Electromagnetic formation flight (EMFF) is an enabling technology for a number of spacecraft mission architectures. The RINGS program will be the first time EMFF is demonstrated in a microgravity environment. Nonlinearities due to magnetic field interactions preclude linear feedback controllers from being used to control the RINGS system. Approximate dynamic programming is explored in this paper as a potential method for developing a controller. Aggregation and cost approximation methods are used to develop the cost-to-go of the system. Direct input and rollout architectures are presented for building a controller based on the cost-to-go. Aggregation and cost approximation methods are both able to produce a valid cost-to-go for the RINGS system. Both direct input and rollout control architectures are able to drive the system to the desired state given a cost-to-go, with the rollout architecture performing the same as a direct input controller. Overall, dynamic programming was successful in developing a working RINGS controller.

INTRODUCTION

The use of electromagnets for maintaining a spacecraft formation, or electromagnetic formation flight (EMFF), is an enabling technology for a various number of spacecraft mission architectures. Electromagnetic formation flight was originally conceived as a potential enabling technology for NASA's Terrestrial Planet Finder (TPF) mission. The TPF mission sought to use satellite-based interferometry to search for Earth-like planets. Interferometry in space requires an array of spacecraft spinning around a central point. While there are other methods to achieve a spinning array in space, the original design for TPF called for thrusters that continuously fired to provide the centrifugal force. However, thrusters have a limited amount of fuel and can cloud the sensitive optics on the spacecraft. Electromagnetic formation flight could solve the issue of limited fuel by harvesting energy from the sun. EMFF would also not interfere with sensitive optics. The TPF mission was eventually canceled, but the idea of electromagnetic formation flight lives on. Besides the TPF mission, EMFF is an enabling technology for a number of space missions, including robotic assembly, fractionated spacecraft architectures, and remote sensing.

While EMFF has been demonstrated on the ground, it has yet to be demonstrated in space. The Resonant Inductive Near-field Generation System (RINGS) program, which is set to launch
to the International Space Station in June of 2013, will be the first time EMFF is demonstrated in a microgravity environment. The RINGS system will integrate onto the SPHERES control platform, which has been onboard the International Space Station since 2006. The RINGS system is a pair of electromagnets, each with 100 turns of wire in a 68 cm diameter circle. The RINGS can produce a sinusoidal current of up to 18A RMS. This creates an oscillating magnetic field which the RINGS use to actuate against each other for both position and attitude control. A picture of the RINGS flight units can be seen in Figure 1.

Figure 1. RINGS Flight Units

While previous work has been performed on control of an EMFF system, these controllers assume large separation distances between each spacecraft and that the system has three orthogonal coils on each spacecraft; neither is the case for RINGS. Instead, RINGS is a nonholonomic, underdamped system with each vehicle in close proximity to each other and only one coil per vehicle. This paper focuses on applying dynamic programming to the control of the RINGS system. Previous EMFF controllers have been developed using Lyapunov functions. However, the assumptions used to derive these controllers will not work for RINGS. Dynamic programming has been used to develop controllers for other types of nonlinear systems, but not for satellite formation flight. Spacecraft control algorithms have typically relied on linear state feedback laws; however, these types of control formulations will not work for the RINGS system because of the nonlinearity introduced by electromagnet interactions.

The rest of the paper is organized as follows. In the theory section the RINGS system is described and then formulated into a dynamic programming problem framework. Aggregation and cost approximation methods are then presented in order to develop the cost-to-go for the system. Methods for developing controllers based on the cost-to-go are subsequently presented. The results section presents the cost-to-go and controller performance for a special case of the RINGS system. The conclusion section summarizes the results and presents ideas for further work.

THEORY

The objective of a RINGS controller is to provide inputs to the RINGS system that drive the state of the system to some desired state. Linear state feedback does not work with the RINGS
system because of the nonlinearities involved with magnetic field interaction. While there are other methods to develop controllers for the RINGS system, few offer the performance and robustness properties associated with controllers developed using dynamic programming. Traditional dynamic programming methods are infeasible for this problem because of the high dimensionality of the system and the continuous state space; however, recent breakthroughs in approximate dynamic programming have enabled the use of dynamic programming for RINGS control.

Dynamic programming relies on Bellman’s principle of optimality. In his work, Richard Bellman writes “An optimal policy has the property that whatever the initial state and the initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.” In other words, the optimal trajectory cost from a particular state is independent of the path used to reach that state.

Figure 2 shows the process used in this paper to develop a controller using dynamic programming. The problem is first formulated based on the description of a system. Then, either aggregation or cost approximation is used to develop an approximation of the cost-to-go from a given state. The cost-to-go approximation is then used to implement a controller using a direct input method or a rollout method.

**System Description**

The RINGS system contains a pair of alternating current electromagnets, each mounted on a control vehicle that contains a set of thrusters. The thrusters will be used to emulate a reaction wheel set, meaning thrusters will only be used to provide torques and not translational forces.

**Table 1. Term Descriptions of Two Coils In Proximity**

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vec{d}_1, \vec{d}_2$</td>
<td>Vector from the center of the coil to a point on the coil</td>
</tr>
<tr>
<td>$i_1, i_2$</td>
<td>Current in coil</td>
</tr>
<tr>
<td>$d\vec{l}_1, d\vec{l}_2$</td>
<td>Incremental segment of coil length</td>
</tr>
<tr>
<td>$\vec{r}$</td>
<td>Vector from origin of $d\vec{l}_1$ to origin of $d\vec{l}_2$</td>
</tr>
<tr>
<td>$\vec{s}$</td>
<td>Vector from center of coil 1 to origin of $d\vec{l}_2$</td>
</tr>
</tbody>
</table>
The forces and torques on one electromagnet from the other can be found by integrating the Biot-Savart law, as seen in Equations (1) and (2), respectively, while the terms used in these equations are defined in Table 1 and Figure 3.\(^\text{11}\)

\[
\vec{\tau}_2 = \frac{\mu_0 i_1 i_2}{4\pi} \oint \left( \oint \vec{\mathbf{r}} \times d\vec{l}_1 \frac{1}{r^2} \right) \times d\vec{l}_2
\]

\[
\vec{\tau}_2 = \frac{\mu_0 i_1 i_2}{4\pi} \oint \vec{\mathbf{a}}_2 \times \left[ \left( \oint \vec{\mathbf{r}} \times d\vec{l}_1 \frac{1}{r^2} \right) \times d\vec{l}_2 \right]
\]

In order to keep the focus of this paper on dynamic programming controller development and not RINGS dynamics, only the axial case of RINGS will be considered. In this special case, the RINGS are assumed to be axially aligned about the center of the coils and fixed in attitude.

\[
\text{Equation (3)}
\]

Figure 4 shows the physical realization of the axial RINGS case. In this setup, there is no torque on either RINGS unit and the force on each unit is parallel to the axial direction. The input to the RINGS system will be the product of \(i_1\) and \(i_2\), since the force is proportional to the current product. Let \(r\) be the separation distance between the two RINGS units, \(\dot{r}\) be the rate of change of the separation distance, \(m\) be the mass of one RINGS unit, and \(F_{\text{ax}}(r, u)\) be the force on a coil found by evaluating Equation (1) in the axial direction with a separation of \(r\) and an input of \(u = i_1 i_2\). The equation of motion for this formulation is seen in Equation (3).
Notice that Equation (3) is nonlinear because of the dependence of the force on the separation distance. Since Equations (1) and (2) have yet to be solved in closed form, the state propagation cannot have also yet to be solved in closed form. However, conventional ordinary differential equations solvers can be used to numerically approximate the state of the system during one control cycle.

**Problem Formulation**

For the scope of this work, infinite horizon dynamic programming will be used. Stationary policies produced by infinite horizon solutions are advantageous because finite time horizon policies require storing the policy (or whatever metric is used to find the optimal policy) as a function of time, adding an extra dimension to the problem. Since the dimension of this problem will already be quite large, reducing the dimension of the problem will be particularly advantageous. However, an infinite horizon formulation will not work for cost function purely based on inputs, as solutions that take an infinitely long time to reach the target state would arise. Therefore, infinite horizon formulations will work best for minimum time and weighted time-input formulations.

There are two general types of infinite horizon frameworks that would be appropriate for a RINGS controller: discounted cost and stochastic shortest path. A discounted cost formulation is more appropriate for a regulator framework because trajectories are assumed to be infinitely long. Discounted cost formulations also have stronger convergence guarantees than stochastic shortest path formulations. A stochastic shortest path formulation is better suited for minimum time to target formulations. Since the objective of this paper is to develop a controller to drive the RINGS system to a state, versus regulating around one, the stochastic shortest path formulation will be used.

Let \( x_k \) and \( u_k \) be the state of and the input to a system at time \( k \), respectively. The axial RINGS system, as derived in the System Description section, can be propagated using ordinary differential equation solvers in order to write the system in the form shown in Equation (4).

\[
x_{k+1} = f(x_k, u_k)
\]  

(4)

The state of the RINGS system as derived System Description section is a vector containing the separation distance \( r \) and the rate of change of the separation distance \( \dot{r} \). The state space of the axial RINGS system is continuous over \( r \in \mathbb{R}^+ \) and \( \dot{r} \in \mathbb{R} \), meaning there are an infinite number of states the RINGS system could take. Since it is impossible to implement traditional dynamic programming methods for an infinite number of states, approximate dynamic programming must be used. The two methods of approximate dynamic programming considered in this paper are aggregation and cost approximation.

Let \( g(x_k, u_k, x_{k+1}) \) be the cost of transitioning from state \( x_k \) to state \( x_{k+1} \) using input \( u_k \). The objective of the controller will be to minimize the cost function described in Equation (5).

\[
J(x_0) = \lim_{N \to \infty} \frac{1}{N} \sum_{k=0}^{N-1} g_k(x_k, u_k, x_{k+1})
\]  

(5)

In the stochastic shortest path formulation of dynamic programming, there is an absorbing termination state \( T \) where the system will remain once that state is reached. This termination...
state is used to write the transition cost for the stochastic shortest path formulation, seen in Equation (6). The termination state contains the target state(s) of the controller.

\[
g_k(x_k, u_k, x_{k+1}) = \begin{cases} 
1, & x_k \in T \\
0, & x_k \in T 
\end{cases} \tag{6}
\]

A key assumption in shortest path formulations is that there is a positive probability of reaching the termination state in finite time, ensuring that Equation (5) is bounded. However, a system may never reach the target state. For RINGS, there are two distinct cases where the system may never reach the set of target states. The first is if the RINGS reach "terminal velocity" where no input will ever bring the RINGS back together regardless of actuation time. This condition is possible because the electromagnetic force diminishes as the RINGS move apart. To handle this case, the system will transition to the termination state after \(\tau\) seconds with cost \(\tau\). The other case is when the RINGS crash into each other. To handle this case, the system will transition to the termination state at cost if the separation distance reaches zero. Assuming that the RINGS can reach the target state(s) if they do not crash together or reach terminal velocity, then there is a positive probability of reaching the termination state in finite time, satisfying the stochastic shortest path formulation assumption.

One disadvantage of the stochastic shortest path formulation is that while formulation includes an absorbing termination state, in reality there is no such state. Depending on the choice of termination state(s), a physical system could enter a termination state and transition out of it during the next control cycle, since the controller does not consider trajectories where the system leave the termination state. Although not examined in this paper, a linear quadratic regulator (LQR) style cost function combined with a discounted dynamic programming formulation could also be used to develop a RINGS controller. Let \(x_t\) be the target state of the system, while \(Q\) and \(R\) are tuning matrices for the state error and control effort, respectively. The transition cost function for the linear quadratic formulation is seen in Equation (7).

\[
g_k(x_k, u_k, x_{k+1}) = (x_k - x_t)^T Q (x_k - x_t) + u_k^T R u_k \tag{7}
\]

A controller using Equation (7) as the transition cost function would consider trajectories after the system has reached the target state, making this type of controller more akin to a regulator, as the linear quadratic regulator name suggests. This transition cost function must be used with the discounted dynamic programming formulation to ensure that Equation (5) remains bounded. As for tuning the tuning matrices \(Q\) and \(R\), there exists methods for tuning LQR costs based on simulation trajectories. Also, notice that Equation (6) only depends on the state, not the input. The stochastic shortest path formulation does not penalize control because using electricity is considered “free,” since electricity can be harvested by a spacecraft via solar panels. If there were to be a cost associated with using electricity for control, then the LQR discounted dynamic programming formulation might be more appropriate than the stochastic shortest path formulation.

**Aggregation Formulation**

The first method for determining the cost-to-go from a particular state is state aggregation. Let \(P\) be the set of all possible states of the system. For dynamic programming to be feasible, the state space must be reduced to a finite number of states. Therefore, subsets of points in \(P\) will be aggregated to make mutually disjoint subsets of \(P\). The states of the system of this formulation are \(i = 0, ..., n\), while \(D_i \subset P, i = 0, ..., n\) represents the set of aggregate states that the discrete state \(i\) represents.
Using the framework described in Figure 5, the aggregate state transition probabilities \( p_{ij}(u) \) can be computed using Equation (8). Since the original formulation of the problem was deterministic, \( p_{xy}(u) = 1 \); however, a stochastic transition probability of the original could still be used in this type of formulation.

\[
p_{ij}(u) = \sum_x d_{ix} \sum_y p_{xy}(u) a_{yj} \quad (8)
\]

While there are many different ways to define the disaggregation and aggregation probabilities, this paper will use a hard aggregation technique, described in Equations (9) and (10).

\[
d_{ix} = \begin{cases} 1, & x \in D_i \\ 0, & x \notin D_i \end{cases} \quad (9)
\]
\[
a_{yj} = \begin{cases} 1, & y \in D_j \\ 0, & y \notin D_j \end{cases} \quad (10)
\]

Using the aggregation formulation described in Figure 5, Bellman’s equation describing the optimal cost-to-go \( J^* \) can be seen in Equation (11).

\[
J^*(i) = g(i, u) + \sum_{j=0}^{n} p_{ij}(u) J^*(j) \quad (11)
\]

To solve for the fixed point of \( J^* \), value iteration, policy iteration, or linear programming can be used. Because of the potentially large scale of this problem, this paper will use value iteration—which is better suited for large scale problems—to solve Bellman’s equation.

For value iteration to be implemented, the term \( p_{ij}(u) \), described in Equation 8, must be calculated. Since \( p_{xy}(u) \) is not explicitly known, it can be calculated using simulation. By sampling \( D_i \) and propagating the sample forward \( q \) times, the law of large numbers states that \( p_{ij}(u) \) can be approximated using Equations (12) and (13), where \( x_s \) is the \( s \)th sample of \( D_i \).

\[
H_I(y) = \begin{cases} 1, & y \in D_j \\ 0, & y \notin D_j \end{cases} \quad (12)
\]
\[ p_{ij}(u) \approx \frac{1}{q} \sum_{s=1}^{q} H_j(f(x_s, u)) \]  

(13)

One disadvantage of aggregation versus cost approximation is that a region of states \( P \) must be defined. Given a fixed number of states, if \( P \) is too large, the aggregate states will become very large, reducing the accuracy of the controller. If \( P \) is too small, optimal trajectories that exit and reenter \( P \) will not be considered. Another disadvantage of aggregation is having to account for states that, by the nature of the system dynamics, exist \( P \) regardless of input. In this case, for the purpose of this paper the system assumed to “time out”, transitioning to the termination state with cost \( \tau \).

**Cost Approximation Formulation**

In a cost approximation formulation, a basis function combined with tuning parameters is used to approximate the cost-to-go of the system. The cost approximation takes the general form of \( \tilde{J}(\phi(x), s) \), where \( \phi(x) \) is a basis function that can be computed given state \( x \), and \( s \) is a set of tuning parameters for the approximation.\(^{14} \) For a linear cost approximation formulation, the cost-to-go is approximated using Equation (14).

\[ \tilde{J}(x, r) = \phi(x)^T s \]  

(14)

The key concept of cost approximation is that an approximation for the cost-to-go can be found using state information combined with a set of pre-computed tuning parameters \( s \). The only items that need to be stored are the basis function and the tuning parameters, as compared to an aggregation formulation where the cost-to-go at every aggregate state must be stored. Cost approximation is especially useful for problems with many different dimensions, since the “curse of dimensionality”\(^{15} \) hampers high dimensional dynamic programming formulations.

There are several methods for computing the tuning parameters, including temporal differences, least squares temporal difference, and least squares policy evaluation. For this paper, the \( s \) vector is updated according to the least squares temporal differences (LSTD) method. The LSTD methods attempt to solve the fixed point described in Equation (15) by simulating trajectories and using those trajectories to develop an approximation for \( C \) and \( d \).

\[ Cs^* = d \]  

(15)

As a trajectory moves from state \( i_k \) to \( i_{k+1} \), \( C_k \) and \( d_k \) are updated according to Equations (16), (17), and (18). When a batch of simulations is complete, \( s_k \) is updated using Equation (19).\(^{16} \) This process is repeated until \( s_k \) converges to \( s^* \).

\[ C_k = (1 - \delta_k)C_{k-1} + \delta_k \phi(i_k) (\phi(i_k) - \phi(i_{k+1}))^T \]  

(16)

\[ C_k = (1 - \delta_k)d_{k-1} + \delta_k \phi(i_k) g(i_k, i_{k+1}) \]  

(17)

\[ \delta_k = \frac{1}{k + 1}, k = 0,1,... \]  

(18)

\[ s_k = C_k^{-1}d_k \]  

(19)

Since the trajectories will eventually terminate, it is necessary to restart the trajectories. To ensure sufficient exploration, a random feasible state is used as the starting state of the system.
Controller Implementation

One of the disadvantages of using dynamic programming for controller development is implementing the controller on the system. Spacecraft are often limited in both data storage space and computational ability, making implementing a dynamic programming controller difficult. Therefore, choosing the proper implementation technique is critical for the feasibility and success of the controller. There are two general methods for implementing a dynamic programming controller: storing the optimal input as a function of the state directly, or storing the cost-to-go and performing a rollout algorithm.

The most intuitive way to implement a dynamic programming controller is to store the optimal input as a function of the state. The on-line controller would simply have to perform a lookup at every control update. To develop such a lookup table, a grid of states would have to be defined and the input associated with each state stored. This implementation works for both aggregation-based and approximation-based dynamic programming solutions.

![Figure 6. Direct Input Controller Architecture](image)

The basic implementation in this form requires storing every input at every sampled state. During a control update, the estimated state is used to find the stored states in the input table and their associated inputs. Those passed to the interpolator, which find the input based on the stored inputs. Finding the proper input could involve linear or cubic interpolation, or could be as simple as a nearest-point lookup. A block diagram of this type of architecture is seen in Figure 6. Notice that the lookup table is a function of the target state(s). This means that a different lookup table is required for different target states.
Figure 7. Rollout Controller Architecture

The other option is to implement a rollout controller. A rollout controller solves Bellman's equation at the current state using the stored cost-to-go as a heuristic. The advantage of this approach over a direct input method is twofold. First, rollout algorithms outperform the base heuristic used in the rollout.\(^{17}\) Second, a rollout controller requires only storing the cost-to-go for every sampled state, while a direct input controller requires storing every input at every sample state. If the system has multiple inputs, this can significantly increase the storage requirements. The downside of rollout controllers is that it requires more on-line computation than direct input controllers. A rollout controller has to compute the minimum of Bellman's equation, meaning it must predict the state at the next control cycle for a set of inputs. This requires propagating the system for every set of inputs, which can be computationally expensive. A block diagram of a rollout controller architecture is seen in Figure 7. In this architecture, a propagator takes the current state and finds new states as a function of control inputs by computing the forward ordinary differential equation. At every new point the cost table in interpolated or evaluated to find the cost-to-go as a function of control inputs. Finally, a solver finds the minimum of Bellman’s equation, where the argument of the minimum is the optimal input.

Both direct input controllers and rollout controllers can be combined with some form of data compression. Data compression reduces the amount of storage required for the lookup table at the expense of either performance or on-line computational costs. "Lossless" compression achieves the same performance as an uncompressed lookup table but increases the on-line computational requirements since the lookup table (or at least portions of the lookup table) needs to be decompressed every control cycle. "Lossy" compression decreases the required storage space at the expense of controller performance.

RESULTS
Aggregation Cost-To-Go

The state space examined for state aggregation is the convex hull of the points \([r \ \dot{r}] = [1.1 \pm 0.9 \ 0 \pm 0.4]\) where \(r\) is in meters and \(\dot{r}\) is in meters per second. The target box is the convex hull of \([r \ \dot{r}] = [1 \pm 0.01 \ 0 \pm 0.01]\). The maximum time \(\tau\) was set to 30 seconds. The algorithm used to develop the cost-to-go is seen in the appendix.
Figure 8. Aggregation Cost-To-Go

Figure 8 shows the simulation results of the dynamic program. The upper right region of cost that is greater than 30 is the "terminal velocity" region of the state space. The bottom left region of high cost is the region where no input will prevent the RINGS from crashing. Superimposed on the cost-to-go are sample trajectories where the RINGS start at rest (\( \dot{r} = 0 \)) and attempt to navigate to the set of target states.

Cost Approximation Cost-To-Go

The convex hull of the points \([r \quad \dot{r}] = [1.1 \pm 0.9 \quad 0 \pm 0.4] \) served as the set of initial conditions from which starting trajectories were seeded. As with the aggregation formulation, the target box is the convex hull of \([r \quad \dot{r}] = [1 \pm 0.01 \quad 0 \pm 0.01] \). Using an initial guess of a two-dimensional Taylor series expansion about the point \([r \quad \dot{r}] = [1 \quad 0] \), the cost approximation never converged on an optimal set of parameters. After several iterations, the matrix \( C_k \) would become singular. In addition to that problem, the Taylor series expansion allowed the cost function to have local minima not at the target set that would "trap" the simulation. After trying many different basis functions, one basis function proved fruitful. Let \( h(y) \) be a piecewise continuous function defined in Equation 16. Further, let \( ||\cdot|| \) represent the Euclidian norm and \( x_r \) be the component of the state corresponding to the separation distance \( r \). The three element basis function that converged is seen in Equation 17.

\[
\begin{align*}
  h(y) &= \begin{cases} 
    y, & y \geq 0 \\
    0, & y < 0
  \end{cases} \\
  \phi(x)^T &= [h(x_r - 1) \quad h(-x_r - 1) \quad ||x - [1,0]||]
\end{align*}
\]

(16)  

(17)
Figure 9. Cost Approximation Cost-To-Go

Figure 9 shows the results of the dynamic program. The algorithm used to develop the cost-to-go approximation is included in the appendix. The cost-to-go is plotted, with a set of sample trajectories used to compute the tuning vector $s^*$ superimposed on the plot. The cost-to-go properly reflects the diminishing control authority of the RINGS unit as the RINGS units move apart.

Figure 10. Cost-To-Go Comparison

Figure 10 shows the comparison between aggregation and cost approximation formulations for finding the cost-to-go. The performance metric is the average time to reach the set of target states given a set of initial states where the system is at rest ($\dot{r} = 0$). The results are presented based on the number of divisions per state since the performance of the aggregation algorithm is based on the resolution of the state aggregation. When the number of aggregate states is low, cost approximation outperforms aggregations. At approximately 43 divisions per state aggregation begins to
outperform cost approximation. It is important to note as the memory required to store the cost-to-go table increases exponentially with the number of divisions per state while cost approximation has a fixed storage size (which is typically much smaller than the storage requirements of the aggregation method). It is also significant to note that it may be possible to decrease the mean time to target of the cost approximation formulation by finding a basis function that better matches the actual cost to go. Also included in the plot is the optimal performance of a linear feedback law. The optimal linear feedback law was found by conducting a line search of feedback gains on the separation distance error and the separation distance rate error. As shown in Figure 10, the optimal linear feedback law performed poorly.

Controller Comparison

To compare controllers, the method used to generate the cost-to-go is fixed, with each controller developed based on a singular cost-to-go. For this paper, a cost-to-go based on state aggregation with 99 divisions per state was used.

Figure 11 shows the comparison of controllers. The rollout controller did not significantly outperform the direct input controller; at certain divisions per state the direct input outperformed the rollout algorithm. This is most likely associated with the fact that the cost-to-go as a function of the state is much less smooth than the optimal input as a function of the state. This means the linear interpolation better approximates the optimal input versus the optimal cost-to-go.

Although the rollout controller did not outperform the direct input controller, the rollout still has advantages that may make it more appropriate than a direct input controller. The amount of storage required for direct input storage scales linearly with the number of inputs, while rollout controllers have the same storage requirements regardless of the number of inputs. However, the online computational requirements scale exponentially with the number of inputs for a rollout controller.

CONCLUSION

The most significant result is that, for all permutations of Figure 2, dynamic programming produces a controller that drives the system to the set of target states. Both aggregation and cost
approximation produced a cost-to-go that generally increases a function of distance from the target, which intuitively makes sense. Cost approximation outperformed aggregation methods when a low number of aggregate states were used, but aggregation eventually outperformed cost approximation after approximately 43 divisions per state. Both the direct input and the rollout controller were able to control the RINGS system in simulation. The rollout algorithm did not significantly outperform the direct input method.

The axial RINGS case serves as a unique RINGS case that provides significant insight into the performance of dynamic programming as a control method. The axial case is useful because it only has two states, which makes visualization very easy. In higher dimensions controllers can only be examined by watching for convergence of the algorithms and simulating the results. Nevertheless, this case showed the advantages and disadvantages of aggregation and cost approximation when developing a controller. Using advanced methods for basis function development, such as function parameterization, may help aid in the basis function selection. Examining other cost approximation algorithms, like the least squares policy evaluation or the temporal difference algorithm, may prove fruitful when attempting cost approximation.

Another potential area for exploration is using a different cost function for the controller development. Instead of a shortest path problem, this could just as easily have been formulated as a regulator problem with discounted cost. The advantage of this type of formulation is that it has a better chance of keeping the system at the target. Right now, the system terminates when it reaches the target, but in reality the system would have to remain at the target, meaning overshoot of the physical system would become important.

ACKNOWLEDGMENTS

This work was conducted in conjunction with the RINGS program, which is led by Professor Raymond Sedwick of the University of Maryland with assistance from Professor David Miller of the MIT Space Systems Laboratory. The author would like to also thank Dustan Alinger, Allison Porter, and Alex Buck for their contributions to the RINGS program. The RINGS program is funded by the Defense Advanced Research Projects Agency (DARPA) as part of their INSPIRE program.
APPENDIX: COST-TO-GO ALGORITHMS

Algorithm 1 Aggregation Value Iteration Algorithm

Given $J_0(i)$

\begin{algorithm}
\begin{algorithmic}
\State $k = 0, ..., N$
\For{$i = 0, ..., n$}
\For{$m = 0, ..., p$}
\For{$s = 1, ..., q$}
\State $x_s^k \in D_i$
\State $x_s^{k+1} = f(x_s^k, u_m)$
\If{$x_s^{k+1} \notin C$}
\State $J_k(s) = \tau$
\Else
\State $J_k(s) = J(j)$, where $x_s^{k+1} \in D_j$
\EndIf
\EndFor
\State $J_k(m) = \frac{1}{p+1} \sum_{s=0}^q J_k(s)$
\EndFor
\State $J_{k+1}(i) = 1 + \min_{m \in (1, ..., p)} J_k(m)$
\EndFor
\end{algorithmic}
\end{algorithm}

Algorithm 2 Cost Approximation Iteration Algorithm

Given $r_0$

\begin{algorithm}
\begin{algorithmic}
\State $r_k = r_0$
\For{$k = 0, ..., n$}
\State $x_0 \in D_x$
\State $C^{-1} = [0]$
\State $d^{-1} = [0]$
\For{$i = 0$}
\While{$i < N$}
\State $u_i = \arg\min_{u \in U(i)} g(x_i^i, f(x_i^i, u^i)) + J(\phi(x_i^i, u^i))$
\State $x_i^{i+1} = f(x_i^i, u^i)$
\State $\delta_k = \frac{1}{k+1}$
\State $C_k^i = (1 - \delta_k)C_k^{i-1} + \delta_k \phi(x_i^i)(\phi(x_i^i) - \phi(x_i^{i+1}))$
\State $d_k^i = (1 - \delta_k)d_k^{i-1} + \delta_k \phi(x_i^i)g(x_i^i)$
\If{$x_i^{i+1} \in D_x$}
\State $x_i^{i+1} \in D_x$
\EndIf
\State $i = i + 1$
\EndWhile
\State $r_{k+1} = (C_k^N)^{n-1}d_k^N$
\EndFor
\end{algorithmic}
\end{algorithm}

REFERENCES

smith,[Washington, DC]: National Aeronautics and Space Administration; Pasadena, Calif.: Jet Propulsion Laboratory, California Institute of Technology,[1999](JPL publication; 99-3) 1 (1999).


