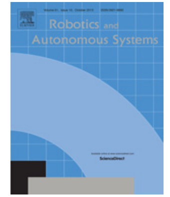
Robotics and Autonomous Systems 62 (2014) 214–228

Contents lists available at ScienceDirect

Robotics and Autonomous Systems

journal homepage: www.elsevier.com/locate/robot



A hierarchical approach for primitive-based motion planning and control of autonomous vehicles

David J. Grymin, Charles B. Neas, Mazen Farhood *

Department of Aerospace and Ocean Engineering, Virginia Tech, Blacksburg, VA 24061, United States

HIGHLIGHTS

- We examine graph-based search methods for motion planning using motion primitives.
- We develop a locally greedy algorithm and compare it to a version of Weighted A*.
- Greedy algorithm is advantageous when utilizing large motion primitive libraries.
- We develop a hybrid control technique for tracking concatenated motion primitives.
- The approach is applied to a hovercraft subject to disturbances and uncertainties.

ARTICLE INFO

Article history:
Received 4 August 2012
Received in revised form
29 June 2013
Accepted 3 October 2013
Available online 18 October 2013

Keywords:
Motion planning
Heuristic search
Hybrid control
Time-varying systems
Linear matrix inequalities

ABSTRACT

A hierarchical approach for motion planning and control of nonlinear systems operating in obstacle environments is presented. To reduce the computation time during the motion planning process, dynamically feasible trajectories are generated in real-time through concatenation of pre-specified motion primitives. The motion planning task is posed as a search over a directed graph, and the applicability of informed graph search techniques is investigated. Specifically, we develop a locally greedy algorithm with effective backtracking ability and compare this algorithm to weighted A* search. The greedy algorithm shows an advantage with respect to solution cost and computation time when larger motion primitive libraries that do not operate on a regular state lattice are utilized. Linearization of the nonlinear system equations about the motion primitive library results in a hybrid linear time-varying model, and an optimal control algorithm using the ℓ_2-induced norm as the performance measure is provided to ensure that the system tracks the desired trajectory. The ability of the resulting controller to closely track the trajectory obtained from the motion planner, despite various disturbances and uncertainties, is demonstrated through simulation.
© 2013 Elsevier B.V. All rights reserved.

1. Introduction

Motion planning for unmanned vehicles involves developing feasible trajectories through an obstacle field from a given initial state to a desired goal state; see, for instance, [1,2]. By using a discretized set of feasible motion primitives, the problem of finding a trajectory from the start to the goal becomes a graph search, a topic that has received a wealth of attention in the literature. This paper takes the approach of utilizing a set of pre-specified motion primitives, i.e. state and control histories defined over finite (or semi-infinite) time intervals, to generate, in real-time, collision-free trajectories from start to goal via graph search methods. As for the execution of the motion plan, the series of motion primitives generated by the planner will correspond to a sequence of predesigned subcontrollers to be applied consecutively.

The notion of constructing a solution from available trajectories is a common approach for vehicle motion planning. Prior methods have used online optimization to determine the trajectory [3], concatenating trim and maneuver trajectories to form a dynamically feasible path from the start state to the goal. In certain scenarios, the solution of such an optimization problem may require more computational effort than can be allotted to the planning task. Deterministic and sampling-based searches over graphs are two broad categories that have received considerable attention related to robot and vehicle trajectory planning in obstacle environments; a comprehensive review of motion planning with respect to unmanned aerial vehicles is given in [4].

Deterministic graph search algorithms use knowledge obtained during the search as well as prior knowledge of the environment to work towards an optimal solution. A heuristic, or rule of thumb, assists in determining the order of expansion during the search. For vehicle motion planning problems, the cost-to-goal is a commonly chosen heuristic. The A* algorithm, a complete and optimal algorithm, uses the path cost to reach each node as well as the future path cost estimate from the heuristic, and traverses the graph by

* Corresponding author. Tel.: +1 540 231 2983; fax: +1 540 231 9632.
E-mail address: farhood@vt.edu (M. Farhood).

0921-8890/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.robot.2013.10.003

expanding nodes with the lowest total path cost. For vehicle motion planning, the length of the path to reach a node can be used for path cost. In some applications, finding the optimal path may become burdensome, and a suboptimal solution is accepted to reduce the computational load. Weighted A* (WA*) relaxes optimality by weighting the heuristic in relation to the cost-to-go, effectively increasing the greediness of the algorithm, and is able to return solutions much faster with a bound on the suboptimal path cost [5,1]. Recent work related to A* based search methods has focused on iteratively improving suboptimal trajectories towards the minimum-cost path; see, for instance, the anytime search heuristic developed in [6,7]. Anytime search attempts to quickly return a feasible yet suboptimal path, and then improve upon this path successively in the time allotted for planning. In [7], for example, successive WA* searches are run with decreasing weight to achieve the best possible path in the given time for computation.

In sampling-based planners, such as the probabilistic roadmap (PRM) and rapidly-exploring random trees (RRTs), completeness is probabilistic; a solution will be returned, should one exist, with a probability converging to one as the number of samples tends towards infinity [8,9]. In practice, the RRT algorithm in particular is capable of returning a path to the goal fairly quickly, even in high-dimensional search spaces and subject to differential vehicle constraints. RRTs quickly examine unexplored regions of the state space, and are able to find paths through complicated obstacle fields with relative ease. The trade-off, however, is in the solution quality, as the path is often erratic due to the random sampling which drives the expansion. Additionally, Karaman and Frazzoli showed that the probability of the RRT algorithm converging to the optimal solution was zero. Their development of the RRT* algorithm, however, provides conditions for asymptotic optimality in addition to probabilistic completeness [10]. This algorithm has since been extended to an anytime framework, in which an initial solution is obtained quickly and then improved upon in the remaining time allotted [11].

The representation of the input and search spaces is also a factor in selecting the method to use. In [12], a discretized set of control inputs was used to compute a path for nonholonomic vehicles, with numerical integration performed during the planning process. Graph search was then utilized over a partitioning of the configuration space to determine a sequence of control inputs that brought the vehicle from its initial position to a goal region. A similar approach was taken by [13], with integration of control actions performed offline and stored for use with an online planner; solutions were obtained by performing a search over a tree. Pre-computed vehicle motions can also be developed that result in a grid-based representation of the configuration space, referred to as a state lattice. In this framework, the state lattice is represented as a directed graph, with vertices corresponding to specific reachable states of the vehicle and edges indicating the dynamically feasible motions which connect the states exactly. Such a representation is resolution complete, i.e. it is complete with respect to the resolution at which the lattice is generated [14,15].

It is important to note that when using pre-computed control input and state histories, the ability of the vehicle to track the resulting motion plan is subject to model accuracy. Unmodeled dynamics, parametric uncertainty, and exogenous disturbances may result in deviations from the original motion plan during execution. In the work of Burridge et al., a sequence of pre-computed feedback controllers is used to bring the system to a desired goal state in the presence of disturbances and obstacles in the robot workspace [16]. This framework has also been used for motion planning using controllers valid over regions of the free space; the vehicle is guided to the goal region by the sequence of controllers, with no path explicitly determined [17,18].

The approach in this paper utilizes a distinct set of motion primitives and entails performing a graph search to find an appropriate dynamically feasible trajectory through an obstacle environment. The set of motion primitives, hereafter referred to as a library, is developed offline. State and control histories for each motion primitive can be obtained through a variety of methods, for instance, by solving an optimization problem involving the nonlinear system equations or by recording human operator control inputs. The task of the motion planner is to then concatenate available motion primitives to find a trajectory from the initial state to the goal. This approach eliminates the need to solve for dynamically feasible state and control histories online. As far as the motion planner is concerned, any dynamically feasible motion primitive can be incorporated into the library. But, since these primitives can be generated experimentally, it is important that the primitive be within the state-space envelope where the derived mathematical model constitutes a reasonably accurate description of the vehicle dynamics. This requirement is imposed because the proposed control approach is model-based, as discussed later. We examine in this paper graph-based search techniques for motion planning, where the graph does not represent a state lattice but rather exhibits a tree structure, and the edges of the graph correspond to pre-specified motion primitives. Specifically, we develop a locally greedy algorithm with effective backtracking ability and compare it to a version of weighted A* based on a tree search. Both algorithms are applied in simulation to a hovercraft system and evaluated in environments composed of known, randomly constructed static obstacle fields. The greedy algorithm shows an advantage with respect to solution cost and computation time when relatively large motion primitive libraries with multiple velocities are utilized.

In addition, the paper provides a hybrid control approach, with the ℓ_2-induced norm as the performance measure, to ensure that the system tracks the desired trajectory generated by the motion planning algorithm despite various disturbances and uncertainties. The hybrid systems of interest in this paper are composed of linear time-varying (LTV) subsystems obtained from linearizing the nonlinear system equations describing the vehicle dynamics about the library of pre-specified primitives. The switching between these subsystems and ultimately their corresponding subcontrollers is dictated by the motion planning algorithm. The synthesis solution is provided in terms of a semidefinite program, and is based on the results of [19,20]. Related to this work is the paper [21] which provides a hybrid dynamics framework for the design of guaranteed safe switching regions using reachable sets. The paper [22] also gives a control algorithm for maneuver-based motion planning, which is robust to a certain class of perturbations.

The paper is organized as follows. Section 2 presents the deterministic search methods used in this paper. Section 3 provides a control result for hybrid LTV systems. Section 4 gives a detailed implementation of the motion planning and control methods on a hovercraft system. This paper serves as an extension of the results presented in the conference paper [23], and provides additional details regarding the implementation of the methodology. The intent of this paper, borrowing terminology from [4], is to provide a framework for *sound* motion planning, where the devised plan guarantees a collision-free trajectory despite possible disturbances, measurement errors, and other uncertainties.

The notation is mostly standard. We denote the set of real $n \times m$ matrices by $\mathbb{R}^{n\times m}$. The adjoint of an operator X is written X^*, and we use $X \prec 0$ to mean it is negative definite. The normed space of square summable vector-valued sequences is denoted by ℓ_2. It consists of elements $x = (x_0, x_1, x_2, \ldots)$, with each $x_k \in \mathbb{R}^{n_k}$ for some n_k, having a finite 2-norm $\|x\|_{\ell_2}$ defined by $\|x\|^2_{\ell_2} = \sum_{k=0}^{\infty} |x_k|^2 < \infty$, where $|x_k|^2 = x_k^* x_k$.

2. Motion planning with a primitive library

The search algorithms in this work make use of a library of pre-specified motion primitives. In [3], motion primitives are defined as state and control trajectories that encompass two classes: trim trajectories and maneuvers. The trim trajectories are composed solely of steady-state motions, whereas (transition) maneuvers are trajectories that begin and end at steady-state conditions. In general, a motion primitive can be any dynamically feasible trajectory, namely a state history and a corresponding control input that satisfy the nonlinear system equations over a finite or a semi-infinite time interval. For graph search purposes, the trim trajectories in our paper, i.e. state and control histories defined over semi-infinite time intervals, will be applied over a pre-specified number of time steps when building the search graph. With this stated, henceforth a distinction will not be made between trim and transition trajectories, and both will be referred to as motion primitives. There are two assumptions necessary for the approach presented. First, we assume that the motion primitives possess translational and rotational symmetry. This allows the motion planning algorithms to concatenate state and control histories during the search process. The second assumption is that the operational environment is bounded, i.e. the configuration space of the vehicle is reasonably constrained in size.

It is permissible to connect a primitive to another only when the final state of the first primitive coincides with (or at least is "very close" to) the initial state of the second one after performing the appropriate translation in position. For certain applications, it may also be necessary to ensure that the motion primitive control input histories are "compatible" across primitives, e.g. any rate limit placed on the input is respected in transitions between connectable primitives. The set of states reachable from the start node by some sequence of library primitives can be represented by a directed graph, where the vertices of the graph correspond to the reachable states and the edges indicate the motion primitives which connect the states. The expansion of nodes using this graph ultimately leads to a dynamically feasible solution path. In certain cases, where the primitives are carefully chosen, the graph may represent a regular state lattice, as given in [15], and consequently, the A* algorithm and its variants can be applied using a lattice-based search. However, for certain systems such as a six degree-of-freedom aircraft, we may want to incorporate a number of desired motion primitives into the library, which may render the task of developing a regular state lattice difficult. Specifically, requiring regularity in translational coordinates of lattice primitives will impose additional constraints on the boundary conditions when determining dynamically feasible state and control histories. Thus, it is worthwhile to study graph-based search techniques, where the graph does not represent a state lattice but rather exhibits a tree structure. For this reason, a version of WA* based on tree search is provided, in addition to an alternative algorithm where the expansion is driven by greedy behavior.

2.1. Weighted A search*

In A*, each node in the graph has three values associated with it: the cost-to-go, $g(n)$, the estimated cost-to-goal, $h(n)$, and the estimated total path cost, $f(n) = g(n) + h(n)$. Note that to retain the guarantee of returning a minimum cost path, the heuristic used in A* searches must be both admissible and consistent. Admissibility requires that the heuristic never overestimates the actual cost-to-goal. A heuristic is consistent if for any nodes n and n', $h(n) \leq h(n') + c(n, n')$, where $c(n, n')$ is the edge cost between n and n'. The search is initiated from the start node and any valid (collision-free) successors are added to the queue of open nodes, $\mathcal{O}$, and the start node is added to the closed set, $\mathcal{C}$. This addition of nodes is referred to as expansion. The search progresses by choosing the node in $\mathcal{O}$ with the lowest total path cost, $f(n)$, expanding this node and placing it in $\mathcal{C}$, and adding any valid successors to $\mathcal{O}$. This process continues iteratively until a node that reaches the goal state, or criteria, is found. A* returns the minimum cost path, should one exist, while expanding the fewest number of nodes necessary to do so [24,1]. A relaxation of A*, developed to reduce the computational effort in the search process, is weighted A*. The optimality requirement of A* is relaxed by computing the estimated total path cost as $f(n) = g(n) + (1 + \epsilon)h(n)$, where $\epsilon > 0$ is the weighting factor. The result of this weighting is goal-driven expansion, settling for suboptimal paths that expand towards the goal state faster [25]. The worst-case path returned by the algorithm is $(1 + \epsilon)f^*(n_{goal})$, where $f^*(n_{goal})$ is the cost of the minimum-cost path. Further information on WA* searches is contained in [26].

We now describe the WA* implementation used in this work. Given a library of N primitives, we define the set $P = \{1, 2, \ldots, N\}$, where each element in P refers to a specific primitive in the library. Let the transition between states be denoted as $x(n') = F(x(n), p)$, where $x(n) \in X$ and $p \in P$, with $X \subset \mathbb{R}^{n_x}$ denoting the set of states reachable from the start node by some sequence of library primitives. Let X_O denote the obstacle space, $X_F = X \setminus X_O$ the obstacle-free space, and $X_G \subset X$ the goal region which is basically a user-defined neighborhood about the goal state. The motion planning problem is then to find a sequence of primitives $p_i \in P$, where $i = 1, 2, \ldots, D$ for some positive integer D, that gives an obstacle-free path x from initial state $x(1) = x_{initial}$ to $x(D + 1) \in X_G$, where $x(i + 1) = F(x(i), p_i)$. The integer D will then denote the depth of the solution path in the tree. Expansion is governed during WA* search by the value of the total path cost at each node, $f(n)$. The child node of n, denoted as n', will have a path length $g(n') = g(n) + c(n, n')$, where $c(n, n')$ is the cost associated with the primitive to reach n' from n, determined during generation of the primitive library. During motion planning, the algorithm will build and maintain a tree $\mathcal{T} = (V, E)$, where V represents the vertices of the tree in X_F and E the directed edges between vertices; each directed edge corresponds to the motion primitive necessary to transition between the associated vertices.

There are several basic functions used during the operation of the algorithm, which we now describe briefly. The function `GetSuccesors`(n, p) applies the motion primitive $p \in P_c(n)$ to n to generate a candidate node n', where $P_c(n)$ is the set of primitives that are compatible with the state $x(n)$. The function `CollisionFree`(n, p) performs collision detection along primitive p starting from $x(n)$ and terminating at $x(n')$. If a collision is detected, then n' is discarded. Otherwise, the function returns n' as a valid candidate expansion node and also returns its heuristic cost $h(n')$. The implementation of collision detection used in this work considers only convex obstacle shapes, in particular rectangles defined by a base, height, and orientation angle. Obstacles are permitted to overlap, allowing for nonconvex regions of X_O. The collision detection as implemented is $O(N_c p_x n_{obs})$ in time complexity, where N_c is the number of compatible motion primitives, p_x the number of positions along each motion primitive that are checked for collision, and n_{obs} the number of rectangular obstacles. The function `InTree`$(n', \mathcal{T})$ searches over the nodes in the tree to find those nodes, denoted n_d and referred to as duplicates, which lie within an ellipsoid, $\mathcal{E}_d$, centered about $x(n')$. A similar approach was proposed by Barraquand and Latombe, however a pre-determined cell representation of the configuration space was utilized instead [12]. The choice of ellipsoid size is analogous to the choice of cell size when partitioning the configuration space; smaller $\mathcal{E}_d$ size values correspond to a finer resolution of the configuration space, and may therefore incur an increase in solution time due to a larger search tree size. The

Algorithm 1 Weighted A*

```
1: T ← AddNode(−, n_start, T, −)
2: O ← n_start, n ← n_start
3: while x(n) ∉ X_G or O ≠ ∅ do
4:     for p ∈ P_c(n) do
5:         n′ ← getSuccessors(n, p)
6:         if depth(n′) ≤ D_max then
7:             if CollisionFree(n, p) then
8:                 n_d ← InTree(n′, T, E_d)
9:                 if n_d ≠ ∅ then
10:                    if g(n_d) < g(n′) then
11:                        n′ not added to T
12:                    else
13:                        T ← AddNode(n, n′, T, p)
14:                        C ← Descendants(n_d)
15:                        O ← n′
16:                else
17:                    T ← AddNode(n, n′, T, p)
18:                    O ← n′
19:    C ← n
20:    n ← min(f(O))
21: return T
```

InTree function is $O(n_{\mathcal{T}} N_c n_{xc})$ in time complexity, where $n_{\mathcal{T}}$ is the number of nodes in the search tree, N_c the number of compatible motion primitives, and n_{xc} the number of dimensions considered for a node to be a duplicate. Descendants(n_d) traverses through the tree to find any nodes that were expanded from n_d and place them in the closed set if necessary. When a valid candidate is chosen for addition to the tree, the function AddNode($n, n', \mathcal{T}, p$) inserts node n' to the tree, creates an edge from n to n' storing the associated primitive p, and assigns a cost associated with this node. In WA* searches, $f(n') = g(n') + (1 + \epsilon)h(n')$, where $g(n') = g(n) + c(n, n')$ and $h(n')$ is the heuristic cost at the new node. The WA* algorithm is *resolution* complete with respect to the finite set of motion primitives utilized, as discussed in [15], contingent on the size of the ellipsoid chosen for duplicate node detection as well as the maximum allowed search depth; this will be clearly illustrated towards the end of the section.

The WA* search, given in Algorithm 1, operates as follows. The search tree $\mathcal{T}$ is initialized and n_{start}, the initial node, is added to the tree and also placed in $\mathcal{O}$ (Lines 1–2). While the state of the current node selected is not within the goal region, X_G, and the open list $\mathcal{O}$ is not empty, the algorithm will iterate over the while loop. The current node is set as n. Then, for any primitive $p \in P_c(n)$, the successor n' from n is obtained by applying the motion primitive p (Line 5). If the depth of the successor nodes is greater than the maximum allowed search depth, the successors will not be added to the tree (Line 6). The algorithm then checks to see if a collision occurs along the primitive p starting from state $x(n)$ (Line 7). If no collision occurs, the algorithm will proceed to check if there are nodes in the search tree that lie within some ellipsoid centered about the candidate successor (Line 8). If this is the case and so there are already "duplicate" nodes n_d in the search tree (Line 9), the cost-to-go of the candidate and duplicates are compared. If $g(n') > g(n_d)$, n' will not be added to the search tree (Lines 10–11). Conversely, if $g(n') < g(n_d)$, any descendants of n_d in the open set will be moved to the closed set $\mathcal{C}$ and no longer considered for expansion; in addition, n' will be added to the tree and placed in the open set $\mathcal{O}$ (Lines 13–15). If no nodes are returned during the check for duplicates, the candidate node is added to the tree and placed in the open set (Lines 17–18). The current node is then placed in the closed list, $\mathcal{C}$, as all successors have been expanded (Line 19). The algorithm proceeds by choosing the node in the open set with the lowest total path cost, $f(n)$, as the next node for expansion (Line 20). The algorithm terminates when it has expanded a node within the goal region (returns the tree) or there are no more nodes remaining in the open set (returns failure).

Remark 1. It is important to elaborate on Line 14 in Algorithm 1. The basic idea is that we want to get rid of duplicate nodes up to a certain tolerance; otherwise, the search may become exhaustive and in some cases even formidable. If we choose the tolerance to be very small, then the current node can probably inherit the descendants of the duplicate with higher cost-to-go without causing significant gaps or discontinuities in the returned trajectory. However, this may lead to scenarios where we have many nodes in the open queue that are "close" to each other with respect to their location within the configuration space, which would increase the computational cost of the algorithm. On the other hand, choosing the tolerance to be relatively large may result in trajectories with gaps, which could prove challenging to track in some agile vehicle applications. In our case, we judiciously choose the tolerance to reduce the number of nodes close to each other in the open set; in addition, to avoid gaps in the returned trajectory, we move all descendants of a duplicate node with a higher cost-to-go, along with the duplicate node itself, to the closed set. This issue does not arise when the graph represents a state lattice, in which case the duplicate nodes will match exactly, and so we simply update the parent node and the path cost of the descendants.

2.2. Greedy and impatient algorithm

A disadvantage of using WA* on a search tree employing the aforementioned approach occurs when updating the tree to reflect a lower cost path to within an ellipsoid of a particular node in the state space. As any descendants from the higher cost node are moved from the open list to the closed list (Algorithm 1, Line 14), computational effort expanding nodes along the higher cost path has essentially been wasted. For this reason, we examine the application of a greedy algorithm that uses a local priority queue and quickly abandons paths deemed unfit. The intent of the algorithm is to add fewer nodes into the search space, thus decreasing the number of nodes that must be examined for potential duplicates. We will henceforth refer to this algorithm as the greedy and impatient (GI) algorithm.

Expansion is governed by the incremental cost-to-go, $c(n, n')$, and the heuristic of potential successors, $h(n')$. The only nodes considered for addition to the tree are the potential successors to the current node, i.e. with a library of N total primitives, the algorithm will have at most N candidates when deciding which node to add to the tree. It should be noted that at each iteration, the GI algorithm adds a *single* node to the search tree; this is a subtle, yet significant, difference from WA* where all valid successors are added to the tree. The inclusion of the incremental cost-to-go allows certain primitives to be penalized, if desired, by adjusting the relative cost between primitives. For a truly greedy algorithm, however, this cost would be neglected. Greedy algorithms will commit to what appear to be promising paths early on in the search, a behavior that can be problematic in complex obstacle environments; the "impatience" of the algorithm is therefore added to overcome this possible downfall. Specifically, if the estimated heuristic cost of the best successor node, n', is greater than that of the current node, n, i.e. $h(n') > h(n)$, the algorithm will become impatient and backtrack to a "watch node". Backtracking to a watch node also occurs when a node has no valid successors, i.e. all primitives applied at this node lead to a collision with an obstacle. The concept of using watch nodes is adopted from [27,23], however the conditions for setting a watch node and the

Algorithm 2 Greedy & Impatient Algorithm

1: $\mathcal{T} \leftarrow \texttt{AddNode}(-, n_{start}, \mathcal{T}, -)$
2: $\mathcal{O} \leftarrow n_{start}, n \leftarrow n_{start}$
3: **while** $x(n) \notin X_G$ or $\mathcal{O} \neq \emptyset$ **do**
4: **for** $p \in P_c(n)$ **do**
5: $n_p \leftarrow \texttt{GetSuccessors}(n, p)$
6: **if** $\texttt{depth}(n_p) \leq D_{max}$ **then**
7: **if** $\texttt{CollisionFree}(n, n_p)$ **then**
8: $q(n) \leftarrow n_p$
9: **if** $q(n) \neq \emptyset$ **then**
10: $n' = \text{argmin}\{h(n_c) + c(n, n_c) \mid n_c \in q(n)\}$
11: $\mathcal{T} \leftarrow \texttt{AddNode}(n, n', \mathcal{T}, p)$
12: $\mathcal{O} \leftarrow n'$
13: $\text{watch}(n') \leftarrow \text{watch}(n)$
14: **if** $h(n') > h(n)$ **then**
15: $n \leftarrow \text{watch}(n)$
16: Continue while loop
17: **else**
18: $\Delta h(n') = h(n') - h(n)$
19: $\Delta h(n) = h(n) - h(n_{parent})$
20: **if** $\Delta h(n') < \Delta h(n)$ **then**
21: $\text{watch}(n') \leftarrow n$
22: $n \leftarrow n'$
23: **else**
24: $\mathcal{C} \leftarrow n$
25: **if** $n = n_{start}$ **then**
26: $n \leftarrow \min(f(\mathcal{O}))$
27: **else**
28: $n \leftarrow \text{watch}(n)$
29: **return** $\mathcal{T}$

overall algorithm structure differ in our case. The watch nodes pinpoint where along the current search path specific greedy behavior has occurred. Namely, a node n is a watch node if $h(n') - h(n) < h(n) - h(n_{parent})$, where n' and n_{parent} are the successor and predecessor (parent) of n, respectively. In a physical sense, a watch node corresponds to a location in the configuration space along the current path where the algorithm selects a node for expansion that decreases the heuristic by an amount more than the decrease in heuristic resulting from the preceding motion primitive. Since there may exist other successors from the watch node that also lead to a decrease in the heuristic function, the algorithm indicates that the watch node should be re-examined if backtracking occurs later in the search along the current path.

The notation and the functions used in the algorithm operation are the same as those used for WA*, with a few notable exceptions. If a candidate node, n', is found to be a duplicate of a node, say n_d, already within the search tree, then n' will be discarded if $g(n') > g(n_d)$. The reason is that n_d corresponds to a location in the configuration space already in the search tree. Permitting n' to be added to the tree may result in repeatedly visiting an area of the configuration space, and consequently unbounded growth of the search tree. Specifically, when nodes are checked for collision in the GI algorithm, the `InTree` check is also performed. If a potential successor is found to be a duplicate of a node already existing in the search tree and has a higher cost-to-go than the node already in the tree, then this potential successor will be treated the same as if the associated primitive led to a collision with an obstacle.

The GI algorithm is given in Algorithm 2. The algorithm begins by creating the search tree, $\mathcal{T}$, adding the start node, n_{start}, to the tree and placing it in the open set, and setting the start node as the current node (Lines 1–2). As in the WA* implementation, while the current node is not within the goal region and there are nodes remaining in the open set, the algorithm iterates over a loop (Line 3). For any primitive $p \in P_c(n)$, the successor n_p is obtained and its depth in the tree is checked to ensure that it does not exceed D_{max} (Lines 4–6). The successor n_p, along with the path from n to n_p, is then checked for collision with obstacles. If the path is collision free, n_p is added to the local priority queue, denoted $q(n)$ (Lines 7–8). If $q(n)$ is nonempty, i.e. node n has collision-free successors, the algorithm chooses the successor, denoted n', with the minimum sum of heuristic and incremental cost (Lines 9–10). The successor n' is then added to the tree and placed in the open list (Lines 11–12). The watch node of n is also set to be the watch node of n'. The algorithm then compares the heuristic of n' to that of n. If $h(n') > h(n)$, the algorithm backtracks to the watch node associated with n (Lines 14–16). Otherwise, the change in heuristic value between the current node, n, and the added successor n' and between the parent of the current node, n_{parent}, and n are computed (Lines 18–19). If $h(n') - h(n) < h(n) - h(n_{parent})$, the watch node of n' becomes n (Lines 20–21). n' then becomes the current node, and the iteration continues (Line 22). If a node has no successors in $q(n)$ due to collisions with obstacles (or running into duplicates), or all compatible primitives have been expanded unsuccessfully, the algorithm moves n to the closed set $\mathcal{C}$ (Line 24). If n is the start node, the node in the open set with the lowest total path cost is chosen for expansion (Line 26); otherwise, the algorithm backtracks to the watch node of n (Line 28). The algorithm terminates when no nodes remain in the open set, or a node is expanded in the goal region, X_G, in which case the search tree, $\mathcal{T}$, is returned.

Before proceeding, it is necessary to provide some commentary regarding the GI algorithm. As stated in the algorithm, a node is only removed from the open set after all primitives from the node have been expanded; it is permissible to therefore revisit any node a finite number of times, equal to the number of compatible motion primitives at the node. To avoid redundant computations, each node has a list associated with it that indicates which primitives have been expanded previously in the search process. The heuristic cost of successors, which will also indicate collision with an obstacle, is stored for each open node as well. Thus when revisiting a node during the search, the algorithm only needs to check whether the remaining available successors are potential duplicate nodes. Though in its worst-case behavior the algorithm resorts to a brute force search over all possible primitives at each reachable node, backtracking to the start node can occur quickly during the search process. This happens, for example, when the vehicle is initially oriented away from the goal state and all primitives lead to an increase in heuristic value. In this scenario, each primitive will initially trigger the backtracking condition and the algorithm will then expand the higher heuristic cost nodes until a path with a decreasing heuristic is possible. The algorithm therefore allows for the heuristic to increase along the resulting path, but only in the event that the algorithm finds it necessary to do so. It should be noted that although the GI algorithm resorts to a WA* approach when returning to the start node via backtracking, there is no theoretically established upper bound on the path cost in this case.

Theorem 1. *If X_G is reachable from $x(1)$ using a finite sequence of motion primitives, then both algorithms will find a path from the start node provided that $\mathcal{E}_d$ is set small enough, and D_{max} large enough.*

Proof. The proof follows from that for Claim 1 in Barraquand and Latombe [12]. Let $\mathcal{T}_\infty \in X_F$ be the infinite tree obtained by recursively applying all motion primitives from the start node that generate collision free successors. Thus $\mathcal{T}_\infty$ denotes all reachable state space configurations in the obstacle-free space. Since the goal region, X_G, is reachable from the initial state, we denote $x(D+1) \in X_G$ as the reachable state within the goal region at depth D in $\mathcal{T}_\infty$. Let $\mathcal{T}_{D_{max}}$ then be a finite subtree of $\mathcal{T}_\infty$, with depth $D_{max} \geq D$. This implies that $\mathcal{T}_{D_{max}}$ contains a collision-free path from the start node

to the goal region. By choosing $\mathcal{E}_d$ small enough, it can be ensured that for any node, n, along the path from $x(1)$ to $x(D+1)$, all other nodes with state space locations within $\mathcal{E}_d$ of $x(n)$ have a cost-to-go which is greater than the cost-to-go of node n. Thus no nodes along the solution path will be removed from $\mathcal{T}_{D_{max}}$. The algorithms will then be guaranteed to find a path connecting the start node to a node within the goal region, X_G. Further, since the search tree is finite, the search will terminate in finite time. ■

As in [12], the above proof is not constructive, and thus there are several implications of Theorem 1. When using a motion primitive library constructed over a state lattice, the size of $\mathcal{E}_d$ can be set arbitrarily small, since all primitives will reach the state space location of other nodes in the tree exactly. Further, if the search space is confined to a finite region of the configuration space, the maximum depth, D_{max}, can be set arbitrarily high. By admitting only nodes with a lower cost-to-go at a location in the state space, the algorithms prevent the tree from infinite growth (repeatedly visiting areas of the state space).

For motion primitive libraries that are not generated over a state lattice, the choice of $\mathcal{E}_d$ and D_{max} may have more severe consequences. Specifically, if the algorithm terminates and returns a failure to find a path to the goal, either the size of $\mathcal{E}_d$ was too large, D_{max} was too small, or there is not a solution to the problem. Consider, for example, a set of obstacles through which there is a unique path from the start to goal, defined by the sequence of primitives $\{p_1, \ldots, p_{n-1}, \ldots, p_D\}$, where $x(1)$ is the start node, $x(n)$ some point in the state space along the path, and $x(D+1) \in X_G$. Let $\{p_1, \ldots, p_{m-1}\}$ be the sequence of motion primitives for some other subpath to node m, where $x(n)$ lies within an ellipsoid $\mathcal{E}_d$ centered about $x(m)$. Further, let $g(m) < g(n)$, i.e. the cost-to-go to reach $x(m)$ is less than the cost-to-go for $x(n)$. In this scenario the node n and its descendants (the path to the goal region) would be discarded, and the node m retained instead. Since the path passing through n is the only path from the start node to the goal region, there is no path from m to the goal. Thus the planners would return that there is no solution when clearly this is not the case. Some smaller choice of $\mathcal{E}_d$, however, would retain node n in the search tree.

Remark 2. Unlike best-first searches such as A* and its variants, greedy search does not provide an upper bound on the total path cost in general [24,5,28,29]. Greedy search does, however, have several advantages over best-first searches in some problem domains. Due to the fact that all successors of a node are placed in the search tree, best-first search techniques may require substantial amounts of memory. The set of open nodes must be searched and sorted to determine the next node for expansion. On the other hand, in locally greedy search, there is only one active node and a potentially shorter list to sort when choosing the next node in the iteration. Thus, the computation required to choose the next node for expansion at each iteration scales linearly with the number of successors available (which is at most the size of the primitive library in our case).

3. Hybrid LTV control

As aforementioned, the desired state and corresponding control trajectories will be generated in real-time using a library of pre-specified motion primitives. This section gives a systematic optimal algorithm, using the ℓ_2-induced norm as the performance measure, for the control of nonlinear systems about such trajectories.

Consider the problem of designing a feedback controller to some vehicle. The closed-loop system is shown in Fig. 1, where w and z denote the exogenous disturbances and errors to be controlled, respectively, and y denotes the measurements and u the control input. This closed-loop system can be viewed as a map from w to z, denoted by $w \mapsto z$. As z represents the errors caused by the disturbances, we would like to design a controller that would minimize the effect of the disturbances w on z. In other words, we would like a controller that would make the map $w \mapsto z$ "small" according to some measure. One popular performance measure is the ℓ_2-induced norm, defined by

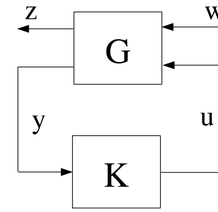


Fig. 1. Closed-loop system, where G is the plant and K the feedback controller, with w, z, y, and u denoting the exogenous disturbances, errors to be controlled, measurements, and applied control, respectively.

$$\|w \to z\|_{\ell_2 \to \ell_2} = \sup_{\|w\|_{\ell_2} \neq 0} \frac{\|z\|_{\ell_2}}{\|w\|_{\ell_2}}.$$

The basic control philosophy in this case is to treat the worst case scenario disturbances. This is a very sound approach when we do not know what we are up against: plan for the worst and optimize [30]. This so-called H_∞ approach addresses the questions of modeling and disturbance uncertainties, and it is in fact the most pursued approach for robust control ever since the formal framework was first formulated in [31].

Given a nonlinear vehicular system and an associated library of primitives, linearizing the nonlinear system equations about some pre-specified primitive, labeled i, from this library and then discretizing using zero-order hold sampling result in general in an LTV discrete-time system, say $G^{(i)}$, defined by the following state-space equation:

$$\begin{bmatrix} x_{k+1} \\ z_k \\ y_k \end{bmatrix} = \begin{bmatrix} A_k^{(i)} & B_{1k}^{(i)} & B_{2k}^{(i)} \\ C_{1k}^{(i)} & D_{11k}^{(i)} & D_{12k}^{(i)} \\ C_{2k}^{(i)} & D_{21k}^{(i)} & 0 \end{bmatrix} \begin{bmatrix} x_k \\ w_k \\ u_k \end{bmatrix}, \qquad x_0 = 0,$$

for $w \in \ell_2$, where x_k is the value of the state vector at discrete time k. The vectors x_k, z_k, w_k, y_k, and u_k are real and may have time-varying dimensions which we denote by n_{xk}, n_{zk}, n_{wk}, n_{yk}, and n_{uk}, respectively. Given our problem setup, $G^{(i)}$ may be a finite-horizon, a periodic, or an eventually periodic system depending on the primitive in question. An eventually periodic system arises when linearizing the nonlinear system equations about an eventually periodic trajectory. Such a trajectory can be arbitrary for an initial amount of time, but then settles into a periodic orbit; a special case of this is when a system transitions between two operating points. Finite horizon and periodic systems are subclasses of eventually periodic systems, and so, we will assume without loss of generality that $G^{(i)}$ is an eventually periodic system, specifically an (h_i, q_i)-eventually periodic system as defined next.

Definition 1. An LTV system $G^{(i)}$ is (h_i, q_i)-eventually periodic for some integers $h_i \geq 0$, $q_i \geq 1$ if each of its state-space matrix sequences is (h_i, q_i)-eventually periodic; for instance, the sequence $A_k^{(i)}$ would be of the form

$$\underbrace{A_0^{(i)}, \ldots, A_{h_i-1}^{(i)}}_{h_i \text{ terms}}, \underbrace{A_{h_i}^{(i)}, \ldots, A_{h_i+q_i-1}^{(i)}}_{q_i \text{ terms}}, \underbrace{A_{h_i}^{(i)}, \ldots, A_{h_i+q_i-1}^{(i)}}_{q_i \text{ terms}}, \ldots.$$

Periodic systems correspond to the case where the finite horizon length h_i is zero, whereas finite horizon systems are $(h_i, 1)$-eventually periodic with the periodic portions of the state space

sequences set to zeros. Linearizing the nonlinear system equations about all the pre-specified primitives in the library results in a hybrid system G composed of eventually periodic subsystems $G^{(i)}$ for $i = 1, 2, \ldots, N$, where N is the number of library primitives. Note that we refer to $G^{(i)}$ as an eventually periodic model in general, even though, depending on the corresponding primitive, it may actually be a linear time-invariant (i.e., periodic with period length equal to one), a periodic, or a finite horizon model. This is acceptable because, as mentioned before, all these models are special cases of the eventually periodic model, and the following synthesis result can be appropriately adjusted to reflect the different types of models comprising the hybrid system.

Suppose that plant G is controlled by a feedback hybrid controller $K = \{K^{(1)}, \ldots, K^{(N)}\}$, where $K^{(i)}$ is (h_i, q_i)-eventually periodic and defined by the state-space equation:

$$\begin{bmatrix} \hat{x}_{k+1} \\ u_k \end{bmatrix} = \begin{bmatrix} \hat{A}_k^{(i)} & \hat{B}_k^{(i)} \\ \hat{C}_k^{(i)} & \hat{D}_k^{(i)} \end{bmatrix} \begin{bmatrix} \hat{x}_k \\ y_k \end{bmatrix}, \qquad \hat{x}_0 = 0,$$

with $\hat{x}_k \in \mathbb{R}^{p_k}$. Note that subcontroller $K^{(i)}$ corresponds to subsystem $G^{(i)}$, and hence is associated with primitive i. It is always possible to apply the standard ℓ_2-induced norm control approach [19,20] to design a subcontroller $K^{(i)}$ for subsystem $G^{(i)}$. However, such an approach will not guarantee stability, let alone performance, across switching boundaries. To elaborate, as the motion plan is in the form of a reference concatenated primitive trajectory, implementing this plan boils down to executing consecutively a series of subcontrollers associated with the sequence of primitives composing the reference trajectory. If we are to apply the standard ℓ_2-induced norm control approach to design the subcontrollers *separately*, then there are no theoretically established guarantees that the closed-loop system will exhibit a stable behavior as we switch between subcontrollers. The hybrid control approach proposed herein also uses the ℓ_2-induced norm as the performance measure, but it incorporates all possible connections between compatible library primitives into the control design. Specifically, in addition to the standard synthesis conditions associated with each subsystem $G^{(i)}$, we include coupling conditions which reflect the possible connections between compatible primitives. As a result, this approach comes with stability and performance guarantees across switching boundaries. But, since including the coupling conditions necessitates incorporating the standard synthesis conditions for all the subsystems simultaneously into the control synthesis program, the hybrid approach can become computationally intensive in the case of large primitive libraries.

The next definition expresses precisely our synthesis goal.

Definition 2. A feedback controller $K = \{K^{(i)}\}_{i=1}^N$ is a γ-admissible synthesis for hybrid plant G if the closed-loop system in Fig. 1 is asymptotically stable and the performance inequality $||w \to z||_{\ell_2 \to \ell_2} < \gamma$ is achieved.

The following definitions will be convenient:

$$\mathcal{F}_1(R_k, R_{k+1}, \gamma, k) = J_k^* R_k J_k - V_{1k}^* R_{k+1} V_{1k} + M_k^* M_k - \gamma^2 V_{2k}^* V_{2k},$$

and

$$\mathcal{F}_2(S_k, S_{k+1}, \gamma, k) = \begin{bmatrix} W_k^* S_{k+1} W_k - U_{1k}^* S_k U_{1k} - U_{2k}^* U_{2k} & L_k^* \\ L_k & -\gamma^2 I \end{bmatrix},$$

where V_{1k}, V_{2k}, U_{1k}, and U_{2k} are defined by the following equalities:

$$\operatorname{Im}\begin{bmatrix} V_{1k}^* & V_{2k}^* \end{bmatrix}^* = \operatorname{Ker}\begin{bmatrix} B_{2k}^* & D_{12k}^* \end{bmatrix},$$

$$\begin{bmatrix} V_{1k}^* & V_{2k}^* \end{bmatrix}\begin{bmatrix} V_{1k}^* & V_{2k}^* \end{bmatrix}^* = I,$$

$$\operatorname{Im}\begin{bmatrix} U_{1k}^* & U_{2k}^* \end{bmatrix}^* = \operatorname{Ker}\begin{bmatrix} C_{2k} & D_{21k} \end{bmatrix},$$

$$\begin{bmatrix} U_{1k}^* & U_{2k}^* \end{bmatrix}\begin{bmatrix} U_{1k}^* & U_{2k}^* \end{bmatrix}^* = I,$$

and $J_k = A_k^* V_{1k} + C_{1k}^* V_{2k}$, $M_k = B_{1k}^* V_{1k} + D_{11k}^* V_{2k}$, $W_k = A_k U_{1k} + B_{1k} U_{2k}$, $L_k = C_{1k} U_{1k} + D_{11k} U_{2k}$. We also define the set $\mathcal{B}_i$ to consist of all the library primitives j that can succeed primitive i.

Theorem 2. *Consider a hybrid system $G = \{G^{(i)}\}_{i=1}^N$, where each subsystem $G^{(i)}$ is (h_i, q_i)-eventually periodic. Then, there exists a γ-admissible hybrid synthesis $K = \{K^{(i)}\}_{i=1}^N$ to G, where $K^{(i)}$ is also (h_i, q_i)-eventually periodic, if, for $i = 1, 2, \ldots, N$ and $k = 0, 1, \ldots, h_i + q_i - 1$, there exist positive definite matrices $R_k^{(i)}$ and $S_k^{(i)}$ such that*

$$\mathcal{F}_1^{(i)}\left(R_k^{(i)}, R_{k+1}^{(i)}, \gamma, k\right) \prec 0, \qquad \mathcal{F}_2^{(i)}\left(S_k^{(i)}, S_{k+1}^{(i)}, \gamma, k\right) \prec 0,$$

$$\begin{bmatrix} R_k^{(i)} & I \\ I & S_k^{(i)} \end{bmatrix} \succeq 0, \qquad (1)$$

with $R_{h_i+q_i}^{(i)} = R_{h_i}^{(i)}$, $S_{h_i+q_i}^{(i)} = S_{h_i}^{(i)}$, and, for all $j \in \mathcal{B}_i$,

$$R_{h_i}^{(i)} = R_0^{(j)}, \qquad S_{h_i}^{(i)} = S_0^{(j)}. \qquad (2)$$

The notation $\mathcal{F}_a^{(b)}$ is as defined previously with the superscript b indicating that the state-space data used correspond to subsystem $G^{(b)}$.

Note that if a library primitive exhibits a periodic behavior, it may be desirable to connect to compatible successive primitives at various instants in the period, as opposed to just the beginning of the period as depicted in conditions (2). In such a case, we rewrite (2) as

$$R_{h_i+c}^{(i)} = R_0^{(j)}, \qquad S_{h_i+c}^{(i)} = S_0^{(j)},$$

where $0 \le c < q_i$ specifies these instants.

Proof. Say the sequence of library primitives to be traversed is $m_1, m_2, \ldots, m_s$, where the time-intervals in which the system executes these primitives are $[0, k_1]$, $[k_1 + 1, k_2], \ldots, [k_{s-1}, \infty[$, respectively. Note that the length of these intervals may be longer than the sum of the finite horizon length and period of the associated primitives since the system may traverse the periodic parts more than once. Define the matrix sequences $R_k \equiv R_t^{(m_i)}$ and $S_k \equiv S_t^{(m_i)}$ for $i = 1, \ldots, s$, $k = k_{i-1} + 1, \ldots, k_i$, with $k_0 = -1$, $k_s = \infty$, and t defined as follows:

$$t = \begin{cases} \hat{t} = k - k_{i-1} - 1 & \text{for } \hat{t} < h_{m_i} + q_{m_i} \\ h_{m_i} + \left((\hat{t} - h_{m_i}) \bmod q_{m_i}\right) & \text{for } \hat{t} \ge h_{m_i} + q_{m_i}. \end{cases}$$

These matrix sequences solve the synthesis conditions for the eventually periodic system $\bar{G}$, whose A-matrix, for example, is defined as $\bar{A}_k \equiv A_t^{(m_i)}$ for i, k, t as aforementioned. Invoking [20, Corollary 13] completes the proof. ■

The matrices $R_k^{(i)}$ and $S_k^{(i)}$ are obtained by solving a semidefinite feasibility problem (1)–(2). Customized primal–dual interior point methods are typically used to solve such semidefinite programs and are known to outperform the barrier method [32]. The reader is referred to [33] for a collection of performance tests on various semidefinite program solvers such as SeDuMi [34] and SDPT3 [35]. The solutions $R_k^{(i)}$ and $S_k^{(i)}$ can then be used to construct the subcontroller $K^{(i)}$ offline, as shown in [19]. Note that the procedure given in [19] for constructing admissible LTV controllers is a generalization of the method developed in [36] for the LTI case. The work in [19] considers admissible syntheses that stabilize the closed-loop system and guarantee that $\|w \mapsto z\|_{\ell_2 \to \ell_2} < 1$. The assumption that $\gamma = 1$ is made for simplicity and without any loss of generality since this γ can always be absorbed into $G^{(i)}$. To apply the procedure of [19], note that a γ-admissible synthesis for $G^{(i)}$ is a 1-admissible synthesis for $\bar{G}^{(i)}$, where $\bar{G}^{(i)}$ has the same system realization as $G^{(i)}$ except that $\bar{C}_{1k}^{(i)} = \frac{1}{\gamma} C_{1k}^{(i)}$, $\bar{D}_{11k}^{(i)} = \frac{1}{\gamma} D_{11k}^{(i)}$, and $\bar{D}_{12k}^{(i)} = \frac{1}{\gamma} D_{12k}^{(i)}$.

Remark 3. The synthesis solution in Theorem 2 is provided in terms of a semidefinite program with coupled temporal constraints associated with the pre-specified primitives as well as coupling conditions across switching boundaries prescribed by a set of rules (these rules specify the acceptable ways to connect the primitives). For the synthesis conditions to have a solution, it is necessary that any linear time-invariant, periodic, or eventually periodic constituent models of the hybrid system be stabilizable and detectable. Such requirements are not necessary for finite-horizon models (linearized about transition maneuvers) in general unless the transition maneuver can be connected to itself. As for implementation, the motion planning algorithm issues a policy which is in the form of a sequence of primitives composing a reference trajectory that leads to some desired goal state. The controller executes this policy; namely, the series of primitives corresponds to a sequence of subcontrollers to be applied consecutively. In other words, the motion planning algorithm determines the scheduling scheme of the subcontrollers constituting the hybrid control system.

As implied from the proof of Theorem 2, if we consider a concatenated primitive trajectory, linearize the nonlinear system equations about this trajectory, and then formulate the standard ℓ_2 induced norm synthesis problem for the resulting LTV model, the hybrid synthesis solutions can be used to construct feasible solutions to this LTV synthesis program. In other words, the sequence of subcontrollers of the hybrid control system corresponding to the sequence of primitives composing the reference trajectory constitutes a stabilizing (with γ-level performance) controller to the concatenated primitive trajectory.

4. Four-thruster hovercraft example

4.1. Motion planning

Three motion primitive libraries are developed to demonstrate the hierarchical process presented. These libraries are then used to comparatively evaluate the WA* and GI algorithms in finding suitable trajectories through environments with randomly placed obstacles. The four-thruster hovercraft model is shown in Fig. 2, and the equations of motion of the hovercraft are given below:

$$m\ddot{x} + b_t\dot{x} = (u_1 - u_3)\cos\theta + (u_2 - u_4)\sin\theta, \quad (3)$$

$$m\ddot{y} + b_t\dot{y} = (u_1 - u_3)\sin\theta + (u_4 - u_2)\cos\theta, \quad (4)$$

$$J\ddot{\theta} + b_r\dot{\theta} = L(u_1 - u_2 + u_3 - u_4). \quad (5)$$

The translational and rotational data for this system are as follows: mass $m = 1.731$ kg, half-radius $L = 0.15$ m, translational friction $b_t = 3.7 \times 10^{-3}$ N s/m, rotational friction $b_r = 3.65 \times 10^{-4}$ N s m/rad, and moment of inertia $J = 2.363 \times 10^{-2}$ kg m^2. Each thruster can exert a force of magnitude at most 3 N.

The libraries of motion primitives are composed of both steady-state and transition trajectories, similar to [3], with trim trajectories applied over a finite time horizon when building the search graph. For the hovercraft, the steady-state motion is forward in the body frame, where forward is defined along the $\theta = 0$ direction. The first library contains two sets of five primitives, each corresponding to final heading angle changes of [90, 45, 0, −45, −90] degrees. That is, for each change of the final heading angle, there are two primitives of different lengths. The initial and final velocities of the transition primitives are all equal to 1 m/s in the direction of the vehicle heading, and the steady-state primitives also have a constant velocity of 1 m/s. Given such a velocity setup, we say the library operates at a velocity of 1 m/s. The library primitives are judiciously chosen to form a state lattice with sampling of 0.5 m in x and y displacement. The second library has 5 primitives and is similar to the first in primitive types and velocity, however the

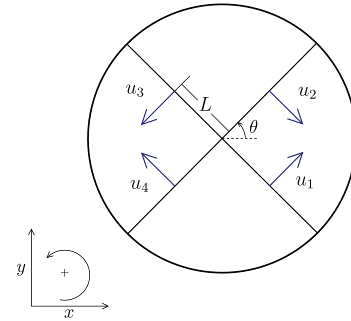


Fig. 2. Four-thruster hovercraft.

primitives do not form a state lattice. The final library considered contains primitives with the same final heading angles as the first library and can operate at velocities of 0.5 m/s, 1 m/s, and 2 m/s. This library also contains acceleration and deceleration primitives at constant heading angle that switch between the aforementioned velocities. There are 19 total primitives in this library, and clearly not all of them are compatible with each other.

Determining dynamically feasible state and control input histories for each primitive can be accomplished through solution of an optimal control problem [37]. Pseudospectral optimal control software tools such as DIDO [38] and GPOPS [39] specifically address the solution of such problems and have been applied to a variety of systems including autonomous ground, aerial, and surface vehicles as well as spacecraft; a review of applications of pseudospectral optimal control can be found in [40]. For the four-thruster hovercraft, the task of obtaining the state and control histories for each primitive is formulated as a convex optimization program, and then solved using the modeling system CVX [41], along with the solver SDPT3 [35]. Specifically, the constraints of this program consist of the equations of motion which are discretized using zero-order hold sampling with sampling time $T = 0.05$ s, initial and final conditions on the state, a bound of 2.5 N on the magnitudes of the control inputs, a rate limit of 20 N/s on the control input histories, and affine equations relating positions to velocities based on Simpson's 1/3 rule. Note that as the equations of motion are nonlinear in the angular displacement θ, we first estimate an admissible θ-trajectory, and then, using this trajectory the equations of motion become affine in the program variables. The objective function is a weighted sum of a quadratic smoothing function of the state and control histories and a quadratic penalty function on the control inputs.

Fig. 3 shows a graph representing the state lattice associated with the first library. As mentioned before, the first library consists of two sets of primitives. The first set, shown in black in Fig. 3, can be applied to nodes with a heading angle equal to a multiple of 90°. The second set can be applied to nodes with a heading angle equal to an odd multiple of 45° and is shown in red in Fig. 3. Fig. 4 shows the motion primitives operating at 0.5 m/s, 1 m/s, and 2 m/s. Note that the 1 m/s primitives compose the second library. In the first library, the motion primitives with a constant heading angle are applied for 12 or 14 time steps depending on the heading of the vehicle, where a time step is equal to 0.05 s; all other motion primitives in this library are applied for 18 time steps. All primitives in the second and third libraries are applied for 12 time steps, except for the 90 degree turns at 2 m/s, which are applied for 18 time steps. The reference control inputs for a 90 degree turn at 1 m/s are shown in Fig. 5.

Initially, the vehicle is at the origin (0, 0) with a heading angle of 45° and a velocity of 1 m/s when using the first or second libraries and 0.5 m/s when using the third library. The goal region is defined by a Euclidean ball of radius 0.3 m, centered about the point $(x, y) = (12, 12)$. The performance of the GI algorithm is compared

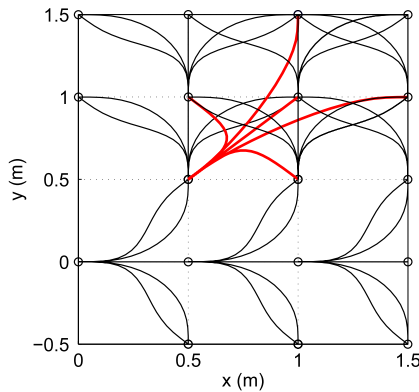


Fig. 3. State Lattice Motion Primitives. (For interpretation of the references to colour in the text, the reader is referred to the web version of this article.)

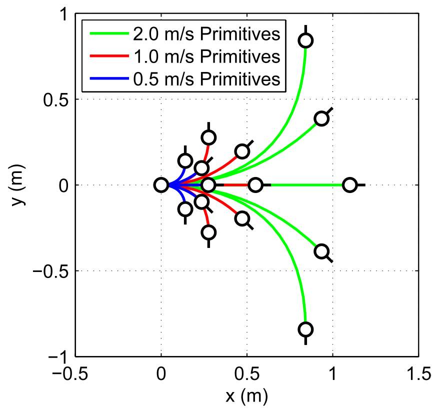


Fig. 4. Motion Primitives at 0.5, 1, 2 m/s. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

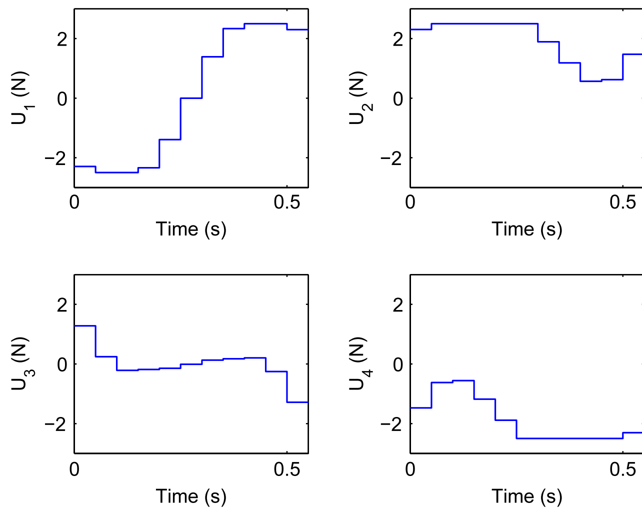


Fig. 5. 90° turn control input histories.

with that of WA*. The implementation of the GI algorithm will not make use of the cost-to-go, $c(n, n')$, and will be governed solely by the heuristic, $h(n)$. In the WA* case, three values for the weighting of the heuristic are used, namely $\epsilon = 0.5, 1.5, 4$. The heuristic used for both algorithms is the Euclidean distance from the position of the current node in the state space to that of the goal node. More complex heuristics, which take into account obstacle edges that intersect the ray projecting from the current node to the goal node, can be used. Based on the few attempts we carried out, however, these heuristics tend to result in an increased computation time over the Euclidean distance heuristic in our example, with a negligible improvement in the path cost. Note that the lengths of the primitives composing the third library are carefully chosen so that the vehicle can maneuver through dense obstacle areas using the lower velocity primitives and speed up in sparse regions where the higher velocity primitives do not lead to collisions.

All computations are carried out in MATLAB on a Dell Desktop with a 3.07 GHz quad-core Intel Xeon CPU and 6 GB of RAM running Windows 7. The performance measures used are the returned path cost and the wall clock time. It is probably better to use the number of floating point operations (FLOPS) executed as a performance measure rather than the wall clock time, where the FLOP counts can be obtained using, for instance, the Lightspeed MATLAB Toolbox [42]. However, we have noticed that the FLOP results are conformable with the wall clock time ones, and so we opt to use the latter for simplicity. A total of 2000 tests are run in environments with random obstacle placement and orientation. In 1000 of the tests, each of the obstacles has a 1 m × 1 m square shape, while in the remaining tests obstacles are restricted to be rectangular in shape with the length of the edges varying from 2 m to 4 m. Although all obstacles are thus assumed to be convex, obstacles are permitted to both overlap and have arbitrary angular orientations, allowing for complex (nonconvex) configurations; collision detection is performed for each convex obstacle. For each test case, the percentage of the obstacle field coverage is the ratio of the total area of all obstacles, accounting for overlap, to the area allotted for obstacle placement. The paths are further restricted to maintain a distance of at least 0.6 m from the obstacles in order to allow for some leeway when executing the motion plan in the presence of disturbances. A potential successor is considered to be a duplicate of another node in the tree if the latter node has the same heading angle and resides in a Euclidean ball of radius 0.13 m, centered about the potential successor; when using the third library, the velocity of the nodes must also be the same.

The mean path costs versus percentage of obstacle field coverage for each motion primitive library are given in Fig. 6, with error bars indicating the 95% confidence interval for the mean value. The path costs for the GI algorithm using the state lattice (first library) motion primitives are higher than those for WA* across the entire range of obstacle coverage. For the library with 1 m/s primitives (second library), the path costs for the GI algorithm are approximately equal to those for WA* with $\epsilon = 1.5$, verified using Student's t-test [43]. The second library also results in paths that are of lower cost than those generated using the first library. This is not an unexpected result, as the lattice library is created to operate over a uniform sampling of the translational coordinates in the configuration space, which places restrictions on the length of the motion primitives. Creating state lattice motion primitives over a finer discretization of the translational coordinates may lead to shorter path lengths. For the library with motion primitives at multiple velocities (third library), the path costs are lower than those obtained using the second library for WA* with $\epsilon = 0.5$. The third library returns path costs that are higher than those obtained using the second library for all other choices of ϵ and for the GI algorithm as well. In this case, the GI algorithm returns path costs that are approximately equal to those for WA* with $\epsilon = 4$, and higher than those for WA* with $\epsilon = 1.5$, again verified via t-test. The mean number of nodes placed in the search tree by each algorithm for the three motion primitive libraries is given in Fig. 7. For the first two libraries, as expected the GI algorithm adds fewer nodes, since in this case a single node is added to the search tree per iteration. For the third motion primitive library, the WA* algorithm expands more nodes during the search process than the number expanded using the first two libraries, however the opposite is true of the GI algorithm. This is indicative of the GI algorithm utilizing the higher

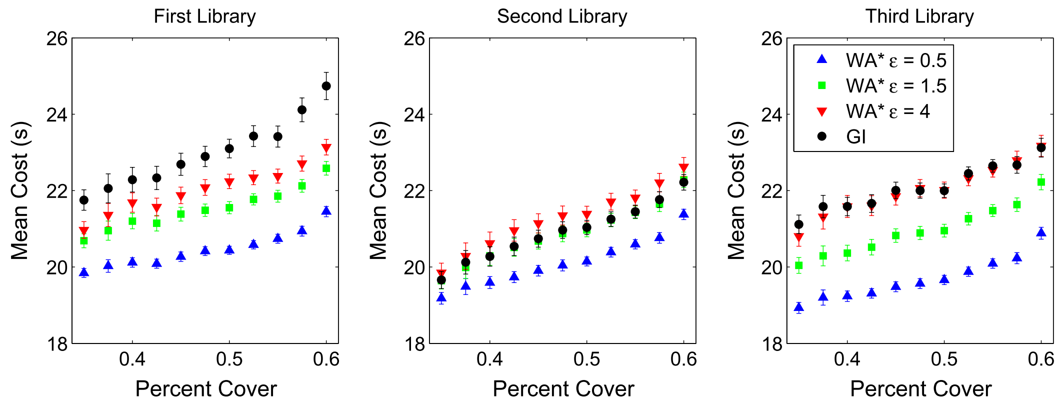


Fig. 6. Mean path costs for all motion primitive libraries.

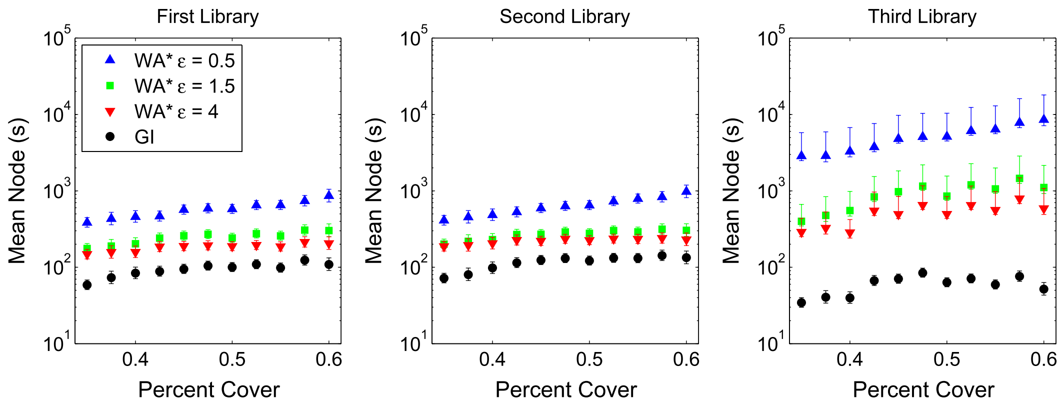


Fig. 7. Mean number of nodes expanded for all motion primitive libraries.

velocity motion primitives and finding a solution at a shallower depth in the tree, resulting in slightly higher path costs.

Fig. 8 shows the mean solution time versus percent obstacle coverage for each algorithm and motion primitive library, with error bars indicating the 95% confidence interval. The first library shows a slight advantage over the second library with respect to computation time. This is expected, as the motion planner using primitives constructed over a state lattice is able to update the parent of a node if a lower cost path is found to that particular node. In the case of the second library, however, duplicate nodes with lower cost result in any descendants of the higher cost path being placed into the closed list. The GI algorithm is comparable with respect to computation time to WA* with $\epsilon = 1.5$ when using the first library; in the case of the second library, the GI algorithm requires slightly less computation time than WA* with $\epsilon = 4$. The equivalence of computation times is again verified via t-test. For WA*, the third library results in increased computation time compared to the first two libraries for all the ϵ values that we consider; in addition, the upper confidence limits on the mean computation times are significantly larger, which indicates that there are a number of cases with high solution times. The GI algorithm, on the other hand, has lower mean solution times than those obtained using the first and second libraries. Note that the number of nodes expanded in the search tree directly affects the computation time because of the computational demand of the `InTree` function. As displayed by Fig. 7, the GI algorithm places a far lower number of nodes in the search tree than WA* when utilizing the third motion primitive library. This leads to the WA* algorithm checking a longer list of potential duplicate nodes, resulting in increased computation time. Maximum solution times versus percent obstacle coverage are given in Fig. 9. From these plots, it is clear that, for the first two motion primitive libraries, the maximum computation times for the GI algorithm are comparable to those for WA* with $\epsilon = 4$. For the third library, the maximum solution times for the GI algorithm are lower than those for WA* with any of the values of ϵ considered. When using the first library, the WA* algorithm with $\epsilon = [0.5, 1.5, 4]$ returns solutions quicker than the GI algorithm in [4.4%, 53%, 75%] of the test cases, respectively. With the second library, WA* returns solutions in less time in [4.7%, 14.6%, 26.3%] of the test cases for $\epsilon = [0.5, 1.5, 4]$, respectively. Finally, WA* with $\epsilon = [0.5, 1.5, 4]$ returns solutions faster than the GI algorithm when using the third library in [1.2%, 2.2%, 2.65%] of the test cases, respectively.

Figs 10–11 give paths generated by the algorithms using the three libraries for a specific obstacle environment. The (x, y) position of nodes placed in the search tree are shown in black, and the resulting solution paths are given in green. The lattice-based library planners return solutions in about 0.08 s (WA*, $\epsilon = 0.5$) and 0.04 s (GI). Computation times for the second library are around 0.4 s (WA*, $\epsilon = 0.5$) and 0.3 s (GI), and the third library results are computed in approximately 1.6 s (WA*, $\epsilon = 0.5$) and 0.05 s (GI). The times needed to traverse the paths in Fig. 10 are 20 s, 23.4 s, and 21 s, respectively. In Fig. 11, the times required to traverse the paths are 20.1 s, 22.8 s, and 13.2 s, respectively. Observe that, using the third library, the GI algorithm generates a far faster trajectory than WA* (13.2 s vs 21 s). Notice also that the WA* algorithm can end up placing a large number of nodes in the tree when a state lattice is not utilized, as reflected by the density of nodes along the obstacles in Fig. 10 for the second and third libraries. In this particular obstacle environment, the WA* algorithm with the lowest ϵ value returns a solution faster than when using higher weighting factors in the cases of the first and third libraries, and the GI algorithm in these cases is far faster than WA* for all three weighting values. When using the second library,

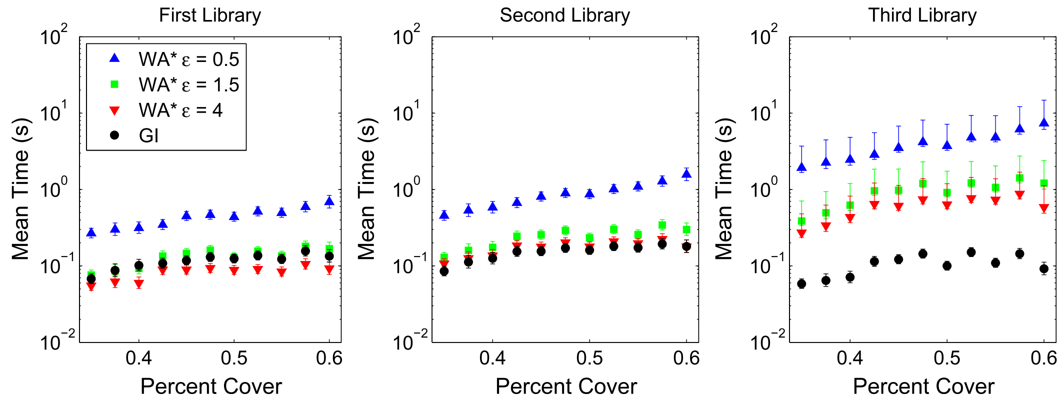


Fig. 8. Mean solution times for all motion primitive libraries.

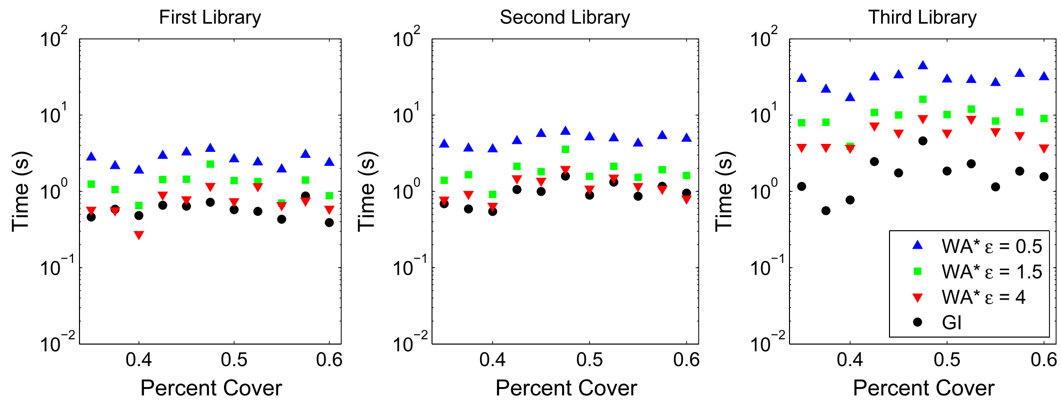


Fig. 9. Maximum solution times for all motion primitive libraries.

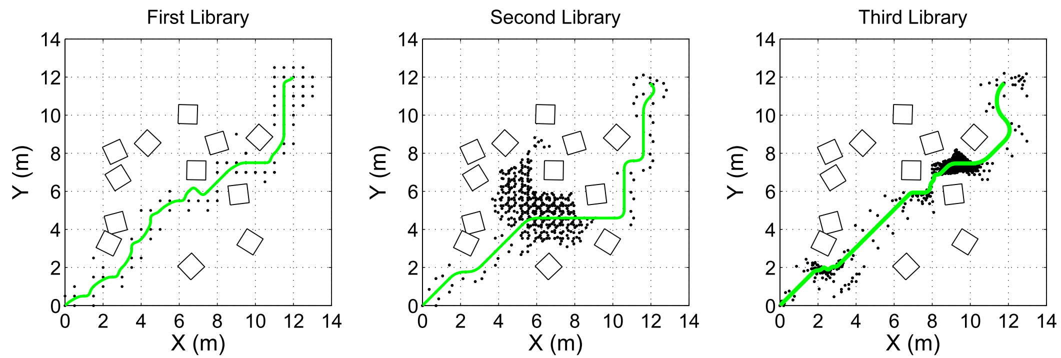


Fig. 10. WA* paths with $\epsilon = 0.5$ for all motion primitive libraries.

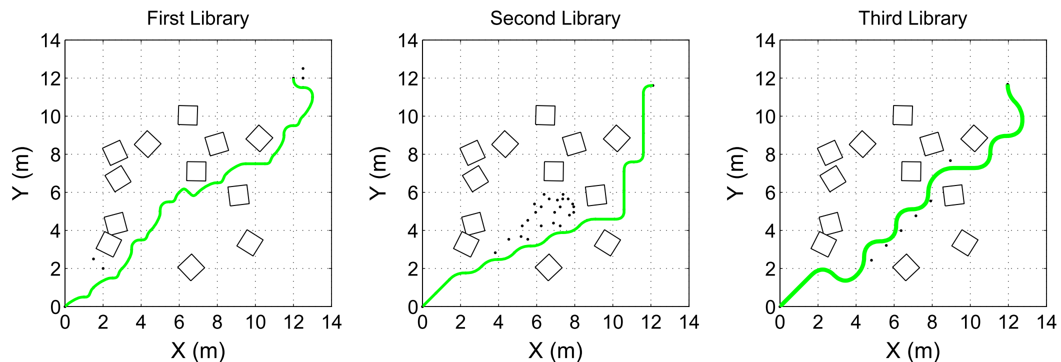
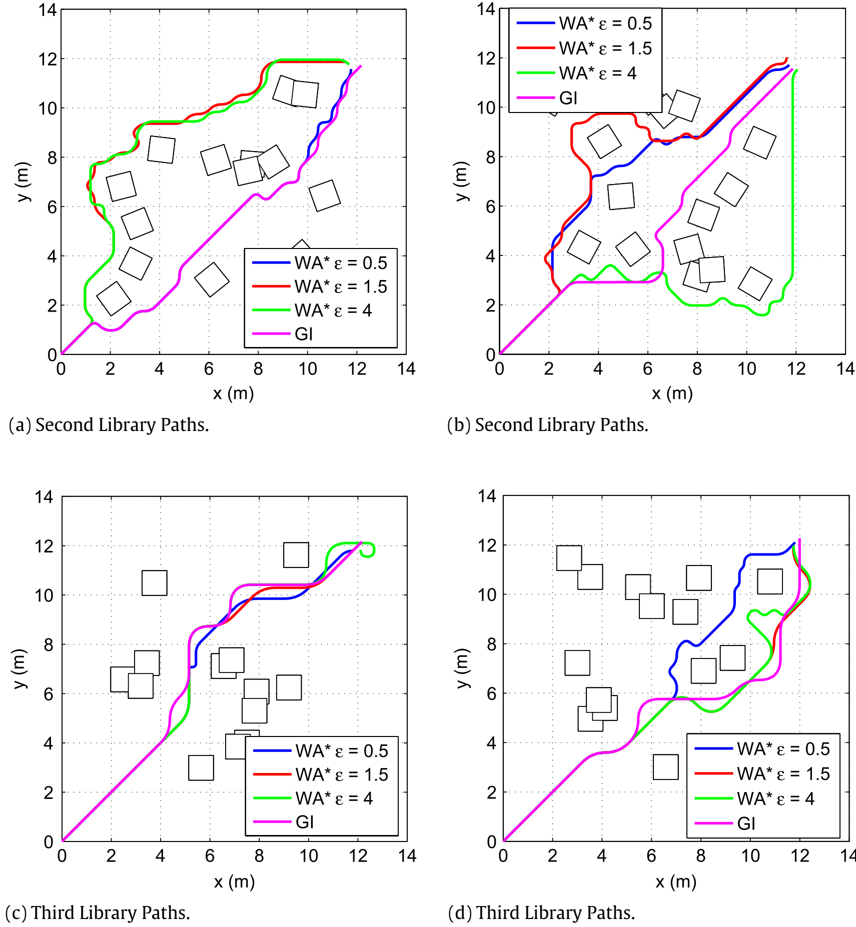


Fig. 11. GI paths for all motion primitive libraries.



(a) Second Library Paths.
(b) Second Library Paths.
(c) Third Library Paths.
(d) Third Library Paths.

Fig. 12. Solution Path Comparisons—Second and Third Libraries. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

however, the WA* algorithm with $\epsilon = 4$ is the fastest and returns a solution in about 0.14 s. Direct comparisons of paths found using the second library are given in Fig. 12(a)–(b), and examples of paths returned using the third library are given in Fig. 12(c)–(d). These figures demonstrate that paths returned by the GI algorithm are comparable to those returned using WA*.

4.2. Hybrid control

The equations of motion given in Eqs. (3)–(4) are derived in the inertial reference frame and the dynamics subsequently vary with respect to rotations about the z-axis. Using a body-axis reference frame eliminates the dependence on θ, and the resulting transformed equations of motion are given in Eqs. (6)–(8). The motion primitives utilized in the previous section are also transformed into the body-axis reference frame using Eqs. (9)–(10).

$$m\ddot{x}_b + b_t\dot{x}_b = m\dot{\theta}\dot{y}_b + u_1 - u_3 \tag{6}$$

$$m\ddot{y}_b + b_t\dot{y}_b = -m\dot{\theta}\dot{x}_b + u_4 - u_2 \tag{7}$$

$$J\ddot{\theta} + b_r\dot{\theta} = L(u_1 - u_2 + u_3 - u_4) \tag{8}$$

$$\dot{x}_b = \dot{x}\cos\theta + \dot{y}\sin\theta \tag{9}$$

$$\dot{y}_b = -\dot{x}\sin\theta + \dot{y}\cos\theta. \tag{10}$$

Defining the state column vector $v = (x_b, y_b, \theta, \dot{x}_b, \dot{y}_b, \dot{\theta})$ and the input column vector $u = (u_1, u_2, u_3, u_4)$, the equations of motion (6)–(8) can be equivalently written as $\dot{v} = f(v, u)$, where $f(\cdot,\cdot)$ is defined in the obvious way. Assuming that the input and state approximately follow the (reference) input and state trajectories associated with the finite-horizon library primitive i of length h_i, then the errors between the actual and reference variables, namely $\bar{v}^{(i)} = v - v_r^{(i)}$ and $\bar{u}^{(i)} = u - u_r^{(i)}$, will be small enough to satisfy the following state-space equation:

$$\dot{\bar{v}}^{(i)} = A_c^{(i)}(t)\,\bar{v}^{(i)} + B_{2c}^{(i)}(t)\,\bar{u}^{(i)},$$

where $A_c^{(i)}(t) = \left.\frac{\partial f}{\partial v}\right|_{(v_r^{(i)}, u_r^{(i)})}$ and $B_{2c}^{(i)}(t) = \left.\frac{\partial f}{\partial u}\right|_{(v_r^{(i)}, u_r^{(i)})}$.

We will assume that the hovercraft is subject to exogenous disturbances in the form of torques as well as forces in the x and y directions; also, these disturbances, like the input, are applied in discrete-time with a sampling frequency of 20 Hz. In addition to these disturbances which constitute the first three components of the vector-valued signal $w(t)$, we also consider errors (noise) in the measurements of x, y, and θ, which correspond to the last three components of the signal $w(t)$. Then, the state-space description of the continuous-time LTV model is

$$\dot{\bar{v}}^{(i)}(t) = A_c^{(i)}(t)\,\bar{v}^{(i)}(t) + B_{1c}\,w(t) + B_{2c}^{(i)}(t)\,\bar{u}^{(i)}(t),$$

where, for simplicity, we take

$$B_{1c} = \begin{bmatrix} 0_{3\times3} & 0_{3\times3} \\ I_3 & 0_{3\times3} \end{bmatrix}.$$

The corresponding discrete-time model obtained by zero-order hold sampling is given by: $\bar{v}_{k+1}^{(i)} = A_k^{(i)}\,\bar{v}_k^{(i)} + B_{1k}\,w_k + B_{2k}^{(i)}\,\bar{u}_k^{(i)}$, where

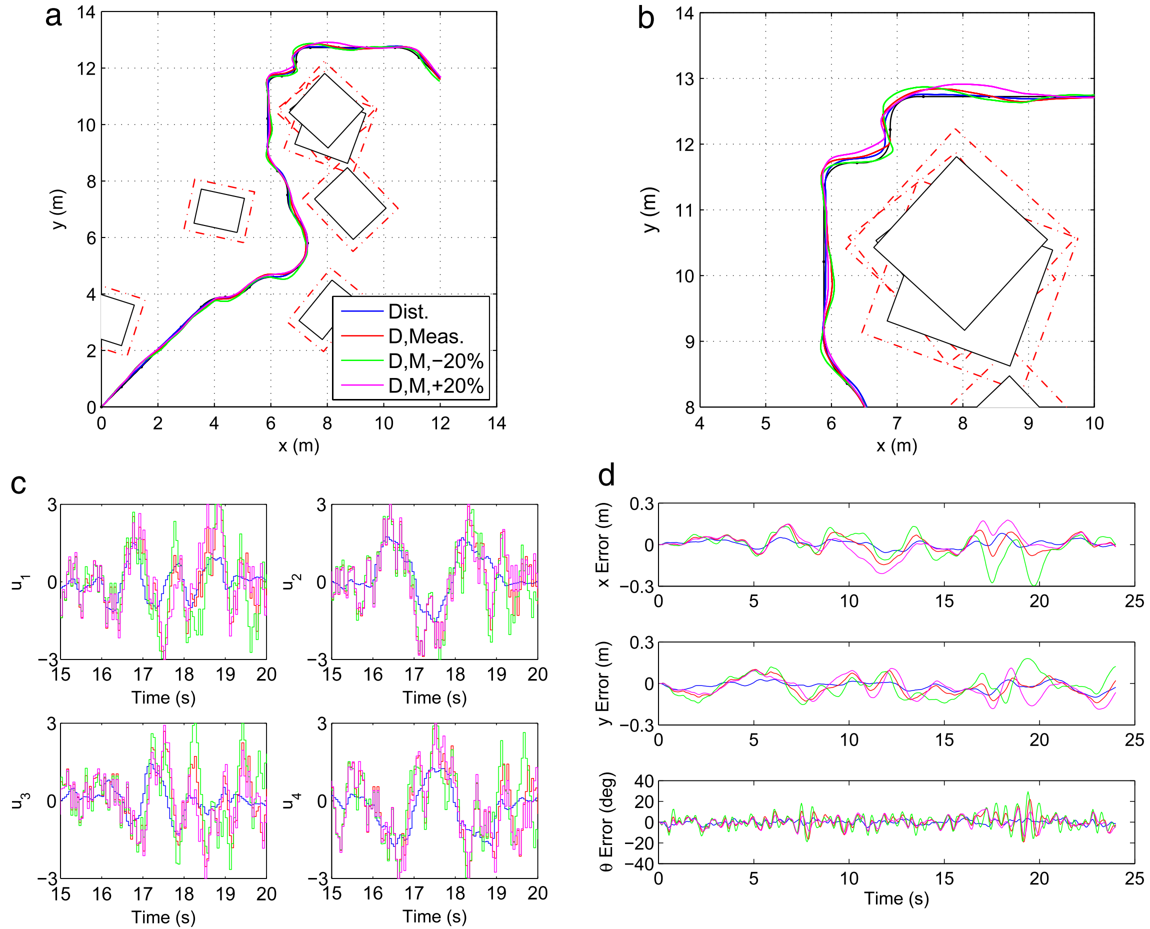


Fig. 13. Simulation Results: (a) feedback simulation paths, (b) simulation paths—zoomed, (c) feedback control inputs, (d) errors in the states. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

$T = 0.05$ s is the sampling time, $\bar{v}_k^{(i)} = \bar{v}^{(i)}(kT)$ for integers $k \geq 0$, $A_k^{(i)} = \Phi^{(i)}((k+1)T, kT)$, $\Phi^{(i)}(\cdot,\cdot)$ being the state transition matrix, $B_{2k}^{(i)} = \int_{kT}^{(k+1)T} \Phi^{(i)}((k+1)T, \tau) B_{2c}^{(i)}(\tau)d\tau$, and B_{1k} is defined similarly to $B_{2k}^{(i)}$. We assume that the states x_b, y_b, and θ are measurable, and as for the exogenous errors to be controlled, we choose in this example to only penalize $\bar{x}_b^{(i)}, \bar{y}_b^{(i)}, \bar{\theta}^{(i)}$, and $\bar{u}_j^{(i)}$ for $j = 1, \ldots, 4$. Then we get the following standard discrete-time finite-horizon model:

$$\begin{bmatrix} \bar{v}_{k+1}^{(i)} \\ z_k \\ p_k \end{bmatrix} = \begin{bmatrix} A_k^{(i)} & B_{1k} & B_{2k}^{(i)} \\ C_1 & D_{11} & D_{12} \\ C_2 & D_{21} & D_{22} \end{bmatrix} \begin{bmatrix} \bar{v}_k^{(i)} \\ w_k \\ \bar{u}_k^{(i)} \end{bmatrix},$$

for $k = 0, 1, \ldots, h_i$, where $w \in \ell_2$, p and z are the measurements and exogenous errors respectively, and, as discussed in the preceding, D_{11} and D_{22} are zeros, $C_1 = \text{diag}(I_3, 0_{4\times3})$, $D_{12} = \begin{bmatrix} 0_{4\times3} & 0.1 I_4 \end{bmatrix}^*$, $C_2 = \begin{bmatrix} I_3 & 0_{3\times3} \end{bmatrix}$, and
$D_{21} = \begin{bmatrix} 0_{3\times3} & \text{diag}(0.05, 0.05, 0.0436) \end{bmatrix}$.

Note that, when linearizing the system equations about the steady-state trajectory, we obtain an LTI (i.e., (0, 1)-eventually periodic) model. Next, given this hybrid model, we seek a hybrid LTV synthesis that would guarantee the stability of the closed-loop system and furthermore ensure that $\|w \mapsto z\|_{\ell_2 \to \ell_2} < \gamma$, where γ is the minimum possible bound, up to a certain tolerance, that is achievable by such a synthesis. Thus, we have to solve the semidefinite programming problem: minimize γ^2 subject to the synthesis conditions (1)–(2) (for a finite horizon model $G^{(i)}$, take h_i equal to the finite horizon length and $q_i = 1$ with the periodic portions of the state space sequences set to zeros; for an LTI model $G^{(j)}$, take $h_j = 0$ and $q_j = 1$). We use `Yalmip` [44] along with SDPT3 [35] for this matter, and find that the minimum achievable $\gamma \approx 0.61$. The elapsed (i.e., wall clock) time for solving the optimization problem is about 14 s (CPU time = 11 s). For the simulation, we relax γ to 1.3 in order to obtain satisfactory controller performance. Note that this excessive relaxation is done to accommodate the significant modeling uncertainties considered such as varying the mass and inertia by ±20% in simulation; however, it is possible to improve the performance while still maintaining satisfactory robustness by judiciously choosing the penalty functions on the control and state errors. The solutions of the synthesis program can then be used to construct a controller, as discussed at the end of Section 3; note that the MATLAB command *basiclmi* is useful for this purpose.

As for the simulation, we subject the hovercraft to iid disturbances, which are generated by the MATLAB function *rand*; these disturbances correspond to forces in the x and y directions, namely $\mathcal{F}_{xk}$ and $\mathcal{F}_{yk}$, as well as torques $\mathcal{T}_k$, applied on the hovercraft in discrete-time with a sampling frequency of 20 Hz, such that $|\mathcal{F}_{xk}|, |\mathcal{F}_{yk}| \leq 1$ N and $|\mathcal{T}_k| \leq 0.15$ N m for all integers $k \geq 0$. We impose the 3 N bound on the control inputs as well as a rate limit of 20 N/s. We also change the mass and moment of inertia of the hovercraft by a factor of 0.8 (20% decrease) or 1.2 (20% increase). Additionally, zero mean Gaussian noise with standard deviation of 0.05 m is applied to the x and y measurements, and, similarly, zero mean Gaussian noise with 2.5° standard deviation is applied to the measurements of θ. Despite these disturbances and uncertainties, the controller is still able to force the hovercraft to

closely track trajectories generated from the library primitives, as shown in Fig. 13(a)–(d) for a typical run. In these figures, the different cases of considered disturbances and uncertainties are labeled as follows: force/torque disturbances only (Dist); force/torque disturbances and measurement noise (D,Meas.); force/torque disturbances, measurement noise and −20% mass and inertia (D, M, −20%); and force/torque disturbances, measurement noise, and +20% mass and inertia (D, M, +20%). The reference and simulations paths through the obstacle field are given in Fig. 13(a). The points on the paths correspond to the locations of the center of gravity of the hovercraft. Since the radius of the hovercraft is 0.3 m which is also the difference between the edges of the concentric rectangles, the paths must not enter the rectangles with dashed edges around the obstacles. During motion planning, the width of the area between the bounding box (dashed rectangle) and the corresponding obstacle is set to 0.6 m to allow for some deviation from the reference trajectory, if necessary, due to disturbances and other uncertainties. The control input and state errors are given in Fig. 13(c)–(d). Despite the measurement noise, disturbances, and uncertainty with respect to mass and inertia, the controller is able to keep the vehicle within 0.3 m of the reference path in both x and y directions, thus avoiding collisions with obstacles in all simulation scenarios.

5. Conclusions

By using a library of pre-specified motion primitives, the task of finding a dynamically feasible trajectory through an obstacle field is reduced to a search over a graph, in which the nodes correspond to the vehicle states and the edges represent the motion primitives. Developing the motion primitives according to a state lattice sampling benefits the search process with respect to computation time. For motion primitives that do not form a state lattice, a tree representation is instead used in order to maintain continuity of states between successive primitives. Consequently, this adversely affects the solution time for the search algorithms since a duplicate node with a higher cost-to-go, which in general will not match exactly the current node, is placed along with its descendants in the closed set to avoid a trajectory with gaps or discontinuities, thus wasting the previous computations. As an alternative, a locally greedy algorithm is presented. This algorithm exhibits improved performance with respect to run time over weighted A*, particularly when considering motion primitive libraries with multiple velocities. The improved performance in solution time can be attributed to the size of the search trees created by each algorithm. Specifically, the check to see whether a location within the configuration space has already been visited is far more computationally demanding for larger search trees, and the weighted A* algorithm constructs much larger search trees than the locally greedy algorithm. Ultimately, the success of both of these approaches is dependent on the heuristic used. Obstacle fields can be created where the Euclidean distance, for instance, is a poor estimate of the true cost-to-goal, resulting in computation times that are unacceptable. In these scenarios, a sampling-based planner may be preferable.

A control approach is also presented, which uses the ℓ_2-induced norm as the performance measure, providing a set of discrete-time controllers that accompany the motion primitive library. The solution returned by the motion planner consists not only of a sequence of motion primitives leading from the initial state to the goal, but also a set of controllers with stability and performance guarantees. An example of the hierarchical process applied to a hovercraft details the development of motion primitives, the design of subcontrollers using convex optimization, the motion planning task, and the execution of the motion plan in simulation subject to exogenous disturbances, measurement noise, and various uncertainties. The resulting hierarchical motion planning and feedback control strategy is able to guide the hovercraft through an obstacle field while avoiding collisions. The benefit of this hierarchical approach is that the feedback control strategy is developed offline in conjunction with the motion primitive library, and thus does not require any additional computation when determining a dynamically feasible trajectory in real-time. Future work includes the investigation of other motion planning algorithms, such as sampling-based methods, in order to determine their applicability to the problem posed herein.

References

[1] N. Nilsson, Problem-Solving Methods in Artificial Intelligence, McGraw-Hill, New York, 1971.
[2] S. LaValle, Planning Algorithms, Cambridge University Press, 2006.
[3] E. Frazzoli, M. Dahlah, E. Feron, Maneuver-based motion planning for nonlinear systems with symmetries, IEEE Transactions on Robotics 21 (6) (2005) 1077–1091.
[4] C. Goerzen, Z. Kong, B. Mettler, A survey of motion planning algorithms from the perspective of autonomous UAV guidance, Journal of Intelligent and Robotic Systems 57 (1–4) (2009) 65–100.
[5] J. Pearl, Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley, Reading, MA, 1984.
[6] E. Hansen, R. Zhou, Anytime heuristic search, Journal of Artificial Intelligence Research 28 (1) (2007) 267–297.
[7] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, S. Thrun, Anytime search in dynamic graphs, Artificial Intelligence 172 (14) (2008) 1613–1643.
[8] L. Kavraki, P. Svestka, J. Latombe, M. Overmars, Probabilistic roadmaps for path-planning in high-dimensional configuration spaces, IEEE Transactions on Robotics and Automation 12 (4) (1996) 556–580.
[9] S. LaValle, J. Kuffner, Randomized kinodynamic planning, International Journal of Robotics Research 20 (5) (2004) 378–400.
[10] S. Karaman, E. Frazzoli, Incremental sampling-based algorithms for optimal motion planning, in: Proceedings of Robotics: Science and Systems, 2010.
[11] S. Karaman, M. Walter, A. Perez, E. Frazzoli, S. Teller, Anytime Motion Planning using the RRT*, in: Proceeding of the 2011 IEEE Conference on Robotics and Automation, Shanghai, China, 2011, pp. 1478–1483.
[12] J. Barraquand, J. Latombe, Nonholonomic multibody mobile robots: controllability and motion planning in the presence of obstacles, Algorithmica 10 (1993) 121–155.
[13] J. Go, T. Vu, J. Kuffner, Autonomous behaviors for interactive vehicle animations, Graphical Models 68 (2) (2006) 90–112.
[14] S. Pancanti, L. Pallottino, S. Salvadorini, A. Bichi, Motion planning through symbols and lattices, in: Proceedings of the 2004 IEEE ICRA, New Orleans, LA, 2004, pp. 3914–3919.
[15] M. Pivtoraiko, R. Knepper, A. Kelly, Differentially constrained mobile robot motion planning in state lattices, Journal of Field Robotics 26 (3) (2009) 308–333.
[16] R. Burridge, A. Rizzi, D. Koditschek, Sequential composition of dynamically dextrous robot behaviors, International Journal of Robotics Research 18 (2009) 534–555.
[17] D. Conner, H. Choset, A. Rizzi, Flow-through policies for hybrid controller synthesis applied to fully actuated systems, IEEE Transactions on Robotics 25 (1) (2009) 136–146.
[18] S. Lindemann, S. LaValle, Simple and efficient algorithms for computing smooth, collision-free feedback laws over given cell decompositions, International Journal of Robotics Research 28 (2009) 600–621.
[19] G. Dullerud, S. Lall, A new approach to analysis and synthesis of time-varying systems, IEEE Transactions on Automatic Control 44 (8) (1999) 1486–1497.
[20] M. Farhood, G. Dullerud, LMI tools for eventually periodic systems, Systems & Control Letters 47 (5) (2002) 417–432.
[21] J. Gillula, H. Huang, M. Vitus, C. Tomlin, Design of guaranteed safe maneuvers using reachable sets: Autonomous quadrotor aerobatics in theory and practice, in: Proceedings of the 2010 IEEE International Conference on Robotics and Automation, 2010, pp. 1649–1654.
[22] R. Sanfelice, E. Frazzoli, A Hybrid Control Framework for Robust Maneuver-Based Motion Planning, in: Proceedings of the 2008 American Control Conference, 2008, pp. 2254–2259.
[23] C. Neas, M. Farhood, A hybrid architecture for maneuver-based motion planning and control of agile vehicles, In *Proceedings of the 18th IFAC World Congress*, Milano, Itay, 2011, pp. 3521–3526.
[24] P. Hart, N. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Transactions on Systems Science and Cybernetics 4 (2) (1968) 100–107.
[25] I. Pohl, Heuristic search viewed as path finding in a graph, Artificial Intelligence 1 (3–4) (1970) 193–204.
[26] R. Ebendt, R. Drechsler, Weighted A* search—unifying view and application, Artificial Intelligence 173 (14) (2009) 1310–1342.
[27] C. Neas, A Greedy Search Algorithm for Maneuver-Based Motion Planning of Agile Vehicles, Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA, 2010.
[28] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Prentice Hall, New York, 2010.

[29] T. Cormen, C. Leiserson, C. Rivest, R.L. Stein, Introduction to Algorithms, Vol. 3, MIT Press, Cambridge, MA, 2009.
[30] M. Green, D. Limebeer, Linear Robust Control, Prentice Hall, 1995.
[31] G. Zames, Feedback and optimal sensitivity: model reference transformations, multiplicative seminorms, and approximate inverses, IEEE Transactions on Automatic Control 26 (2) (1981) 301–320.
[32] S. Boyd, L. Vandenberghe, Convex Optimization, Cambridge University Press, 2004.
[33] H. Mittelmann, Decision tree for optimization software, 2012, http://plato.asu.edu/guide.html.
[34] J.F. Sturm, Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones, Optimization Methods and Software 11–12 (1999) 625–653, version 1.05 available from http://fewcal.kub.nl/sturm.
[35] K. Toh, M. Todd, R. Tutuncu, SDTP3—A Matlab software package for semidefinite programming, Optimization Methods & Software 11 (1999) 545–581.
[36] P. Gahinet, P. Apkarian, A linear matrix inequality approach to H_∞ control, International Journal of Robust and Nonlinear Control 4 (1994) 421–448.
[37] A. Bryson, Y. Ho, Applied optimal control, Hemisphere (1975).
[38] I. Ross, F. Fahroo, Legendre Pseudospectral Approximations of Optimal Control Problems, Vol. 295, Springer-Verlag, 2003.
[39] A. Rao, D. Benson, C. Darby, M. Patterson, C. Fancolin, G. Huntington, Algorithm 902: GPOPS, a MATLAB software for solving multiple-phase optimal control problems using the gauss pseudospectral method, ACM Transactions on Mathematical Software 37 (2) (2010) 1–39.
[40] Q. Gong, W. Kang, N. Bedrossian, F. Fahroo, P. Sekhavat, K. Bollino, Pseudospectral optimal control for military and industrial applications, in: Proceedings of the 46th IEEE Conference on Decision and Control, 2007, pp. 4128–4142.
[41] M. Grant, S. Boyd, CVX: Matlab Software for Disciplined Convex Programming, Version 1.21, 2010, http://cvxr.com/cvx.
[42] T. Minka, The Lightspeed Matlab Toolbox, Efficient Operations for Matlab Programming, Version 2.2, 2007, available from http://research.microsoft.com/en-us/um/people/minka/software/lightspeed/.
[43] J. Devore, Probability and Statistics for Engineering and the Sciences, sixth ed., Duxbury Press, 2003.
[44] J. Lofberg, YALMIP: a toolbox for modeling and optimization in MATLAB, in: Proceedings of the CACSD Conference, Taipei, Taiwan, 2004, pp. 284–289, available from http://control.ee.ethz.ch/~joloef/wiki/pmwiki.php.

David J. Grymin is a Ph.D. student affiliated with the Nonlinear Systems Laboratory in the Department of Aerospace and Ocean Engineering at Virginia Polytechnic Institute and State University (Virginia Tech). His current research interests include aerodynamic system identification, control of unmanned aerial vehicles, and motion planning in complex environments.

Charles B. Neas received his M.S. degree in Aerospace Engineering, in 2010, from Virginia Tech, Blacksburg, Virginia. He is currently working at Robins Air Force Base as a C-130 structural engineer.

Mazen Farhood received his bachelor's degree in Mechanical Engineering from the American University of Beirut, Lebanon, in 1999. He received the M.S. degree in 2001, and the Ph.D. degree in 2005, both in Mechanical Engineering from the University of Illinois at Urbana–Champaign. Since 2008, he has been an assistant professor in the Department of Aerospace and Ocean Engineering at Virginia Tech. From 2007 to 2008, he was a scientific researcher in the Delft Center for Systems and Control, Delft University of Technology, The Netherlands. Prior to that, he was a postdoctoral fellow in the School of Aerospace Engineering at Georgia Institute of Technology. His areas of current research interest include distributed control, motion planning and tracking along trajectories, semidefinite programming, model reduction, and control of agile aerial vehicles.