2009 American Control Conference
Hyatt Regency Riverfront, St. Louis, MO, USA
June 10-12, 2009

NTGsim: A Graphical User Interface for the Nonlinear Trajectory Generation Algorithm

Lyall Jonathan Di Trapani and Tamer Inanc

Abstract— **Nonlinear Trajectory Generation (NTG), developed by Mark Milam et. al. [1], is a software algorithm used to generate trajectories of constrained nonlinear systems in real-time. The goal of this project is 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 the NTG. This new program, called NTGsim, eliminates the need to reprogram the NTG algorithm explicitly each time the user wishes to change a parameter.**

I. Introduction

Nonlinear Trajectory Generation (NTG) developed by Mark Milam et. al. [1] solves constrained nonlinear optimal control problems in real–time. It is based on a combination of nonlinear control theory, spline theory, and sequential quadratic programming [2]. NTG takes the optimal control problem formulation, characterization of trajectory space, and the set of collocation points, and transforms them into an NLP problem. It is then solved using NPSOL, a popular NLP solver, which uses Sequential Quadratic Programming (SQP) to obtain the solution [3].

NTG has been applied to several robotics problems in the literature. In [4] and [5] NTG is used to generate low-observable trajectories for unmanned aerial vehicles. In [6], NTG is used for a missile intercept problem.

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's parameters, constraints and cost functions. If the user whishes to make any changes to his or her optimal control problem, such as alter a spline parameter or modify the cost functions, the user 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.

In this paper we present *NTGsim*, the solution we have developed. 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 support in specifying and solving optimization problems with NTG. Recently, a solution to the problem is proposed to facilitate the use of NTG called OPTRAGEN, a MATLAB toolbox for NTG developed by Bhattacharya et. al. [16]. 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 the NTG which was designed to solve the constrained nonlinear optimal control problems in real-time. On the other hand, our proposed solution NTGsim is based on Java platform and provides real-time application and 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, the reader will be given a brief overview of the NTG algorithm. In the next section, NTGsim will be broken down into its major components and concepts and each one will be discussed. Following that, full installation instruction for the NTGsim will be provided. The final section will give a brief example on how to use NTGsim.

II. Overview of NTG

In order for the user to manipulate NTG, the user must define the constrained nonlinear optimal control problem of interest explicitly in the NTG framework [1]. For such as system to be fully qualified, the NTG algorithm has 46 input parameters [1]. These parameters can be broken up into two different types: *static parameters* and *dynamic functions*.

The static parameters define the number of outputs (splines) and the number of derivatives for each output, the number of cost and constraint fuctions, the placement of knot points, the order and smoothness of the B-splines, and the collocation points for each output; basically, any parameter that can be assigned a discrete value(s.) 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, *the dynamic functions*, comprises the six remaining input parameters. Half of these are the three cost functions-initial, trajectory, and final-

Lyall Jonathan Di Trapani is a Master of Engineering student at the University of Louisville, Louisville, KY 40292 USA (ljditr01@louisville.edu).

Tamer Inanc is an Assistant Professor at the University of Louisville, Louisville, KY 40292 USA (t.inanc@louisville.edu).

which describe the objective(s) of the system. The other half is the three nonlinear constraint functions-initial, trajectory, and final. These last three aptly named functions describe the nonlinear constraints of the system. The reason the six functions are labeled as "dynamic" is 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 time of this project, it was tested on x86 based computers running various Ubuntu and Mandriva Linux distributions. The NTG algorithm is dependent on three static libraries[a]; 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 [8]. 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.

III. 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, discuss each one and explain how modularization and multithreading were used.

A. Data Flow

The following is a rough overview of the data flow through the program. Each concept will be discussed in more detail at a latter time. The data flow of the program is shown in Fig. 1. First, the `InputGui` object receives input from the user. For each input received, the `InputGui` will pass the new data to the `InputData` object. The `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 done setting up the desired parameters, the Controller object will send the `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 [9]. The `libJ2NTG` takes the parameters from within the `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 JNI, to the Controller.

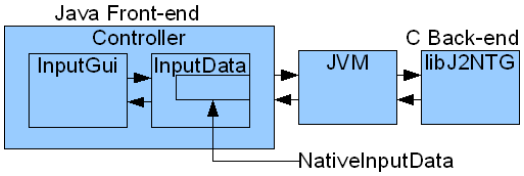

Figure 1: NTGsim Data Flow

As can be seen from the 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 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 NTGsim was not tied to a single operating system (OS). In case NTG was ported to a new operating system, we wanted to have the ability to bring NTGsim along with it to the new OS. Java's platform independent nature [10] made the language a natural fit for this project. Another goal was to create a modular GUI which could be adapted to be used with different trajectory generation algorithms in the future. The object oriented nature of the Java programming language [10], [11], [12] allowed us to easily incorporate this feature. Since the overreaching purpose of this project 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 this project was envisioned as 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 [10]. Thus, future extensions to the NTGsim in order to include an outside data source are very possible with Java's RMI.

Also, in 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 have to at least be able to keep up with the NTG algorithm. Although, in the past, the Java Runtime Environment (JRE) has suffered a 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 [10]. This is in large part due to the advent of the Hotspot Just in Time (JIT) compiler [13].

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 [13]. Another important feature of the Java platform is its ability to automatically generate program

[a] In Linux, static libraries are suffixed by the .a extension and can also be referred to as archives. Dynamic libraries are suffixed by the .so extension and sometimes referred to as shared libraries.

documentation from properly formatted comments. This huge time saver also contributed to our decision to use the Java [12].

Because of Java's lack of pointers and built in memory management, it is much easier to keep bug free [9], [10]. If we had chosen to use C or C++ we would have had the added burden of dealing with pointers and managing NTGsim's memory. Finally, the benefits of developing NTGsim with Java far outweighed the negative.

B. 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 the 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 is ready for NTG execution. `NativeCaller`: responsible for connecting to the native code (in this case NTG) through JNI and returning the results. The `InputData` has a reference to the `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`. The `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`.)[a]

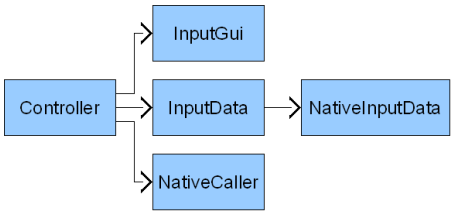

Figure 2: NTGsim Simplified Class Diagram

C. 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 [14]. The three most important objects, the `InputGui`, `InputData`, and the `NativeCaller`, implement an interface which corresponds to their family. For instance, the `NtgInputData` implements the `InputData` interface; the `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 have different behavior.

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

For example, if we wanted to interface the 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 the `NativeCaller` for the Controller object. 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 [14]. A `Factory` interface was created which has methods to create all three of the main objects[b]. An `NtgFactory` subclass creates objects specifically for the current version of NTGsim. For instance, when `createInputData()` is called on the `NtgFactory`, it instantiates an object of the class `NtgInputData` and returns the newly created object. Fig. 3 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, is the concrete factory code. Instead of using the `NtgFactory`, the programmer would create a new `Factory` which creates the appropriate objects.

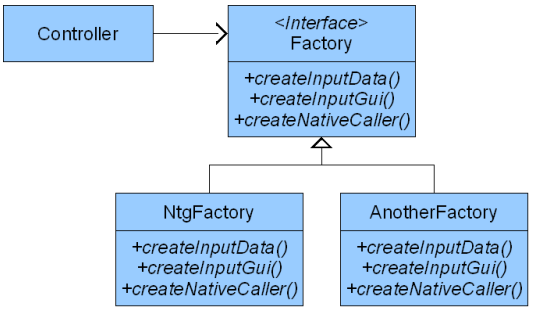

Figure 3: Abstract Factory Design Pattern Class Diagram

D. InputGui

The `InputGui` is the interface between the human user and the NTGsim program. The `InputGui` object gets user input and sends it to the InputObject. It contains all the input widgets[c] used in the GUI as well as the listener objects which respond to the user-triggered widgets' events. The `InputGui` registers the listeners on all the widgets. When

[b] The NativeInputData object was not included in the Factory because it is tightly coupled with the input data. Instead, each concrete InputData class is responsible for instantiating its corresponding NativeInputData.

[c] 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 user inputs data, the affected widget responds by notifying all the objects listening to it. This system is modeled after the observer design pattern [14]. The widget objects are the subjects and the listener objects are the observers.

NTGsim uses over 40 widgets. With this many widgets, a special system was needed to uniformly handle all of the widgets' different events. Four elements were used to ease this problem: (1) 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. (2) A special listener class, called `MultiChangeListener`, 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 the same way; by calling `updateInputData()` on the event's source (the `DataPointable` widget which triggered the envent.) (3) All widgets were placed in a `HashMap`[a] immediately after creation. The `HashMap` key values were based on an `Enumeration` which had one concise, descriptive value for each widget. This allowed for the retrieval of specific widgets from the `HashMap`. (4) Each widget contained a reference to a `DataPointer` which will be discussed in more 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`.

E. 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 the `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 the `InputGui` was created. In this system, each key in the keysets created for the widgets' HashMap has a corresponding `DataPointer` object in the `InputData`. A `DataPointer` object is an inner-class of to the `InputData` which implements the `DataPointer` interface. Having all the inner-classes implements the same interface makes them polymorphic and much easier to manage. The `DataPointer` interface, like the `DataPointable` interface, 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 the `NativeInputData`. The `getData()` method retrieves the pertinent data from the `NativeInputData`, translates it into a form useful to the user, and returns the new data.

By using the `DataPointer` and `DataPointable` interface, the widgets become loosely coupled with their corresponding `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 worry of breaking preexisting code.

F. NativeInputData and NativeCaller

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, the `InputData` keeps its `NativeInputData` object marked as private. This way the `InputData` is the only object who has access to the `NativeInputData` and all of its variables.

The purpose of the `NativeCaller` is to connect with the native code and pass it the `NativeInputData`. The `NativeCaller` uses the Java Native Interface to link the program with a dynamic library, `libJ2NTG.so`, containing the NTG algorithm. To accomplish this, the `NativeCaller` defines a native method; this is a 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 [9].

G. 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 [13]. 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 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 100ms, then the user will notice the lag in the GUI [13]. Therefore, in order to keep the GUI responsive, computationally intensive tasks, such as calling the NTG algorithm, should be executed on a separate thread

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

other than the EDT. Whenever the user requests the Controller to run the NTG algorithm (remember, this occurs on the EDT,) the Controller creates a special `SwingWorker` which invokes the `NativeCaller`'s method. A `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.

H. 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 on. This is illustrated in Fig. 4, which also shows that `libJ2NTG.so` also depends on another dynamic library, `libDynamicFunc.so`. This secondary dynamic library contains the implementation of the six user defined function that NTG calls (3 cost functions and 3 nonlinear constraint functoins.) `libJ2NTG.so` is static and never needs to be change, however, the 6 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 the `libDynamicFunc.so` and then recompile it. He or she does not need to touch the `libJ2NTG.so` whatsoever. The end result is a greatly simplified function editing process.

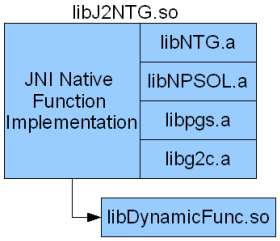

Figure 4: Native Library Structure

As mentioned earlier, `libJ2NTG.so` implements the native method in C. The native method, `genSpline()`, receives the `NativeInputData` as an input parameter. It starts by retrieving each of the parameters to the NTG algorithm from the `NativeInputData`. In the case that 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.

IV. Installation

The following section will explain how to install the NTGsim application. (1) Download and install the Java SE Runtime Environment Version 6 (JRE 6) for the i586 architecture. (2) Download the NTGsim zip file from the University of Louisville website (will be available at http://www.ece.uofl.edu/~t0inan01). (3) Unzip the file in any directory. (4) To run NTGsim from the command line, go to the directory in which you unzipped NTGsim and type `./<PATH_OF_JRE>/java -jar NTGsim/dist/NTGsim.jar`. (5) 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. 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: NTG is freely distributed software; however, the NPSOL library on which NTG depends is not. NPSOL is commercially licensed by Stanford Business Software Inc.

V. Operation And Example

The following defines the Vanderpole problem [16]:

$$min_{x(t),u(t)} \int_0^5 \frac{1}{2}(x_1^2 + x_2^2 + u^2)dt$$

subject to system dynamics

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

subject to terminal constraints

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

The problem can be reduced to a problem with one unknown, z(t) ≡ x1(t), and the optimization problem in terms of z(t) is

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

subject to terminal constraints

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

The first step is to define the cost and nonlinear constraints functions. By default, these functions are set for the Vanderpole 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 helper function, write their implementations in the `DynamicFuncNTG.c` file and be sure to include their prototypes in the `DynamicFuncNTG.h` file.

The next step is to enter the static parameters. The static

parameters are separated into 5 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 key board. Press enter or select a different field to commit the value. 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 finished editing the field or not.

The Vanderpole parameters can be directly loaded into NTGsim by using the "Load Presets" option from the "File" menu. To use the Vanderpole presets instead of manually entering the Vanderpole 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 Vanderpole settings will automatically be loaded into the program. At any point in time, the current state of all the NTGsim's parameters can be printed to 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`.

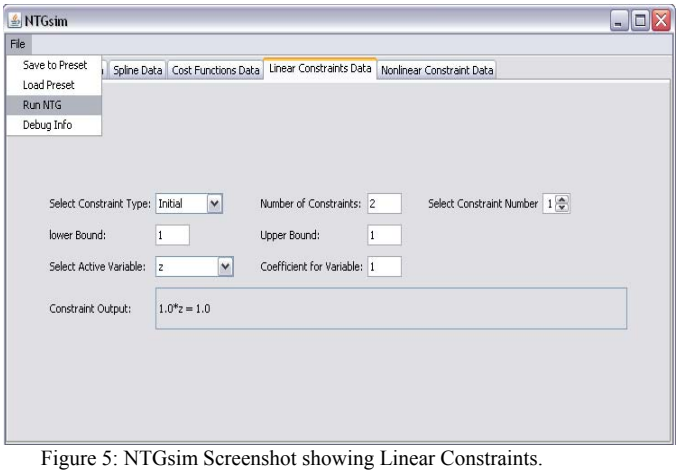

Figure 5: NTGsim Screenshot showing Linear Constraints.

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 'x' in the top-right corner of the GUI. Fig. 5 shows a screenshot of the GUI.

VI. Conclusion

Through this project, 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 segregating the 3 cost functions and 3 nonlinear constraint functions into a separate dynamic library, we have made changing the aforementioned functions much more straight forward since now these functions are not hidden deep within the code.

References

[1] M. B. Milam, "Real-time optimal trajectory generation for constrained dynamical systems," Ph.D. dissertation, California Institute of Technology, 2003.

[2] C. de Boor, *A Practical Guide to Splines*. New York: Springer-Verlag, 2001.

[3] M. K. Muezzinoglu and T. Inanc, "Trajectory Generation in Guided Spaces using NTG Algorithm and Artificial Neural Networks," Proceedings of the 2006 American Control Conf., June 14-16, 2006.

[4] K. Misovec, T. Inanc, J. Wohletz and R.M. Murray, "Low-Observable Nonlinear Trajectory Generation for Unmanned Air Vehicles," Proceedings of the 42nd IEEE Conf. on Decision and Control, December 2003.

[5] T. Inanc, K. Misovec and R. M. Murray, "Nonlinear Trajectory Generation for Unmanned Air Vehicles with Multiple Radars", 43rd IEEE Conference on Decision and Control, December 14-17, 2004.

[6] M.B. Milam, "Missile Interception Research Report," California Institute of Technology Internal Report, 2002.

[7] Lian E-L. and Murray R.M., "Cooperative Task Planning of Multi-Robot Systems with Temporal Constraints," International Conference on Robotics and Automation, 2003.

[8] P.E. Gill, W. Murray, M.A. Saunders and M.H. Wright, "User's Guide for NPSOL 5.0 A Fortran Package for Nonlinear Programming, Systems Optimization Laboratory," Stanford Univ, CA, 1998.

[9] S. Liang, *Java Native Interface: Programmer's Guide and Specification*, Prentice Hall PTR, 1999.

[10] K. Sierra and B. Bates, *Head First Java*, CA: O'Reilly Media Inc., 2005.

[11] J. Gosling, B. Joy, G. Steele, G. Bracha, *The Java Language Specification*, Prentice Hall PTR, 2005, Introduction.

[12] Sun Microsystems, Inc., "Java SE 6 API Javadocs," Available at: http://java.sun.com/javase/6/docs/api/

[13] Andrew Davison, *Killer Game Programming in Java*, O'Reilly Media Inc., 2005.

[14] Eric Freeman and Elisabeth Freeman, *Head First Design Patterns*, CA: O'Reilly Media Inc., 2004.

[15] R. Bhattacharya and P. Singla, "Nonlinear Trajectory Generation Using Global Local Approximations," Proceedings of the 45th IEEE Conference on Decision & Control, December 13-15, 2006.

[16] R. Bhattacharya, "OPTRAGEN: A MATLAB Toolbox for Optimal Trajectory Generation", Proceedings of the 45th IEEE Conference on Decision & Control, December 13-15, 2006, pg 6832-6836.