\end{bmatrix}_B = \begin{bmatrix} I_a(\ddot{\phi} - \ddot{\psi}s\theta - \dot{\psi}\dot{\theta}c\theta) \\ I_t(\ddot{\theta}c\theta + \ddot{\psi}c\theta s\phi - 2\dot{\psi}\dot{\theta}s\phi s\theta + \dot{\psi}^2 c\theta s\theta c\phi) + I_a(\dot{\phi}\dot{\psi}c\theta c\phi - \dot{\phi}\dot{\theta}s\phi - \dot{\psi}^2 c\theta s\theta c\phi + \dot{\theta}\dot{\psi}s\theta s\phi) \\ I_t(-\ddot{\theta}s\phi + \ddot{\psi}c\theta c\phi - 2\dot{\psi}\dot{\theta}c\phi s\theta - \dot{\psi}^2 c\theta s\theta s\phi) - I_a(\dot{\phi}\dot{\psi}c\theta s\phi + \dot{\phi}\dot{\theta}c\phi - \dot{\psi}^2 c\theta s\theta s\phi - \dot{\theta}\dot{\psi}s\theta c\phi) \end{bmatrix}_B$$

Which will be solved for and propagated in Matlab.

Lagrangian Formulation

Gravitational potential energy, referenced from Earth's surface:

$$V = mgx$$

Translational kinetic energy of center of mass, and rotational kinetic energy about the center of mass:

$$\begin{aligned} T &= \frac{1}{2}m(\dot{x}^2+\dot{y}^2+\dot{z}^2)+\frac{1}{2}\vec{\omega}^T\{I\}\vec{\omega} \\ &= \frac{1}{2}m(\dot{x}^2+\dot{y}^2+\dot{z}^2)+\frac{1}{2}I_a p^2+\frac{1}{2}I_t(q^2+r^2) \end{aligned}$$

$$L = T - V$$

$$L = \frac{1}{2}m(\dot{x}^2+\dot{y}^2+\dot{z}^2)+\frac{1}{2}I_a(\dot{\phi}^2-2\dot{\phi}\dot{\psi}s\theta+\dot{\psi}^2s^2\theta)+\frac{1}{2}I_t\left(\dot{\psi}^2c^2\theta+\dot{\theta}^2\right)-mgx$$

Taking derivatives with respect to our generalized coordinates:

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{x}}-\frac{\partial L}{\partial x}=F_1 \qquad \frac{d}{dt}\frac{\partial L}{\partial \dot{y}}-\frac{\partial L}{\partial y}=F_2 \qquad \frac{d}{dt}\frac{\partial L}{\partial \dot{z}}-\frac{\partial L}{\partial z}=F_3$$

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{\phi}}-\frac{\partial L}{\partial \phi}=M_1 \qquad \frac{d}{dt}\frac{\partial L}{\partial \dot{\theta}}-\frac{\partial L}{\partial \theta}=M_2 \qquad \frac{d}{dt}\frac{\partial L}{\partial \dot{\psi}}-\frac{\partial L}{\partial \psi}=M_3$$

Where the generalized forces simply must be transformed as before:

$$\begin{bmatrix} F_1 \\ F_2 \\ F_3 \end{bmatrix}_I = R_\psi^T R_\theta^T R_\phi^T \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix}_B + \begin{bmatrix} -mg \\ 0 \\ 0 \end{bmatrix}_I$$

But the new generalized moments must be explicitly calculated:

$$M_1 = \begin{bmatrix} M_x \\ M_y \\ M_z \end{bmatrix}_B \cdot \frac{\partial \vec{\omega}_B}{\partial \dot{\phi}} \qquad M_2 = \begin{bmatrix} M_x \\ M_y \\ M_z \end{bmatrix}_B \cdot \frac{\partial \vec{\omega}_B}{\partial \dot{\theta}} \qquad M_3 = \begin{bmatrix} M_x \\ M_y \\ M_z \end{bmatrix}_B \cdot \frac{\partial \vec{\omega}_B}{\partial \dot{\psi}}$$

Evaluating these derivatives yields our desired equations of motion for all six generalized coordinates:

$$\begin{bmatrix} \boldsymbol{F_1} \\ \boldsymbol{F_2} \\ \boldsymbol{F_3} \end{bmatrix} = \begin{bmatrix} \boldsymbol{m\ddot{x}} \\ \boldsymbol{m\ddot{y}} \\ \boldsymbol{m\ddot{z}} \end{bmatrix}$$

$$\begin{bmatrix} M_1 \\ M_2 \\ M_3 \end{bmatrix} = \begin{bmatrix} I_a(\ddot{\phi} - \ddot{\psi}s\theta - \dot{\psi}\dot{\theta}c\theta) \\ I_a\left(\dot{\phi}\dot{\psi}c\theta - \dot{\psi}^2 s\theta c\theta\right) + I_t(\ddot{\theta} + \dot{\psi}^2 c\theta s\theta) \\ I_a\left(-\ddot{\phi}s\theta - \dot{\phi}\dot{\theta}c\theta + \ddot{\psi}s^2\theta + 2\dot{\psi}\dot{\theta}s\theta c\theta\right) + I_t(\ddot{\psi}c^2\theta - 2\dot{\psi}\dot{\theta}c\theta s\theta) \end{bmatrix}$$

Which can be formulated into the same result derived via force balance. Isolating the state accelerations, we obtain our final update formulas for the simulator (all forces and moments are in the body frame):

$$\boxed{\begin{aligned}
\ddot{x} &= \frac{1}{m}(F_x c\psi c\theta + F_y(c\psi s\theta s\phi - s\psi c\phi) + F_z(s\psi s\phi + c\psi s\theta c\phi)) - g \\
\ddot{y} &= \frac{1}{m}(F_x s\psi c\theta + F_y(c\psi c\phi + s\psi s\theta s\phi) + F_z(s\psi s\theta c\phi - c\psi s\phi)) \\
\ddot{z} &= \frac{1}{m}(-F_x s\theta + F_y c\theta s\phi + F_z c\theta c\phi) \\[1em]
\ddot{\phi} &= \frac{M_x}{I_a} + \dot{\psi}\dot{\theta}c\theta + \frac{s\theta}{I_t c\theta}\left(M_z c\phi + M_y s\phi + I_a\left(\dot{\phi}\dot{\theta} - \dot{\psi}\dot{\theta}s\theta\right) + 2I_t\dot{\psi}\dot{\theta}s\theta\right) \\
\ddot{\theta} &= \frac{1}{I_t}(0.5(I_a - I_t)\dot{\psi}^2 s2\theta - I_a\dot{\phi}\dot{\psi}c\theta + M_y c\phi - M_z s\phi) \\
\ddot{\psi} &= \frac{1}{I_t c\theta}\left(M_z c\phi + M_y s\phi + I_a(\dot{\phi}\dot{\theta} - \dot{\psi}\dot{\theta}s\theta) + 2I_t\dot{\psi}\dot{\theta}s\theta\right)
\end{aligned}}$$

Appendix 2: Source Code

main.m

```
%% Main Control Script
% CS229 Final Project
% Brian Mahlstedt

% Introduction --------------------------------------------------
----------

% This simulator employs reinforcement learning to optimize the
thruster
% policy for VTOL of the SpaceX Grasshopper vehicle.

% 1) Determine best action for current state
% 2) Run the dynamics to get the next state
% 3) Count the rewards and transitions
% 4) If it hasn't failed yet, return to step 1 and continue simu-
lation
% 5) If it fails, use the total rewards/transitions to update the
Markov
%    Decision Process (MDP) model by calculating a new reward
function R
%    and station transition probability matrix Psa
% 6) Reinitilize state and repeat 1...until learning has con-
verged

cd('C:\Users\Brian\Dropbox\Graduate Work\CS229\Final Pro-
ject\Code')
clear all;close all;clc
tic % begin timing

% Parameter Definition
max_fail = 1000; % Max failures allowed. Should converge before
this.
disp_start = 1; % Start display after this # of failures. Can be
used to
% rush through initial trials and only display after the MDP
model is
% reasonably accurate, or set equal to max_fail to never display
(fastest
% simulation), or set to 1 to visualize from the beginning.
gamma = 0.95; % Discount factor, emphasizes earlier rewards.
tol = 0.01; % Convergence criteria for value iteration.
no_learning_threshold = 20; % Consecutive runs where value itera-
tion
% converges in one iteration (near-perfect MDP model)
dt = .01; % Time step resolution for the dynamics of the simula-
tion.
success_time = 60*60; % If the sim balances for this many sec-
onds, it ends
% and considers itself a success, no matter the state of the RL
algorithm.
ns = 4*4*4*3*3*3+1; % Number of states. The last state denotes
failure.
na = 9; % Number of actions.

% Initialization
consecutive_no_learning = 0; % number of trials with near-perfect
MDP model
nfail = 0; % Number of failures.
iterations = 0;  % Counts simulation iterations.
iter_start = 0; % Number of sim iterations at start of current
trial.
success = 0; % Boolean to indicate the success condition.
numrandact = 0; % Counter for actions that were selected at ran-
dom.

% Starting State
x = 0; y = 0; z = 0;
x_dot = 0; y_dot = 0; z_dot = 0;
psi = 6*pi/180/3*randn; psi_dot = 180*pi/180/3*randn;
theta = 6*pi/180/3*randn; theta_dot = 180*pi/180/3*randn;
phi = 180*pi/180*(1/3*randn+1); phi_dot = 180*pi/180/3*randn;
s = state(psi,psi_dot,theta,theta_dot,phi,phi_dot); % Discretized
state.

% Preallocation
iters2fail = zeros(max_fail); % Sim iterations until failure.
randact = zeros(max_fail); % Percentage of random actions per
failure.
vp_iter = zeros(max_fail); % Number of value or policy iterations
until
% convergence.
Psa_counts = zeros(ns,ns,na); % These count the transitions that
happen
% over multiple timesteps between failures (old state, new state,
action).
Psa = ones(ns,ns,na)/ns; % Each layer is a matrix, where element
(old state,
% new state, action) shows the probability that taking action k
in state i
% will result in state j. This is updated at every failure. Ini-
tialize
% asa uniform distribution (same likelihood of reaching any
state).
R_counts = zeros(ns,2); % Counts the reward over multiple
timesteps between
% failures. The first column is the new state and the rewards ob-
served
% there. The second column counts each time you went to that new
state.
R = zeros(ns,1); % Current reward function associated with each
state.
% Only updated at every failure.
V = zeros(ns,1); % Total expected discounted future reward for
each state.
p = zeros(ns,1); % Optimal Policy. This is an array that simply
contains
% the best action to take in each state to maximize V.

% Animation
if disp_start < max_fail
    warning('off','MATLAB:DELETE:FileNotFound');
    delete('animation.mp4')
    vidObj = VideoWriter('animation.mp4','MPEG-4');
    vidObj.Quality = 100;
    vidObj.FrameRate = 1/dt; % real time
    open(vidObj);
end

% Main Simulation Loop -------------------------------------------
----------

while (nfail < max_fail) && ...
        (consecutive_no_learning < no_learning_threshold)

  % Determine the best action for the current state based on Psa
and V
  % s is just passed to action and reward to check for failure
state.
  [a,coin] = action(Psa,V,s,psi,theta);
  if coin
      numrandact = numrandact+1;
  end

  % Get the next state by simulating the dynamics

[x,y,z,x_dot,y_dot,z_dot,psi,theta,phi,psi_dot,theta_dot,phi_dot]
= ...
      dynamics(a,dt,x,y,z,x_dot,y_dot,z_dot,...
        psi,theta,phi,psi_dot,theta_dot,phi_dot);
  new_s = state(psi,psi_dot,theta,theta_dot,phi,phi_dot); % Dis-
cretize.
  if nfail+1 > disp_start
     simdisplay(x,y,z,psi,theta,phi,(iterations -
iter_start)*dt,...
        iters2fail(nfail)*dt,nfail,vp_iter(nfail),a);
     writeVideo(vidObj,getframe(gcf));
  end
  iterations = iterations + 1;

  % Reward Function
  Rew = reward(new_s,psi,theta,a);

  % Update Transition and Reward Counters
  Psa_counts(s,new_s,a) = Psa_counts(s,new_s,a) + 1;
  R_counts(new_s,:) = R_counts(new_s,:) + [Rew 1];

  % If Simulation Fails
  if (new_s == ns)

    % Compute new MDP model (update Psa and R)
    for k = 1:na % Loop over action layers.
      for i = 1:ns % Loop over old states.
        total = sum(Psa_counts(i,:,k));
        if (total > 0) % Does nothing if state-action hasn't been
tried.
          Psa(i,:,k) = Psa_counts(i,:,k)/total;
          % Calculates probabilies of all next states from this
old state.
        end
      end
    end
    for i = 1:ns
      if (R_counts(i,2) > 0) % Checks if we ever got to that new
state.
        R(i) = R_counts(i,1)/R_counts(i,2);
        % Reward associated with each state, updated every fail-
ure. It is
        % the sum of all the rewards noticed when you got to that
state
        % divided by the number of times you went to that state.
      end
    end

    % Value Iteration to optimize V for this new MDP model
    iter = 0; % Loop counter.
    change = 1; % Just set higher than tol to start while loop.
    new_V = zeros(ns,1); % preallocate V array to compare to pre-
vious
    allVs = zeros(1,na); % Preallocate array of 'V's for each ac-
tion
```

```
    while change > tol % change = ALL differences in the value
vector > tol
      iter = iter + 1;
      for i = 1:ns
          [psi,~,theta,~,~,~] = state2x(i);
          for k = 1:na

              % Synchronous - loops through states, then updates
V(s) all
              % at once
              allVs(k) = reward(i,psi,theta,k) +
gamma*Psa(i,:,k)*V;

%               % Asynchronous - updates each V(s) as it changes
%               allVs(k) = reward(i,x,theta,k) +
Psa(i,:,k)*new_V;

          end
        new_V(i) = max(allVs); % update using a*
      end
      change = max(abs(V - new_V)); % Can change to mean and make
tol tight
      V = new_V;
    end

%     % Policy Iteration to optimize V and pi for this new MDP
model
%     iter = 0; % Loop counter.
%     change = 1; % Just set higher than tol to start while loop.
%     new_p =  zeros(ns,1); % preallocate policy array to compare
to previous
%     for i = 1:ns
%       p(i) = action(Psa,V,i,psi,theta); % determine policy,
just once before loop
%     end
%     while change > tol % change = ALL differences in the value
vector > tol
%       iter = iter + 1;
%       V = bellman(Psa,p,R,gamma); % solve Bellman's equations
for V
%       for i = 1:ns
%           new_p(i) = action(Psa,V,i,psi,theta); % determine
best action
%       end
%       new_V = bellman(Psa,new_p,R,gamma);
%       change = max(abs(V - new_V)); % Can change to mean and
make tol tight
%       p = new_p;
%     end

    % Check if convergence occurred in one iteration (no learn-
ing)
    if (iter == 1)
      consecutive_no_learning = consecutive_no_learning + 1;
    else
      consecutive_no_learning = 0;
    end

    % Update Counters
    nfail = nfail + 1;
    vp_iter(nfail) = iter;
    iters2fail(nfail) = iterations - iter_start;
    iter_start = iterations; % Start time for next trial.
    randact(nfail) = numrandact/iters2fail(nfail); % % of random
actions.
    numrandact = 0; % Restart counter.

    % Calculate Policy pi* from V* (redundant if using policy it-
eration)
    for i = 1:ns
        [psi,~,theta,~,~,~] = state2x(i);
        [p(i),~] = action(Psa,V,i,psi,theta);
    end

    % Print useful info to command window.
    fprintf('Nfails: %i     Viters: %i      Time: %.2f s\n',...
        [nfail iter iters2fail(nfail)*dt])

    % Visualization of MDP Model (the animation is figure 1)
    % R (figure 2)
    figure(2),set(gcf,'units','normalized','posi-
tion',[.505 .55 .24 .36])
    bar(R),axis tight,grid on
    xlabel('State'),ylabel('Average Reward Associated with State,
R(s)')
    title('Visualization of Reward Function')
    % V* (figure 3)
    figure(3),set(gcf,'units','normalized','posi-
tion',[.755 .55 .24 .36])
    bar(V),axis tight,grid on
    xlabel('State')
    ylabel('Total Expected Future Rewards For Being in State and
Acting Optimally, V*(s) ')
    title('Visualization of the Value Function, V*')
    % pi* (policy) (figure 4)
    figure(4),set(gcf,'units','normalized','posi-
tion',[.265 .55 .24 .36])
    bar(p),axis tight,grid on
    xlabel('State'),ylabel('Optimal Action to take in State')
    set(gca,'YTick',[1 2 3 4 5 6 7 8 9],'YTickLabel',...
        {'+y +z';'+y 0z';'+y -z';'0y +z';'0y 0z';'0y -z';'-y
+z';'-y 0z';'-y -z'})
    title('Visualization of Optimal Policy, \pi^*')
```

```
%     % Psa (figures 5 to 4+na)
%     for i = 1:na
%         figure(4+i),stem3(Psa(:,:,i),'markersize',1)
%         xlabel('Old State'),ylabel('New State')
%         zlabel('Probability of Transition')
%         title(['Psa for Action ' num2str(i)])
%         set(gcf,'units','normalized','position',...
%             [.505+(i-1)*.495/na .06 .48/na .36])
%     end

    % Reinitialize State.
    % (initial conditions map the entire space to force RL to ex-
plore)

    % Uniform
%     x = 0; y = 0; z = 0; x_dot = 0; y_dot = 0; z_dot = 0;
%     psi = 12*pi/180*(2*rand-1);
%     psi_dot = 180*pi/180*(2*rand-1);
%     theta = 12*pi/180*(2*rand-1);
%     theta_dot = 180*pi/180*(2*rand-1);
%     phi = 12*pi/180*(2*rand-1);
%     phi_dot = 180*pi/180*(2*rand-1);

    % Normal
    x = 0; y = 0; z = 0; x_dot = 100; y_dot = 0; z_dot = 0;
    psi = 6*pi/180/3*randn;
    psi_dot = 180*pi/180/3*randn;
    theta = 6*pi/180/3*randn;
    theta_dot = 180*pi/180/3*randn;
    phi = 180*pi/180*(1/3*randn+1);
    phi_dot = 180*pi/180/3*randn;

    s = state(psi,psi_dot,theta,theta_dot,phi,phi_dot);

  else % If the simulation didn't fail, simply continue
      s = new_s;
  end

  % If the inverted pendulum balances for XXXs, end simulation as
success
  if (iterations-iter_start)*dt > success_time
      iters2fail(nfail+1) = iterations - iter_start; % Last ele-
ment = success.
      success = 1;
      break
  end
end

% Simulation End. Visualization & Analysis ----------------------
----------

% Truncate unused elements of preallocated arrays
iters2fail(iters2fail == 0) = [];
randact(nfail+1:end) = [];
vp_iter(vp_iter == 0) = [];

% Plot Learning Curve
figure,hold on,semilogy(iters2fail*dt,'k');
% Compute Simple Moving Average
window = 50;
i = 1:window;
w = ones(1,window) ./ window;
weights = filter(w,1,iters2fail*dt);
x1 = window/2:size(iters2fail*dt,2)-(window/2);
h = plot(x1,weights(window:size(iters2fail*dt,2)), 'r--');
set(h, 'LineWidth', 2);
title('Learning Curve')
xlabel('Trial Number')
ylabel('Time to Failure [sec]')
warning('off','MATLAB:legend:IgnoringExtraEntries');
% legend('Time to Failure','Moving Average','location','south-
east')
if success
    plot(nfail+1,iters2fail(end)*dt,'og','MarkerSize',5,...
        'MarkerFaceColor','g') % Plots a green dot if ends in
success.
end

% Calculate total computation time
t = toc;
h = floor(t/60^2);
m = floor((t-h*60^2)/60);
s = t-h*60^2-m*60;
fprintf('Time elapsed: %i hour(s), %i minute(s), %.2f sec-
ond(s)\n',[h m s])

% Plot value/policy iterations vs. number of failures
figure,plot(1:nfail,vp_iter)
xlabel('Number of Failures')
ylabel('Value/Policy Function Iterations to Convergence')
title('Value/Policy Iterations vs. Number of Failures')

% Plot the percentage of random actions taken per failure
figure,plot(1:nfail,randact*100)
xlabel('Number of Failures')
ylabel('Percentage of Random Actions Taken')
title('Percentage of Random Actions vs. Number of Failures')

% Animation Output
if disp_start < max_fail
    close(vidObj);
    winopen('animation.mp4')
end
```

dynamics.m

```
function [new_x,new_y,new_z,new_x_dot,new_y_dot,new_z_dot,...

new_psi,new_theta,new_phi,new_psi_dot,new_theta_dot,new_phi_dot]
= ...
        dynamics(action,dt,x,y,z,x_dot,y_dot,z_dot,...
            psi,theta,phi,psi_dot,theta_dot,phi_dot)
% Simulates the dynamics of the rocket via the equations of mo-
tion.
%                       . . .                .    .    .
% State vector = [X Y Z X Y Z psi theta phi psi theta phi]
%
% States are propagated using a variable order RungaKutta method.
% Input and output angular values are radians.
% Psi Theta Phi = Yaw Pitch Roll (3-2-1 Euler)

% Parameters for Simulation Dynamics ----------------------------
----------

% Atmosphere
g = 3.71; % Gravitational parameter. [m/s^2] Earth = 9.81, Mars =
3.71

% Grasshopper
h = 30; % Height. [m]
w = 4; % Width. [m]
r = w/2; % Radius. [m]
v = pi*r^2*h; % Volume. [m^3]
p = 600; % Adjusted homogenous density. (RP-1 = 900) [kg/m^3]
m = v*p; % Mass. [kg]
Ia = m*r^2/2; % Axial moment of inertia. [kg-m^2]
It = m/12*(3*r^2+h^2); % Transverse moment of inertia. [kg-m^2]

% Engines
F_m1d = 650e3; % Full force of M1D (147kip) [N]
F_lat = 50e3; % Full force of lateral cold gas thrusters [N]

% Introduction of Randomness ------------------------------------
----------

action_flip_prob = 0.05; % probability action is not as intended
if rand < action_flip_prob
    actions = 1:9;
    actions(action) = []; % remove intended action from consider-
ation
    action = actions(randi(8)); % pick new unintended action at
random (2 = 1-na)
end

% Determine Action ----------------------------------------------
----------

% 1 - M1D fires (+x), lat fires (+y), lat fires (+z)
% 2 - M1D fires (+x), lat fires (+y), lat off   (+0)
% 3 - M1D fires (+x), lat fires (+y), lat fires (-z)
% 4 - M1D fires (+x), lat off   (+0), lat fires (+z)
% 5 - M1D fires (+x), lat off   (+0), lat off   (+0)
% 6 - M1D fires (+x), lat off   (+0), lat fires (-z)
% 7 - M1D fires (+x), lat fires (-y), lat fires (+z)
% 8 - M1D fires (+x), lat fires (-y), lat off   (+0)
% 9 - M1D fires (+x), lat fires (-y), lat fires (-z)

if action == 1
    Fx = F_m1d; Fy = F_lat;  Fz = F_lat;
elseif action == 2
    Fx = F_m1d; Fy = F_lat;  Fz = 0;
elseif action == 3
    Fx = F_m1d; Fy = F_lat;  Fz = -F_lat;
elseif action == 4
    Fx = F_m1d; Fy = 0;      Fz = F_lat;
elseif action == 5
    Fx = F_m1d; Fy = 0;      Fz = 0;
elseif action == 6
    Fx = F_m1d; Fy = 0;      Fz = -F_lat;
elseif action == 7
    Fx = F_m1d; Fy = -F_lat; Fz = F_lat;
elseif action == 8
    Fx = F_m1d; Fy = -F_lat; Fz = 0;
elseif action == 9
    Fx = F_m1d; Fy = -F_lat; Fz = -F_lat;
end

% More Introduction of Randomness -------------------------------
----------

force_noise_factor = 0.2; % multiplied by between 1-.. and 1+..
no_force_prob = 0.01; % Force is 0 with this probability

Fx = Fx*(1+force_noise_factor*(1-2*rand));
Fy = Fy*(1+force_noise_factor*(1-2*rand));
Fz = Fz*(1+force_noise_factor*(1-2*rand));

if rand < no_force_prob
  Fx = 0; Fy = 0; Fz = 0;
end

% Calculate Moments From Forces ---------------------------------
----------

Mx = 0; % No thruster can exert a moment about the axial direc-
tion.
% (No direct roll actuator).
My = Fz*h/2; % [N*m]
Mz = -Fy*h/2; % [N*m]

% Equations of Motion -------------------------------------------
----------
x_acc =
1/m*(Fx*cos(psi)*cos(theta)+Fy*(cos(psi)*sin(theta)*sin(phi)-...
    sin(psi)*cos(phi))+Fz*(sin(psi)*cos(phi)+...
    cos(psi)*sin(theta)*cos(phi)))-g;
y_acc = 1/m*(Fx*sin(psi)*cos(theta)+Fy*(cos(psi)*cos(theta)+...

sin(psi)*sin(theta)*sin(phi))+Fz*(sin(psi)*sin(theta)*cos(phi)-..
.
    cos(psi)*sin(phi)));
z_acc = 1/m*(-
Fx*sin(theta)+Fy*cos(theta)*sin(phi)+Fz*cos(theta)*cos(phi));
psi_acc =
1/It/cos(theta)*(Mz*cos(phi)+My*sin(phi)+Ia*(phi_dot*theta_dot...
    -
psi_dot*theta_dot*sin(theta))+2*It*psi_dot*theta_dot*sin(theta));
theta_acc = 1/It*(.5*(Ia-It)*psi_dot^2*sin(2*theta)-
Ia*phi_dot*psi_dot...
    *cos(theta)+My*cos(phi)-Mz*sin(phi));
phi_acc =
Mx/Ia+psi_dot*theta_dot*cos(theta)+sin(theta)/It/cos(theta)*...
    (Mz*cos(phi)+My*sin(phi)+Ia*(phi_dot*theta_dot-
psi_dot*theta_dot...
    *sin(theta))+2*It*psi_dot*theta_dot*sin(theta));

% Return New State Variables (using Euler's method) -------------
----------

new_x  = x + dt*x_dot;
new_y  = y + dt*y_dot;
new_z  = z + dt*z_dot;
new_x_dot = x_dot + dt*x_acc;
new_y_dot = y_dot + dt*y_acc;
new_z_dot = z_dot + dt*z_acc;
new_psi = psi + dt*psi_dot;
new_theta = theta + dt*theta_dot;
new_phi = mod(phi + dt*phi_dot,2*pi); % modulus 360
new_psi_dot = psi_dot + dt*psi_acc;
new_theta_dot = theta_dot + dt*theta_acc;
new_phi_dot = phi_dot + dt*phi_acc;
```

action.m

```
function [a coin] = action(Psa,V,s,psi,theta)
% Determines the best action for the current state based on Psa and V
% ( find pi(s) = argmax(a) { sum_over_all_states_s' [Psa(s')V(s')] }
% If there is a tie amongst best actions, one is selected randomly from the
% top choices.

na = size(Psa,3);
A = zeros(na,2);
% The first column is the expected value of total future discounted
% rewards for taking that action in the current state. E_{s' ~ Psa) [V(s')]
% The second column is the original index of that expected value. I will
% sort later, so these are for information retention.

for i = 1:na
    A(i,:) = [reward(s,psi,theta,i)+Psa(s,:,i)*V i];
end
A = flipud(sortrows(A)); % Place best actions first.
A(A(:,1)~=A(1,1),:) = []; % Truncate actions that aren't (tied for) best.
num_tied = size(A,1); % Number of actions tied for best.
a = A((floor(num_tied*rand)+1),2); % Picks an action randomly from those
% that are tied (even if there is only 1).

if num_tied > 1
    coin = 1; % to indicate a random action was chosen
else
    coin = 0;
end
```

reward.m

```
function R = reward(s,psi,theta,a)
% This function receives the state and action and calculates the reward.

sa_weight = 1; % to weight state regulation over thrust minimization

if any(a == [1 3 7 9]) % two lateral thrusters fire
    ease = 0;
elseif any(a == [2 4 6 8]) % one lateral thruster fires
    ease = .5;
elseif a == 5% neither lateral thruster fires
    ease = 1;
end

if s == 1729
    R = -1;
else
    R = sa_weight*(...
        .5*(1-abs(psi)/(12*pi/180)) + .5*(1-abs(theta)/(12*pi/180)))...
            +(1-sa_weight)*...
        ease;
end

% Rewards more for being vertical not firing.
% All have between 0 (failure) and 1 (psi = 0 or theta = 0 or F = 0) reward,
% which is weighted based on a cost function ala LQR control, for a total
% reward between 0 and 1, except for the failure state R = -1.
```

state.m

```
function state = state(psi,psi_dot,theta,theta_dot,phi,phi_dot)
% This function returns a discretized value for a continuous state vector.
% There are 4 angular position ranges for psi (yaw), 4 for theta (pitch),
% and 4 for phi (roll). This yields the discrete state output as one of
% 4*4*4*3*3*3 = 1728 permutations plus 1 for the 'failure' state. A finer
% discretization produces a better policy, but a larger state space
% requiring longer computations.

% Define limits for discrete state regions
% (outside of these = failure, except for phi (roll) which is cyclic)
psi_lim = pi/180*[-6 -1 0 1 6];
psi_dot_lim = pi/180*[-180 -45 45 180];
theta_lim = pi/180*[-6 -1 0 1 6];
theta_dot_lim = pi/180*[-180 -45 45 180];
phi_lim = pi/180*[0 90 180 270 360];
phi_dot_lim = pi/180*[-180 -45 45 180];

% Calculate number of states based upon number of discrete ranges
psi_tot = length(psi_lim)-1;
psi_dot_tot = length(psi_dot_lim)-1;
theta_tot = length(theta_lim)-1;
theta_dot_tot = length(theta_dot_lim)-1;
phi_tot = length(phi_lim)-1;
phi_dot_tot = length(phi_dot_lim)-1;
numst = psi_tot * psi_dot_tot * theta_tot * theta_dot_tot * ...
    phi_tot * phi_dot_tot; % not including failure state

% Calculate multiplier of permutations in subdivisions
phi_dot_mult = numst/phi_dot_tot;
phi_mult = numst/phi_dot_tot/phi_tot;
theta_dot_mult = numst/phi_dot_tot/phi_tot/theta_dot_tot;
theta_mult = numst/phi_dot_tot/phi_tot/theta_dot_tot/theta_tot;
psi_dot_mult = numst/phi_dot_tot/phi_tot/theta_dot_tot/theta_tot/psi_dot_tot;
psi_mult = numst/phi_dot_tot/phi_tot/theta_dot_tot/theta_tot/psi_dot_tot/psi_tot;

% Find ranges that the current state elements fall within
psi_intvl = interval(psi,psi_lim);
psi_dot_intvl = interval(psi_dot,psi_dot_lim);
theta_intvl = interval(theta,theta_lim);
theta_dot_intvl = interval(theta_dot,theta_dot_lim);
phi_intvl = interval(phi,phi_lim);
phi_dot_intvl = interval(phi_dot,phi_dot_lim);

% Determine state by summing subsequent contributions from state elements
if any([psi_intvl psi_dot_intvl theta_intvl theta_dot_intvl ...
        phi_intvl phi_dot_intvl]==0)
    state = numst+1;
else
    state = phi_dot_mult*(phi_dot_intvl-1)+...
        phi_mult*(phi_intvl-1)+...
        theta_dot_mult*(theta_dot_intvl-1)+...
        theta_mult*(theta_intvl-1)+...
        psi_dot_mult*(psi_dot_intvl-1)+...
        psi_mult*(psi_intvl-1)+...
        +1; % to shift states back up by one
end
```

state2x

```
function [psi,psi_dot,theta,theta_dot,phi,phi_dot] = state2x(s)
% This function takes the discretized value for a continuous state vector
% and returns the 'average' of the corresposding state elements.

% Define limits for discrete state regions
% (outside of these = failure, except for phi (roll) which is cyclic)
psi_lim = pi/180*[-6 -1 0 1 6];
psi_dot_lim = pi/180*[-180 -45 45 180];
theta_lim = pi/180*[-6 -1 0 1 6];
theta_dot_lim = pi/180*[-180 -45 45 180];
phi_lim = pi/180*[0 90 180 270 360];
phi_dot_lim = pi/180*[-180 -45 45 180];

% Calculate number of states based upon number of discrete ranges
psi_tot = length(psi_lim)-1;
psi_dot_tot = length(psi_dot_lim)-1;
theta_tot = length(theta_lim)-1;
theta_dot_tot = length(theta_dot_lim)-1;
phi_tot = length(phi_lim)-1;
phi_dot_tot = length(phi_dot_lim)-1;
numst = psi_tot * psi_dot_tot * theta_tot * theta_dot_tot * ...
    phi_tot * phi_dot_tot; % not including failure state

% Calculate multiplier of permutations in subdivisions
phi_dot_mult = numst/phi_dot_tot;
phi_mult = numst/phi_dot_tot/phi_tot;
theta_dot_mult = numst/phi_dot_tot/phi_tot/theta_dot_tot;
theta_mult = numst/phi_dot_tot/phi_tot/theta_dot_tot/theta_tot;
psi_dot_mult = numst/phi_dot_tot/phi_tot/theta_dot_tot/theta_tot/psi_dot_tot;
psi_mult = numst/phi_dot_tot/phi_tot/theta_dot_tot/theta_tot/psi_dot_tot/psi_tot;

if s == 1729 % failure
    psi = psi_lim(1);
    psi_dot = psi_dot_lim(1);
    theta = theta_lim(1);
    theta_dot = theta_dot_lim(1);
    phi = phi_lim(1);
    phi_dot = phi_dot_lim(1);
    % reward will then be zero from the state
else
    phi_dot_intvl = ceil(s/phi_dot_mult);
    phi_intvl = ceil((s-phi_dot_mult*(phi_dot_intvl-1))/...
        phi_mult);
    theta_dot_intvl = ceil((s-phi_dot_mult*(phi_dot_intvl-1)-...
        phi_mult*(phi_intvl-1))/theta_dot_mult);
    theta_intvl = ceil((s-phi_dot_mult*(phi_dot_intvl-1)-...
        phi_mult*(phi_intvl-1)-theta_dot_mult*(theta_dot_intvl-1))/theta_mult);
    psi_dot_intvl = ceil((s-phi_dot_mult*(phi_dot_intvl-1)-...
        phi_mult*(phi_intvl-1)-theta_dot_mult*(theta_dot_intvl-1)-...
        theta_mult*(theta_intvl-1))/psi_dot_mult);
    psi_intvl = ceil((s-phi_dot_mult*(phi_dot_intvl-1)-...
        phi_mult*(phi_intvl-1)-theta_dot_mult*(theta_dot_intvl-1)-...
        theta_mult*(theta_intvl-1)-psi_dot_mult*(psi_dot_intvl-1))/psi_mult);

    % Find ranges that the current state elements fall within
    psi = interval2x(psi_intvl,psi_lim);
    psi_dot = interval2x(psi_dot_intvl,psi_dot_lim);
    theta = interval2x(theta_intvl,theta_lim);
    theta_dot = interval2x(theta_dot_intvl,theta_dot_lim);
    phi = interval2x(phi_intvl,phi_lim);
    phi_dot = interval2x(phi_dot_intvl,phi_dot_lim);
end
```

simdisplay.m

```
function simdisplay(x,y,z,psi,theta,phi,time_elapsed,tlast,nfail,val_iter,a)
% 'Animates' the simulation by updating the rocket position at each
% timestep, visualizing the movement within a figure.

figure(1) % Figure 1 always denotes the simulation display
set(gcf,'doublebuffer','on')
set(gca,'nextplot','replacechildren');
cla

% Plot Grasshopper & M1D
h = 30; r = 2; % [m]
[xc yc zc] = cylinder;
[xs ys zs] = sphere;
gray = [.5 .5 .7];
orange = [1 0.5 0.2];
origin = [x y z]; % center of mass to rotate about
hold on
p1 = surf(xc*r-y,yc*r-z,zc*h-h/2+x,'facecolor',gray,'edgecolor',gray); % cylinder
rocket_rotate(p1,psi,theta,phi,origin);
p2 = surf(xs*r-y,ys*r-z,zs*r-h/2+x+h,'facecolor',gray,'edgecolor',gray); % nose
rocket_rotate(p2,psi,theta,phi,origin);
p3 = surf(xs*r-y,ys*r-z,zs*r-h/2+x,'facecolor','r','edgecolor','r'); % M1D
rocket_rotate(p3,psi,theta,phi,origin);
p4 = surf(xs*.66*r-y,ys*.66*r-z,zs*.66*r-r-h/2+x,'facecolor',orange,'edgecolor',orange); % M1D
rocket_rotate(p4,psi,theta,phi,origin);
p5 = surf(xs*.33*r-y,ys*.33*r-z,zs*.33*r-1.66*r-h/2+x,'facecolor','y','edgecolor','y'); % M1D
rocket_rotate(p5,psi,theta,phi,origin);

% Conditionally Plot Lateral Thrusters
if any(a == [1 2 3])
    p6 = surf(xs*r/4+r-y,ys*r/4-z,zs*r/4+r-h/2+x,'facecolor','r','edgecolor','r'); % pushes my +y
    rocket_rotate(p6,psi,theta,phi,origin);
    p7 = surf(xs*r/8+1.25*r-y,ys*r/8-z,zs*r/8+r-h/2+x,'facecolor','y','edgecolor','y'); % pushes my +y
    rocket_rotate(p7,psi,theta,phi,origin);
end
if any(a == [7 8 9])
    p8 = surf(xs*r/4-r-y,ys*r/4-z,zs*r/4+r-h/2+x,'facecolor','r','edgecolor','r'); % pushes my -y
    rocket_rotate(p8,psi,theta,phi,origin);
    p9 = surf(xs*r/8-1.25*r-y,ys*r/8-z,zs*r/8+r-h/2+x,'facecolor','y','edgecolor','y'); % pushes my -y
    rocket_rotate(p9,psi,theta,phi,origin);
end
if any(a == [1 4 7])
    p10 = surf(xs*r/4-y,ys*r/4+r-z,zs*r/4+r-h/2+x,'facecolor','r','edgecolor','r'); % pushes my +z
    rocket_rotate(p10,psi,theta,phi,origin);
    p11 = surf(xs*r/8-y,ys*r/8+1.25*r-z,zs*r/8+r-h/2+x,'facecolor','y','edgecolor','y'); % pushes my +z
    rocket_rotate(p11,psi,theta,phi,origin);
end
if any(a == [3 6 9])
    p12 = surf(xs*r/4-y,ys*r/4-r-z,zs*r/4+r-h/2+x,'facecolor','r','edgecolor','r'); % pushes my -z
    rocket_rotate(p12,psi,theta,phi,origin);
    p13 = surf(xs*r/8-y,ys*r/8-1.25*r-z,zs*r/8+r-h/2+x,'facecolor','y','edgecolor','y'); % pushes my -z
    rocket_rotate(p13,psi,theta,phi,origin);
end

hold off
view(3),axis equal
xlabel('x [m]'),ylabel('y [m]'),zlabel('z [m]')

% Labels
% axis([-3 3 -1 1.5]),pbaspect([1 2.5/6 1])
set(gca,'Position',[.1 .33 .8 .6])
title('Animation'),xlabel('y [m]'),ylabel('z [m]'),zlabel('x [m]')

% Add Real Time to Plot (slows down simulation)
set(0,'ShowHiddenHandles','on')
delete(findobj('HorizontalAlignment','center')) % delete previous
annotation('textbox',[.55 .15, .35, .05],'String',...
    ['Time Elapsed: ' num2str(time_elapsed,'%.2f') ' sec'],...
    'HorizontalAlignment','center','BackgroundColor','w');
annotation('textbox',[.55 .05, .35, .05],'String',...
    ['Time Elapsed, Last: ' num2str(tlast,'%.2f') ' sec'],...
    'HorizontalAlignment','center','BackgroundColor','w');
annotation('textbox',[.1 .15, .35, .05],'String',...
    ['Number of Failures: ' num2str(nfail,'%i')],...
    'HorizontalAlignment','center','BackgroundColor','w');
annotation('textbox',[.1 .05, .35, .05],'String',...
    ['Value Iterations, Last: ' num2str(val_iter,'%i')],...
    'HorizontalAlignment','center','BackgroundColor','w');

drawnow
```

rocket_rotate.m

```
function rocket_rotate(h,psi,theta,phi,origin)
% Inputs the Euler Angles (3-2-1 yaw/pitch/roll) in radians and rotates
the
% 3D graphic given by handle h.

% Define Passive Rotation Matrices
R_psi = [cos(psi) sin(psi) 0 ; -sin(psi) cos(psi) 0 ; 0 0 1];
R_theta = [cos(theta) 0 -sin(theta) ; 0 1 0 ; sin(theta) 0 cos(theta)];

rotate(h,[0 -1 0]',psi*180/pi,origin) % matlab's -y is my z
rotate(h,R_psi'*[-1 0 0]',theta*180/pi,origin) % matlab's -x is my y
(rotated once)
rotate(h,R_theta'*R_psi*[0 0 1]',phi*180/pi,origin) % matlab's z is my
x (rotated twice)
```

bellman.m

```
function V = bellman(Psa,p,R,gamma)
% This function solves Bellman's equations for the value function ac-
cording
% to some policy.

ns = length(R); % number of states
A = zeros(ns,ns); % preallocating the array that will be AV = R
for i = 1:ns
    A(i,:) = Psa(i,:,p(i));
end
A = A-diag(ones(1,ns)/gamma);
V = A\(R/-gamma);
```

interval.m

```
function intvl = interval(val,limits)
% This function takes a value and a vector of limits for the value, and
% then outputs the interval that the value falls between. The output is
0
% if the value is outside of the range difined by the limits. Limits
must
% be in ascending order.
%
% For example
% interval(1.5, [1 3.5 4 17 150]) = 1
% interval(6, [1 3.5 4 17 150]) = 3
% interval(149.9, [1 3.5 4 17 150]) = 4
% interval(.5, [1 3.5 4 17 150]) = 0
% interval(200, [1 3.5 4 17 150]) = 0

for j = 1:length(limits)
    if val < limits(j)
        intvl = j-1;
        return
    end
end
intvl = 0; % if return is never invoked, the value is higher than the
high
% limit and we return 0 to indicate failure.
```

interval2x.m

```
function val = interval2x(intvl,limits)
% This function takes an interval and a vector of limits for intervals,
and
% then outputs the mean of the respective interval. Limits must be in
% ascending order.
%
% For example
% interval2x(1, [1 2 4 7 11]) = 1.5
% interval2x(2, [1 2 4 7 11]) = 3
% interval2x(3, [1 2 4 7 11]) = 5.5
% interval2x(4, [1 2 4 7 11]) = 9

val = mean([limits(intvl) limits(intvl+1)]);
```

rotations.m

```
clear all;close all;clc
```

Kinematic Relations

```
syms p t s % phi [0,360) theta (-90,90) psi [0,360)

Rp = [1 0 0 ; 0 cos(p) sin(p) ; 0 -sin(p) cos(p)];
Rt = [cos(t) 0 -sin(t) ; 0 1 0 ; sin(t) 0 cos(t)];
Rs = [cos(s) sin(s) 0 ; -sin(s) cos(s) 0 ; 0 0 1];

I2B = Rp*Rt*Rs; % Inertial Frame to Body Frame
B2I = I2B'; % Body Frame to Inertial Frame

E2B = [1 0 -sin(t) ; 0 cos(p) cos(t)*sin(p) ; 0 -sin(p)
cos(t)*cos(p)];
% euler angle derivatives to body rates [p;q;r]=E2B*[pd;td;sd]
B2E = simplify(inv(E2B)); % singularity at theta = 90deg
% body rates to euler angle derivatives [pd;td;sd]=B2E*[p;q;r]
```

Force Balance

```
% Newton's Equations

syms m g % mass gravity
syms Fx Fy Fz % forces in the body frame

dummy = 1/m*(B2I*[Fx;Fy;Fz]-[m*g;0;0]);
xdd_fb = dummy(1); ydd_fb = dummy(2); zdd_fb = dummy(3);

% Euler's Equations

syms sd sdd pd pdd td tdd % derivatives of Euler angles
syms Ia It % axial and tranverse moments of inertia

pp = pd-sd*sin(t);
qq = sd*sin(p)*cos(t)+td*cos(p);
rr = sd*cos(t)*cos(p)-td*sin(p);
% double letters are body rates to not overload p with phi, the
Euler angle

ppdot = pdd-sdd*sin(t)-sd*td*cos(t);
qqdot = tdd*cos(p)-td*pd*sin(p)+sdd*cos(t)*sin(p)+...
    sd*(pd*cos(p)*cos(t)-td*sin(p)*sin(t));
rrdot = -tdd*sin(p)-td*pd*cos(p)+sdd*cos(t)*cos(p)+...
    sd*(-td*sin(t)*cos(p)-pd*sin(p)*cos(t));

syms Mx My Mz % moments in body frame

sdd_fb = solve(My-(It*qqdot-pp*rr*(It-Ia)),sdd);
tdd_fb = simplify(solve(subs(Mz-(It*rrdot+pp*qq*(It-
Ia)),sdd,sdd_fb),tdd));
sdd_fb = simplify(subs(sdd_fb,tdd,tdd_fb));
pdd_fb = simplify(solve(subs(Mx-Ia*ppdot,sdd,sdd_fb),pdd));
% dd = double dot

fprintf('************************************************************
****\n\n')
fprintf('Force Balance \n\n')
fprintf('************************************************************
****\n\n')
fprintf('psi double dot = \n')
pretty(sdd_fb),
fprintf('\n\ntheta double dot = \n')
pretty(tdd_fb)
fprintf('\n\nphi double dot = \n')
pretty(pdd_fb)
```

```
****************************************************************
Force Balance

****************************************************************

psi double dot =

  Mz cos(p) + My sin(p) + Ia pd td - Ia sd td sin(t) + 2 It sd td
sin(t)
  ---------------------------------------------------------------
-------
                                It cos(t)


theta double dot =

                2
  Ia sin(2 t) sd
  --------------- - Ia pd cos(t) sd + My cos(p) - Mz sin(p)     2
         2                                                     sd
sin(2 t)
  ------------------------------------------------------------ - ---
---------
                             It
2


phi double dot =

  Mx
  -- + sd td cos(t) + (sin(t) (Mz cos(p) + My sin(p) + Ia pd td -
  Ia

     Ia sd td sin(t) + 2 It sd td sin(t))) / (It cos(t))
```

Lagrangian Formulation

```
% Since Matlab cannot take generic derivatives with respect to
time, I've
% derived (and double-checked) the following formulas; this
section simply
% performs the form manipulation.

M3 = Mx*-sin(t) + My*cos(t)*sin(p) + Mz*cos(t)*cos(p);
M2 = My*cos(p) - Mz*sin(p);
M1 = Mx;

tdd_l = simplify(solve(...
    M2-(It*(tdd+sd^2*cos(t)*sin(t))+Ia*(pd*sd*cos(t)-
sd^2*sin(t)*cos(t)))...
    ,tdd));
pdd_l = solve(M1-Ia*(pdd-sdd*sin(t)-sd*td*cos(t)),pdd);
sdd_l = simplify(solve(subs(...
    M3-(Ia*(-pdd*sin(t)-
pd*td*cos(t)+sdd*sin(t)^2+2*sd*td*sin(t)*cos(t))+...
    It*(sdd*cos(t)^2-2*sd*td*cos(t)*sin(t))),pdd,pdd_l),sdd));
pdd_l = simplify(subs(pdd_l,sdd,sdd_l));

fprintf('************************************************************
****\n\n')
fprintf('Lagrangian Formulation \n\n')
fprintf('************************************************************
****\n\n')
fprintf('psi double dot = \n')
pretty(sdd_l),
fprintf('\n\ntheta double dot = \n')
pretty(tdd_l)
fprintf('\n\nphi double dot = \na')
pretty(pdd_l)
```

```
****************************************************************

Lagrangian Formulation
```

```
************************************************************

psi double dot =

  Mz cos(p) + My sin(p) + Ia pd td - Ia sd td sin(t) + 2 It sd td
sin(t)
  -----------------------------------------------------------------
-------
                                It cos(t)


theta double dot =

                2
  Ia sin(2 t) sd
  --------------- - Ia pd cos(t) sd + My cos(p) - Mz sin(p)     2
         2                                                     sd
sin(2 t)
  ------------------------------------------------------- - ---
---------
                             It
2


phi double dot =
a
  Mx
  -- + sd td cos(t) + (sin(t) (Mz cos(p) + My sin(p) + Ia pd td -
  Ia

     Ia sd td sin(t) + 2 It sd td sin(t))) / (It cos(t))
```

Verification

```
% specify random values
p = 2*pi*rand;
t = -pi/2+pi*rand;
s = 2*pi*rand;
pd = -1+2*rand;
td = -1+2*rand;
sd = -1+2*rand;
Ia = 5+10*rand;
It = 15+10*rand;
Mx = 10*rand;
My = 10*rand;
Mz = 10*rand;

fprintf('**********************************************************
****\n\n')
fprintf('Tests - subsequent answers should be identical:\n\n')
fprintf('**********************************************************
****\n\n')
subs(sdd_fb)
subs(sdd_l)
subs(tdd_fb)
subs(tdd_l)
subs(pdd_fb)
subs(pdd_l)
```

```
************************************************************

Tests - subsequent answers should be identical:

************************************************************


ans =

    2.4850


ans =

    2.4850


ans =

   -0.1570


ans =

   -0.1570


ans =

   -2.0141


ans =

   -2.0141
```