

Chase:

A MatLab Waypoint Generator For The Cal Poly Flight Simulator

by

Patrick LeBeau

Aeronautical Engineering Department

California Polytechnic State University

San Luis Obispo

August 2003

Statement of Disclaimer

As this project is the result of a class assignment, it has been graded and accepted as fulfillment of the course requirements. Acceptance does not imply technical accuracy or reliability. Any use of the information in this report is done at the risk of the user. California Polytechnic State University at San Luis Obispo and its staff cannot be held liable for any use or misuse of this project.

Page of Approval

TITLE

Chase: A MatLab Waypoint Generator For The Cal Poly Flight Simulator

AUTHOR

Patrick LeBeau

DATE SUBMITTED

August 25th, 2003

Senior Project Advisor

Signature

Department Chairman

Signature

Table of Contents

| <u>Section</u> | <u>Page</u> |
|--|-------------|
| List of Figures | v |
| Abstract | 6 |
| Introduction | 7 |
| Procedure | 8 |
| GUI Creation | 8 |
| Waypoint Generator Routine | 9 |
| Discussion of Results | 12 |
| Chase User's Guide..... | 12 |
| Known Issues & Areas For Improvement | 15 |
| Conclusions | 16 |
| Appendix A: Source Code | 17 |

List of Figures

| <u>Figure Name</u> | <u>Page</u> |
|------------------------------------|-------------|
| Figure 1: GUIDE Blank Pallet | 8 |
| Figure 2: Chase Window | 12 |

Abstract

The Cal Poly Flight Simulation Laboratory was in need of the addition of computer-controlled aircraft in its world environments to allow for handling tasks that would help evaluate the design of the piloted aircraft. A graphical user interface was created using MatLab code to generate the tabular text file of time and space coordinates required by the 3DLinx key frame navigation tool. The resulting program, Chase, is a simple, user-friendly tool allowing for the completion of this task, as well as being able to be adapted to the future needs of the sim.

Introduction

The Cal Poly Flight Simulation Lab has provided a resource for aerospace engineering students to apply their knowledge of aerodynamics, structures and controls, as well as to develop their computer programming skills. The simulator was built and is maintained by students with an interest in flight simulation. More than just a hobby, however, recent aircraft design students have used the simulator to validate and support their work for submission to the AIAA Team Undergraduate Design Competition. It has also been used to perform grant work for companies such as the Lockheed Martin Aeronautics Company in Palmdale (The Skunk Works) and NASA Ames.

The flight sim is run with a MatLab Simulink model. The aircraft controls are modeled through a table lookup system of control coefficients. The coefficients are calculated based on the methods detailed in the design texts by Roskam. The graphical world through which the pilot navigates is modeled in the 3DLinx environment.

To improve the functionality of the flight sim, this project is focused on adding additional, realistically behaving aircraft to the flight sim world. These aircraft can be tanker or fighter aircraft. This addition will serve both to increase interest in the simulator and to provide students with a handling task that could be used to evaluate various control system arrangements including stick force feedback and control surface sizing.

To make this process as easy as possible, it was decided to implement a graphical user interface (GUI) designed in the MatLab environment to generate the text file of waypoints needed to control an object in the 3DLinx graphics world. These text files can then be easily uploaded to an object in a given 3DLinx world file to create the desired result.

Procedure

Development of this code began with use of the MatLab tool GUIDE for creation of the graphical user interface (GUI). Then aircraft equations of motion were used to build a waypoint generation routine into the GUI code structure.

GUI Creation

GUI creation utilized the MatLab tool GUIDE, launched by typing `<guide>` at the MatLab command line. This launches a pallet area representative of the window that the program will launch (Fig. 1). Various interfaced can then be added to this window such as push buttons, toggle buttons, radio buttons, text fields and edit fields. Bringing up the property inspector allows the designer to set the various properties for each element of the new GUI.

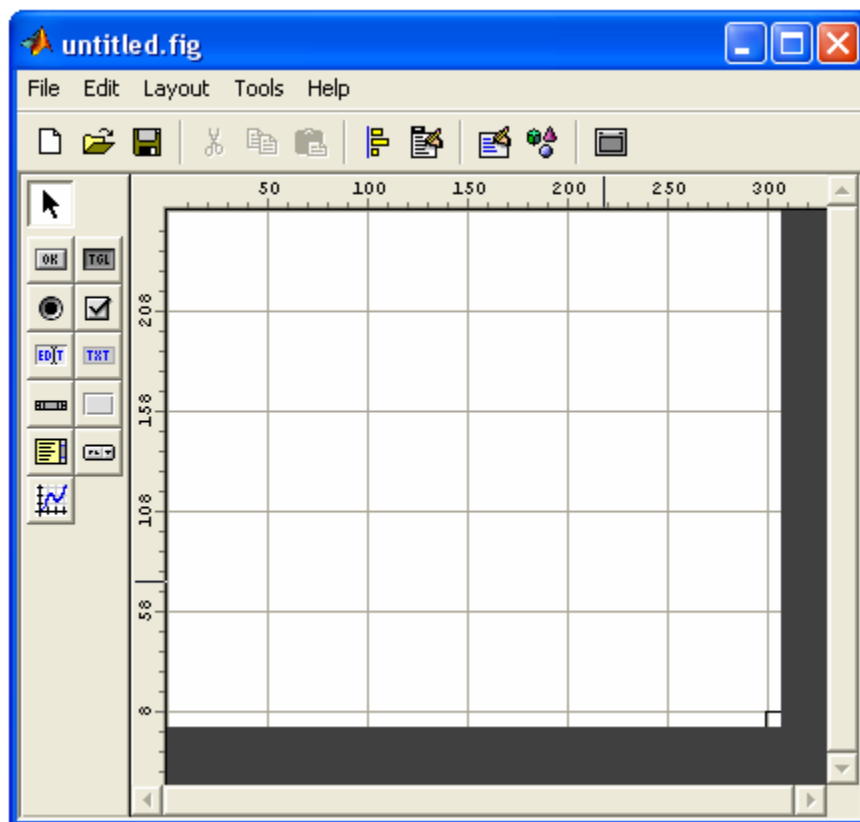


Figure 1: GUIDE Blank Pallet

Once the GUI is designed, it is saved in two parts: a *.fig file and a corresponding *.m file. Both of these files are necessary for the GUI to operate. The *.fig file stores all of the graphical information for the window and is edited by the GUIDE program. The *.m file holds all of the MatLab code that is to be executed when a user interacts with the GUI. Every time a new feature is added to the *.fig file, MatLab will automatically add a new callback to the bottom of the *.m file for the new graphical item. These callbacks function as subroutines, and will be executed every time a user performs a certain action on the graphical item, such as entering a value in an edit field or depressing a push button. For more detailed information on GUI design with GUIDE, consult the MatLab help files.

It is important to note that because of the naming system used in the *.m file, the files must retain their original names, Chase.fig and Chase.m, to function correctly. To allow for future editions of this code, a copy of the original files will be saved as ChaseV01.fig and ChaseV01.m. Those wishing to incorporate changes should work on the Chase.fig and Chase.m files and then save a copy of their version as ChaseV02, etc. An old version of the code can be reinstated by creating a copy of a ChaseVXX version and renaming it over the current Chase files.

Waypoint Generator Routine

To generate the waypoints for a specified aircraft path, the user must first identify the desired flight path in some way. Chase uses its own code to describe various maneuvers, allowing the user to piece together several different maneuvers as segments of a larger path. Chase then breaks down the path into its segments and creates the waypoints of each

individually, stacking one after the other, to generate the total path. The process of creating this path description is detailed in the *Chase User's Guide* section of this report.

All values needed across multiple callbacks or subroutines are stored in the variable `cp`, a global structure array. At any time while running the chase program, these variables may be accessed by typing in `<global cp>` at the MatLab command line. For example, `cp.ic` stores all the variables for the values of the initial conditions set by the user.

All of the calculations and value storage required to write the waypoints are performed under the `push_CreateFile_Callback` subroutine. No data is generated prior to the user clicking on the `<Create File>` push button.

The first section of code parses out the path description by segments, delimited by the semicolon `<;>` character. Each of these segments is broken down into its component pieces, delimited by the comma `<,>` character. The components are stored in the `cp.seg` array. This process is repeated for each segment.

Once the path parsing is complete, the values are used to calculate the waypoints for the described straight or turning segment. To calculate the vehicle's motion throughout the described path, equations of motion are necessary. All calculations performed for each segment use a standard aircraft body axis coordinate system defined as `x` out the nose, `y` out the right wing and `z` straight down, and all measurements are in feet or degrees. However, the 3DLinx environment has a highly non-standard coordinate system with `x` out the right wing, `y` straight up and `z` out the back of the aircraft. In addition, all measurements of distance are in meters. The more standard approach used in this program should be much easier to follow, and all values will be adjusted for use by 3DLinx once the calculations are complete.

This program is design to allow for level flight only. All aircraft equations of motion are calculated presuming that the vertical component of the lift vector is equal to the weight of the aircraft. This is used to determine the total lift the aircraft must generate at a specific bank angle, which in turn drives the lateral acceleration of the vehicle, and thus the lateral acceleration. This lateral acceleration produces a change in the velocity vector. As the total speed of the vehicle is assumed to be constant, a given lateral acceleration will produce a given yaw rotation rate of the velocity vector and thereby vehicle. The equations of motion for the aircraft are applied in finite time step increments to create waypoints in time and space.

Each segment begins by calculating a terminating condition for the segment, such as a set length of time or a set change in heading. The code will then ensure that the aircraft rolls to the proper bank angle and continues in that attitude until the termination condition is reached. When the aircraft is rolling, the average value of its bank angle over the time increment is used to calculate the total rotation. When the aircraft is turning, the average value of the heading over the time increment is used to determine the total displacement.

When the vehicle is at a fixed bank angle, level or otherwise, the time increments are set to one second. When the vehicle is rolling, these increments are reduced to one tenth of a second in an attempt to create a smooth transition from one attitude to the next.

Once all the waypoint calculations are complete. They are adjusted and reordered to match the 3DLinx world's coordinate system. The user is then asked to determine the name and location of the output text file, and the file is created, written and saved.

Discussion of Results

This project has created a graphical user interface (GUI) to aid in the creation of text files to import into the 3DLinx worlds used by the Cal Poly Flight Sim. Following are instructions for its intended use and known issues to avoid.

Chase User's Guide

To launch the program, begin MatLab and ensure that it is in the working directory or in the MatLab search path (under <File/Set Path...>) and type <chase> at the command prompt. This will bring up a GUI with multiple fields for data entry (Figure 2).

The screenshot shows a MATLAB GUI window titled "<Student Version> : Chase". The window is divided into three main sections:

- Initial Conditions:** A section with six input fields for: X (ft), Y (ft), Altitude (ft), Heading (deg), Airspeed (kts), and Bank Angle (deg).
- Flight Path:** A section containing a "Path Desc." text field, two sub-sections for "Straight" and "Turn" settings, and two "Add Segment" buttons.
 - Straight:** Radio buttons for "Time (sec)" and "Distance (ft)", with a corresponding input field for "Distance (ft)".
 - Turn:** Radio buttons for "Bank Angle (deg)", "g Loading", "Time (sec)", and "Heading Change (deg)", with input fields for "Bank Angle (deg)", "Time (sec)", and "Heading Change (deg)".
- Error Message Center:** A section with a text area displaying "No Error" and a "Clear" button.

At the bottom right of the window is a "Create Text File" button.

Figure 2: Chase Window

Begin by inputting the desired initial conditions in the first six fields for X and Y displacement, altitude, heading, airspeed and bank angle. These will determine the starting point for the aircraft, and if the 3DLinx object is set to loop, it will return to this point at the end of its flight path.

After the initial conditions are set, the user must designate a flight path for the aircraft. This description may be entered manually in the <Path Desc.> field or input through use of the <Straight> and <Turn> tools. Path descriptions are made up of several segments, each described by entering <A,B,X,C,Y;> where:

A – The letter <s> or <t>, denoting a straight or a turn

If the section is a straight, then:

B – The letter <t> or <d>, denoting a value of time in seconds or distance in feet.

X – A number value, corresponding to the state of B.

C and Y – Will be blank, resulting in an input of <A,B,X;>.

If the section is a turn, then:

B – The letter or <g>, denoting a value of bank angle in degrees or g loading.

X – A number value, corresponding to the state of B. A positive value corresponds to right turns, a negative value to left turns.

C – The letter <t> or <h>, denoting a value of time in seconds or heading change in degrees

Y – A number value corresponding to the state of C.

Note that each segment must end with a semicolon <;>, including the final segment.

To fill in the path description with the <Straight> or <Turn> tools, simply select the radio button for the desired value input, type the value into the nearby edit field and press the <Add Segment> button.

When the desired path is described, press the <Create Text File> button to build and save the waypoints in a text file.

If an anticipated error is occurred, a beep will sound and a description will appear in the <Error Message Center>. To perform the desired action, simply address the displayed message. The message center can be cleared at any time with the <Clear> button, as it will not clear itself. As such, it only displays the last error, not the current state of the GUI. Operations can still be performed while the <Error Message Center> is not cleared. Unanticipated errors will be communicated by MatLab in the main MatLab window.

Once the text file is created, it is ready to be imported into a 3DLinx model. Open a new standard exe file in Visual C. Right click on the tool bar on the left of the screen to add the 3DLinx button. Use the button to draw a 3DLinx window in the exe file. Right click in the 3DLinx window and select properties. Right click on the 3DLinx text at the top of the properties tree to load a world used by the sim.

Pick an existing entity or insert a new entity to load the waypoint file to. In the object properties window, set the object navigation to key frame type. One of the tabs under this tab will allow for importing waypoints from a file. Ensure that the input mode popup next to the input button is set to Euler (it does not look like it can scroll, but look closely to see the buttons) before inputting the file. If the program gives an error about the number of columns in the file, the text file may need to be opened in WordPad (Not Notepad) to arrange the columns properly and saved.

Once the waypoints have been successfully input, return to the first tab in the navigator area and set the value of play to <1>. The vehicle will now play through the described path. If the looping box is checked, the vehicle will play through the path repeatedly.

Trail and error is necessary to get the start and end points to match up. Because it is not assumed that a turn will be followed by a straight, the aircraft does not automatically return to level flight at the end of a turn. The time it takes for the aircraft to level off afterward will cause a slight overshoot of the desired heading change. To remedy this, it is advised to create a file with the theoretical heading changes desired, for instance 180 degrees, look at the output file to determine the overshoot by when the bank returns to zero, perhaps 181.5 degrees, and then recreate the text file undershooting the turns by the proper amount, 178.5 degrees in this theoretical case.

Known Issues & Areas For Improvement

In its current form, Chase will not handle very small inputs well. An input asking for a heading change of two degrees or less will be overshoot by the roll in to start the turn unless the turn rate is fairly small.

There is currently no capability to allow for an altitude change or any change in airspeed. This would constitute a moderate amount of changes to the program, requiring editing of both the *.m file and the *.fig file.

As is mentioned at the end of the previous section, it is tricky to capture a precise heading.

It may be desirable to allow the nose of the aircraft to separate from the velocity vector, allowing for a more natural angle of attack to be induced by a turn.

Conclusions

Chase is a user friendly GUI for generating flight path waypoints for use in the Cal Poly Flight Sim Lab. It allows a user to create a flight model for aircraft in the 3DLinx environment through very simple process. These models can be used to create handling qualities tasks for the evaluation of several different aspects of aircraft design. It allows room for future projects to add to its functionality and adapt to need that develop as the sim is upgraded.

Appendix A: Source Code

```

function varargout = Chase(varargin)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% CHASE Application M-file for Chase.fig                                %
%   FIG = CHASE launch Chase GUI.                                     %
%   CHASE('callback_name', ...) invoke the named callback.         %
%                                                                 %
% Last Modified by GUIDE v2.0 18-Aug-2003 19:35:44                   %
%                                                                 %
% Created by: Patrick LeBeau                                         %
%                                                                 %
% MODIFICATION LOG                                                  %
% Modified by: Patrick LeBeau                                       %
% Date: 8/24/03                                                     %
% Description: Finished debugging of waypoint generator routines    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if nargin == 0 % LAUNCH GUI
    fig = openfig(mfilename,'reuse');

    % Use system color scheme for figure:
    set(fig,'Color',get(0,'defaultUicontrolBackgroundColor'));

    % Generate a structure of handles to pass to callbacks, and store it.
    handles = guihandles(fig);
    guidata(fig, handles);

    if nargout > 0
        varargout{1} = fig;
    end

    % Setup global variables for value storage
    global cp;
    cp.ic.x = [];
    cp.ic.y = [];
    cp.ic.alt = [];
    cp.ic.head = [];
    cp.ic.speed = [];
    cp.ic.bank = [];
    cp.path = '';
    cp.seg = {};
    cp.waypts = [];
    cp.error='No Error';

elseif ischar(varargin{1}) % INVOKE NAMED SUBFUNCTION OR CALLBACK

    try
        [varargout{1:nargout}] = feval(varargin{:}); % FEVAL switchyard
    catch
        disp(lasterr);
    end

end

```

```

%| ABOUT CALLBACKS:
%| GUIDE automatically appends subfunction prototypes to this file, and
%| sets objects' callback properties to call them through the FEVAL
%| switchyard above. This comment describes that mechanism.
%|
%| Each callback subfunction declaration has the following form:
%| <SUBFUNCTION_NAME>(H, EVENTDATA, HANDLES, VARARGIN)
%|
%| The subfunction name is composed using the object's Tag and the
%| callback type separated by '_', e.g. 'slider2_Callback',
%| 'figure1_CloseRequestFcn', 'axis1_ButtondownFcn'.
%|
%| H is the callback object's handle (obtained using GCBO).
%|
%| EVENTDATA is empty, but reserved for future use.
%|
%| HANDLES is a structure containing handles of components in GUI using
%| tags as fieldnames, e.g. handles.figure1, handles.slider2. This
%| structure is created at GUI startup using GUIHANDLES and stored in
%| the figure's application data using GUIDATA. A copy of the structure
%| is passed to each callback. You can store additional information in
%| this structure at GUI startup, and you can change the structure
%| during callbacks. Call guidata(h, handles) after changing your
%| copy to replace the stored original so that subsequent callbacks see
%| the updates. Type "help guihandles" and "help guidata" for more
%| information.
%|
%| VARARGIN contains any extra arguments you have passed to the
%| callback. Specify the extra arguments by editing the callback
%| property in the inspector. By default, GUIDE sets the property to:
%| <MFILENAME>('<SUBFUNCTION_NAME>', gcbo, [], guidata(gcbo))
%| Add any extra arguments after the last argument, before the final
%| closing parenthesis.

% -----
function varargout = edit_ICX_Callback(h, eventdata, handles, varargin)
% Stub for Callback of the uicontrol handles.edit_ICX.

global cp;
cp.ic.x = str2num(get(handles.edit_ICX,'string'));

% -----
function varargout = edit_ICY_Callback(h, eventdata, handles, varargin)
% Stub for Callback of the uicontrol handles.edit_ICZ.

global cp;
cp.ic.y = str2num(get(handles.edit_ICY,'string'));

% -----
function varargout = edit_ICAlt_Callback(h, eventdata, handles, varargin)
% Stub for Callback of the uicontrol handles.edit_ICAlt.

```

```

global cp;
cp.ic.alt = str2num(get(handles.edit_ICAlt,'string'));

% -----
function varargout = edit_ICHead_Callback(h, eventdata, handles, varargin)
% Stub for Callback of the uicontrol handles.edit_ICHead.

global cp;
cp.ic.head = str2num(get(handles.edit_ICHead,'string'));

% -----
function varargout = edit_ICAirspeed_Callback(h, eventdata, handles, ...
    varargin)
% Stub for Callback of the uicontrol handles.edit_ICAirspeed.

global cp;
cp.ic.speed = str2num(get(handles.edit_ICAirspeed,'string'));

% -----
function varargout = edit_ICBank_Callback(h, eventdata, handles, varargin)
% Stub for Callback of the uicontrol handles.edit_ICBank.

global cp;
cp.ic.bank = str2num(get(handles.edit_ICBank,'string'));

% -----
function varargout = edit_FPDesc_Callback(h, eventdata, handles, varargin)
% Stub for Callback of the uicontrol handles.edit_FPDesc.

global cp;
cp.path = get(handles.edit_FPDesc,'string');

% -----
function varargout = edit_S1_Callback(h, eventdata, handles, varargin)
% Stub for Callback of the uicontrol handles.edit_S1.

% -----
function varargout = edit_T1_Callback(h, eventdata, handles, varargin)
% Stub for Callback of the uicontrol handles.edit_T1.

% -----
function varargout = edit_T2_Callback(h, eventdata, handles, varargin)
% Stub for Callback of the uicontrol handles.edit_T2.

% -----
function varargout = radio_STime_Callback(h, eventdata, handles, varargin)

```

```

% Stub for Callback of the uicontrol handles.radio_STime.

set(handles.radio_SDist,'value',0);

% -----
function varargout = radio_SDist_Callback(h, eventdata, handles, varargin)
% Stub for Callback of the uicontrol handles.radio_SDist.

set(handles.radio_STime,'value',0);

% -----
function varargout = radio_TBank_Callback(h, eventdata, handles, varargin)
% Stub for Callback of the uicontrol handles.radio_TBank.

set(handles.radio_Tg,'value',0);

% -----
function varargout = radio_Tg_Callback(h, eventdata, handles, varargin)
% Stub for Callback of the uicontrol handles.radio_Tg.

set(handles.radio_TBank,'value',0);

% -----
function varargout = radio_TTime_Callback(h, eventdata, handles, varargin)
% Stub for Callback of the uicontrol handles.radio_TTime.

set(handles.radio_THead,'value',0);

% -----
function varargout = radio_THead_Callback(h, eventdata, handles, varargin)
% Stub for Callback of the uicontrol handles.radio_THead.

set(handles.radio_TTime,'value',0);

% -----
function varargout = push_SAdd_Callback(h, eventdata, handles, varargin)
% Stub for Callback of the uicontrol handles.push_SAdd.

global cp;

if isempty(str2num(get(handles.edit_S1,'string')))
    cp.error = 'Input Values Must Be Numbers';
    edit_Error_Callback(h, eventdata, handles, varargin);
elseif ~get(handles.radio_STime,'value') & ...
    ~get(handles.radio_SDist,'value')
    cp.error = 'Time Or Distance Button Must Be Selected';
    edit_Error_Callback(h, eventdata, handles, varargin);
elseif get(handles.radio_STime,'value')
    cp.path = [cp.path,'s,t,',get(handles.edit_S1,'string'),''];
    set(handles.edit_FPDesc,'string',cp.path);
elseif get(handles.radio_SDist,'value')

```

```

        cp.path = [cp.path,'s,d,',get(handles.edit_S1,'string'),''];
        set(handles.edit_FPDesc,'string',cp.path);
    else
        cp.error = 'Unknown Error In Chase.m: push_SAdd_Callback';
        edit_Error_Callback(h, eventdata, handles, varargin);
    end

% -----
function varargout = push_TAdd_Callback(h, eventdata, handles, varargin)
% Stub for Callback of the uicontrol handles.push_TAdd.

global cp;

if isempty(str2num(get(handles.edit_T1,'string'))) | ...
    isempty(str2num(get(handles.edit_T2,'string')))
    cp.error = 'Input Values Must Be Numbers';
    edit_Error_Callback(h, eventdata, handles, varargin);
elseif ~get(handles.radio_TBank,'value') & ~get(handles.radio_Tg,'value')
    cp.error = 'Bank Angle or g Loading Button Must Be Selected';
    edit_Error_Callback(h, eventdata, handles, varargin);
elseif ~get(handles.radio_TTime,'value') & ...
    ~get(handles.radio_THead,'value')
    cp.error = 'Time or Heading Change Button Must Be Selected';
    edit_Error_Callback(h, eventdata, handles, varargin);
elseif get(handles.radio_TBank,'value')

    if abs(str2num(get(handles.edit_T1,'string'))) > 83.62
        cp.error = 'Bank Angle Must Be < 83.62 Degrees';
        edit_Error_Callback(h, eventdata, handles, varargin);
    else
        cp.path = [cp.path,'t,b,',get(handles.edit_T1,'string'),''];

        if get(handles.radio_TTime,'value')
            cp.path = [cp.path,'t,',get(handles.edit_T2,'string'),''];
        elseif get(handles.radio_THead,'value')
            cp.path = [cp.path,'h,',get(handles.edit_T2,'string'),''];
        end

        set(handles.edit_FPDesc,'string',cp.path);
    end

elseif get(handles.radio_Tg,'value')

    if abs(str2num(get(handles.edit_T1,'string'))) > 9
        cp.error = 'g Load Must Be < 9 g''s';
        edit_Error_Callback(h, eventdata, handles, varargin);
    else
        cp.path = [cp.path,'t,g,',get(handles.edit_T1,'string'),''];

        if get(handles.radio_TTime,'value')
            cp.path = [cp.path,'t,',get(handles.edit_T2,'string'),''];
        elseif get(handles.radio_THead,'value')
            cp.path = [cp.path,'h,',get(handles.edit_T2,'string'),''];
        end

        set(handles.edit_FPDesc,'string',cp.path);
    end
end

```

```

        end

else
    cp.error = 'Unknown Error In Chase.m: push_TAdd_Callback';
    edit_Error_Callback(h, eventdata, handles, varargin);
end

% -----
function varargout = edit_Error_Callback(h, eventdata, handles, varargin)
% Stub for Callback of the uicontrol handles.edit_Error.

global cp;

beep;
set(handles.edit_Error, 'string', cp.error);
set(handles.edit_Error, 'BackgroundColor', [1 0 0]);
pause(0.1);
set(handles.edit_Error, 'BackgroundColor', get(0, ...
    'defaultUicontrolBackgroundColor'));
pause(0.1);
set(handles.edit_Error, 'BackgroundColor', [1 0 0]);
pause(0.1);
set(handles.edit_Error, 'BackgroundColor', get(0, ...
    'defaultUicontrolBackgroundColor'));

% -----
function varargout = pushbutton_Clear_Callback(h, eventdata, handles, ...
    varargin)
% Stub for Callback of the uicontrol handles.pushbutton_Clear.

global cp;

cp.error = 'No Error';
set(handles.edit_Error, 'string', cp.error);
set(handles.edit_Error, 'BackgroundColor', get(0, ...
    'defaultUicontrolBackgroundColor'));

% -----
function varargout = push_CreateFile_Callback(h, eventdata, handles, ...
    varargin)
% Stub for Callback of the uicontrol handles.push_CreateFile.

global cp;

cp.path = get(handles.edit_FPDesc, 'string');

if isempty(cp.ic.x | cp.ic.y | cp.ic.alt | ...
    cp.ic.head | cp.ic.speed | cp.ic.bank) % Check for I.C.
    cp.error = 'All Initail Conditions Must Be Set';
    edit_Error_Callback(h, eventdata, handles, varargin);
    path_ok = 0;
elseif isempty(cp.path) % Check for path desctiption
    cp.error = 'Path Description Required';
    edit_Error_Callback(h, eventdata, handles, varargin);

```

```

    path_ok = 0;
else % Parse out flight path description segments
    path_delim = findstr(cp.path,',' );
    seg_num = length(path_delim);
    cp.seg{seg_num,5} = [];

    for i=(1:seg_num)

        if i == 1
            current_seg = cp.path(1:path_delim(i)-1);
        else
            current_seg = cp.path(path_delim(i-1)+1:path_delim(i)-1);
        end

        seg_delim = findstr(current_seg,',' );
        cp.seg{i,1} = current_seg(1);
        cp.seg{i,2} = current_seg(3);

        if current_seg(1) == 's'
            cp.seg{i,3} = str2num(current_seg(seg_delim(2)+1:end));
        elseif current_seg(1) == 't'
            cp.seg{i,3} = str2num(current_seg(seg_delim(2)+1: ...
                seg_delim(3)-1));
            cp.seg{i,4} = current_seg(seg_delim(3)+1);
            cp.seg{i,5} = str2num(current_seg(seg_delim(4)+1:end));
        end

    end

end

    path_ok = 1;
end

if path_ok
    % Generate flight path waypoints (time, x, y, z, phi, theta, psi)

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Note: The 3DLinX world used by the sim has a non-standard body axis %
    % orientation, as follows: %
    % x - out the right wing %
    % y - up %
    % z - out the tail %
    % Also, all units of distance are in meters. For ease of %
    % computation, all calculations will be performed in standard %
    % body axes format and in feet. The waypoints will be adjusted %
    % for the 3DLinX world after they have all been determined. %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    cp.waypts = [];
    time = 0;
    xpos = cp.ic.x;
    ypos = cp.ic.y;
    zpos = -cp.ic.alt;
    phi = cp.ic.bank;
    theta = 0;
    psi = cp.ic.head;
    speed = cp.ic.speed*1.6878; % kts to ft/s

```

```

p = 50; % Roll rate in deg/s

cp.waypts(1,1) = time;
cp.waypts(1,2) = xpos;
cp.waypts(1,3) = ypos;
cp.waypts(1,4) = zpos;
cp.waypts(1,5) = phi;
cp.waypts(1,6) = theta;
cp.waypts(1,7) = psi;

for j = (1:seg_num) % Perform calculations for each segment (loop)
    dt = 1;

    if cp.seg{j,1} == 's' % Calculate straight waypoints

        % Set segment time
        if cp.seg{j,2} == 't'
            t_seg = cp.seg{j,3};
        elseif cp.seg{j,2} == 'd'
            t_seg = cp.seg{j,3}/speed;
        end

        t_end = time + t_seg;
        dt = 1;

        while phi ~= 0 % Roll level (loop)
            oldphi = phi;
            oldpsi = psi;

            if abs(phi) < p/10
                phi = 0;
            elseif phi > 0
                phi = phi-p*(dt/10);
            elseif phi < 0
                phi = phi+p*(dt/10);
            end

            % Calculate and record waypoints
            time = time+dt/10;
            avgphi = (oldphi+phi)/2;
            dpsi = atan(32.2*(dt/10)/speed*tan(avgphi*pi/180))*180/pi;
            psi = psi+dpsi;
            avgpsi = (oldpsi+psi)/2;

            if psi < 0
                psi = psi+360;
            elseif psi >= 360
                psi = psi-360;
            end

            if avgpsi < 0
                avgpsi = avgpsi+360;
            elseif avgpsi >= 360
                avgpsi = avgpsi-360;
            end

            dx = speed*dt/10*cos(avgpsi*pi/180);

```

```

xpos = xpos+dx;
dy = speed*dt/10*sin(avgpsi*pi/180);
ypos = ypos+dy;

cp.waypts(end+1,1) = time;
cp.waypts(end,2) = xpos;
cp.waypts(end,3) = ypos;
cp.waypts(end,4) = zpos;
cp.waypts(end,5) = phi;
cp.waypts(end,6) = theta;
cp.waypts(end,7) = psi;
end % End roll level (loop)

while time < t_end % Straight time steps (loop)

% Calculate and record waypoints
if t_end-time < 1 & t_end-time > 0
    dt = abs(t_end-time);
    time = t_end;
else
    dt = 1;
    time = time+dt;
end

dx = speed*dt*cos(psi*pi/180);
xpos = xpos+dx;
dy = speed*dt*sin(psi*pi/180);
ypos = ypos+dy;

cp.waypts(end+1,1) = time;
cp.waypts(end,2) = xpos;
cp.waypts(end,3) = ypos;
cp.waypts(end,4) = zpos;
cp.waypts(end,5) = phi;
cp.waypts(end,6) = theta;
cp.waypts(end,7) = psi;
end % End straight time steps (loop)

% End calculate straight waypoints
elseif cp.seg{j,1} == 't' % Calculate turn waypoints

if cp.seg{j,2} == 'b' % Set segment bank angle
    bank_seg = cp.seg{j,3};
elseif cp.seg{j,2} == 'g'
    if cp.seg{j,3} > 0
        bank_seg = acos(1/cp.seg{j,3})*180/pi;
    elseif cp.seg{j,3} < 0
        bank_seg = -acos(1/-cp.seg{j,3})*180/pi;
    end
end

if cp.seg{j,4} == 't' % Time-based calculation
    t_seg = cp.seg{j,5};
    dt = 1;
    t_end = time+t_seg;

    while phi ~= bank_seg % Match bank angle 1 (loop)

```

```

oldphi = phi;

if abs(phi-bank_seg) < p/10
    phi = bank_seg;
elseif phi > bank_seg
    phi = phi-p*(dt/10);
elseif phi < bank_seg
    phi = phi+p*(dt/10);
end

% Calculate and record waypoints
time = time+dt/10;
avgphi = (oldphi+phi)/2;
oldpsi = psi;
dpsi = atan(32.2*(dt/10)/speed*tan(avgphi*pi/180))...
    *180/pi;
psi = psi+dpsi;
avgpsi = (oldpsi+psi)/2;

if psi < 0
    psi = psi+360;
elseif psi >= 360
    psi = psi-360;
end

if avgpsi < 0
    avgpsi = avgpsi+360;
elseif avgpsi >= 360
    avgpsi = avgpsi-360;
end

dx = speed*dt/10*cos(avgpsi*pi/180);
xpos = xpos+dx;
dy = speed*dt/10*sin(avgpsi*pi/180);
ypos = ypos+dy;

cp.waypts(end+1,1) = time;
cp.waypts(end,2) = xpos;
cp.waypts(end,3) = ypos;
cp.waypts(end,4) = zpos;
cp.waypts(end,5) = phi;
cp.waypts(end,6) = theta;
cp.waypts(end,7) = psi;
end % End match bank angle 1 (loop)

while time < t_end % Turn time steps (loop)

% Calculate and record waypoints
if t_end-time < 1 & t_end-time > 0
    dt = abs(t_end-time);
    time = t_end;
else
    dt = 1;
    time = time+dt;
end

oldpsi = psi;

```

```

dpsi = atan(32.2*dt/speed*tan(phi*pi/180))*180/pi;
psi = psi+dpsi;

if psi < 0
    psi = psi+360;
elseif psi >= 360
    psi = psi-360;
end

avgpsi = (oldpsi+psi)/2;

if avgpsi < 0
    avgpsi = avgpsi+360;
elseif avgpsi >= 360
    avgpsi = avgpsi-360;
end

dx = speed*dt*cos(avgpsi*pi/180);
xpos = xpos+dx;
dy = speed*dt*sin(avgpsi*pi/180);
ypos = ypos+dy;

cp.waypts(end+1,1) = time;
cp.waypts(end,2) = xpos;
cp.waypts(end,3) = ypos;
cp.waypts(end,4) = zpos;
cp.waypts(end,5) = phi;
cp.waypts(end,6) = theta;
cp.waypts(end,7) = psi;
end % End turn time step (loop)

% End time-based calculation
elseif cp.seg{j,4} == 'h' % Heading-based calculation
    psi_change = cp.seg{j,5};
    dpsi_counter = 0;

    if bank_seg < 0
        psi_change = -psi_change;
    end

    while phi ~= bank_seg % Match bank angle 2 (loop)
        oldphi = phi;

        if abs(phi-bank_seg) < p/10
            phi = bank_seg;
        elseif phi > bank_seg
            phi = phi-p*(dt/10);
        elseif phi < bank_seg
            phi = phi+p*(dt/10);
        end

        % Calculate and record waypoints
        time = time+dt/10;
        avgphi = (oldphi+phi)/2;
        oldpsi = psi;
        dpsi = atan(32.2*(dt/10)/speed*tan(avgphi*...
            pi/180))*180/pi;
    end
end

```

```

dpsi_counter = dpsi_counter+dpsi;
psi = psi+dpsi;

if psi < 0
    psi = psi+360;
elseif psi >= 360
    psi = psi-360;
end

avgpsi = (oldpsi+psi)/2;

if avgpsi < 0
    avgpsi = avgpsi+360;
elseif avgpsi >= 360
    avgpsi = avgpsi-360;
end

dx = speed*dt/10*cos(avgpsi*pi/180);
xpos = xpos+dx;
dy = speed*dt/10*sin(avgpsi*pi/180);
ypos = ypos+dy;

cp.waypts(end+1,1) = time;
cp.waypts(end,2) = xpos;
cp.waypts(end,3) = ypos;
cp.waypts(end,4) = zpos;
cp.waypts(end,5) = phi;
cp.waypts(end,6) = theta;
cp.waypts(end,7) = psi;
end % End match bank angle 2 (loop)

dt = 1;
dpsi = atan(32.2*dt/speed*tan(phi*pi/180))*180/pi;

while dpsi_counter ~= psi_change % Turn heading steps
    % (loop)
    if abs(psi_change-dpsi_counter) < abs(dpsi)
        dt = abs((psi_change-dpsi_counter)/dpsi);
        dpsi = psi_change-dpsi_counter;
    else
        dt = 1;
    end

    time = time+dt;
    oldpsi = psi;
    psi = psi+dpsi;
    dpsi_counter = dpsi_counter+dpsi;

    if psi < 0
        psi = psi+360;
    elseif psi >= 360
        psi = psi-360;
    end

    avgpsi = (oldpsi+psi)/2;

    if avgpsi < 0

```

```

        avgpsi = avgpsi+360;
    elseif avgpsi >= 360
        avgpsi = avgpsi-360;
    end

    dx = speed*dt*cos(avgpsi*pi/180);
    xpos = xpos+dx;
    dy = speed*dt*sin(avgpsi*pi/180);
    ypos = ypos+dy;

    cp.waypts(end+1,1) = time;
    cp.waypts(end,2) = xpos;
    cp.waypts(end,3) = ypos;
    cp.waypts(end,4) = zpos;
    cp.waypts(end,5) = phi;
    cp.waypts(end,6) = theta;
    cp.waypts(end,7) = psi;
end % End turn heading steps (loop)

end % End heading-based calculation

end % End calculate turn waypoints

end % End segment calculations (loop)

% Convert from standard body coordinate system (t,x,y,z,phi,theta,psi)
% to 3DLinX system (t,y,-z,-x,theta,-psi,-phi)

[m,n] = size(cp.waypts);
temp(m,n) = 0;
temp(1:m,1) = cp.waypts(1:end,1);
temp(1:end,2) = cp.waypts(1:end,3)*0.3048;
temp(1:end,3) = -cp.waypts(1:end,4)*0.3048;
temp(1:end,4) = -cp.waypts(1:end,2)*0.3048;

temp(1:end,5) = cp.waypts(1:end,6);
temp(1:end,6) = -cp.waypts(1:end,7);
temp(1:end,7) = -cp.waypts(1:end,5);

temp = temp';

% Create and save the text file

[filename,pathname] = uiputfile('*.txt','Save Chase Text File');

if filename

    if isempty(findstr('.txt',filename))
        filename = [filename, '.txt'];
    end

    fid = fopen([pathname,filename], 'w');
    fprintf(fid, ['Chase KeyFrame Navigator Waypoints', ...
        '(t y -z -x theta -psi -phi)\n'], temp);
    fprintf(fid, '%7.6f %7.6f %7.6f %7.6f %7.6f %7.6f %7.6f\n', temp);
    fclose(fid);
else

```

```
        cp.error = 'Save Was Unsuccessful';  
        edit_Error_Callback(h, eventdata, handles, varargin);  
    end  
  
end  
  
% The End
```