Int. J. Appl. Math. Comput. Sci., 2010, Vol. 20, No. 2, 305–316
DOI: 10.2478/v10006-010-0023-5

NTGSIM: A GRAPHICAL USER INTERFACE AND A 3D SIMULATOR FOR NONLINEAR TRAJECTORY GENERATION METHODOLOGY

LYALL JONATHAN DI TRAPANI *, TAMER INANC **

* Twelfth Air Force
Davis-Monthan Air Force Base, 6901 E Broadway Blvd, Tucson, AZ, USA
e-mail: lyall.ditrapani@dm.af.mil

**Electrical and Computer Engineering Department
University of Louisville, Lutz Hall 447, Louisville, KY, USA
e-mail: t.inanc@louisville.edu

Nonlinear Trajectory Generation (NTG), developed by Mark Milam, is a software algorithm used to generate trajectories of constrained nonlinear systems in real-time. The goal of this paper is to present an approach to make NTG more user-friendly. To accomplish this, we have programmed a Graphical User Interface (GUI) in Java, using object oriented design, which wraps the NTG software and allows the user to quickly and efficiently alter the parameters of NTG. This new program, called NTGsim, eliminates the need to reprogram the NTG algorithm explicitly each time the user wishes to change a parameter.

Keywords: trajectory generation, optimal control, Java, GUI.

1. Introduction

Nonlinear Trajectory Generation (NTG) developed by (Milam, 2003) solves constrained nonlinear optimal control problems in real-time. This methodology performs a parameterization of the solution of the system by piecewise polynomials, called B-Spline functions (de Boor, 2001). Identifying plant output and/or state variables as algebraic functions of the control input thus enables translating the dynamical constraints of the optimal control problem as algebraic nonlinear constraints posed on polynomial coefficients. NTG then employs a built-in nonlinear programming toolbox to tackle the resulting static optimization problem (Muezzinoglu and Inanc, 2006).

NTG has been applied to several robotics problems in the literature. In (K. Misovec and Murray, 2003; Inanc *et al.*, 2004), NTG is used to generate low-observable trajectories for unmanned aerial vehicles. In (Milam, 2002), IT is used for a missile intercept problem. In (Lian and Murray, 2003), NTG is extended to a multi-vehicle problem with precedence constraints.

Unfortunately, the current state of NTG is somewhat counterintuitive to use. Each optimal control problem requires the user to write a program which details the problem parameters, constraints and cost functions. This program would make a call to the NTG function. Once written, the program must then be compiled and linked with the NTG library which contains the NTG algorithm. If the user wishes to make any changes to his or her optimal control problem, e.g., alter a spline parameter or modify the cost functions, he or she must open up the source code of his or her program, make the appropriate changes, recompile, and finally link with the NTG library once again. On top of being hideously time consuming, this process also increases the chance of introducing bugs to the program each time the process is repeated. It is clear that NTG is not the most user friendly software package. This is exactly the problem which is addressed here.

In this paper we present NTGsim, the solution we have developed, with preliminary results presented in (Trapani and Inanc, 2009). NTGsim is a graphical user interface for NTG. By creating a GUI around NTG, we hope to increase ease of use and accessibility, to eliminate unnecessary recompilation and to provide support in specifying and solving optimization problems with NTG. Recently, a solution to the problem has been proposed to facilitate the use of NTG called OPTRAGEN, a MATLAB toolbox for NTG developed by (Bhattacharya, 2006). However, OPTRAGEN, being a MATLAB toolbox for

NTG, is different from our solution, NTGsim. OPTRAGEN obviously requires MATLAB to be used and it does not provide real-time application of NTG, which was designed to solve constrained nonlinear optimal control problems in real-time. On the other hand, our proposed solution, NTGsim, is based on the Java platform and provides real-time application and a built-in 3D simulator (currently being developed) to quickly simulate the designed trajectories without depending on other commercial software tools such as MATLAB.

The structure of this paper is as follows. First, a brief overview of the NTG algorithm is provided in Section 2. In the next section, the bulk of the paper, NTGsim will be broken down into its major components and concepts, and each one will be discussed. Following that, full installation instruction for NTGsim will be provided. The final section will give a brief example on how to use NTGsim based on the van der Pol optimal control problem.

2. Overview of nonlinear trajectory generation methodology

The NTG algorithm is the state-of-the-art and it solves constrained nonlinear optimal control problems in real-time (Milam, 2003). It is based on a combination of nonlinear control theory, spline theory, and sequential quadratic programming. NTG takes the optimal control problem formulation, the characterization of the trajectory space in terms of approximation functions and transforms them into a Non-Linear Programming (NLP) problem. It is then solved using NPSOL, a popular NLP solver, which uses sequential quadratic programming to obtain the solution.

This section gives a brief introduction to the NTG algorithm and the underlying techniques. The general nonlinear optimal control problem is stated first. Then, the outline of the NTG algorithm is summarized.

2.1. Optimal control problem. Consider a general dynamical (control) system

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t), \mathbf{u}(t)), \tag{1}$$

where $\mathbf{x}(t)$ is the state of the system and $\mathbf{u}(t)$ is the control input. For optimal control, we would like to choose $\mathbf{u}(t)$ such that some cost function is minimized and constraints are enforced. That is, given a cost function of the form

$$J = \Phi_0(\mathbf{x}(t_0), \mathbf{u}(t_0), t_0) + \int_{t_0}^{t_f} L(\mathbf{x}(t), \mathbf{u}(t), t)\,\mathrm{d}t + \Phi_f(\mathbf{x}(t_f), \mathbf{u}(t_f), t_f), \tag{2}$$

we would like to choose $\mathbf{u}(t)$ for $t \in [t_0, t_f]$ which minimizes J subject to the constraints of the form

$$\begin{array}{lrcl} \text{Initial} & lb_0 & \leq \Psi_0(\mathbf{x}(t_0), \mathbf{u}(t_0), t_0) & \leq ub_0, \\ \text{Trajectory} & lb_t & \leq \Psi_t(\mathbf{x}(t), \mathbf{u}(t), t) & \leq ub_t, \\ \text{Final} & lb_f & \leq \Psi_f(\mathbf{x}(t_f), \mathbf{u}(t_f), t_f) & \leq ub_f. \end{array} \tag{3}$$

Note that the cost function J is composed of an initial condition cost, $\Phi_0(\cdot)$, an integral cost over the trajectory, $L(\cdot)$, and a final condition cost, $\Phi_f(\cdot)$. The constraints are similarly partitioned. Here lb and ub represent lower and upper bounds, respectively. Equations (2) and (3) are standard in optimal control and are further explained in (Milam, 2003; Bryson and Ho, 1975).

In most cases, the dynamics (1) and constraints (3) are too complicated for the minimization of (2) to be solved analytically, so numerical algorithms must be used to obtain the solutions. To solve optimal control problems numerically, they are often transformed into nonlinear programming problems. The nonlinear trajectory generation methodology is the state-of-the-art algorithm for transforming the optimal control problem given in Eqn. (2) to an NLP problem, and solving it in real-time (Milam, 2003; Milam *et al.*, 2000).

2.2. Nonlinear trajectory generation methodology. If the cost and constraints are evaluated at discrete points in the interval $[t_0, t_n]$, it is possible to translate the optimization problem, defined by (2) and (3), into the following NLP problem in C_j:

$$\min_{\vec{C} \in \mathbb{R}^p} F(\vec{C})$$

subject to

$$L \leq G(\vec{C}) \leq U,$$

where $\vec{C} = [C_1 \ \cdots \ C_p]^T$. $F(\vec{C})$ is our transformed cost function, and $G(\vec{C})$ is the transformation of the constraints, with L and U being the lower and upper bounds, respectively. The discrete points, C_i, at which the cost and constraints are evaluated are known as *collocation points*.

There are three steps in the NTG algorithm:

1. The first step is to exploit any differential flatness of (1) to find a new set of outputs of the system so that the system dynamics can be mapped down to a lower-dimensional space, with the property that all the states and controls of the original system can be recovered from the new lower-dimensional representation. The idea is that it will be easier and computationally more efficient to solve a lower dimensional problem by finding an output $z = z_1, \ldots, z_q$ of the form
$$z = A\left(x, u, u^{(1)}, \ldots, u^{(r)}\right), \tag{4}$$
where $u^{(i)}$ denotes the i-th derivative of u with respect to time. If Eqn. (1) is differentially flat, then

the states and inputs of the system, (x, u), can be completely established from (5). If there is no flat output or one cannot find a flat output, then (x, u) can still be completely determined from the lowest dimensional space possible given in Eqn. (6). A necessary condition for the existance of such outputs is given in (Milam *et al.*, 2000).

$$(x, u) = B\left(z, z^{(1)}, \ldots, z^{(s)}\right), \tag{5}$$

$$(x, u) = B_1\left(z, z^{(1)}, \ldots, z^{(s_1)}\right), \tag{6}$$

$$0 = B_2\left(z, z^{(1)}, \ldots, z^{(s_2)}\right),$$

where $z^{(i)}$ denotes the i-th derivative of z with respect to time.

2. The second step in NTG is to further represent these outputs in terms of the B-spline functions as

$$z_i(t) = \sum_{j=1}^{p_i} B_{j,r_i}(t) C_j^i, \tag{7}$$

where B_j and C_j^i represent known B-spline basis functions and unknown B-spline coefficients, respectively (de Boor, 2001).

3. Finally, to determine the coefficients of the B-spline functions, C_j^i, with the sequential quadratic programming package NPSOL (Gill *et al.*, n.d.), the cost function and constraints given in (2) and (3) are re-formulated in terms of the B-spline coefficients, yielding a nonlinear programming problem.

2.3. Using NTG. The NTG algorithm generates trajectories of constrained nonlinear systems. In order for the user to manipulate NTG, he or she must define the constrained nonlinear optimal control problem of interest explicitly in the NTG framework (Milam, 2003). For such a system to be fully qualified, the NTG algorithm has 46 input parameters (Milam, 2003). These parameters can be broken up into two different types: static parameters and dynamic functions.

Static parameters define the number of outputs (splines) and the number of derivatives for each output, the number of cost and constraint functions, the placement of knot points, the order and smoothness of the B-splines, and the collocation points for each output. Basically, any parameter can be assigned a discrete value. We have labeled these parameters as "static" because they do not require the NTG algorithm to be recompiled each time they change. The static parameters can be passed to the NTG algorithm with the use of a well designed GUI without altering the NTG algorithm.

The second group of parameters, dynamic functions, comprises the remaining six input parameters. Half of these are the three cost functions—initial, trajectory, and final, which describe the objective(s) of the system. The other half are the three nonlinear constraint functions—initial, trajectory, and final. These last three aptly named functions describe the nonlinear constraints of the system. The reason why the six functions are labeled as "dynamic" is because they do require the NTG algorithm to be recompiled each time they are changed.

The current version of NTG only runs on operating systems which implement the Portable Operating System Interface (POSIX). During the preparation of this paper, it was tested on x86 based computers running various Ubuntu and Mandriva Linux distributions. The NTG algorithm is dependent on three static libraries: libNPSOL.a, libpgs.a, and libg2c.a. The NPSOL and PGS libraries were written in the FORTRAN programming language. The NPSOL library is needed to do the actual nonlinear problem solving (Gill *et al.*, n.d.). The PGS library takes care of spline related functions and the G2C library is needed to understand the FORTRAN symbols used in NPSOL and PGS. The entire NTG algorithm, along with the three static libraries, is packaged into a single static library called libNTG.a. The libNTG.a library does not do anything by itself. It needs another program to wrap the library and feed it all 46 of its parameters.

3. Program structure

This section will explain the general flow of data through NTGsim and the reasons why Java was chosen as the programming language. Next, it will break down the application into its major components and discuss each one. It will also explain how modularization and multithreading were used in developing NTGsim.

3.1. Data flow. The following is a rough overview of the data flow through the program. Each concept will be discussed in greater detail further on. The data flow of the program is shown in Fig. 1. First, the InputGui object receives input from the user. For each input received, InputGui will pass the new data to the InputData object. InputData will then translate the new data from the problem domain into the format the NTG algorithm expects. The transformed data will be stored in the NativeInputData object. Once the user is sets up the desired parameters, the Controller object will send NativeInputData to the native library, libJ2NTG, using the Java Native Interface (JNI.) The JNI allows a connection from the Java program, through the Java Virtual Machine (JVM) to the native library (Liang, 1999). LibJ2NTG takes the parameters from within NativeInputData and feeds them to the NTG algorithm, which in turn computes the coefficients of the spline(s) representing the generated trajectory(s.) The coefficients are written to a text file on the hard drive

and then returned across the JVM, using the JNI, to the Controller.

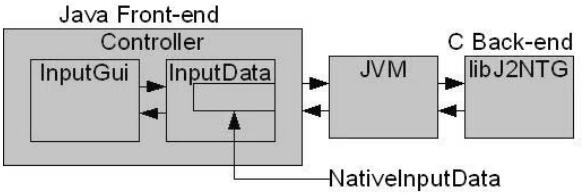

Fig. 1. NTGsim data flow.

3.1.1. Reasons for Java. As can be seen from Fig. 1, using Java as opposed to the C programming language adds an extra layer of complexity to the program flow. If the C language was used instead to program NTGsim, the interfacing layer through the JNI would not be necessary. However, developing the software in Java allowed us to obtain many of our key objectives while facilitating the production of more robust software. One of our goals was to ensure that NTGsim was not tied to a single Operating System (OS). If NTG were ported to a new operating system, we would like to have the ability to bring NTGsim along with it to the new OS. Java's platform independent nature (Gosling and McGilton, 1996; Sierra and Bates, 2005) made the language a natural choice for this purpose.

Another goal was to create a modular GUI which could be adapted to the use with various trajectory generation algorithms in the future. The object oriented nature of the Java programming language (Sierra and Bates, 2005; Gosling *et al.*, 2005) allowed us to easily incorporate this feature. The overall strength of Java class libraries was another win for Java (Sierra and Bates, 2005; Sun Microsystems, 2006).

By taking advantage of these libraries, we had access to thousands of classes written by professional software engineers (Sierra and Bates, 2005). Java's Swing and Remote Method Invocation (RMI) are two examples of Application Programming Interfaces (APIs) from the class library which we will now discuss in greater detail. Since the overreaching purpose of the work presented here was to design a GUI, ensuring that the chosen language had a quality widget toolkit available was top priority.

Java's Swing and the underlying Abstract Windowing Toolkit (AWT) fit the bill nicely. Their power, flexibility and consistent cross-platform presentation were all wins for Java over other languages. Since NTGsim was envisioned as a trajectory generation system for autonomous robots, having a way to remotely connect with the NTGsim program and feed it input data was very appealing. For instance, being able to connect to a GPS or an overhead camera system would be very useful. RMI easily provides this functionality (Sierra and Bates, 2005). Therefore, future extensions to the program in order to include an outside data source are very possible with Java's RMI.

Also, in the light of the real-time nature of NTG, we needed to ensure that NTGsim would also be able to run in real-time. This means that the GUI component would at least have to be able to keep up with the NTG algorithm. Although, in the past, the Java Runtime Environment (JRE) has suffered from negative reputation with respect to its performance, today the JRE's performance is quite close to that of C++ and certainly up to the task of running a responsive GUI (Sierra and Bates, 2005). This is in large part due to the advent of the Hotspot Just in Time (JIT) compiler (Davison, 2005).

In GUI applications, it is important to prevent GUI hang-ups where the application appears to be unresponsive. With its built-in multithreading ability, Java is again a good choice (Davison, 2005). Another important feature of the Java platform is its ability to automatically generate program documentation from properly formatted comments. This huge time saver also contributed to our decision to use the Java language (Sun Microsystems, 2006). Because of Java's lack of pointers and built in memory management, it is much easier to stay bug free (Liang, 1999; Gosling and McGilton, 1996; Sierra and Bates, 2005). Finally, the benefits of developing the software with Java far outweighed the negative.

3.2. Overview of the class diagram. Overall, the program is broken up into four distinct parts. These four parts can be seen in Fig. 2.

- `InputGui`: responsible for retrieving user input and sending it to InputObject.
- `InputData`: responsible for translating the user input into the NTG format.
- `NativeInputData`: a simple data structure which holds the processed input data. All data in this object are ready for NTG execution.
- `NativeCaller`: responsible for connecting to the native code (in this case, NTG) through JNI and returning the results.

`InputData` has a reference to `NativeInputData`. A fifth object, the Controller, is also shown in Fig. 2. This object is responsible for controlling the overall program flow and the communication with the native library, `libJ2NTG`. `InputGui`, `InputData`, and `NativeCaller` are contained within the Controller object, which is also composed of the top-level window (`JFrame`) and the menu bar (`JMenuBar`.)[1]

[1] `JFrame` and `JMenuBar` are standard components in Java's Swing GUI toolkit.

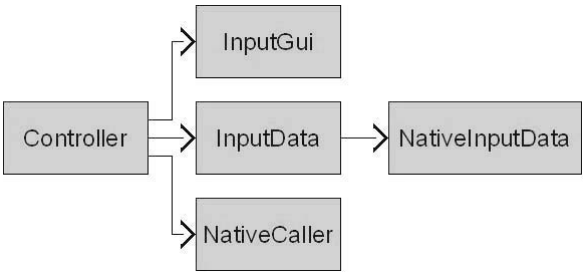

Fig. 2. NTGsim simplified class diagram.

3.3. Modularization. As discussed earlier, we wanted to keep the top-level objects as modular as possible in order to aid in future extension to the program, such as swapping out the back end algorithm, NTG, for another trajectory generation algorithm. In order to accomplish this, the strategy design pattern was used (Freeman and Freeman, 2004). The three most important objects, `InputGui`, `InputData`, and `NativeCaller`, implement an interface which corresponds to their family. For instance, `NtgInputData` implements the `InputData` interface; `NtgCaller` implements the `NativeCaller` interface, and so on. The Controller object only references the interfaces of the classes and not the concrete classes directly. This allows the creation of families of objects which all implement the same interface and can be used interchangeably even though they exhibit different behavior. An example class diagram of this design pattern can be seen in Fig. 3[2]. For instance, if we wanted to interface NTGsim with a different trajectory generation algorithm, we would merely create an object which implements the `NativeCaller` interface and calls the new algorithm, and then provide this new object as `NativeCaller` for the Controller object.

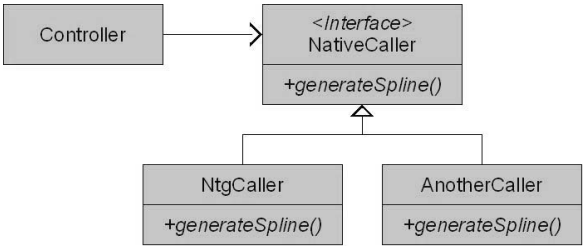

Fig. 3. Strategy design pattern class diagram.

Another scenario would be if we needed to accept input from a sensor instead of a human user. An example of a sensor system could be a Global Positioning System (GPS) used for tracking a vehicle. In this case we would create a new `GpsInputGui` object that implements the `InputGui` interface and communicates with a sensor instead of a human user. The rest of the system would not need to change. By creating modular objects, we have isolated potential change into separate compartments, preventing the change in one object from affecting another object.

To further help modularization, the abstract factory design pattern was used to encapsulate the instantiation of the main objects (Freeman and Freeman, 2004). A Factory interface was created which has methods to create all three of the main objects. An `NtgFactory` subclass creates objects specifically for the current version of NTGsim. For instance, when `createInputData()` is called on `NtgFactory`, it instantiates an object of the class `NtgInputData` and returns the newly created object. Figure 4 shows the class diagram of the factory design pattern. However, all the methods take interfaces as input parameters and likewise return interfaces. This means the `createInputData()` method returns an object which implements the `InputData` interface; the `createInputGui()` method returns an object which implements the `InputGui` interface, and so on. By returning interfaces, the Controller object only needs to know about the three different interfaces and not the exact concrete subclass which it will be using. Because the object creation process is encapsulated, the only code which needs to be changed when using a different object, as with the two examples above, is the concrete factory code. Instead of using `NtgFactory`, the programmer would create a new `Factory` which creates the appropriate objects.

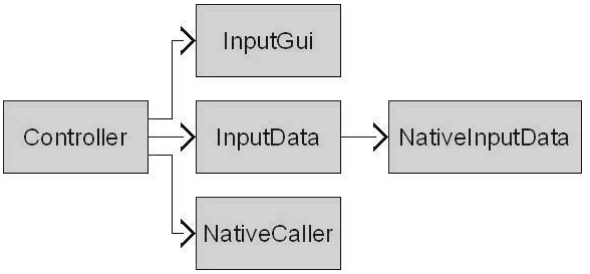

Fig. 4. Abstract factory design pattern class diagram.

3.4. InputGui. `InputGui` is the interface between the human user and the NTGsim program. The `InputGui` object gets user input and sends it to `InputObject`. It contains all the input widgets[3] used in the GUI as well as the listener objects which respond to the user-triggered widgets' events. `InputGui` registers the listeners on all the widgets. When the user inputs data, the affected widget responds by notifying all

[2]The return types and input parameters of the methods were omitted due to space constraints.

[3]A widget is a term used to describe a GUI component with which the user interacts. Some examples of widgets are buttons, menus, and combo-boxes.

the objects listening to it. This system is modeled after the observer design pattern (Freeman and Freeman, 2004). The widget objects are the subjects and the listener objects are the observers. This can be seen in Fig. 5, where `WidgetA` classes trigger `ActionEvents`[4], `WidgetB` classes trigger `ChangeEvents`, `WidgetC` classes trigger `ItemEvents`, and so on.

NTGsim uses over 40 widgets. With so many widgets, a special system was needed to uniformly handle all of the widgets' different events. Four elements were used to ease this problem:

- All the widgets were extended to implement a custom interface called `DataPointable`. This interface has three methods: `updateInputData()`, `redisplay()`, and `setDataPointer()`. Having a common interface allowed the widgets to be treated polymorphously.

- A listener class, called `MultiChangeListener`, as seen in Fig. 5, was created which can listen for any event triggered by the GUI widgets as long as the registered widgets implements the `DataPointable` interface. With this class, all events are handled in the same way: by calling `updateInputData()` on the event's source (the `DataPointable` widget which triggered the envent).

- All widgets were placed in `HashMap`[5] immediately after creation. The `HashMap` key values were based on an Enumeration which had one concise, descriptive value for each widget. This allowed the retrieval of specific widgets from `HashMap`.

- Each widget contained a reference to `DataPointer`, which will be discussed in greater detail in the `InputData` section.

The three methods in the `DataPointable` interface have very simple purposes. `updateInputData()` will pass the new user input to the `InputData` object, `redisplay()` will redisplay the widget with the current values taken from the `InputData` object, and `setDataPointer()` sets a reference to the supplied `DataPointer`.

Putting the widgets in `HashMap` allowed the widgets to be treated as a collection and also allowed `InputGui` to perform operations over the entire collection using a minimal amount of code. This is made possible by Java implementation of the iterator design pattern (Freeman and Freeman, 2004). To make things even easier, the "for each" construct was used (for (Element e: `Collection<Element>`)). This even eliminates the need to explicitly transform the collection into `Iterator`. For example, registering all the widgets with `MultiListener` was a simple matter of using the "for each" construct to loop through all the elements and call `element.add(multiChangeListener)` on each element. This system allowed new widgets to be added to the GUI at any time. As long as the widgets implemented the `DataPointable` interface, virtually no code changes were needed except to update the keyset.

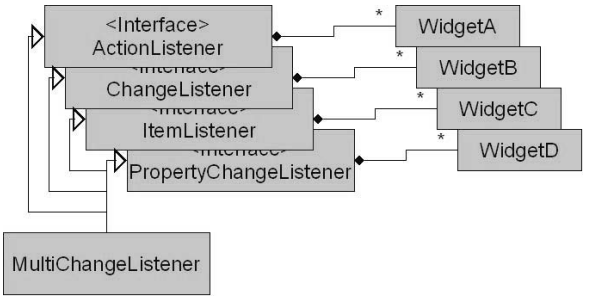

Fig. 5. Modified observer design pattern class diagram.

3.5. InputData. As mentioned earlier, the `InputData` object takes the user input retrieved by the `InputGui` object, translates it into the format expected by the NTG algorithm, and stores it in the `NativeInputData` object. However, as previously stated, NTGsim has over 40 widgets. That means over 40 different input types. More explicitly, it means that `InputData` needs to understand over 40 different input messages and how to handle and translate each one. To deal with this problem, a complementary system to the one used with InputGui was created. In this system, each key in the keysets created for the widgets' HashMap has a corresponding `DataPointer` object in `InputData`.

A `DataPointer` object is an inner-class of to `InputData` which implements the `DataPointer` interface. Having all the inner-classes implementations, the same interface makes them polymorphic and much easier to manage. The `DataPointer` interface, like the `DataPointable` one, is very simple. It has only two methods, `setData()` and `getData()`. The `setData()` method translates the user input to the NTG format and places it in the `NativeInputData` object. It also updates all dependent parameters in `NativeInputData`. The `getData()` method retrieves the pertinent data from `NativeInputData`, translates them into a form useful to the user, and returns the new data.

By using the `DataPointer` and `DataPointable` interfaces, the widgets become loosely coupled with their correspond-

[4]Event objects are used messenger objects. They are the input parameters to the functions invoked on the observer object when an important event occurs.

[5]`HashMap` is the standard key/value pair collection in the Java language.

ing DataPointers. This allows widgets and DataPointers to be changed, added, and removed without affecting the other objects. It also simplifies programming the GUI since each widget/DataPointer combo can be dealt with one at a time without worrying about breaking the preexisting code.

3.6. NativeInputData. This object contains all the input parameters needed by the NTG algorithm in the format NTG expects. It has very few methods. It is intended to be used as a "dumb" data structure as opposed to being a "first class object." All of the access modifiers of its members are public to allow easy access by the InputData object. However, InputData keeps its NativeInputData object marked as private. This way, InputData is the only object who has access to NativeInputData and all of its variables.

3.7. NativeCaller. The purpose of NativeCaller is to connect with the native code and pass it NativeInputData. NativeCaller uses the Java native interface to link the program with a dynamic library, libJ2NTG.so, containing the NTG algorithm. To accomplish this, NativeCaller defines a native method—with the "native" qualifier and no body (the body of the method is implemented in C.) By using the "native" key word, the Java compiler knows that this method will be implemented in C or C++ and will not generate any errors due to its missing body (Liang, 1999).

3.8. Multithreading. NTGsim always invokes the NativeCaller's method on a separate thread. This is to prevent the GUI from becoming unresponsive. To understand why this is, one must understand how threading works with Java's Swing. The Event Dispatch Thread (EDT) is the thread on which all the GUI related activities occur such as rendering the GUI and executing Events (Davison, 2005). By running all GUI activities on a single private thread, the GUI can operate consistently and safely. However, if a method with significant computational demand is executed on the EDT, the thread will become tied up with the said method and will be unable to handle user triggered events. If the computation time of the method is short, the delay will be unnoticeable. However, if the delay is long, on the order of 100 ms, then the user will notice the lag in the GUI (Davison, 2005). Therefore, in order to keep the GUI responsive, computationally intensive tasks, such as calling the NTG algorithm, should be executed on a separate thread other than on the EDT. Whenever the user requests the Controller to run the NTG algorithm (remember, this occurs on the EDT), the Controller creates special SwingWorker which invokes the NativeCaller's method. SwingWorker is an object which runs on a separate worker thread apart from the EDT. With this implementation, the NTGsim GUI remains responsive even though another thread is busy calculating a new trajectory with NTG.

3.9. libJ2NTG. libJ2NTG.so is the dynamic library which houses the implementation of the native method as well as the static NTG library and all the static libraries on which NTG depends. This is illustrated in Fig. 6. Also shown in Fig. 6 is the fact that libJ2NTG.so depends on yet another dynamic library, libDynamicFunc.so. This secondary dynamic library contains the implementation of the six user defined functions that NTG calls (three cost functions and three nonlinear constraint functoins.) libJ2NTG.so is static and never needs to be changes. However, the six user defined functions may change with different constraints and objectives. By creating a separate dynamic library, libDynamicFunc.so, for the dynamic parts, we have effectively encapsulated the changing part of the native code. Therefore, if the user wants to change the trajectory cost function, he or she need only open and edit libDynamicFunc.so and then recompile it. He or she does not need to touch libJ2NTG.so whatsoever. The end result is a greatly simplified function editing process.

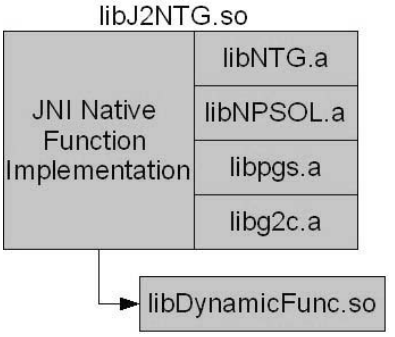

Fig. 6. Native library structure.

As mentioned earlier, libJ2NTG.so implements the native method in C. The native method, genSpline(), receives NativeInputData as an input parameter. It starts by retrieving each of the parameters to the NTG algorithm from NativeInputData. In the case when the parameter is a primitive array, the method must tell the JVM to lay out the array's elements contiguously and to pin down the array in its memory. This guarantees that the JVM will not move around the array or any of the array's elements until the method releases the array. Doing this allows the native method to perform pointer arithmetic. Next, the native method passes all the new parameters to the NTG algorithm. Once the algorithm completes execution, the native method releases the primitive arrays and returns the NativeOutputData object.

4. Simulator

While the main GUI was created to assist the user in getting input into the NTG algorithm, the Simulator component was designed with a complementary goal in mind. The purpose of the Simulator is to assist the user in understanding and viewing the output generated by the NTG algorithm. To accomplish this task, the Simulator actually draws the trajectories in three dimensions (3D) and animates them over time. In order to understand how this is done, one must first understand how NTG generates output and how NTGsim processes this output on the way to being rendered by the Simulator.

4.1. Output data pipeline. The NTG algorithm provides as output an array of double precision floating point numbers which represent the values of the control points (also called coefficients) of the B-spline trajectories. It is very difficult to just look at these numbers and understand exactly what they mean. Instead of leaving the user with this array of numbers as the final output, NTGsim performs a number of steps to make the output trajectories more meaningful to the user. These steps form a pipeline from the NTG algorithm to the simulator as seen in Fig. 7.

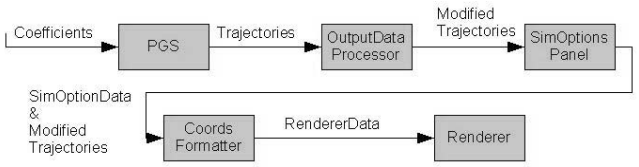

Fig. 7. Output data pipeline.

After the NTG algorithm finishes execution, the `libJ2NTG.so` library converts the control points produced by NTG into the actual values of the trajectories. It relies on the PGS static library to accomplish this task. Once complete, the output trajectory values are sent back across the JNI to the `Controller` object. The `Controller` object then sends the trajectory values to `OuputDataProcessor`, which will scale the time values according to the optimized time if the user has selected to optimize time. When the user tells NTGsim to simulate the trajectories, the modified trajectory values are passed to the Simulator. The Simulator then takes the modified trajectories and gives them to `SimOptionsPanel`, which gathers input related to the simulation from the user. From this information, it creates a `SimOptionsData` object and passes it back to the Simulator, which then takes the modified trajectories and the `SimOptionsData` and hands them over to the `CoordsFormatter` object. This object uses the data to create `RendererData`, which the Renderer object will use to draw the trajectories onto the screen.

4.2. Hardware acceleration. As stated earlier, the Simulator will animate the trajectories in 3D over time. This means that the 3D coordinates must be transformed from their 3D model space to the world coordinate system and then projected onto the 2D monitor screen. Since it will be animating, this must occur several times per second. Doing all these geometric transformations can be very computationally intensive. If run on a software renderer, the Simulator would consume a great deal of CPU cycles and system memory. Instead, it was decided to hardware accelerate the 3D rendering. This means that the Graphics Processing Unit (GPU) on the machine will do the rendering instead of the CPU. Today's GPUs are highly specialized processors with multiple parallel processing units especially designed to do 3D coordinate transformations and pixel rasterization operations. Therefore, hardware accelerating the rendering will not only free up the CPU to do GUI processing activities, giving the impression of a more responsive application. However, in order to access the GPU, an API is needed to communicate with the GPU's driver. OpenGL is the industry standard for cross-platform 3D graphics APIs. However, NTGsim, and by extension, the Simulator, are written in Java by utilizing the JOGL API. The JOGL provides Java with a one to one mapping of OpenGL commands through the JNI. The results of using the JOGL are impressive. The Simulator is very responsive and is always able to animate in real-time.

4.3. Interfaces. The Simulator user interface was designed to be as user friendly as possible. All the major commands were mapped to the mouse buttons. The user can easily rotate the 3D graph by holding down the right mouse button and dragging the mouse in the direction he or she wishes to rotate. Using the middle mouse button, the user can zoom in and zoom out the graph. The user can also pan the graph left, right, up, or down by holding down the middle mouse button and dragging the mouse in the direction he or she wishes to pan. The simulator also has key bindings to allow the user to look directly down the x, y or z-axis. These functions are mapped to the x, y, and z keyboard keys. Since the Simulator uses parallel projection, looking down one of these axes will give the impression of a 2D scene. An additional function has been mapped to the r key. This is the "Reset" function. When the r key is pressed, the coordinate system will be reset to its default values, centering the graph in the middle of the screen.

A mini-GUI was created to obtain the user's desired simulation setup. This GUI, called `SimOptionsPanel`, can be seen in Fig. 8. The first column of check-boxes allows the user to enable or disable each of the three axes. The second column allows the user to select which active variable is desired to be plotted over the corresponding axis. The third column

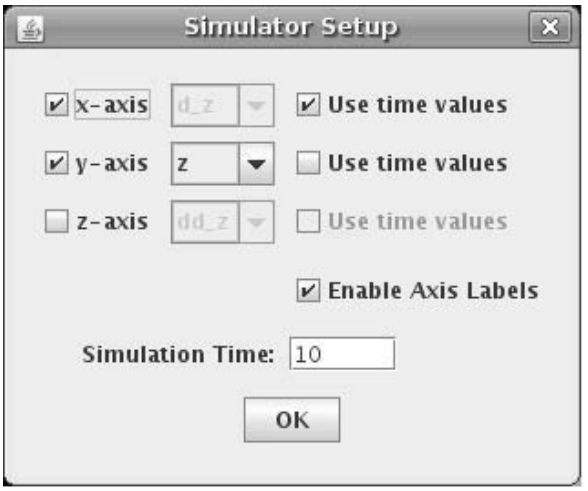

Fig. 8. SimOptionsPanel used to obtain the desired Simulator options from the user.

gives the user the option to plot the time values over the corresponding axis instead of an active variable. The *Enable Axis Labels* check-box toggles the labels and tick mark values on or off. The *Simulation Time* text field allows the user to specify a simulation time. The actual simulation time will be scaled to meet the user specified simulation time. This option is present for situations where the optimal control problem dictates an unreasonable simulation time (such as a time in microseconds or hours.) The final widget, the "OK" button, simply closes `SimOptionsPanel` and launches the Simulation window with the user specified options.

5. Installation

The following explains how to install NTGsim:

- Download and install Java SE Runtime Environment Version 6 (JRE 6) for the i586 architecture available on `http://java.sun.com/javase/downloads/index.jsp`. Instructions for downloading and installing the JRE can also be found at the above website.
- Download the NTGsim zip file from the University of Louisville website (will be available at `http://arcs.louisville.edu`).
- Unzip the file in any directory.
- To run NTGsim from the command line, go to the directory in which you unzipped NTGsim and type `./<PATH_OF_JRE>/java-jarNTGsim/dist/NTGsim.jar`.
- Replace `<PATH_OF_JRE>` with the file path of the `JRE` java binary. If the path is already set as an environment variable, it is not necessary to include it in the command.

Please ensure that you download and install the 32-bit version (i.e., i586) of the JRE and not the 64-bit one. NTGsim is built on 32-bit native libraries and will not work with a 64-bit JRE unless the libraries are recompiled from source to 64-bit. Note that NTG is freely distributed software. However, the NPSOL library on which NTG depends is commercially licensed by Stanford Business Software Inc. For more details, visit `http://www.sbsi-sol-optimize.com/asp/sol_npsol.htm`.

6. Operation and an example

This section of the paper will explain how to operate NTGsim and give an example by working through the van der Pol oscillator problem with NTGsim. The following defines the van der Pol oscillator problem:

$$\min \int_0^5 (x_1^2 + x_2^2 + u^2)\,\mathrm{d}t$$

subject to the dynamics

$$\begin{aligned} \dot{x}_1 &= x_2, \\ \dot{x}_2 &= -x_1 + (1 - x_1)^2 x_2 + u, \end{aligned}$$

and constraints

$$\begin{aligned} x_1(0) &= 1, \\ x_2(0) &= 0, \\ -x_1(5) + x_2(5) &= 1. \end{aligned}$$

The problem can be reduced to that with one unknown, $z(t) = z_1(t)$, and the optimization problem in terms of $z(t)$ is

$$\min_{z(t)} \int_0^5 \left[z^2 + \dot{z}^2 + \{\ddot{z} + z - (1 - z^2)\dot{z}\}^2\right] \mathrm{d}t$$

subject to the constraints

$$\begin{aligned} z(0) &= 1, \\ \dot{z}(0) &= 0, \\ -z(5) + \dot{z}(5) &= 1. \end{aligned}$$

The first step is to define the cost and nonlinear constraints functions. By default, these functions are set for the van der Pol example. However, to change the cost and nonlinear constraints functions, first, navigate to the `/DynamicFuncNTG` folder in the directory in which you unzipped NTGsim. Next, open the `DynamicFuncNTG.c` file in a text editor and modify the functions as needed. Then recompile the `DynamicFuncNTG` library using the included makefile

located in the current directory. If you wish to use the helper function, save the implementation in the file `DynamicFuncNTG.c` and be sure to include its prototypes in the file `DynamicFuncNTG.h`.

The next step is to enter the static parameters. The static parameters are divided into five different categories: Output Variable Data, Spline Data, Cost Function Data, Linear Constraints Data, and Nonlinear Constraints Data. Navigate between these different sections by clicking on the desired tab to the top of the window pane. In order to allow the user to enter the parameter values, the NTGsim interface has three different widgets: checkboxes, combo-boxes, and text-fields. Click on a checkbox to select it. Click a second time to deselect it. For combo-boxes, clicking on the box will expose a drop-down menu of possible choices. Click on the desired selection. Text fields allow the user to directly enter a value. Click on the text field and the mouse pointer will transform into a text cursor inside the text field. Enter the desired value by typing on the keyboard. Press "Enter" or select a different field to commit the value. Note that if you click on the menu before committing the value in a text field, the value will not be stored as a parameter because NTGsim has no way of knowing if you have finished editing the field or not.

The van der Pol parameters can be directly loaded into NTGsim by using the "Load Presets" option from the "File" menu. To use the van der Pol presets instead of manually entering the van der Pol settings, download the `vanderpole.ntg` file from the University of Louisville website. Select "Load Presets" from the "File" menu. A file browser window will open. Navigate to the directory in which you downloaded the `vanderpole.ntg` file and click "Open". The van der Pol settings will be automatically loaded into the program. At any point in time, the current state of all NTGsim parameters can be printed using the console by selecting "Debug Info" from the "File" menu.

Finally, select "Run NTG" from the "File" menu. NTG will run and its output will be displayed in the console window from which you launched NTGsim. Also, a text file containing the coefficients of the spline will be saved in the same directory as the NTGsim folder under the name `coeff1.txt`. Selecting "Simulate Trajectory" from the "Actions" menu will open the Simulator and allow the user to view the trajectories in 3D and animate them over time.

When working on any project, if the user wishes to save the current settings, he or she can select "Save Presets" from the "File" menu. A file browser will then pop up and the user can select the directory and the file name of the file to save. Click on "Save" when finished. By convention, NTGsim files end with ".ntg". In order to give the user ultimate control in naming his or her save files and to avoid conflicts with other programs, the above convention is not strictly enforced by NTGsim. When finished using NTGsim, click on the "×" in the top-right corner of the GUI. Figure 9 shows a screenshot of the GUI. Figures 10–12 show the simulated trajectory and some of the capabilities of the simulator presented in Section 4.

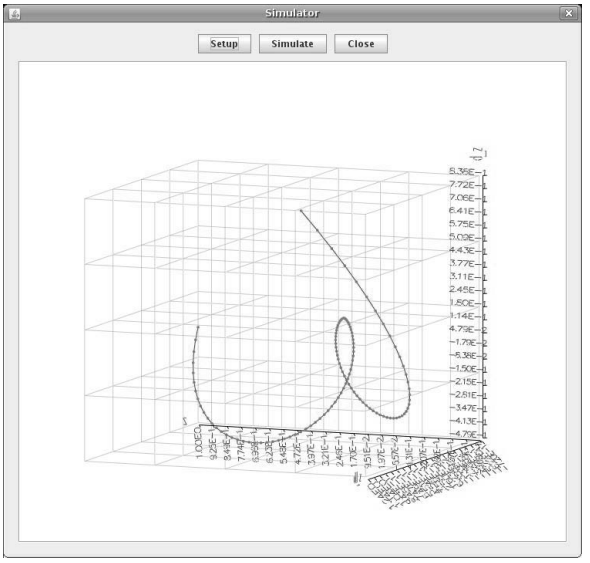

Fig. 10. Simulator showing a rotated view of the trajectory in 3D.

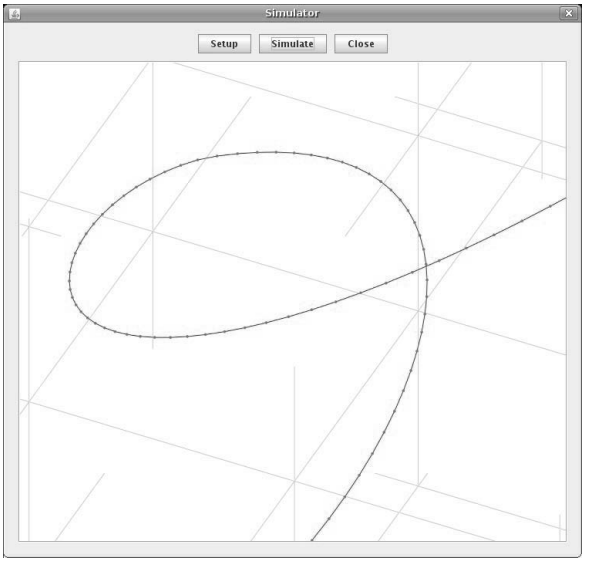

Fig. 11. Zoomed-in view of the 3D simulated trajectory.

7. Conclusion

Through the work presented here, we have greatly simplified the use of NTG. By providing a GUI for the NTG algorithm, NTGsim has given the end user an intuitive and efficient way of altering NTG's static parameters. Also, by

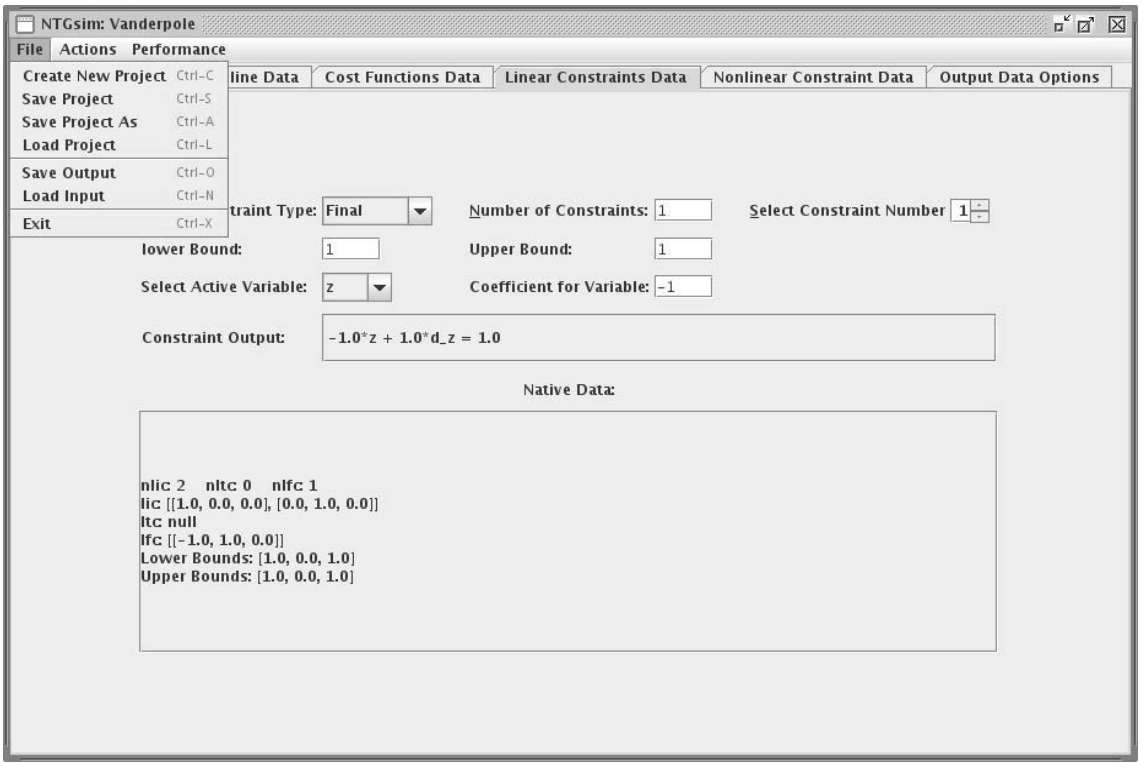

Fig. 9. NTGsim screenshot showing the linear constraints.

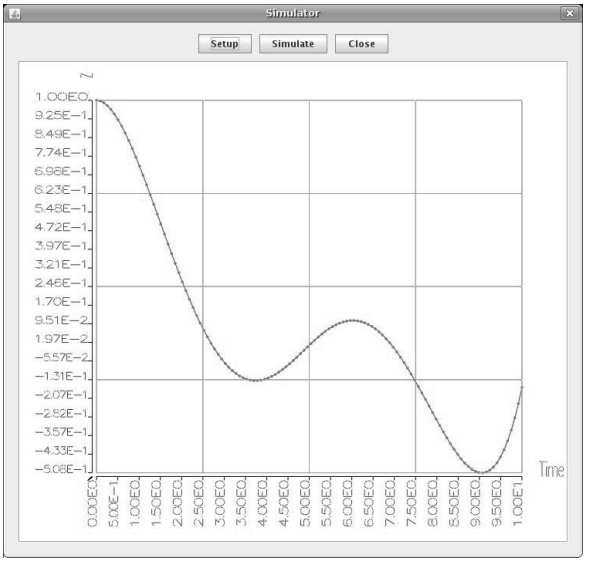

Fig. 12. Simulator viewed down the Z-axis showing a 2D plot.

segregating the three cost functions and three nonlinear constraint functions into a separate dynamic library, we have made changing the aforementioned functions much more straightforward since now these functions are not hidden deep within a mess of code. The Simulator gives the user access to a 3D visualization of the output and also animates the trajectories over time. With the inclusion of the Simulator, the user no longer needs to depend on third party applications to render the trajectories, such as Matlab or Techplot. The next step in making the NTG algorithm more user-friendly is to incorporate the dynamic function editing process directly into the GUI and streamline the entire process.

References

Bhattacharya, R. (2006). Optragen: A Matlab toolbox for optimal trajectory generation, *Proceedings of the 45th IEEE Conference on Decision and Control, San Diego, CA, USA*, pp. 6832–6836.

Bryson, A.E.J. and Ho, Y.C. (1975). *Applied Optimal Control: Optimization, Estimation and Control*, Taylor and Francis, Levittown, PA.

Davison, A. (2005). *Killer Game Programming in Java*, O'Reilly Media Inc., Sebastopol, CA.

de Boor, C. (2001). *A Practical Guide to Splines*, Springer-Verlag, New York, NY.

Freeman, E. and Freeman, E. (2004). *Head First Design Patterns*, O'Reilly Media Inc., Sebastopol, CA.

Gill, P.E., Murray, W., Saunders, M. and Wright, M. (n.d.). *NPSOL—Nonlinear Programming Software*, Stanford Business Software Inc., Mountain View, CA.

Gosling, J., Joy, B., Steele, G. and Bracha, G. (2005). *The Java Language Specification*, Prentice Hall PTR, Englewood Cliffs, NJ.

Gosling, J. and McGilton, H. (1996). Original Java whitepaper, `http://java.sun.com/docs/white/langenv/`.

Inanc, T., Misovec, K. and Murray, R.M. (2004). Nonlinear trajectory generation for unmanned air vehicles with multiple radars, *Proceedings of the 43th IEEE Conference on Decision and Control, Atlantis, Paradise Island, Bahamas*, pp. 3817–3822.

Milam, M. (2002). Missile interception research report, *California Institute of Technology Internal Report*, `http://www.cds.caltech.edu/~milam/research/res.htm`.

Milam, M.B. (2003). *Real-Time Optimal Trajectory Generation for Constrained Dynamical Systems*, Ph.D. thesis, California Institute of Technology, Pasadena, CA.

Milam, M., Mushambi, K. and Murray, R. (2000). A new computational approach to real-time trajectory generation for constrained mechanical systems, *Proceedings of the 39th IEEE Conference on Decision and Control, Sydney, Australia*, pp. 845–851.

Misovec K., Inanc T., J.W. and Murray, R.M. (2003). Low-observable nonlinear trajectory generation for unmanned air vehicles, *Proceedings of the 42nd IEEE Conference on Decision and Control*, *Maui, HI, USA,* pp. 3103–3110.

Muezzinoglu, M. K. and Inanc, T. (2006). Trajectory generation in guided spaces using artificial neural networks and ntg algorithm, *Proceedings of the American Control Conference, Minneapolis, MN, USA*, pp. 5776–5781.

Lian, F.-L. and Murray, R. (2003). Cooperative task planning of multi-robot systems with temporal constraints, *Proceedings of the International Conference on Robotics and Automation, Taipei, Taiwan*, pp. 2504–2509.

Liang, S. (1999). *Java Native Interface: Programmer's Guide and Specification*, Prentice Hall PTR, Englewood Cliffs, NJ.

Sierra, K. and Bates, B. (2005). *Head First Java*, O'Reilly Media Inc., Sebastopol, CA.

Sun Microsystems, I. (2006). Java se 6 api javadocs, `http://java.sun.com/javase/6/docs/api/`.

Trapani, L. J. D. and Inanc, T. (2009). Ntgsim: A graphical user interface for the nonlinear trajectory generation algorithm, *Proceedings of the American Control Conference, ACC 2009, St. Louis, MO, USA*, pp. 402–407.

Lyall Jonathan Di Trapani completed a B.Sc. and an M.Eng. in electrical engineering at the University of Louisville in 2006 and 2007, respectively. Both degrees were awarded with the highest honors. During his time at the University of Louisville, he earned the Air Force ROTC Distinguished Graduate Award and the Speed School Distinguished Graduate Award. He is currently pursuing an M.S. in computer science from Southern Methodist University. He has been serving in the US Air Force as a communications engineer since 2007.

Tamer Inanc received his B.Sc. degree from Dokuz Eylul University, Izmir, Turkey, in 1991, and his M.Sc. and Ph.D. degrees from Pennsylvania State University, University Park, USA, in 1996 and 2002, respectively. Between 2002 and 2004, he was a postdoctoral scholar at the California Institute of Technology, Pasadena, USA. He joined the University of Louisville in 2004 as an assistant professor in the Electrical and Computer Engineering Department. He won a merit scholarship in 1993 from Turkey, a travel grant award from the IEEE in 1999, the 2006 Kentuckiana Metroversity Award for Instructional Development and the 2008 Innovations in Technology for Teaching and Learning Awards, University of Louisville.

Received: 31 July 2009
Revised: 24 October 2009