SimSect Hybrid Dynamical Simulation Environment

User's Manual

Uluç Saranli

Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2110, USA
ulucs@eecs.umich.edu

Abstract

This report is the user's manual of SimSect, a flexible environment for the simulation of hybrid dynamical systems. Hybrid dynamical systems are systems where the continuous dynamics undergo discrete changes upon detection of predefined state based events. The simulation of such systems involve accurate detection of events and handling of the transition between different continuous systems.

This report also includes the details of two example hybrid dynamical systems: A spring loaded inverted pendulum (SLIP) runner and a compliant hexapod. The former example illustrates the basic elements of programming models with SimSect, while the latter implements a much more complicated model, addressing many common issues arising when defining complex dynamical models with SimSect.

CSE Technical report: CSE-TR-436-00

1 Introduction

1.1 What is SimSect?

SimSect is a general purpose hybrid dynamical system integration environment. It was mainly built to simulate a hexapod robot with compliant legs. It can. however, be used to define arbitrary hybrid dynamical system (systems with piecewise continuous dynamics), and numerically compute their state trajectories from arbitrary initial conditions. It also incorporates an interface to *Geomview*, a 3D visualization environment (`http://www.geom.umn.edu`) to visualize the state of the dynamical system in a flexible manner.

If you have questions or comments about SimSect or this manual you can e-mail them to ulucs@eecs. umich.edu.

1.2 Authors

SimSect is originally written by Uluc Saranli, together with the Spring-Loaded Inverted Pendulum(SLIP) and the Compliant Hexapod Model definitions.

1.3 Overview

SimSect is designed to be used for integrating a well defined hybrid dynamical flow. In doing so, the user can specify several parameters of the dynamical system, as well as the initial conditions without having to modify the source or recompile the program. The output of SimSect is a set of user defined functions, evaluated over the entire computed trajectory.

SimSect can either be used as a batch simulation tool or as a real-time visualization environment for the dynamical system. The former case can proceed without any user interaction, where the initialization files are usually created through a scripting language such as Perl, and the results are obtained in a set of output files. In the latter scenario, however, the user modifies the initialization file, runs the simulation and observes the results through the visualization.

The definition of a model in SimSect involves writing C code corresponding to various components of the hybrid dynamical system. These include the definition of the vector field on different discrete charts of the system, the chart boundaries in the form of zero level sets of scalar functions and the transition functions which map the system state on one chart to the system state on the following chart in the case of a boundary crossing. For a more detailed discussion on the hybrid system concepts used in SimSect, refer to Section 3.1.

2 Using SimSect

2.1 System Requirements

SimSect currently runs on Intel x86 based Linux platforms. Visualization of the simulation requires Geomview, available from `http://www.geomview.org`.

2.2 Invocation

SimSect consists of a single executable `SimSect`. It accepts only one argument, which specifies the *initialization file*, through which most of the system configuration is done. The default name for the initialization file is `SimSect.rc`.

```
Usage: SimSect [-c filename]
```

2.3 Output Files

Upon completion of execution, SimSect creates three output files. The main output file is `SimSect.data`, which includes an ascii dump of the auxiliary functions defined by the model. By default, the data in this file is collected at every integration time step. The sampling period can be redefined in the initialization file to avoid very large data files. The `SimSect.initial` and the `SimSect.param` files include the model initial conditions and the model parameters, respectively. These two files have the same format as the initialization file (see Section 2.4.1).

2.4 The Initialization File

System configuration and the specification of model parameters as well as the model initial conditions can be done through the initialization file. Unless explicitly specified through the command line parameter, `SimSect.rc` is the default initialization file.

2.4.1 File Format

In SimSect, various parameters and variables are defined as *symbols*, whose values can then be accessed by both the integrator and the model specification to read user specified settings. The initialization file consists of assignments to these symbols in the following syntax.

$$symbol\_name = value;$$

The *symbol_name* can be any string starting with a letter and including letters, digits or the underscore character. The symbols in SimSect can be of type real number or string. The *value* field hence can be either a real number or a string enclosed in quotes. Finally, lines starting with the # symbol are considered as comments and are ignored. Figure 1 illustrates an example init file.

2.4.2 Summary of Integrator Symbols

This section summarizes the symbols predefined by the SimSect integrator subsystem. Model specific symbol definitions will be discussed in conjunction with the details of each particular model (see Sections 4 and 5).

- `systemName (default = "hexapod")`

  The name of the dynamical model to be integrated. Currently valid values are “hexapod”, “akh” and “slip”. If an invalid value is specified, the model defaults to hexapod.

```
### This is a comment
recordPeriod = 0.01;
measurePeriod = 0.1;
dataBaseName = "hexapod";
useGeomview = 1;
framerate = 30;

# Integration engine related parameters
finalTime = 5;
maxChartCount = 128000;
stopPrecision = 1e-10;
maxStopIterations = 128;
tolerance = 1e-5;
maxTimeStep = 1e-3;
minTimeStep = 1e-15;
rkPower = 0.33;
```

Figure 1: An example initialization file for SimSect.

- `dataBaseName (default = "SimSect")`

  The base filename for the output files. The suffixes `.data`, `.initial` and `.param` are appended to this to obtain the output filenames.

- `useGeomview (default = 0`

  Flag to enable the link to Geomview for the visualization of the simulation. The model definition must provide a visualization mapping function for this option to work properly.

- `stopPrecision (default = 1e-10)`

  Numerical precision tolerance for the stopping functions. The boundary crossings between different charts in the hybrid system are detected up to this precision in the function value.

- `maxStopIter (default = 128)`

  Maximum number of iterations that the stopping iterator will be allowed to go through before giving up on trying to find the boundary crossing. The failure to find a crossing within this limit strongly suggests that there is a discontinuity in the boundary function.

- `recordPeriod (default = 0.0)`

  The time period for the recording of trajectory data which will be saved in the output files. Special stopping functions are defined to compute the trajectory points at exact multiples of this value, resulting in accurate recordings of the system trajectory at a given frequency.

- `measurePeriod (default = 0.0)`

  The time period for displaying a user message at particular time intervals. This is a convenient way of interactively displaying the integration progress. However, a very small value for this symbol might result in excessive output of messages.

- `maxChartCount (default = 128)`

  The maximum number of times chart transitions can occur. In cases where vector fields in neighboring charts point towards each other, chattering may occur, which usually throws the integrator into a state where chart transitions occur at very high frequencies. When the number of transitions exceed this value, the integration stops with an error and the results of the integration up to this point are saved.

- `finalTime (default = 1.0)`

  This is the final time for the integration. Upon reaching this point in time, the integration stops and the computed trajectory is saved in the data files. The integrator actually defines a specific stopping function for this purpose, so the final integration time corresponds to this value up to the stopping precision.

- `tolerance (default = 1e-4)`

  This is the state error tolerance setting of the Runge Kutta integrator.

- `maxTimeStep(default = 1e-2)`

  The maximum time step limit for the integration. The time step is constrained to be below this value by the Runge Kutta algorithm for numerical stability purposes. This value, in conjunction with the tolerance determine the accuracy of the integration algorithm,

- `minTimeStep (default = 1e-15)`

  The minimum allowable time step for integration. The time step used by the integration algorithm is not allowed to go below this value.

- `power (default = 0.33)`

  The time step adjustment power for the Runge Kutta algorithm.

3 Programming with SimSect

3.1 Hybrid Dynamical Systems

A large number of dynamical systems that we are interested in analyzing, including the hexapod model that we present in this report, cannot be represented with a single dynamical flow. They are hybrid dynamical systems, which are mixtures of discrete and continuously varying events. This section describes a formal definition of hybrid dynamical systems, and is mostly quoted from the formalism described in [1], with some minor modifications.

We assume that the problem domain is decomposed into the form

$$V \;=\; \bigcup_{\alpha\in I} V_\alpha$$

where I is a finite *index set* and V_α is an open, connected subset of $\mathbf{R}^n$. Each element in this union is called a *chart*. Each chart has associated with it a vector field, $f_\alpha : V_\alpha \times \mathbf{R} \to \mathbf{R}^n$. Notice that the charts are not required to be disjoint. Moreover, on the intersection set $V_\alpha \bigcap V_\beta$, continuity, or even agreement of the vector fields are not required for $\alpha, \beta \in I$. For each $\alpha \in I$, the chart V_α must enclose a *patch*, an open subset U_α satisfying $\overline{U}_\alpha \subset V_\alpha$. The boundary of U_α is assumed piecewise smooth and is referred to as the *patch boundary*. Together, the collection of charts and patches is called an *atlas*.

For each $\alpha \in I$ there is a finite set of *boundary functions*, $h_{\alpha,i} : V_\alpha \to R$, $i \in J_\alpha^{bf}$, and real numbers called *target values*, $C_{\alpha,i}$, for $i \in J_\alpha^{\rm bf}$ that satisfy the condition: For $x \in V_\alpha$ where $\alpha \in I$, we require

$$x \in U_\alpha \;\text{ if and only if }\; h_{\alpha,i}(x) - C_{\alpha,i} > 0 \;\text{ for all } i \in J_\alpha^{\rm bf} \quad .$$

Thus, a patch is to be considered the domain on which a collection of smooth functions are positive. The boundary of a patch is assumed to lie within the set:

$$\bigcup_{i\in J_\alpha^{\rm bf}} h_{\alpha,i}^{-1}\left(\{C_{\alpha,i}\}\right) \qquad\qquad \text{for } \alpha \in I \; .$$

Conceptually, the evolution of the system is viewed as a sequence of trajectory segments where the endpoint of one segment is connected to the initial point of the next by a transformation. It follows that time may be divided into contiguous periods, called *epochs*, separated by instances where *transition functions* are applied at times referred to as *events*. The transition functions are maps which send a point on the boundary of one patch to a point in another (not necessarily different) patch in the atlas.

Within this framework, an orbit in the flow of a hybrid dynamical system which begins at a time t_0 and terminates at t_f may be completely described. A *trajectory*, hence, is a curve $\gamma : [t_0, t_f] \to V \times I$ together with an increasing sequence of real numbers $t_0 < t_1 < \cdots < t_m = t_f$ that satisfies three properties:

- Each time interval (t_i, t_{i+1}) corresponds to an epoch and there exists a designated α so that $\gamma(t)$ lies entirely in $\overline{U}_\alpha \times \{\alpha\}$ for all $t \in (t_i, t_{i+1})$.
- For $t \in [t_i, t_{i+1})$ and the unique α specified above, $t \to \pi_1\left(\gamma(t)\right)$ is an integral curve of the vector field f_α.
- $\lim\limits_{t\to t_{i+1}^-} \pi_1\left(\gamma(t)\right) = y$ exists, $y \in S_\alpha$ and $T_\alpha(y) = \lim\limits_{t\to t_{i+1}^+} \gamma(t)$.

3.2 System Architecture

SimSect isolates the integration algorithms from the definition of the dynamical system. Unlike the model implementation, which is different for every system model, the integration engine is a static component of the system. This abstraction makes the implementation of additional model definitions and the modification of existing models much simpler.

Defining a dynamical system model consists of providing the hybrid components described in Section 3.1. The model definition is usually programmed by the user and includes functions corresponding to the following list of tasks.

- Initialize the partition structure, the initial state and the initial chart.
- Define the properties of a particular chart, and the boundary functions that will be used.
- Compute the vector field for the individual charts.
- Compute boundary functions, identified by a certain index that the chart initialization determines for the current chart.
- Perform chart transitions by computing the next system state and chart.
- Validate a chart by checking whether a given trajectory point lies in the chart.
- Compute an auxiliary function, mainly used for data collection purposes.
- Compute the homogeneous transformations for visualization of the system trajectory using Geomview.

The integration algorthms invoke these functions to compute necessary components of the computation.

3.3 The Integrator Engine

The overall structure of SimSect is depicted in Figure 2, where the example hexapod model components is also included. This section describes the the integrator engine and its interface to the model definition. The subsections summarize C implementations of various components. The reader should refer to the actual source code for implementation details.

3.3.1 Iterators

Most numerical algorithms use some form of iteration, where a particular procedure is repetitively executed until a certain condition is satisfied. Several components of the numerical integration procedures in SimSect have this structure.

On several levels, SimSect makes use of an abstract *iterator* concept. An iterator is a computational object which consists of an initialization procedure, an iterating function, a termination condition and a wrap-up procedure. When an iterator is invoked, the initialization is followed by repeated execution of the iterating function until the termination condition is met. The wrap-up procedure is then invoked and terminates the iteration.

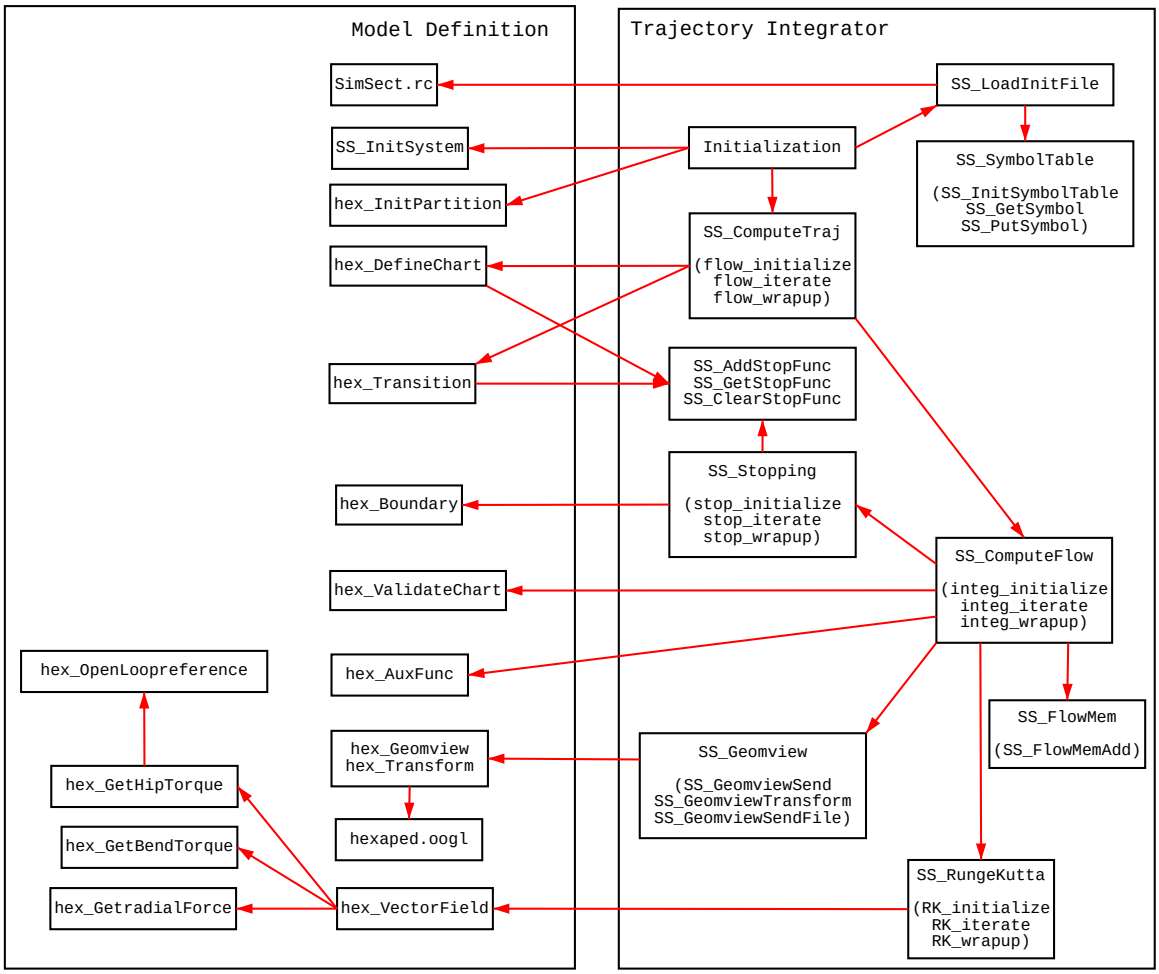


Figure 2: System architecture of SimSect with an example model instance of the Compliant Hexapod Model.

The files `SS_Iterator.c` and `SS_Iterator.h` define functions to initialize and use iterators at a somewhat abstract level. The functions defined by this subsystem are summarized below.

- `SS_InitIterator(...)`

  Setup an iterator by defining the appropriate initialization, iteration, interruption and wrapup procedures.

- `SS_Iterate(...)`

  Invoke an iterator by first calling its initialization procedure, then repeatedly calling its iteration procedure until the termination flag is set and then calling its wrapup procedure.

SimSect implements a hierarchy of several iterators implementing different components of the integration. The following sections describe various iterators in SimSect

3.3.2 The Runge Kutta Iterator

This iterator implements the 4th order adaptive time step Runge Kutta integration algorithm for a single time step. The iteration accomplishes the refinement of the time step until the state error falls below a certain tolerance, determining the time step for computing the next trajectory point.

This module consists of the files `SS_RungeKutta.c` and `SS_RungeKutta.h`. The list below summarizes the functions exported by this module.

- `SS_InitRK(...)`

  Sets up a Runge Kutta iterator with appropriate system states and parameters.

- `SS_UpdateRK(...)`

  Updates the current chart for the trajectory computation.

- `SS_RunRK(...)`

  Carries out the iteration and computes the next trajectory point as well as updating the time step.

- `SS_RK_NextStep(...)`

  Provides external access to the iteration function, without going through the whole iteration process with wrapup etc.

3.3.3 The Flow Iterator

The flow iterator is the module which repeatedly calls `SS_RunRK()` to compute the trajectory inside a particular chart, until a boundary crossing is detected. The functionality of the flow computation module is detailed in Figure 3. Given an initial condition, this module iterates through successive trajectory points, using the Runge-Kutta module to advance through time steps. Upon detection of a boundary crossing, the stopping iterator is invoked, refining the last time step until the boundary crossing time is determined up to the required precision.

This module is implemented by the files `SS_ComputeFlow.c` and `SS_ComputeFlow.h`. The following functions provide access to the functionality of this module.

- `SS_InitFlow(...)`

  Initializes necessary structures for the flow iterator.

- `SS_UpdateFlow(...)`

  Updates the current chart for a flow iterator.

- `SS_PurgeFlow(...)`

  Purges memory structures allocated by the InitFlow function for a flow iterator.

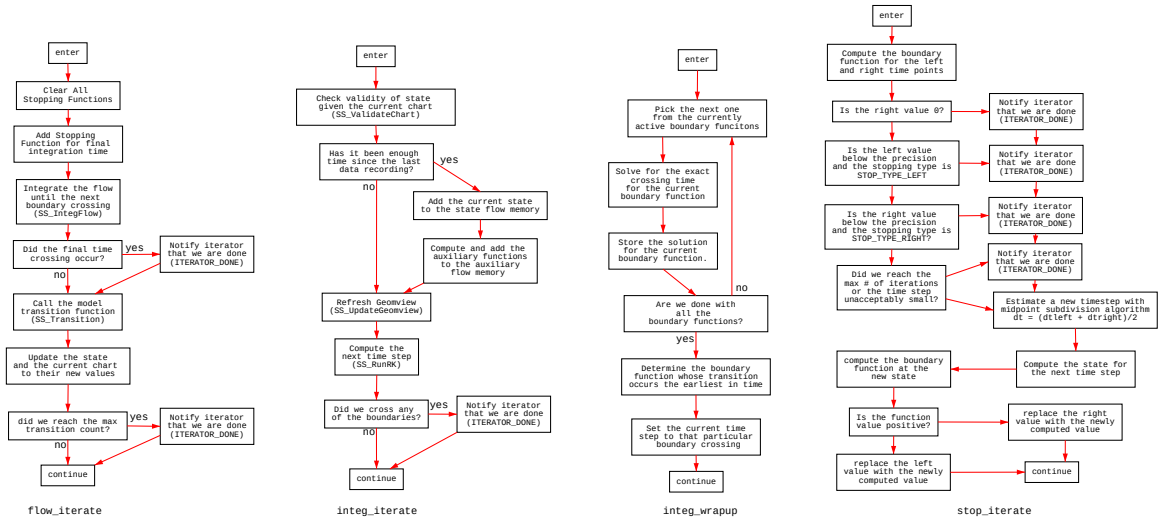


Figure 3: Flowcharts for major functions in the SS_ComputeTraj, SS_ComputeFlow and SS_Stopping modules. The `flow_iterate` is the iteration function for the trajectory iterator. The `integ_integrate` and `integ_wrapup` are the iteration and wrapup functions of the flow iterator, and `stop_iterate` is the iteration function for the stopping function iterator.

- `SS_IntegFlow(...)`

  Integrates the dynamical system model in the current chart, until a boundary crossing is detected by invoking the flow iterator. Then, it uses the stopping iterator to detect the exact crossing time and stops the integration at that time instant. Note that this function also takes care of defining and handling the measurement stopping function introduced in Section 2.4.2 as well as the recording of trajectory points through the use of the flow memory (see Section 3.4.1).

3.3.4 The Stopping Iterator

This module is responsible from refining the last time step of the integration in a chart to determine the exact boundary crossing time. Figure 3 depicts the algorithm for the iteration function of this module. This iterator is invoked from within the wrapup function of the flow iterator, and uses a midpoint subdivision type algorithm to refine the time step where the boundary crossing is detected. It can handle multiple crossings and returns the crossing which is the earliest in time.

The functions associated with this module are also defined in the files `SS_ComputeFlow.c` and `SS_ComputeFlow.h`. The stopping iterator is somewhat hidden in the flow computation module, so it does not have any external functions which provide access to its iteration functions. The exported function are mainly for defining and deleting stopping functions and are heavily used by both the integrator engine itself (for defining final time stopping functions, measurement stopping functions etc.) and by the model definition to define and specify boundary functions.

- `SS_AddStopFunc(...)`

  Adds a stopping function to the current list of active stopping functions. These are scalar valued functions whose zero crossings are detected and iteratively identified up to a certain precision by the stopping iterator. These zero crossings may occur either when a boundary crossing occurs, or when a system defined condition (such as the final integration time) is encountered. The return value is an index which will later be used to access the state of the added stopping function.

- `SS_ClearStopFunc(...)`

  Clears all currently active stopping functions. Usually called at the end of each chart transition, in which case the model defines new boundary functions.

- `SS_GetStopFunc(...)`

  Returns the state of the boundary function. The model definition frequently uses this function to check which boundary function initiated the chart transition to decide on the proper course of action.

3.3.5 The Trajectory Iterator

The trajectory iterator module is one level higher than the flow iterator and is responsible from iterating through successive charts. Each execution of the iteration corresponds to the invocation of a complete flow iteration, resulting in the computation of the trajectory from the current state until the next boundary crossing. Figure 3 gives the flowchart for the main iteration loop for this module.

Note that this module is also responsible from defining the final time stopping function as well as calling various components of the model to initialize the new partition structure.

The function associated with this module are located in the files `SS_ComputeTraj.c` and `SS_ComputeTraj.h`

- `SS_InitTraj(...)`

  Initializes necessary structures for the trajectory iterator.

- `SS_PurgeTraj(...)`

  Purges memory structures allocated by the InitTraj function for a trajectory iterator.

- `SS_IntegTraj(...)`

  Integrates the dynamical system from an initial condition until either the final time or the maximum chart count is reached. The integration also stops in case of an error. The integration is accomplished by invoking the flow iterator for each chart, calling appropriate transition functions for boundary crossings. This is the main entry point to the integrator and is called right after the initialization, carrying out the integration. Following the invocation of this function, the trajectory data is saved and the system exits (see Section 3.3.6 for details).

3.3.6 The Top Level

This section describes what happens at the topmost level of SimSect, from within which all the other modules are invoked.

The file `SimSect.c` is the entry point to the system. The main sequence of tasks carried out by the system from startup to the end are as follows:

- Initialize the symbol table and load the initialization file.
- Initialize Geomview subsystem.
- Call `SS_InitSystem`. This function chooses the model to be integrated and calls its initialization function. Note that to add a new model, this function should be modified to incorporate the new model's initialization system.
- Initialize the flow memory for storing the auxiliary function data.
- Initialize the flow memory for storing the state trajectory data.
- Call the `InitPartition` function of the model to setup the initial condition for the integration as well as any other initializations that the model may want to execute.
- Initialize the trajectory integrator.
- Save the initial conditions and the model parameters into appropriate output files.
- Integrate the model system until the trajectory iterator exits, which is either at the final integration time or some error has occurred.
- Save the trajectory data and auxiliary functions into appropriate output file, cleanup and exit.

3.4 Utility Modules

3.4.1 The Flow Memory Utility Module

In SimSect, the data points resulting from the integration of the model (either the auxiliary functions or the state variables) are stored in a certain data structure called *flow memory*. SimSect provides facilities to create, maintain and delete flow memories, making it possible to collect and save data in a convenient way.

The files `SS_FlowMem.c` and `SS_FlowMem.h` implement the functions associated with this utility module. The provided functions and their descriptions are summarized below.

- `SS_FlowMemInit(...)`

  Allocates and initializes a flow memory structure to be used by the other utility functions in this module. An uninitialized flow memory object cannot be used by the utility functions. Note that the number of elements (doubles) in a data entry are specified with this function at the time of initialization.

- `SS_FlowMemAdd(...)`

  Adds a data entry to the flow memory. The internal implementation of the flow memory allocates heap memory in chunks of a predefined size and fills them in with the data supplied through this function. Hence, the user does not have to worry about space allocation and memory management.

- `SS_FlowMemGet(...)`

  Returns a particular data entry from the flow memory, indexed by an integer. The data entries are ordered chronologically by the order they are added into the memory by the `SS_FlowMemAdd` function.

- `SS_FlowMemDump(...)`

  Outputs the contents of a flow memory in an ascii file where each row corresponds to a single data entry with the appropriate number of elements. This file can easily be imported by numerical computation packages such as it Matlab.

- `SS_FlowMemClear(...)`

  Clears all the data in a flow memory and deallocates the allocated heap memory.

3.4.2 The Symbol Table Utility Module

The symbol table module provides facilities for creating, maintaining and accessing symbol tables. Its main use in SimSect is to handle the interface between the definitions in the initialization file and different components of the system including the integrator itself as well as the model definition.

Symbols in SimSect are named variables which can hold either a real number or a string value. Utility functions to setup symbol tables and to create and access symbols are defined in files `SS_SymbolTable.c`, `SS_SymbolTable.h` and `SS_Parameter.c`.

- `SS_InitSymbolTable(...)`

  Allocates and initializes an empty symbol table structure.

- `SS_PurgeSymbolTable(...)`

  Deallocates an initialized symbol table structure. All the symbols in the table are discarded.

- `SS_NewSymbol(...)`

  Creates a named symbol and allocates memory for the data structure.

- `SS_GetSymbol(...)`

  Looks up and returns a symbol from a symbol table, indexed by the symbol name.

- `SS_PutSymbol(...)`

  Inserts a symbol into a symbol table. The symbol structure must be properly allocated and initialized. If a symbol of the same name already exists in the table, then it is replaced by the newly inserted value.

- `SS_PutDoubleSymbol(...)`

  Creates and initializes a symbol structure with the real number value supplied. This is a shorter and more convenient way of creating symbols if the user does not want to do the initialization of the symbol structure. Uses `SS_PutSymbol()` for insertion and hence checks for duplicates.

- `SS_PutStringSymbol(...)`

  Similar to `SS_PutDoubleSymbol()`, inserts a string valued symbol into a symbol table.

3.4.3 Parser and Initialization File Utility Modules

This module handles the parsing of the initialization file. Normally, the programming of the model will not find this model useful in any way, but we will include a brief description of its functionality in this section for completeness.

This module consists of the files `SS_Tokenizer.c`, `SS_Tokenizer.h`, `SS_SimScript.c`, `SS_SimScript.h` and `SS_InitFile.c`. The exported procedures and their functionality are as follows.

- `SS_InitTokenizer(...)`

  Creates and initializes a tokenizer object.

- `SS_OpenTokenizer(...)`

  Starts the tokenizer with a particular file or string input. This is the first step to be taken before any module can use the tokenizer.

- `SS_GetNextToken(...)`

  Identifies and retrieves the next token from the input stream.

- `SS_CloseTokenizer(...)`

  Ends the tokenization from the current stream and closes the stream.

- `SS_EndOfInput(...)`

  Checks whether the input stream has reached its end.

- `SS_RunScript(...)`

  Opens a specified stream and interprets the script read from the stream. Here, the script format is the initialization file format described in 2.4.1.

- `SS_LoadInitFile(...)`

  Loads and interprets the initialization file with the filename supplied as an argument. This is the function used by the top level SimSect execution. It is simply a wrapper to SimScript facilities, which in turn use the parser and the tokenizer for their functionality.

3.4.4 The Root Finder Utility Module

Some of the controllers used in the example models require the computation of the roots of a scalar function. Consequently, SimSect now provides a Newton's method root finder as a utility module. This module also uses the iterator facilities of SimSect as it relies on an iterative algorithm. Please refer to the source code for the details of the algorithm, which is a very simple root finder.

The files associated with this module are `SS_RootFinder.c` and `SS_RootFinder.h`. The function provided by the module are summarized below.

- `SS_InitRootFinder(...)`

  Initializes the specified root finder object.

- `SS_FindRoot(...)`

  Finds a zero of the function specified in the arguments, closest to a starting point supplied in another argument.

3.4.5 Visualization Utility Module

SimSect implements an interface with Geomview, a three dimensional graphics package, for the visualizing the simulation of the model. This interface is "real-time", in the sense that the visual is updated as the simulation progresses. Currently, there is no off-line visualization option.

SimSect interfaces with Geomview through named Unix pipes, through which it uses Geomview's command language to control its behavior. If the symbol `useGeomview` is set to 1 in the initialization file, SimSect invokes Geomview and establishes a pipe connection for communication. If the attempt to execute and connect to Geomview fails, the subsystem is disabled.

SimSect assumes a particular way in which the model state is visualized. The model definition must provide functions to initialize the initial three dimensional geometry, together with appropriate transformation matrices (see [3] for a detailed account of how to construct geometries in Geomview). The real time visualization is then accomplished by updating the transformation matrices associated with the geometry in a way that is particular to the model and the design choices of the model programmer.

The functions in the files `SS_Geomview.c` and `SS_Geomview.h` implement the visualization utilities of SimSect. The following functions are provided by this module.

- `SS_GeomviewInit(...)`

  Initializes the visualization subsystem. If the symbol `useGeomview` is set to 0 in the initialization file (the default is 1), then this function returns without doing anything. Otherwise, it attempts to execute Geomview or establish connection to an existing Geomview execution. If successful, it resets Geomview and prepares it for the upcoming visualization tasks. This function is called from the SimSect top level execution.

- `SS_GeomviewSend(...)`

  Sends a string to Geomview, which should be in `Geomview Command Language (gcl)`.

- `SS_GeomviewWait(...)`

  Waits for Geomview to finish whatever it is currently doing and then returns.

- `SS_GeomviewTransform(...)`

  Sends a number of transformation matrices to Geomview, redefining existing transformation matrices with names `ttt0, ttt1, ttt2, ....` This function is used by the model definition to update the state of the visual image based on the current state. Essentially, this operation updates the visual, reflecting the new positions and orientations of the geometries which depend on the redefined transformation matrices.

- `SS_GeomviewSendFile(...)`

  Opens the specified text file and sends the contents to Geomview. This function is most commonly used by the model in defining the initial geometry of the visualization, by loading a static `oogl` file during initialization.

- `SS_GeomviewClose(...)`

  Closes the connection to Geomview.

- `SS_GeomviewClear(...)`

  Deletes all the objects in Geomview.

3.4.6 Miscellaneous Utility Functions

The files `SS_Util.c` and `SS_Util.h` provides several utility functions to the model programmer. Among those functions are general purpose matrix and vector manipulation procedures, special purpose matrix and vector facilities for manipulating vectors in three dimensions and rotation matrices as well as some other functional tools. Please refer to the source code for details on these utilities and their usage.

3.5 The Model Definition

Programming a model in SimSect involves providing several functions defining different components of the hybrid system formalism as well as functions to interface to the visualization subsystem. The following list summarizes functions that the programmer must provide in order to complete the definition of the model. The following subsection describes each of these functions in detail, including their inputs outputs and the requirements on their functionality.

- Setup model structures, define state and parameter symbol names.
- Initialize the partition structure, the initial state and the initial chart.
- Compute the vector field for the individual charts.
- Define the properties of a particular chart, and determine the boundary functions that will be used, defining stopping functions as necessary.

- Compute boundary functions, identified by a certain index that the chart initialization determines for the current chart.
- Perform chart transitions by computing the next system state and chart.
- Validate a chart by checking whether a given trajectory point lies in the chart.
- Compute an auxiliary function, mainly used for data collection purposes.
- Compute the homogeneous transformations for visualization of the system trajectory using Geomview.

3.5.1 The State Space Structure

One of the first things that the model programmer needs to determine is the structure of the state space in each of the charts of the system model. In SimSect, there are two different types of state space variables that are commonly used.

The first type corresponds to the usual continuous state variables and by convention occupy the topmost part of the state space (i.e. states 0, 1, 2, ...). Note that each chart may have a different number of these continuous states, among which the conversion will be carried out by the transition function.

The second type of states are all other components of the model that require memory. These are mainly bookkeeping states, which change with transitions between different charts. Due to the fact that the vector field definition is unable to retain any memory, these states become critical in keeping track of the discrete changes that take place in the system. Examples of such states are counts of chart transitions, the foot placement location for the SLIP model, mechanisms to keep track of currently active tripods in the hexapod model etc.

3.5.2 Initializing the Model

This is the first component of any model called by SimSect. This function is called from within `SS_InitSystem`, which looks at the parameter *systemName* to figure out which model is to be integrated. Hence, when a new model is to be defined, the programmer needs to modify `SS_InitSystem` to call the model initialization function. This static way of specifying the model initialization functions is because SimSect currently does not have any means of dynamically loading precompiled model definitions during execution. Hence, their entry points must be hard coded in SimSect.

The model initialization function needs to carry out the tasks in the list below.

- Fill in the model function fields in the global `SS_Data` struct. The fields that need to be filled out are SS_Data.ipFunc, SS_Data.vfFunc, SS_Data.dcFunc, SS_Data.trFunc, SS_Data.vcFunc, SS_Data.auxFunc and SS_Data.geomFunc, each of which corresponding to one of the model functions described in the following sections.
- Call `SS_SetupSystem`, specifying the number of states, the initial state, the names of state variables, the number of parameters, the default parameters (before loading the init file), the parameter symbol names, the number of charts and the number of

auxiliary functions. The initial state and the default parameters should be pointers to arrays of doubles, which are then copied by this function. The state and parameter names must be pointers to arrays of strings with the corresponding symbols.

- Perform any setup or initialization needed by the model. The initialization file is processed before `SS_InitSystem` is called. Hence, the system parameters, initial conditions etc. can be accessed by this function to perform the initialization.

3.5.3 Initializing the Partition

The function pointer in `SS_Data.ipFunc` after the model initialization points to this function. The main purpose of this function is to finalize the initial conditions and determine which chart the system will start from, in addition to any modifications to the parameters, and/or model states. It is called after all the system initializations are complete and right before the integration starts. The prototype for this function is

```
int InitPartition(double* xnew, int* newChart, double x[],
                  double p[], int chart);
```

The fields indicated by `xnew` and `newChart` must be filled out by the function and they will be the initial state and the initial chart for the integration to come.

3.5.4 The Vector Field

This is the function pointed to by the variable `SS_Data.vfFunc`. Its main purpose is to compute the vector field for the model at a given state and chart, defining the flow. Usually, this is the most complicated function in a model definition because it implements the dynamics of the model as well as other components such as controllers etc. Naturally, all the functionality does not need to be implemented in a single function body. A more modular approach with multiple procedures is possible and should be considered when possible. The prototype for this function is

```
int VectorField(double* xdot, double x[], double p[], int chart);
```

The array pointed to by the argument `xdot` must be filled in by the function with the computed vector field.

One of the major mistakes made in implementing a model is to use the vector field function to incorporate discrete changes in system structures from within the vector field. The calls to the vector field function do not occur in chronological order. Hence, even if the integrator computes the flow at a particular time, this does not necessarily mean that the integration has reached that point. Consequently, the assumption that the time (which is one of the states, usually) that the flow is computed at can be used to make discrete decisions about the structure of the system is wrong. The vector field should properly define a (not necessarily continuous) function of the state space, without any other internally maintained discrete state. It is up to the hybrid integrator together with the transition functions to perform those discrete structural changes to the model, which will then reflect themselves back to the vector field in the fact that a new chart has been entered.

3.5.5 Defining a Chart

The variable SS_Data.dcFunc points to this function after initialization. The main purpose of this function is to define a chart after each chart transition (which occurs after boundary crossings). The definition of the chart almost always means defining the stopping functions associated with the new chart, based on the new boundary functions which become active following the transition. This function will use the stopping function facilities of the integrator to create and define the stopping functions.

Note that the places in which the stopping function are accessed (transition function etc.) are separate from this function. Consequently, the indices returned by the stopping function definition utilities must either be stored in some place accessible from those other procedures, or be based on a common indexing mechanism which can later be used to recover the indices for the stopping functions. These indices are assigned to stopping function in the order they are defined, starting from 1. Hence, if the order of their definition is carefully chosen, it will be possible to know in other parts of the model what these indices will correspond to. Example systems of later sections will make this clearer.

The prototype for this function is

```
int DefineChart(SS_ComputeFlow* cf, int chart);
```

Note that the cf argument is necessary to be able to define stopping functions, which are local to the particular flow iterator they are defined in. See Section 3.5.6 for details on how to define and add stopping functions for a particular chart.

3.5.6 The Boundary Functions

Unlike the other functions provided by the model, the definition of the boundary functions is somewhat transparent to the integrator and is not reported through any of the variables in the global struct SS_Data. Instead, the stopping function definitions include the pointers to the corresponding boundary functions.

SS_Stopping.h defines the C type for stopping function definitions as

```
typedef struct {
  int      index;  /* Indices for the boundary function */
  int      type;   /* Crossing for the boundary function */
  int      state;  /* The current state of each function */
  double   value;
  double   timeStep;
  int      (*action)(double* val, double x[], double p[],
                     int chart, int ind);
} SS_StopFunc;
```

When defining boundary functions in, for example, the DefineChart function, the model programmer needs to fill in the index, type and action fields of this struct prior to calling SS_AddStopFunc. The type field is either STOP_TYPE_LEFT or STOP_TYPE_RIGHT, specifying whether the boundary function crossing should be detected from the left or right, resulting a time step either before, or after the crossing, respectively (this is necessary because the

crossing is detected only up to a certain precision). The `index` and `action` fields define what boundary function should be used to evaluate the stopping function. The index field is mainly provided as a convenience for models where it makes most sense to parameterize certain boundary functions with an index, corresponding, for example, to the current chart.

Consequently, whatever function is assigned to the `action` field of any stopping function specification must define the corresponding boundary function for all possible indices that they might be called with. As it can be seen from the prototype, the boundary functions take the current state, model parameters and the current chart, and return the value of the boundary function corresponding to the specified index.

In any of the model functions, the programmer might retrieve one of the boundary functions through `SS_GetStopFunc` and check the function value in the `value` field of the above struct. This is, for example, the way in which the transition function determines which boundary function caused the transition and decide on the appropriate course of action.

3.5.7 The Transition Function

The transition function is the function pointed to by the variable `SS_Data.trFunc` after model initialization. The main purpose of this function is to determine the new state and chart after a boundary crossing. Any zero of any stopping function encountered during integration stops the flow iteration and calls the transition function to determine what the appropriate course of action is. It is then up to this function to decide whether this crossing corresponds to a boundary function, and if so perform necessary changes to the state (such as coordinate transformations, discontinuous changes due to impacts etc.) and determine the new chart. There are also instances where state changes occur even though the current chart does not change.

The prototype for the transition function is

```
int Transition(int* trans, int* newChart, double* xnew,
               double x[], double p[], int chart);
```

The arguments `trans`, `newChart` and `xnew` must be filled out by the function. `trans` indicates whether a chart transition has occurred. This is used by the trajectory iterator to decide whether other function calls such as the `DefineChart` are required. The `newChart` and `xnew` are the new chart and the new state after the transition and will be computed according to the particular aspects of the model.

3.5.8 Validating a Chart

This function is used by the integrator to determine whether the current state is valid for the current chart, mainly for debugging purposes. An invalid state corresponds to such cases where the partitioning of the state space prescribes a particular chart which is different than the current chart as imposed by the integrator. This might correspond, for instance, to a case where the toe of a hopper is underground while the hopper is in the *flight* chart. The `SS_Data.vcFunc` variable points to this function after initialization.

The prototype for this function is

```
int ValidateChart(int* inChart, double x[], double p[], int chart);
```

The `inChart` argument returns a flag indicating whether the given state is in the specified chart or not.

3.5.9 The Auxiliary Functions

The auxiliary function mechanism in SimSect is provided for the model programmer to be able to record arbitrary functions of the system state and output them to a file. By use of this mechanism, SimSect can directly output integrated trajectories in the desired coordinate system, making it more convenient to process the output. This also saves disk space and memory in cases where the variables of interest are very few compared to the number of system states.

The auxiliary function is stored in `SS_Data.auxFunc` after model initialization. The prototype for this function is

```
int Auxiliary(double *f, double x[], double p[], int chart);
```

This function fills in the array supplied in the argument `f` with all the computed auxiliary functions given the current state and the chart. Note that the number of auxiliary functions remains the same throughout the integration and specified during the model initialization. Exceeding that number in filling the output will most likely crash the system.

The computed functions can be anything, from things as trivial as the current time, to complicated coordinate transformations of the current state. There is no restriction on what the model programmer defines as auxiliary functions, as long as the number of functions is correctly specified.

3.5.10 Geomview Interface and Visualization

The interface of the model to Geomview involves two components: the initialization and the update functions. Typically, the initialization function is called during the model initialization and is responsible from setting up the initial geometry. The integrator does not make any function calls specifically for the initialization of Geomview apart from its invocation and resetting. It is up to the model to initially configure Geomview geometry and visualization options.

The second function is the one pointed to by the variable `SS_Data.geomFunc` after the model initialization. This is the the function which updates Geomview at a particular framerate by sending the updated transformation matrices to Geomview. The prototype for this function is

```
int UpdateGeomview(double x[], double p[], int chart);
```

This function is called by the flow iterator at every time step. Most often, the model programmer defines a model parameter `frameRate`, which determines the period with which the Geomview updates are sent out. Then, this function usually computes the transformation matrices for various objects in the geometry and send them to Geomview through the use of the `SS_GeomviewSend` utility function. Moreover, a common practice is to enclose the update commands in the following form

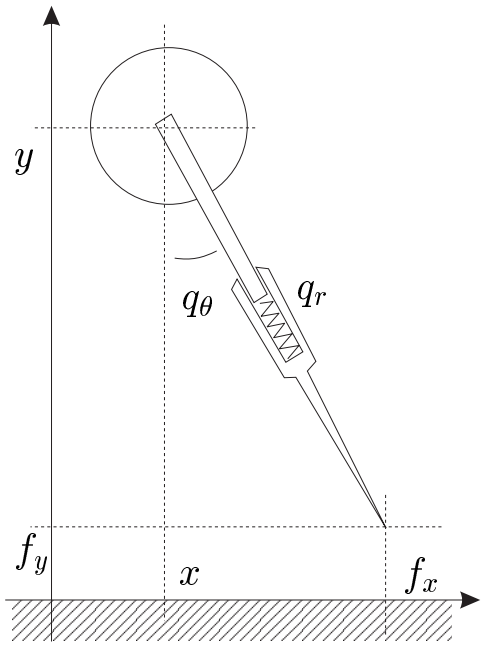


Figure 4: The spring loaded inverted pendulum(SLIP) leg model.

```
int UpdateGeomview(double *x, double *p, int chart) {
  double        f[256];
  int           num;

  SS_GeomviewSend(&SS_Data.geomview, "(freeze Camera)");
  num = Transform(f, x, p, chart);
  SS_GeomviewTransform(&SS_Data.geomview, num, f);
  SS_GeomviewSend(&SS_Data.geomview, "(redraw Camera)");
}
```

This ensures that camera updates are not done during the update, avoiding unnecessary display updates which would slow down the visualization. Note that this example does not implement the framerate feature and would be extremely slow due to the very high number of Geomview updates at every time step.

4 The Spring-Loaded Inverted Pendulum: An Example Dynamical System

4.1 The Continuous Mathematical Model

The SLIP model considered in this example is shown in Figure 4. The leg is assumed massless and the body a point mass at the hip joint. During stance the leg is free to rotate around its toe and the mass is acted upon by a radial spring with potential $U(q_r)$. In flight the mass

is considered as a projectile acted upon by gravity. We assume there are no losses in either the stance or flight phases.

Under these assumptions the stance dynamics are given by

$$\begin{aligned} \ddot{x} &= \frac{D_{q_r} U(k, q_r) \sin(q_{\theta t})}{m} \\ \ddot{y} &= -\frac{D_{q_r} U(k, q_r) \cos(q_{\theta t})}{m} - g \end{aligned}$$

where k is the spring constant and m is the body mass.

4.2 The Hybrid Locomotion Model

During steady state locomotion, our model goes through four consecutive charts: ascent, descent, compression and decompression. Among these, ascent and descent form the *flight* phase, whereas compression and decompression form the *stance* phase. The model allows transitions between these modes only in the sequential order that is given above.

The transition from ascent to descent occurs when the vertical velocity of the body during flight changes sign from positive to negative. The transition from descent to compression occurs when the tip of the leg during descent touches the ground. Then, the system goes into decompression when the spring reaches maximal compression and it starts decompression. Finally, the ascent phase is reentered when the leg reaches maximal extension during decompression.

The boundary functions associated with each of these transitions are defined below.

$$\begin{aligned} b_{asc \to desc}(x, y) &= \dot{y} \\ b_{desc \to comp}(x, y) &= y - q_{rt} \cos(q_{\theta t}) \\ b_{comp \to decomp}(x, y) &= -\dot{q}_r \\ b_{decomp \to asc}(x, y) &= q_{rl} - q_r \end{aligned}$$

where q_{rt}, q_{rl} and $q_{\theta t}$ are the touchdown leg length, liftoff leg length and the touchdown leg angles, respectively. Note that q_r and q_θ can easily be written as functions of x and y through a polar coordinate transformation.

In our simple model $q_{\theta t}$ is actually used as a control variable. We assume that during flight we are able to swing the leg to any desired angle relative to the ground, which then determines when and at which configuration the leg touches the ground.

4.3 The SimSect Implementation of SLIP

This section describes the implementation of the SLIP model described above in SimSect. Most of the example code is included for illustration purposes and should clarify many of the concepts involved in defining a hybrid model in SimSect.

4.3.1 The State Space

The state space for the SimSLIP implementation is defined as follows

$$\mathbf{x} := [x, y, \dot{x}, \dot{y}, f_x, f_y, q_{\theta t}]$$

The SimSLIP implementation defines the following state names for initialization file purposes.

$$\mathbf{x_{names}} := [\texttt{x}, \texttt{y}, \texttt{xdot}, \texttt{ydot}, \texttt{footx}, \texttt{footx}, \texttt{touchdown\_angle}]$$

Note that the states f_x, f_y and $q_{\theta t}$ are bookkeeping states and are only updated at chart transitions.

4.3.2 The Model Parameters

The SLIP model makes use of several parameters configurable from within the initialization file. The parameters and their associated names are defined as follows.

$$\begin{aligned} \mathbf{p} &:= [m, q_{rt}, q_{rl}, g_{st}, g_{fl}, fps, q_{r0}, i, j, k] \\ \mathbf{p_{names}} &:= [\texttt{body\_mass}, \texttt{q\_rt}, \texttt{q\_rl}, \texttt{g\_stance}, \texttt{g\_flight}, \texttt{framerate}, \texttt{q\_0}, \texttt{spri}, \texttt{sprj}, \texttt{k}] \end{aligned} \quad (1)$$

Note that i, j and q_{r0} parametrize the spring law in the following form.

$$\begin{aligned} U_{(i,j)}(q_r, q_{r0}, k) &:= \frac{k}{|i.j|} P_i(-sgn(j)[P_j(q_r) - P_j(q_{r0})]) \\ P_l(x) &:= x_l, l \in N \end{aligned}$$

4.3.3 Some Macro Definitions

The following macro definitions are used in all the function definitions and make the model code much more readable.

```
/* Parameters */
#define M_BODY          p[0]
#define Q_RT            p[1]
#define Q_RL            p[2]
#define GSTANCE         p[3]
#define GFLIGHT         p[4]
#define FRAMESPERSEC    p[5]
#define SPR_REST        p[6]
#define SPR_I           p[7]
#define SPR_J           p[8]
#define SPR_K           p[9]
```

```
#define NUM_OF_PARAMS   (10)

/* Names of charts */
#define ASCENT_CHART        0
#define DESCENT_CHART       1
#define COMPRESSION_CHART   2
#define DECOMPRESSION_CHART 3
```

4.3.4 Model Initialization

The first task of the model initialization function is to define the default initial state and default parameters for the model as well as their names. This is then followed by informing the integrator which functions the model definition provides and a call to `SS_SetupSystem` to finalize the initialization.

```
int slip_InitSystem(void)
  static const double state[] = {
    0.0, 0.9, 1.0, 0.0, 0.0, 0.0, 0.0
  };
  static const char *stateNames[] = {
    "x", "y", "xdot", "ydot", "footx", "footy", "touchdown_angle"
  };
  static const double params[] = {
    50.48, 1.0, 1.0, 10, 10, 120, 1.0, 1.0, -2.0, 1000
  };
  static const char *paramNames[] = {
    "body_mass", "q_rt", "q_rl", "g_stance", "g_flight",
    "framerate", "q_0", "spri", "sprj", "k"
  };
  static int numStates = NUM_OF_STATES;
  static int numParams = NUM_OF_PARAMS;
  static int numPartitions = 4;
  static int numAuxFunc = 6;

  SS_Data.vfFunc = slip_VectorField;
  SS_Data.trFunc = slip_Transition;
  SS_Data.vcFunc = slip_ValidateChart;
  SS_Data.dcFunc = slip_DefineChart;
  SS_Data.ipFunc = slip_InitPartition;
  SS_Data.geomFunc = slip_Geomview;
  SS_Data.auxFunc = slip_AuxFunc;

  return SS_SetupSystem(numStates, state, stateNames, numParams, params,
                        paramNames, numPartitions, numAuxFunc);
}
```

4.3.5 Initializing the Partition

The initialization of the partition in the SLIP model involves determining which chart the specified initial confitions correspond to and then initialize the Geomview component.

The model assumes that the initial leg length is q_{rt}. Together with the initial leg angle and body position, this determines whether the tip of the leg is touching the ground or not. The velocity of the body, in turn, determines which chart in flight or stance the system is initially in.

```
int slip_InitPartition(double* xnew, int* new_chart, double x[],
                       double p[], int chart)
{
  double tipy = x[1] - Q_RT*cos(x[6]);

  memcpy(xnew, x, NUM_OF_STATES*sizeof(double));

  if (tipy < 0) { /* Start at stance */
    double tipx = x[0] + x[1]*tan(x[6]);
    double qrdot = (x[0] * x[2] + x[1] * x[3])
                   / sqrt(x[0] * x[0] + x[1] * x[1]);

    xnew[4] = tipx;
    xnew[5] = 0.0;

    if (qrdot < 0) { /* This is compression */
      *new_chart = COMPRESSION_CHART;
      SS_Message("Starting in COMPRESSION\n");
    } else {
      *new_chart = DECOMPRESSION_CHART;
      SS_Message("Starting in DECOMPRESSION\n");
    }

  } else { /* Start at flight */
    if (x[3] > 0) {  /* This is ascent */
      *new_chart = ASCENT_CHART;
      SS_Message("Starting in ASCENT\n");
    } else {
      *new_chart = DESCENT_CHART;
      SS_Message("Starting in DESCENT\n");
    }
  }

  slip_GeomviewInit(p);

  return NO_ERROR;
}
```

4.3.6 The Vector Field

The flight charts and the stance charts have different vector field definitions. During ascent and descent, the body is acted upon by gravity only. During compression and decompression, it also feels the spring force in addition to gravity. Note that the function `spring_derivative()` defines the derivative of the potential function defined in Equation 1 and hence gives the negative of the spring force.

```
static double spring_derivative(double qr, double qr0, double k,
                                int i, int j)
{
  if (i == 2 && j == 1) return k*(qr-qr0);

  return k*i/(fabs(i)*fabs(j))*pow(-(pow(qr,j)-pow(qr0,j))*j/fabs(j),i-1)
        *(-fabs(j)*pow(qr,j-1));
}

int slip_VectorField(double* xdot, double x[], double p[], int chart)
{
  if (chart == ASCENT_CHART || chart == DESCENT_CHART) {
    xdot[0] = x[2];
    xdot[1] = x[3];
    xdot[2] = 0.0;
    xdot[3] = -GFLIGHT;
    xdot[4] = xdot[5] = 0.0;
  } else if (chart == COMPRESSION_CHART || chart == DECOMPRESSION_CHART) {
    double      legx = x[0] - x[4];
    double      legy = x[1] - x[5];
    double      theta = atan2(legx, legy);
    double      rho = sqrt(legx*legx + legy*legy);
    double      force;

    force = -spring_derivative(rho, SPR_REST, SPR_K, SPR_I, SPR_J);

    xdot[0] = x[2];
    xdot[1] = x[3];
    xdot[2] = force*sin(theta)/M_BODY;
    xdot[3] = force*cos(theta)/M_BODY - GSTANCE;
    xdot[4] = xdot[5] = 0.0;
  } else {
    SS_FatalError("slip_VectorField", INVALID_PARAM_ERROR);
    return INVALID_PARAM_ERROR;
  }

  return NO_ERROR;
}
```

4.3.7 Defining the Charts

After the partition is initialzed and after each chart transition, the `slip_DefineChart()` function is called. Depending on what the next chart is, this function creates the necessary stopping functions. In the SLIP model, each chart has only one associated stopping function (in addition to the system defined final time and measurement stopping functions), which correspond to the boundary functions defined in Section 4.2.

```
int slip_DefineChart(SS_ComputeFlow *cf, int chart)
{
  int           errno;
  SS_StopFunc   sf;

  sf.type = STOP_TYPE_RIGHT;
  sf.action = slip_Boundary;

  if (chart == ASCENT_CHART) {
    sf.index = 0;
  } else if (chart == DESCENT_CHART) {
    sf.index = 1;
  } else if (chart == COMPRESSION_CHART) {
    sf.index = 2;
  } else if (chart == DECOMPRESSION_CHART) {
    sf.index = 3;
  } else {
    SS_FatalError("slip_Transition", INVALID_PARAM_ERROR);
    return INVALID_PARAM_ERROR;
  }

  if ((errno = SS_AddStopFunc(cf, &sf)) < 0)
    return errno;

  return NO_ERROR;
}
```

Note that the `index` field in the stopping function structure determines which boundary function is evaluated by that particular stopping function. The boundary function are defined with the following piece of code.

```
int slip_Boundary(double* val, double x[], double p[], int chart, int index)
{
  switch (index) {
  case 0:
    /* The ascent-descent boundary */
    *val = x[3];
    break;
```

```
  case 1:
    /* The descent-compression boundary */
    *val = x[1] - Q_RT*cos(x[6]);
    break;

  case 2: {
    /* The compression-decompression boundary */
    double legx = x[0] - x[4], legy = x[1] - x[5];
    double xdot = x[2], ydot = x[3];

    *val = - (legx * xdot + legy * ydot) / sqrt(legx * legx + legy * legy);
    break;
  }
  case 3: {
    /* The decompression-ascent boundary */
    double legx = x[0] - x[4], legy = x[1] - x[5];

    *val = Q_RL - sqrt(legx*legx + legy*legy);
    break;
  }
  default:
    SS_FatalError("slip_Boundary", INVALID_PARAM_ERROR);
    return INVALID_PARAM_ERROR;
  }

  return NO_ERROR;
}
```

4.3.8 The Transition Function

The transition function determines the new state and chart after a boundary crossing. In the SLIP model, there is only one possible cyclic chart sequence: ascent → descent → compression → decompression, which is implemented by the transition function. The state of the system does not change except during a transition from descent to compression (touchdown transition) and a transition from decompression into ascent (liftoff transition). The former transition sets, the states f_x and f_y to their new values after the foot touches the ground. The latter transition, however, sets the state $q_{\theta t}$ to the negative of the liftoff angle to preserve neutrally stable symmetric SLIP orbits.

```
int slip_Transition(int* trans, int* newChart, double* xNew, double x[],
                    double p[], int chart)
{
  SS_StopFunc   sf;
```

```
memcpy(xNew, x, NUM_OF_STATES*sizeof(double));

SS_GetStopFunc(&sf, 1);

if (sf.state == STOP_FUNC_CROSSING) {

  switch (chart) {
  case ASCENT_CHART:

    *newChart = DESCENT_CHART;
    SS_Message("Transition: ASCENT -> DESCENT\n");
    break;

  case DESCENT_CHART:

    xNew[4] = x[0] + Q_RT*sin(x[6]);
    xNew[5] = x[1] - Q_RT*cos(x[6]);

    *newChart = COMPRESSION_CHART;
    SS_Message("Transition: DESCENT -> COMPRESSION\n");
    break;

  case COMPRESSION_CHART:

    *newChart = DECOMPRESSION_CHART;
    SS_Message("Transition: COMPRESSION -> DECOMPRESSION\n");
    break;

  case DECOMPRESSION_CHART: {
    double    legx = x[0] - x[4];
    double    legy = x[1] - x[5];

    /* Make the trivial touchdown decision */
    xNew[6] = atan2(legx, legy);

    *newChart = ASCENT_CHART;
    SS_Message("Transition: DECOMPRESSION -> ASCENT\n");
    break;
  }
  default:
    SS_FatalError("slip_Transition", INVALID_PARAM_ERROR);
    return INVALID_PARAM_ERROR;
  }

  *trans = SS_TRUE;
```

```
  } else {
    *trans = SS_FALSE;
  }

  return NO_ERROR;
}
```

Note that the state/chart combination which evaluate to negative boundary function should also be considered invalid states. In our simple illustrative example, however, we do not explicitly check for these cases in the chart validation.

4.3.9 The Auxiliary Functions

The data to be recorded is computed by the auxiliary functions. For our SLIP example, we record the time together with the position and velocity of the body. Note that the number of auxiliary functions is defined in the model initialization function and must be consistent with the number of variables filled in by the auxiliary function component.

```
int slip_AuxFunc(double *f, double x[], double p[], int chart)
{
  f[0] = x[TIME];
  f[1] = x[0];
  f[2] = x[1];
  f[3] = x[2];
  f[4] = x[3];

  return NO_ERROR;
}
```

4.3.10 Chart Validation

The simple SLIP model has only one potential mode of failure: The hopper toppling over and the body hitting the ground. In any one of the charts, if $y < 0$ at any point on the trajectory, the state-chart combination becomes invalid. The following implementation of the chart validation function ensures proper detection of this failure mode.

```
int slip_ValidateChart(int* in_chart, double x[], double p[], int chart)
{
  *in_chart = SS_TRUE;

  if (x[1] < 0) { /* The body is underground */
    SS_Message("Error: Body toppled over\n");
    *in_chart = SS_FALSE;
  }
```

```
  return NO_ERROR;
}
```

4.3.11 The Geomview Interface

The first component of the Geomview interface is the initialization function. This function loads the geometry file `slip.oogl` into Geomview and then defines a 5x100 grid platform that the runner will be running on. The details of the platform construction are not important. However, the interested reader might refer to [3] for details of how to create and manipulate geometries in Geomview.

```
#define GRID_XSTEPS     5
#define GRID_YSTEPS     100

void slip_GeomviewInit(double p[])
{
  int           countx, county;
  int           color_flag;
  char          str[2048];
  double        ptx, pty, height;

  SS_GeomviewClear(&SS_Data.geomview);
  SS_GeomviewSendFile(&SS_Data.geomview, "slip.oogl");
  SS_GeomviewSend(&SS_Data.geomview, "(backcolor Camera 0.25 0.21 0.5)");


  SS_GeomviewSend(&SS_Data.geomview, "(freeze Camera)\n");
  /* Define the platform with the specified number of grid cells */
  SS_GeomviewSend(&SS_Data.geomview,
                  "(read geometry { define platform_geom {CMESH ");
  sprintf(str, "%i %i\n", (int)GRID_XSTEPS + 3, (int)GRID_YSTEPS + 3);
  SS_GeomviewSend(&SS_Data.geomview, str);

  for (county = 0; county < GRID_YSTEPS + 3; county++) {
    for (countx = 0; countx < GRID_XSTEPS + 3; countx++) {

      color_flag = (((countx+county) % 2) == 1);

      height = 0.0;
      if (countx % ((int) GRID_XSTEPS + 2) == 0) {
        ptx = -0.5 + countx*(1.0 / (GRID_XSTEPS+2));
        height = -0.1;
      } else {
        ptx = -0.5 + (countx-1) * 1.0 / GRID_XSTEPS;
      }
```

```
      if (county % ((int) GRID_YSTEPS + 2) == 0) {
        pty = -1.0 + county * (25.0 / (GRID_YSTEPS+2));
        height = -0.1;
      } else {
        pty = -1.0 + (county-1) * 25.0 / GRID_YSTEPS;
      }

      sprintf(str, " %f %f %f %f %f %f %f\n",
              ptx, pty, height,
              (color_flag)?0.333:0.2352,
              (color_flag)?0.4196:0.7019,
              (color_flag)?0.1843:0.443,
              1.);
      SS_GeomviewSend(&SS_Data.geomview, str);
    }
  }
  SS_GeomviewSend(&SS_Data.geomview, "}})");
  SS_GeomviewSend(&SS_Data.geomview, "(redraw Camera)\n");
}
```

The next component of the interface is the function which gets called by the integrator at every trajectory data point. The main purpose of this function is to implement the appropriate frame rate by periodically sending updates of the transformation matrices to Geomview. Note that the actual computation of the transformation matrcies is done in `slip_Transform`, which is also detailed below. This kind of decomposition of the interface into initialization, handling of the frame rate and the computation of the transforms is very typical.

```
int slip_Geomview(double x[], double p[], int chart)
{
  double        f[48];
  int           num;
  static double lastTime = 0;
  static        int frameNo = 0;

  if (x[TIME] == 0) {
    lastTime = 0;
    frameNo = 0;
  }

  if (x[TIME] >= lastTime) {

    SS_GeomviewSend(&SS_Data.geomview, "(freeze Camera)\n");
    num = slip_Transform(f, x, p, chart);
```

```
    SS_GeomviewTransform(&SS_Data.geomview, num, f);
    SS_GeomviewSend(&SS_Data.geomview, "(redraw Camera)\n");

    frameNo++;
    lastTime += 1.0 / FRAMESPERSEC;
  }

  return NO_ERROR;
}
```

The following function computes the transformation matrices for the SLIP geometry and the camera to be sent to Geomview. In the exampel SLIP model, the hopper is defined as a single object consisting of a sphere and a vector for the leg. The first transformation matrix rotates and positions this object according to the current system state. The second transormation matrix sets up the camera such that it follows the hopper from a convenient angle to provide a nice visual animation of the simulation.

```
int slip_Transform(double *f, double *x, double *p, int chart)
{
  double angle;
  double z_angle = 0.9*PI/2, x_angle = 1.8*PI/4;
  double camera_x;

  memset(f, 0, 2*16*sizeof(double));

  /* This section transforms the body object to its current position
     and orients the leg based on what the leg angle is. Note that
     during stance, the foot position relative to the body position
     determines the leg angle. However, during flight, we assume that
     the leg immediately goes to its touchdown position because it
     has no mass.
  */
  if (chart == COMPRESSION_CHART || chart == DECOMPRESSION_CHART) {
    angle = atan2(-(x[0]-x[4]), x[1]);

    f[0] = 1.0;
    f[5] = f[10] = cos(angle);
    f[9] = -(f[6] = sin(angle));
    camera_x = f[13] = x[0];
    f[14] = x[1];
    f[15] = 1.0;
  } else {

    f[0] = 1.0;
    f[5] = f[10] = cos(x[6]);
```

```
    f[9] = -(f[6] = sin(x[6]));
    camera_x = f[13] = x[0];
    f[14] = x[1];
    f[15] = 1.0;
  }

  /* This transformation matrix follows the hopper from a nice angle. */
  f[16 + 0] = cos(z_angle);
  f[16 + 1] = sin(z_angle);
  f[16 + 2] = 0;
  f[16 + 4] = -cos(x_angle) * sin(z_angle);
  f[16 + 5] = cos(x_angle) * cos(z_angle);
  f[16 + 6] = sin(x_angle);
  f[16 + 8] = sin(x_angle) * sin(z_angle);
  f[16 + 9] = -sin(x_angle) * cos(z_angle);
  f[16 + 10] = cos(x_angle);

  f[16 + 12] = 3.8;
  f[16 + 13] = camera_x;
  f[16 + 14] = 1.4;
  f[16 + 15] = 1.0;

  return 2;
}
```

5 The Compliant Hexapod: A More Complicated Example

This section describes the details of the Compliant Hexapod Model definition in SimSect. The hexapod model originally motivated the SimSect project and is one of the example model definitions.

5.1 The Continuous Mathematical Model

In this section, we derive the continous dynamics of the hexapod within a particular chart. The definition of the hybrid model is completed in Section 5.3, with a specification of the chart transitions.

Section 5.1.1 describes the derivation of the physical model and assumptions followed by Section 5.1.2 and 5.1.3, where the force and torque vectors acting on the hexapod body are computed. Finally, Section 5.1.4 gives the equations of motion for the hexapod followed by Section 5.1.8 where several coordinate transformations involved in the computations are given.

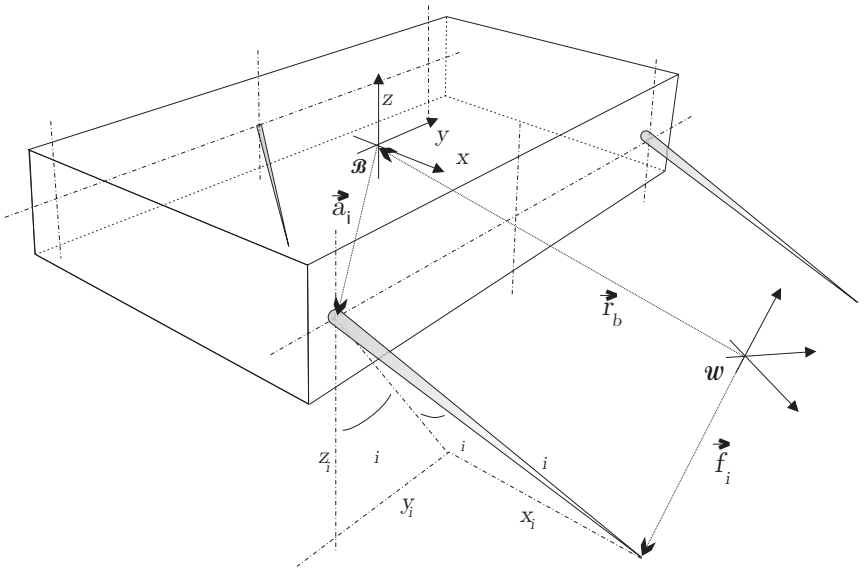


Figure 5: The compliant hexapod model.

5.1.1 The System Structure

Figure 5 shows the structure of the hexapod model. The system consists of a rigid body with six degrees of freedom, whose position and orientation are described by $\mathbf{r_b}$ and $\mathbf{R_b}$, respectively. Two coordinate frames, $\mathcal{B}$ and $\mathcal{W}$ are defined, the former attached to the hexapod body and the latter an inertial frame where the dynamics are formulated.

The legs are attached to the rigid body at fixed points $\mathbf{a_i}$, in the body frame. Each leg has complete spherical freedom. Each leg ha a very small mass m_t at the toe, useful in modeling the flight behavior of the legs as well as a simple friction model during stance. Note that $(\mathbf{r_b}, \mathbf{R_b})$, $\mathbf{v_i}$ and $\mathbf{f_i}$ are related through a simple coordinate transformation (see Section 5.1.8).

Associated with each leg, there is a radial and a torsional spring on the θ direction, as well as torque control on the ϕ degree of freedom. These springs and the hip actuation result in forces and torques being applied to the rigid body. In Section 5.1.2, we derive these forces and torques for a single generalized leg, leading to the formulation of the system dynamics in Section 5.1.4.

5.1.2 Analysis of a Single Leg

Our formulation of the equations of motion for the hexapod model individually computes the ground reaction forces for each leg. To this end, it suffices to analyze a generic leg parametrized by its attachment and touchdown points, $\mathbf{a_i}$ and $\mathbf{f_i}$, respectively (see Figure 6). The force and torque balance on the massless leg result in the following equalities.

Coordinate Frames	
$\mathcal{W}$	inertial world frame
$\mathcal{B}$	body frame
Leg states and parameters	
$\mathbf{a_i}$	leg attachment point in $\mathcal{B}$
$\mathbf{f_i}$	toe position in $\mathcal{W}$
$\mathbf{v_i} := [\theta_i, \phi_i, \rho_i]^T$	current leg state in spherical body coordinates
$\overline{\mathbf{v}}_\mathbf{i} := [v_{x_i}, v_{y_i}, v_{z_i}]^T$	current leg state in Cartesian body coordinates
leg_i	stance flag for leg i
ρ_{td}	leg length at touchdown
ρ_{lo}	leg length at liftoff
k_{r_i}, d_{r_i}	Radial leg spring and damping constants
k_{θ_i}, d_{θ_i}	Angular leg spring and damping constants
ρ_{0_i}, θ_{0_i}	Radial and angular spring rest positions
K	Exponential saturation coefficient for the radial leg spring
States	
$\mathbf{r_b}$	body position
$\mathbf{R_b}$	body orientation
$\dot{\mathbf{r}}_\mathbf{b}$	translational velocity of body
$\mathbf{w_b}$	angular velocity of body
$\mathbf{x} := [\mathbf{R_b}, \mathbf{w_b}, \mathbf{r_b}, \dot{\mathbf{r}}_\mathbf{b}, \mathbf{f_i}, \dot{\mathbf{f}}_\mathbf{i}]$	Lumped state vector used in the implementation
Forces and Torques	
F_{r_i}	radial leg spring force
τ_{θ_i}	leg bending torque in θ_i direction
τ_{ϕ_i}	leg hip torque in ϕ_i direction
System Parameters	
$\mathbf{M_0}$	body inertia matrix in body coordinates
$\mathbf{M}$	body inertia matrix in world coordinates
m_b	body mass
m_t	toe mass
$h_t(x, y)$	Height of the terrain at cartesian coordinates (x,y)
$n_t(x, y)$	Normal vector to terrain surface at coordinates (x,y)
Controller Parameters	
t_c	Period of rotation for a single leg
ϕ_s	Slice of leg sweep for the slow phase
Actuator Model	
K_w, K_τ	Motor speed and torque constants
i_{a_i}, v_{a_i}	Motor armature current and voltage
k_g	Motor gear ratio
w_{s_i}	Motor shaft speed
τ_{s_i}	Motor shaft torque

Table 1: Notation used for the Compliant Hexapod model.

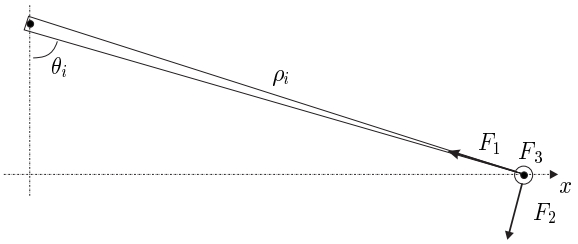


Figure 6: *Analysis of a single leg in the plane defined by the leg and the z-axis of* $\mathcal{B}$.

$$\begin{aligned} F_1 &= F_{r_i} \\ F_2 &= \frac{\tau_{\theta_i}}{\rho_i} \\ F_3 &= \frac{\tau_{\phi_i}}{\rho_i \cos\theta_i} \end{aligned}$$

The body experiences the negative of the ground reaction force on the leg, yielding effective force and torque vectors acting on the center of mass. Projection of these on $\mathcal{B}$ for each leg $i = 1, ..., 6$ yields,

$$\begin{aligned} \mathbf{F_i} &= \begin{bmatrix} -\sin\theta_i & -\cos\theta_i & 0 \\ -\cos\theta_i \sin\phi_i & \sin\theta_i \sin\phi_i & -\cos\phi_i \\ \cos\theta_i \cos\phi_i & -\sin\theta_i \cos\phi_i & -\sin\phi_i \end{bmatrix} \cdot \\ &\qquad \begin{bmatrix} F_{r_i} \\ \tau_{\theta_i}/\rho_i \\ \tau_{\phi_i}/(\rho_i \cos\theta_i) \end{bmatrix} \\ \tau_{\mathbf{i}} &= (\overline{\mathbf{v}}_{\mathbf{i}} + \mathbf{a_i}) \times \mathbf{F_i} \end{aligned}$$

Note that in the equations above, one also needs to specify the spring and damper models for the radial and lateral angular degrees of freedom for each leg. In the compliant hexapod model, we chose to have linear springs and viscous dampers for both degrees of freedom. Moreover, the radial spring saturates with an exponential force past a certain length, modeling the limited extension range of actual leg implementations.

$$\begin{aligned} F_{r_i} &:= \begin{cases} -k_{r_i}(\rho_i - \rho_{0_i}) - d_{r_i}\dot{\rho}_i & \rho_i < \rho_{l_i} \\ -K\exp(\rho_i - \rho_{0_i}) - k_{r_i}(\rho_i - \rho_{0_i}) - d_{r_i}\dot{\rho}_i & \text{otherwise.} \end{cases} \\ \tau_{\theta_i} &:= -k_{\theta_i}(\theta_i - \theta_{0_i}) - d_{\theta_i}\dot{\theta}_i \end{aligned} \tag{2}$$

5.1.3 Total Force and Torque on the Body

The cumulative effect of all the legs on the body is simply the sum of the individual contributions, together with the gravitational force. Expressed in $\mathcal{W}$, the force and torque vectors are given by

$$\mathbf{F_T} = \begin{bmatrix} 0 \\ 0 \\ -m_b g \end{bmatrix} + \mathbf{R_b} \sum_{i=1}^{6} leg_i \mathbf{F_i} \tag{3}$$

$$\tau_\mathbf{T} = \mathbf{R_b} \sum_{i=1}^{6} leg_i \tau_\mathbf{i} \tag{4}$$

where we define

$$leg_i := \begin{cases} 0 & leg\ i\ is\ in\ flight \\ 1 & leg\ i\ is\ in\ stance \end{cases}$$

5.1.4 Rigid Body Dynamics

The dynamics of a rigid body under external force and torque actuation is governed by the following equations [2].

$$\begin{aligned} \ddot{\mathbf{r}}_\mathbf{b} &= \frac{\mathbf{F_T}}{m_b} \\ \mathbf{M}\dot{\mathbf{w}}_\mathbf{b} &= -J(\mathbf{w_b})\mathbf{M}\mathbf{w_b} + \tau_\mathbf{T} \\ \dot{\mathbf{R}}_\mathbf{b} &= J(\mathbf{w_b})\mathbf{R_b} \end{aligned}$$

where we have

$$\begin{aligned} J(\begin{bmatrix} w_x & w_y & w_z \end{bmatrix}^T) &:= \begin{bmatrix} 0 & -w_z & w_y \\ w_z & 0 & -w_x \\ -w_y & w_x & 0 \end{bmatrix} \\ \mathbf{M} &:= \mathbf{R_b}\mathbf{M_0}{\mathbf{R_b}}^{-1} \end{aligned}$$

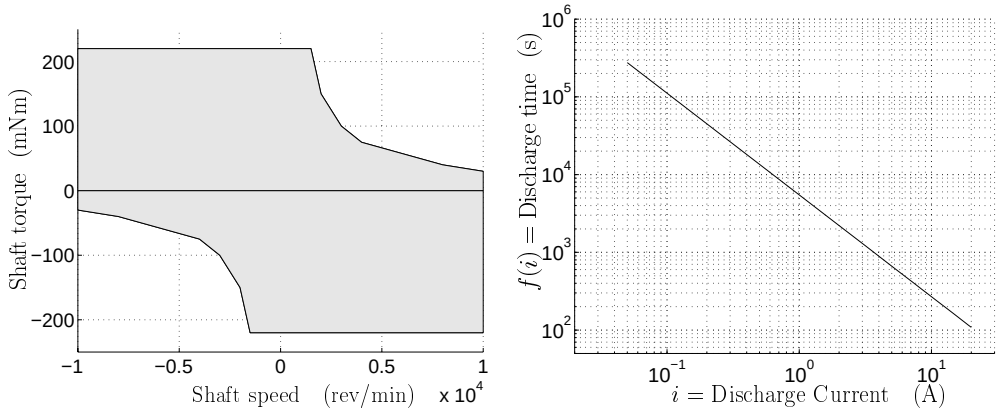
5.1.5 Feet Dynamics

If a leg is in flight, then its toe experiences the negative of the force computed in Section 5.1.2 for that particular leg. Hence, the associated vector field takes the following form.

$$\begin{bmatrix} \mathbf{f_i} \\ \dot{\mathbf{f}}_\mathbf{i} \end{bmatrix} = \begin{bmatrix} \dot{\mathbf{f}}_\mathbf{i} \\ -\mathbf{F_i}/m_t \end{bmatrix}$$

If, on the other hand, a leg is in stance, then the vector field incorporates a first order friction model to model the motion of the toe.

$$\begin{aligned} \dot{\mathbf{f}}_\mathbf{i} &= \begin{cases} 0 & \texttt{if} \quad \| F_t \| < k_f \| F_n \| \\ \frac{\|F_t\| - k_f \|F_n\|}{\|F_t\|} & \texttt{otherwise} \end{cases} \\ \ddot{\mathbf{f}}_\mathbf{i} &= 0 \end{aligned}$$



(a) (b)

Figure 7: *(a) Torque-speed curve for the Maxon RE118751 20W DC motor. (b) Battery discharge curve for the Panasonic 12V-2.2A battery.*

where F_t and F_n are the components of $\mathbf{F_i}$ tangent and normal to the terrain surface, respectively and k_f is the surface friction coefficient.

Using these equations, the vector field for each toe is defined based on the mode of the corresponding leg.

5.1.6 Actuator Model

In the compliant hexapod, we also incorporate a simple model of the hip actuation. This model imposes realistic limitations on the torque capabilities of the actuators. Figure 7(a) portrays the torque-speed curve for a Maxon RE118751 DC motor. This curve characterizes the maximum torque deliverable by the motor as a function of shaft speed.

In addition to these torque limits, the model also incorporates an estimation scheme for motor voltages v_{a_i} and currents i_{a_i} using the commanded leg torques, under the assumption that the electrical dynamics are negligible relative to the mechanical dynamics. In consequence, we have

$$\begin{aligned} i_{a_i} &= \tau_{\phi_i}/(K_\tau k_g) \\ v_{a_i} &= i_{a_i} r_a + K_w w_{\phi_i}/k_g \end{aligned}$$

where K_τ and K_w are motor constants, k_g is the gear reduction ratio of 33:1 at the motor shaft and r_a is the motor armature resistance. Note that in this simple formulation, the only influence of the actuator model on the mechanical dynamics is through the limits on the maximum available torque. In fact, v_{a_i} and i_{a_i}, are not computed by SimSect, but are extracted later from the simulation data.

5.1.7 Battery Model

The discharge characteristics of off-the-shelf small batteries are usually given by plots of discharge time vs constant discharge current. Figure 7(b) is such a discharge curve for the Panasonic 12V 2.2Ah lead-acid battery used. In our discharge model, we use this curve, together with an approximation of the PWM electronics driving the DC motors to estimate the duration of autonomous operation.

First, we derive a method for estimating battery discharge in terms of a continuous time function of varying discharge current. This can be accomplished with

$$\frac{dC(t)}{dt} = -\frac{1}{f(i_a(t))}$$

where $C(t)$ is the percent "energy" left in the battery, and $f(i)$ is the battery discharge curve in functional form. During the hexapod operation, all six motors draw current and contribute to the battery discharge together. Due to the H-bridge output stages of the motor drives, the motor currents add up, yielding the battery lifetime equation

$$1 - \int_0^t \frac{d\sigma}{f(\sum_{i=1}^{6} |i_i(\sigma)|)} = 0.$$

Our battery model detects in simulation the zero crossing of this function, which yields the effective lifetime of the battery. Note that this model does not take into account more elaborate components such as the battery voltage drop as a function of current and discharge time or the effects of ambient temperature. Similar to the electrical motor model, the battery model is implemented outside the SimSect simulation environment.

5.1.8 Relevant Coordinate Transformations

- Positional leg states in the body frame

$$\begin{aligned} \overline{\mathbf{v}}_\mathbf{i} &= [v_{x_i}, v_{y_i}, v_{z_i}]^T = {\mathbf{R}_\mathbf{b}}^{-1} (\mathbf{f_i} - \mathbf{r_b}) - \mathbf{a_i} \\ \mathbf{v_i} &= [\theta_i, \phi_i, \rho_i]^T = \begin{bmatrix} \arctan(x_i/\sqrt{y_i^2 + z_i^2}) \\ \arctan 2(y_i, -z_i) \\ \sqrt{x_i^2 + y_i^2 + z_i^2} \end{bmatrix} \end{aligned}$$

- Leg velocities in the body frame

$$\begin{aligned} \dot{\overline{\mathbf{v}}}_\mathbf{i} &= [\dot{v}_{x_i}, \dot{v}_{y_i}, \dot{v}_{z_i}]^T = {\mathbf{R}_\mathbf{b}}^{-1} \left(\dot{\mathbf{f}}_\mathbf{i} - \dot{\mathbf{r}}_\mathbf{b} - \dot{\mathbf{R}}_\mathbf{b} (\overline{\mathbf{v}}_\mathbf{i} + \mathbf{a_i}) \right) \\ \dot{\mathbf{v}}_\mathbf{i} &= [\dot{\theta}_i, \dot{\phi}_i, \dot{\rho}_i]^T = \begin{bmatrix} (D + A v_{z_i})/(\sqrt{C} F) \\ (-v_{z_i} \dot{v}_{y_i} + v_{y_i} \dot{v}_{z_i})/C \\ E/\sqrt{F} \end{bmatrix} \end{aligned}$$

- Leg accelerations in the body frame

$$\begin{aligned}
\ddot{\overline{\mathbf{v}}}_\mathbf{i} &= [\ddot{v}_{x_i}, \ddot{v}_{y_i}, \ddot{v}_{z_i}]^T = \mathbf{R_b}^{-1}\left(\ddot{\mathbf{f}}_\mathbf{i} - \ddot{\mathbf{r}}_\mathbf{b} - \ddot{\mathbf{R}}_\mathbf{b}(\overline{\mathbf{v}}_\mathbf{i} + \mathbf{a_i}) - 2\dot{\mathbf{R}}_\mathbf{b}\dot{\overline{\mathbf{v}}}_\mathbf{i}\right) \\
\ddot{\mathbf{v}}_\mathbf{i} &= [\ddot{\theta}_i, \ddot{\phi}_i, \ddot{\rho}_i]^T \\
&= \begin{bmatrix} \frac{-2v_{x_i}(Av_{z_i}+D)^2+(C+v_{x_i}^2)(3B^2v_{x_i}-2BC\dot{v}_{x_i}+C^2\ddot{v}_{x_i}-Cv_{x_i}(\dot{v}_{y_i}^2+\dot{v}_{z_i}^2+v_{y_i}\ddot{v}_{y_i}+v_{z_i}\ddot{v}_{z_i}))}{C^{3/2}(C+v_{x_i}^2)^2} \\ 2B(v_{z_i}\dot{v}_{y_i} - v_{y_i}\dot{v}_{z_i}) - C(v_{z_i}\ddot{v}_{y_i} - v_{y_i}\ddot{v}_{z_i})/C^2 \\ -(4E^2 + 4F(\dot{v}_{x_i}^2 + \dot{v}_{y_i}^2 + \dot{v}_{z_i}^2 + v_{x_i}\ddot{v}_{x_i} + v_{y_i}\ddot{v}_{y_i} + v_{z_i}\ddot{v}_{z_i}))/(4F^{3/2}) \end{bmatrix}
\end{aligned}$$

where

$$\begin{aligned}
A &:= v_{z_i}\dot{v}_{x_i} - v_{x_i}\dot{v}_{z_i} \\
B &:= v_{y_i}\dot{v}_{y_i} + v_{z_i}\dot{v}_{z_i} \\
C &:= v_{y_i}^2 + v_{z_i}^2 \\
D &:= v_{y_i}^2\dot{v}_{x_i} - v_{x_i}v_{y_i}\dot{v}_{y_i} \\
E &:= v_{x_i}\dot{v}_{x_i} + v_{y_i}\dot{v}_{y_i} + v_{z_i}\dot{v}_{z_i} \\
F &:= v_{x_i}^2 + v_{y_i}^2 + v_{z_i}^2
\end{aligned}$$

and

$$\begin{aligned}
\ddot{\mathbf{r}}_\mathbf{b} &= \frac{\mathbf{F_T}}{m_b} \\
\dot{\mathbf{w}}_\mathbf{b} &= \mathbf{M}^{-1}(-J(\mathbf{w_b})\mathbf{M}\mathbf{w_b} + \tau_\mathbf{T}) \\
\dot{\mathbf{R}}_\mathbf{b} &= J(\mathbf{w_b})\mathbf{R_b} \\
\ddot{\mathbf{R}}_\mathbf{b} &= J(\dot{\mathbf{w}}_\mathbf{b})\mathbf{R_b} + J(\mathbf{w_b})\dot{\mathbf{R}}_\mathbf{b}
\end{aligned}$$

- Toe coordinates in the world frame

$$\begin{aligned}
\overline{\mathbf{v}}_\mathbf{i} &= \begin{bmatrix} \rho_i \sin\theta_i \\ \rho_i \cos\theta_i \sin\phi_i \\ -\rho_i \cos\theta_i \cos\phi_i \end{bmatrix} \\
\mathbf{f_i} &= \mathbf{R_b}\left(\overline{\mathbf{v}}_\mathbf{i} + \mathbf{a_i}\right) + \mathbf{r_b}
\end{aligned}$$

5.2 The Open-Loop Control of Locomotion

In this section, we describe a four-parameter family of open-loop controllers for the hexapod model, which achieves forward and backward running as well as in-place and differential turning in the abscence of any sensory feedback of the rigid body and leg states. The only feedback occurs locally at the actuators to implement a PD controller for each hip, tracking certain periodic reference trajectories. The algorithms that we describe in this section are tailored to demonstrate the intrinsic stability properties of the compliant hexapod morphology and emphasize its ability to operate without a sensor-rich environment.

An alternating tripod pattern governs both the running and turning controllers, where the legs forming the left and right tripods are synchronized with each other and are 180^o out of phase with the opposite tripod, as shown in Figure 8.

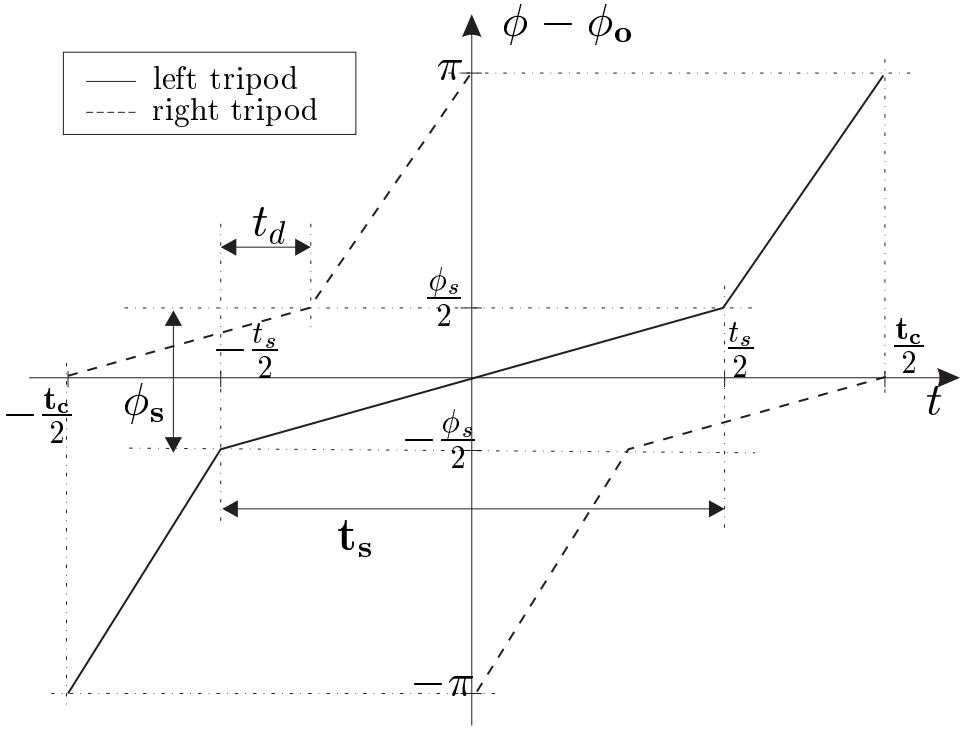


Figure 8: *The motion profiles for left and right tripods.*

5.2.1 Running

The running controller's target trajectories for each tripod are periodic functions of time, parametrized by four variables: t_c, t_s, ϕ_s and ϕ_o. The period of both profiles is t_c. In conjunction with t_s, it determines the duty factor of each tripod. In a single cycle, both tripods go through their slow and fast swing phases, covering ϕ_s and $2\pi - \phi_s$ of the complete rotation, respectively. The duration of double support t_d (where all six legs are in contact with the ground) is determined by the duty factors of both tripods. Finally, the ϕ_o parameter offsets the motion profile with respect to the vertical (see Figure 8). Note that both profiles are monotonically increasing in time; but they can be negated to obtain backward running.

Control of running behavior is achieved by modifying these parameters for a particular desired behavior during locomotion.

5.2.2 Turning

We have developed two different controllers for two qualitatively different turning modes: turning in place and differential turning during running.

The controller for turning in place employs the same leg profiles as for running except that contralateral sets of legs rotate in opposite directions. This results in the hexapod turning in place in the direction determined by the rotational polarity of the left and right sets of legs. Note that the tripods are still synchronized internally, maintaining three supporting legs on the ground. Similar to the control of the forward locomotion speed, the rate of turning

depends on the choice of the particular motion parameters, mainly t_c and ϕ_s.

In contrast, we achieve turning during forward locomotion by introducing differential perturbations to the forward running controller parameters for contralateral legs. In this scheme, t_c is still constrained to be identical for all legs, which admits differentials in the remaining profile parameters, ϕ_o and t_s, while ϕ_s remains unchanged. Two new gain parameters, Δt_s and $\Delta\phi_o$ are introduced. Consequently, turning in $+x$(right) direction is achieved by using $\mathbf{u}_l = [t_c, t_s + \Delta t_s, \phi_s, \phi_o + \Delta\phi_o]$ and $\mathbf{u}_r = [t_c, t_s - \Delta t_s, \phi_s, \phi_o - \Delta\phi_o]$ for the legs on the left and right sides, respectively.

5.3 The Hybrid Hexapod Model

5.3.1 The State Space

The state space of the hybrid hexapod model is more than just the state of the rigid body. The states of each leg are also needed in order to be able to compute the vector field of the previous section. Consequently, the following lumped vector defines the state space for the hybrid model.

$$\mathbf{x} \;:=\; [\mathbf{R_b}, \mathbf{w_b}, \mathbf{r_b}, \dot{\mathbf{r}}_\mathbf{b}, \mathbf{f_i}, \dot{\mathbf{f}}_\mathbf{i}]$$

5.3.2 The Partition Structure

The mathematical model of the previous section does not address any of the discrete transitions that take place during locomotion. Discrete events in the model come from the collisions of the feet with the ground, changing the states of the flags in Equation 3.

Each leg can be in one of two modes, stance or flight. Hence, the compliant hexapod model has a total of $2^6 = 64$ discrete charts capturing all the possible structural configurations of the hexapod. Note that this structure does not take into account the possible collision of neither the rigid body nor the legs themselves with the ground. It is assumed that collisions occur only between the toes and the ground surface.

In each of these 64 charts, the model obeys the continuous dynamics of the previous section, with the appropriate leg flags leg_i.

5.3.3 Chart Transitions

When the rigid body and the legs are moving, each of the legs can transition from stance to flight and vice versa independently. Consequently, it is possible to transition from each of the 64 charts defined in the previous section, to the remaining 63. In this section, we describe a structured way of defining these transition conditions together with their associated boundary functions.

In a particular chart, we define transition conditions associated with each leg, depending on their current mode. There are two types of boundary functions, "liftoff" and "touchdown" associated with each leg which is either in stance or flight, respectively.

- Liftoff condition

The liftoff boundary function is the component of the ground reaction force acting on the toe, normal to the ground surface. When this force becomes positive, the leg is forced to lift off. This function takes the following form.

$$b_i^l(\mathbf{x}) \;:=\; -\langle n_t(f_{x_i}, f_{y_i})|\mathbf{F_i}\rangle$$

This condition alone, however, is not adequate to ensure proper definition of the liftoff model. The exponential component in the radial spring definition of Equation 2 is critical in proper functioning of the liftoff condition.

- Touchdown condition

  During flight, a leg touches down when its toe reaches the ground. The boundary function associated with this condition takes the following form.

$$b_i^t(\mathbf{x}) \;:=\; \left\langle \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T |\mathbf{f_i}\right\rangle - h_t(f_{x_i}, f_{y_i})$$

  Upon detection of this boundary crossing, the veocity of the toe $\dot{\mathbf{f}}_\mathbf{i}$ is set to zero. The model then checks the normal omponent of the ground reaction force. The chart transition does not occur if this force is positive, to avoid an invalid chart-state pair.

  The main reason why such an exception is handled by the model is the damping in the leg. Due to the plastic collision and the toe velocity being set to zero, there is a step change in the derivatives of the spherical leg states at every touchdown. This in turn results in discontinuities in the ground reaction force as a result of damping in the leg, yielding this exception. Note that because the normal toe force is positive, the toe starts traveling away from the surface, validating the skipping of the touchdown transition.

5.4 Implementation of the Model

5.4.1 Configuration Parameters

Table 2 summarizes the configuration parameter symbol names for the compliant hexapod model, together with their default values.

Table 2: Compliant hexapod model configuration parameters.

Parameter	Default	Description
Hexapod platform parameters		
body_mass	6	Body mass (kg)
leg_mass	0.01	Toe mass (kg)
friction_coeff	0.5	Surface friction coefficient
q_rt	0.18	Touchdown leg length (m)
q_rl	0.181	Liftoff leg length (m)
g	10	Gravitational acceleration (m/s^2)
max_torque	3	Maximum leg torque output(Nm)
max_speed	10000	No load motor rotation speed (rpm)
speed_constant	61.261056	K_w (rad/Vs)
torque_constant	16.3e-3	K_τ (Nm/A)
armature_resist	1.34	r_a (ohm)
gear_ratio	0.0333	k_g
torque_loss	.5	Actuator efficiency (unused)
rho0_k...rho5_k	500, 1000, 500, 500, 1000, 500	Radial leg spring constants (N/m)
rho0_d...rho5_d	10, 20, 10, 10, 20, 10	Radial leg damping constants (Ns/m)
rho0_r0...rho5_r0	0.2, 0.2, 0.2, 0.2, 0.2, 0.2	Radial leg spring rest lengths (m)
theta0_k...theta5_k	100, 200, 100, 100, 200, 100	Angular leg spring constants (Nm/rad)
theta0_d...theta5_d	0.4, 0.8, 0.4, 0.4, 0.8, 0.4	Angular leg damping constants (Nms/rad)
theta0_r0...theta5_r0	0.2, 0.2, 0.2, -0.2, -0.2, -0.2	Angular leg spring rest lengths (rad)
attach0_x, attach0_y, attach0_z	0.1, -0.2, 0	Leg 0 attachment point coordinates (m)
attach1_x, attach1_y, attach1_z	0.1, 0, 0	Leg 1 attachment point coordinates (m)
attach2_x, attach2_y, attach2_z	0.1, 0.2, 0	Leg 2 attachment point coordinates (m)
attach3_x, attach3_y, attach3_z	-0.1, -0.2, 0	Leg 3 attachment point coordinates (m)
attach4_x, attach4_y, attach4_z	-0.1, 0.2, 0	Leg 4 attachment point coordinates (m)
attach5_x, attach5_y, attach5_z	-0.1, 0.2, 0	Leg 5 attachment point coordinates (m)
continued on next page		

terrain_type	Height function
"flat"	$h(x, y) := 0$
"sinusoid"	$h(x, y) := a_1 \sin(a_2 x) \cos(a_3 y)$
"sloped"	$h(x, y) := \begin{cases} a_1 x + a_2 y + a_3 & y > 0 \\ a_3 & otherwise \end{cases}$

Table 3: Height functions for different types of terrain. a_1, a_2 and a_3 correspond to terrain_arg1, terrain_arg2 and terrain_arg3, respectively.

continued from previous page		
Parameter	Default	Description
J_11, J_12, J_13	0.2, 0, 0	Body intertia matrix first row
J_21, J_22, J_23	0, 0.08 0	Body intertia matrix second row
J_31, J_32, J_33	0, 0.,0.4	Body intertia matrix third row
terrain_type	"flat"	Terrain type
terrain_arg1	0.03	First argument for terrain function
terrain_arg2	15	Second argument for terrain function
terrain_arg3	15	Third argument for terrain function
Controller related parameters		
control_type	"open-loop"	Controller type
KP	10	Proportional control gain
KD	0.5	Derivative control gain
Open-loop controller parameters		
cycle_time	1	t_c (s)
sweep_angle	0.75	ϕ_s (rad)
flight_time	0.4	t_f (s)
leg_offset	0	ϕ_o (rad)
Visualization related parameters		
grid_xsize	5.0	X-axis size of the terrain grid
grid_ysize	5.0	Y-axis size of the terrain grid
grid_xsteps	50	Number of X-steps for the terrain grid
grid_ysteps	50	Number of Y-steps for the terrain grid
framerate	120	Visualization frame rate
frame_width	640	Width of the frame for save to disk.
frame_height	480	Height of the frame for save to disk

5.4.2 Terrain Surface

The hexapod model implementation currently supports three types of terrain. The configuration parameter terrain_type selects the surface to be used in the simulation. There are also three other parameters modulating the way the currently selected terran is generated.

The functions terrain_height() and terrain_normal() in SS_HexTerrain.c defines the height and surface normal functions for different types of terrain.

5.4.3 Initializing the Partition

The function `hex_InitPartition` is called during system initialization and sets up the initial state of the model. In the hexapod model, its main function is to determine which legs are in stance and which legs are in flight, based on the state of the body and assuming an alternating tripod posture as the initial conditions. This is accomplished by looking at the rest states of the legs at an alternate tripod posture and determining which legs should be touching the ground based on their toe positions.

The initialization also involves the setting up of the visualization subsystem by loading the appropriate geometry file into Geomview and sending the necessary initialialization commands.

5.4.4 Defining the Chart

The function `hex_DefineChart` performs the definition of a new chart. Following a transition, the trajectory iterator removes all the previously defined stopping functions. Consequently, the definition of the chart involves redefining all the stopping functions to be used in the new chart.

The `hex_Boundary` defines all the boundary functions associated with the liftoff and touchdown conditions. The indices corresponding to each of these functions are as follows.

Liftoff:	2*leg_no
Touchdown:	2*leg_no+1

The chart definition then adds stopping functions with the liftoff boundary functions for legs in stance and with the touchdown boundary functions for legs in flight.

5.4.5 Chart Validation

In the hexapod model, there are many cases where the state may be inconsistent with the current chart that the system is in. The following list summarizes these cases and gives the action taken by the `hex_ValidateChart` function. Essentially, most of these conditions invalidate cases where one or more of the boundary functions associated with a chart have negative values corresponding to anomalous cases.

- For all charts, if $\mathbf{r_b}_z < h_t(\mathbf{r_b}_x, \mathbf{r_b}_y)$, then the body toppled over and is underground. Return an error state to stop integration.
- For a leg in stance, if $\rho_i > \rho_{0_i}$, the leg length exceeded the liftoff length. Display a warning and continue integration.
- For a leg in stance, if $\langle n_t(f_{x_i}, f_{y_i}) | \mathbf{F_i} \rangle > 0$, the normal leg force is positive. Return an error state to stop integration.
- For a leg in flight, if $\left\langle \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T | \mathbf{f_i} \right\rangle - h_t(f_{x_i}, f_{y_i}) < 0$, the leg is underground. Return an error state to stop the integration.

5.4.6 The Vector Field

The function `hex_VectorField` defines the vector field for the compliant hexapod model, on the state space defined in Section 5.3.1. Its definition is included in `SS_HexVectorField.c`. However, `SS_HexForces.c` and `SS_HexTransform` define many of the coordinate transformations and computations of the components for the vector field. Please refer to the source code for the details.

The vector field associated with the first half of the state space, specifically $[\mathbf{R_b}, \mathbf{w_b}, \mathbf{r_b}, \dot{\mathbf{r}}_\mathbf{b}]$, is that of rigid body dynamics and was derived in Section 5.1.4. Note that the computation of the leg forces uses all the state variables including the foot locations.

The second half of the state space consists of the positions and velocities of the toes. The vector fields associated with these are defined in Section 5.1.5.

5.4.7 Visualization

The hexapod model exploits the interconnection of SimSect to Geomview. When the subsystem is on, it visualizes a three dimensional model of the hexapod as it is locomoting over the specified terrain.

The file `hexaped.oogl` includes the initial definition of the hexapod geometry, together with the rigid body, a flat ground, all six legs, the feet and all the associated transformation matrices. This file is loaded into Geomview during initialization for the creation of all objects necessary for the real-time update of the scene.

The file `SS_HexGeomview.c` implements the required functions for the visualization interface. The main two functions are `hex_GeomviewInit` and `hex_Transform`. The first function handles the initialization of the scene by loading the `hexaped.oogl` file into Geomview and by creating and defining the terrain object using the `terrain_height` function

The second function computes the transformation matrices for all the objects in the scene that need to be relocated. Normally, those are the rigid body, the legs and the feet. Consequently, this function computes 13 transformation matrices as functions of the system state. These matrices are then sent to Geomview by the wrapper function `hex_Geomview` function, resulting in the update of the visualized scene.

Note that the wrapper function also implements a frame rate feature to avoid excessive update of the display which would unacceptably slow the integration.

5.4.8 Support Functions

The files `SS_HexForces` and `SS_HexTransform` implements several functions to support the implementations of the functions described above.

One of the most important support functions is `hex_syncStates`. It computes several values used in the vector field as well as the controller. Starting from the main state variables $\mathbf{x}$, it fills in all the required fields in `SS_HexapedData` in the correct order, making sure that at the end, the struct contains consistent values for the current data point. Hence, this function is to be called first by any function that use the associated data, to ensure consistency. Note that this function first compares the current time to the last time it has been called and therefore does not carry out the same computation more than once.

This synchronization function uses many of the other support functions for its functionality. Below is a list of those functions summarizing their functionality. Moreover, the order that they are listed in reflects the order that they should be called in to ensure that the value returned is up to date and consistent. The reason for this requirement is that most of these functions use each other to compute certain things and to avoid redundant computations, imposing an order is necessary.

- `hex_BodyPos(...)`

  Returns the position of the rigid body in $\mathcal{W}$.

- `hex_BodyVel(...)`

  Returns the velocity of the rigid body in $\mathcal{W}$.

- `hex_BodyR(...)`

  Return the rotation matrix for the rigid body.

- `hex_Bodyw(...)`

  Returns the angular velocity of the rigid body in $\mathcal{W}$.

- `hex_BodyRd(...)`

  Returns the derivative of the body rotation matrix.

- `hex_FootPos(...)`

  Returns the position of a particular toe in $\mathcal{W}$.

- `hex_FootVel(...)`

  Returns the velocity of a particular toe in $\mathcal{W}$.

- `hex_LegStateCart(...)`

  Returns the cartesian vector state $\overline{\mathbf{v}}_\mathbf{i}$ for a particular leg.

- `hex_LegVelCart(...)`

  Returns the cartesian vector velocity $\dot{\overline{\mathbf{v}}}_\mathbf{i}$ for a particular leg.

- `hex_LegAngles(...)`

  Returns the leg state in polar coordinates.

- `hex_LegVel(...)`

  Returns the leg velocity in polar coordinates.

- `hex_ComputeLegForceTorque(...)`

  Compute and return the forces and torques produced by a particular leg.

- `hex_FootAccel(...)`

  Compute and return the acceleration of a toe in $\mathcal{W}$.

- `hex_ComputeForceTorque(...)`

  Compute and return the total force and torque on the rigid body in $\mathcal{W}$.

- `hex_BodyAccel(...)`

  Returns the acceleration of the rigid body in $\mathcal{W}$.

- `hex_Bodywd(...)`

  Computes and returns the derivative of the body angular velocity vector in $\mathcal{W}$.

- `hex_BodyRdd(...)`

  Computes and returns the second derivative of the body rotation matrix.

- `hex_LegAccelCart(...)`

  Computes and returns the second derivative of the cartesian leg state vector $\overline{\mathbf{v}}_\mathbf{i}$

- `hex_LegAccel(...)`

  Computes and returns the second derivative of the polar leg state vector $\mathbf{v}_\mathbf{i}$

Note that the functions specific to individual legs must be called for all the legs before proceeding with the `hex_ComputeForceTorque()` function. Please refer to the source code for the particulars of the SimSect implementation.

5.4.9 Defining Controllers

In the hexapod model, the controllers also have their own hybrid structures with vector fields and transition functions. The file `SS_HexControl.c` implements the interface to possible different controllers supported by the model. The configuration parameter `control_type` selects the active controller for the simulation.

The functions in `SS_Control.c` route the hybrid simulation system calls to the appropriate controller. When defining a new controller, the user must provide functions for each of these components and modify `SS_HexControl.c` accordingly.

This structure gives the user the flexibility to define separate charts and vector fields for the controller itself, independent of the structure of the hexapod. However, the current implementation of the integration engine requires the state space and the chart structure to be unique for the overall system, which necessitates some care in the controller implementation. The bit fields in the chart number associated with the hexapod model and the vector field elements of the hexapod should not be modified in the controller functions, unless it is explicitly required for controller functionality.

Currently, the only controller that is implemented is the open-loop controller described in Section 5.2 (`control_type = "open-loop"`). Its implementation is done in `SS_OpenLoop.c`.

5.5 The Implementation of the Open-Loop Controller

5.5.1 Controller Vector Field

The SimSect implementation of the open-loop controller introduces six new states, corresponding to the reference trajectories for each leg. The vector fields associated with each of these states are determined based on the current controller chart, which, for each leg, encodes the current phase of the motion profile of Figure 8.

The fields `SS_OL.legSpeed[leg_no]` hold the vector fields for each of the legs, and are computed at each controller chart transition based on the upcoming parameter set and phase of the leg. The function `hex_OpenLoopVectorField()` implements the controller vector field.

5.5.2 Controller Partition Structure

Throughout the open loop reference trajectory generation, each leg goes through four phases: the two halves of the slow phase and the two halves of the fast phase ($T_0 := [0, t_s/2], T_1 = [t_s/2, t_c/2], T_2 = [-t_c/2, -t_s/2], T_3 = [-t_s/2, 0]$. See Figure 8). Consequently, there are 4^6 different possible combinations of leg phases, yielding 4096 different controller charts. The definition of the controller partition assigns two bits to each leg in the `chart` value of the simulation. Bits 17-6 encode the phases for all 6 legs, while bits 5-0 encode the current touchdown states for the legs.

Note that the transition associated with these chart components are purely time dependent. This underlines the open-loop nature of the controller, where the hexapod state does not affect the controller state in any way except the PD controller which implements local feedback at each leg.

5.5.3 Controller Transitions

The transition function associated with the controller is where the vector field of the corresponding leg is updated. Moreover, changes in controller parameter values take place during transitions from leg phase 3-¿0 and leg phase 1-¿2, where all the legs are assumed to undergo the transition at the same time.

The SimSect implementation of the hexapod model incorporates only two of the controller parameters, t_c and ϕ_s. These parameters are enforced by `SS_OpenLoopVectorField`, which computes the derivatives of the reference trajectories. The vector field for each of the reference trajectory states is given below.

Two functions, `openLoopBoundary` and `halfCycleBoundary` implement the four boundary functions for each leg. While `openLoopBoundary` detects the transition from the slow to fast leg swing, `halfCycleBoundary` detects the middle of slow and fast phases. The transitions initiated by `openLoopBoundary` only update the controller vector field, whereas the transitions of `halfCycleBoundary` are also responsible from changing the controller parameters t_c, t_s, ϕ_s and ϕ_o. See the function `hex_OpenLoopTransition` for details.

5.5.4 The PD controller

The function `hex_OpenLoopControl` implements six simple PD controllers around the reference trajectory encoded by the states associated with the controller. This is where the hip

torques for each leg are computed as a function of the current leg position and the desired reference position.

5.6 Future Work on the Hexapod Model

The current hexapod model has many flaws. The first and the most problematic one is the chattering between different leg modes. Chattering in hybrid dynamical systems occurs when the vector fields of two neighboring charts point toward their common boundary, and the trajectory has to slide along the boundary. The SimSect integrator currenty cannot handle such situations, and ends up switching between two charts at very short time intervals.

Another possible modification to the current model is incorporating a different ground contact model. Currently, collisions with the ground are plastic and cause the small foot mass to lose all its energy. This in turn, results in step changes in the leg damping forces, which sometimes puts the system in an invalid state-chart combination. In consequence, the integrator fails to compute the trajectory. Different ground models include high damping, high stiffness ground contact models, where the feet can penetrate the ground and loses its energy in a continuous fashion. However, this approach might lead to a more complicated model which is undesirable for analysis purposes. Moreover, the high damping and stiffness coefficients for the ground may unacceptably slow down the numerical integration. Modifications to the integration engine may be required.

References

[1] J. G. A. Back and M. Myers. *A Dynamical Simulation Facility for Hybrid Systems.* DsTool documentation.

[2] P. C. Hughes. *Spacecraft Attitude Dynamics.* John Wiley & Sons, New York, 1986.

[3] M. Phillips. *Geomview Software Manual.* The Geometry Center.