THE OHIO STATE UNIVERSITY :: DEPARTMENT OF MECHANICAL ENGINEERING

SLASIM: A Suspension Analysis Program

Presented in Partial Fulfillment of the Requirements for Graduation with Distinction at The Ohio State University

Sage Wolfe

Spring 2010

Defense Committee:
Dr. Dennis A. Guenther, Advisor
Dr. Gary J. Heydinger

Abstract

SAE competitions such as Baja SAE and Formula SAE offer undergraduate engineering students excellent preparatory experience for a career in the automotive sector. Students design and build race cars from scratch, but often do not have the necessary background knowledge to design vehicles that function optimally. Suspensions in particular are often left unanalyzed due to the difficulty posed by both graphical analysis and multibody kinematics packages. The experience gained by students would be much more valuable if they were able to easily analyze their designs. SLASIM, a new MATLAB-based program, attempts to provide a powerful yet user-friendly utility for the novice suspension designer. It is GUI-based and made freely available under the GNU General Public License. While not as powerful as commercially available kinematics packages, SLASIM is a useful tool for rapid design iteration. Finally, since it is open source, it provides an easy platform for modification and enhancement.

Acknowledgements

First and foremost, I would like to thank Denny Guenther for his help and support throughout this project.

I would also like to thank Jeff Chrstos for his advice while planning this project.

Thanks to Professor Gary Kinzel, Professor Bob Siston, and Clayton Petry for help with kinematics.

Data was taken from the Formula SAE team at The Ohio State University (Formula Buckeyes). Thanks to Kyle Merical, Team Captain, for providing this data.

Some graphics were adapted from Dr. Steven Timmins at the University of Delaware. Thanks to him for permission to use his CAD screenshots.

Thanks to Milliken & Milliken for writing a great book.

Table of Contents

Appendix A: SLASIM MATLAB Code

Appendix B: SLASIM User's Guide

Table of Figures

Introduction

Background

The Society of Automotive Engineers (SAE) sponsors several student competitions each year. Two of the most popular competitions are Formula SAE and Baja SAE. Both involve the design and fabrication of a complete vehicle. Formula vehicles are small, open-wheeled race cars (which bear some resemblance to a Formula 1 car), typically powered by 600cc sport motorcycle engines. Baja SAE is designed to mimic, on a smaller scale, the Baja 1000 race which takes place annually in the Baja California Peninsula of Mexico. Although the Baja 1000 has many classes of vehicles, the Baja SAE vehicles resemble off-road go-karts or dune buggies (with a large roll cage).

Both teams offer great experience to undergraduate engineering students; experience which is even a requirement at some companies (1). In both cases, the teams are charged with creating an entire race-ready vehicle (Baja SAE teams are provided with an engine). For undergraduates with little design experience, this can be a somewhat daunting task. A second-year student will likely not have taken a course in either kinematics or finite element analysis, but they are tasked with designing a suspension. As a result, suspensions (along with chassis and other components) are often designed so that they work in a sub-optimal (or even poor) manner.

If students were able to intelligently design suspensions, the experience they gained from their student project teams would be much more valuable. There are two common ways to analyze the kinematics and geometry of a suspension design: graphically, or using a multibody kinematics package. Graphical analysis has the advantage of being simple and low-cost, but suffers from being very slow and only in two dimensions. Also, if done on paper (as opposed to line drawings in CAD), the accuracy may be questionable. Graphical analysis discourages iterative improvements due to its labor intensity.

Multibody kinematics packages solve the issues of accuracy and iteration, but are also labor intensive due to the fact that most students must first learn to use the program before actually analyzing any designs. They are also expensive to license (if they do not have a license available through their university).

SLASIM attempts to present an alternative. Since SLASIM is written in MATLAB, most students will have little trouble using the program. It is also accompanied by a brief User's Guide (Appendix B) which explains how to use the program. Since it is GUI-based, the user does not even need to know how to program in MATLAB — they only need to type the command to launch the GUI, enter data, and press 'Run.' The program then returns a number of plots and suspension parameters for the designer to examine. It will be made available freely, under the terms of the GNU Public License (2).

After the program has completed its calculations and displayed its output, the input and results are saved. The user can then tweak their design, run the program again, and have new results — all within a matter of minutes. While this program lacks the advanced capabilities of a multibody kinematics program (such as collision/interference detection), it is much more powerful than graphical analysis.

Since SLASIM is open-source, free, and written in MATLAB, it also has great potential for other (even unseen) uses. For example, it could be used as a module within a vehicle dynamics analysis program (used to generate kinematics data for use in a simulation). Student project teams could also add other features easily. If spring rate data were known, for example, the program could be adapted to analyze the front and rear suspensions and calculate a multitude of vehicle dynamics metrics. The addition of optimization would prove especially beneficial. The program may even be useful for amateur racers or hobbyists who wish to see what effect a certain suspension modification may have on their vehicle.

Project Objective

In order to decide what the program ought to do, Milliken's *Race Car Vehicle Dynamics* (3) was consulted. Based on this, as well as interviews with experts (4), it was decided which parameters SLASIM would calculate. Table 1, below, shows the parameters calculated by SLASIM. Static parameters are those which are single-valued and only calculated at ride height. Parameters which are plotted, such as camber, are calculated at various positions and plotted as a function of suspension position (in jounce or rebound).

Table 1: Output parameters

Parameter	Type
Caster angle	Static
Mechanical trail	Static
Kingpin angle (inclination)	Static
Scrub	Static
Spindle length	Static
Suspension throughout travel, front view	Plot
Suspension in jounce, 3D	3D Plot
Camber	Plot
Bump steer	Plot
Roll center height	Plot
Roll center location (cornering)	Plot
Front anti-dive	Plot
Rear anti-squat	Plot
Rear anti-lift	Plot

The goal of the program is to solve problems for the end user. The static parameters are needed largely to ensure that they are reasonable in nature (e.g., positive trail). The plotted parameters are those which are tweaked by the user to enhance performance (e.g., tuning negative camber gain with suspension jounce).

Program Functionality

User Input

In order to analyze the suspension, the user must input the position of the suspension at ride height, as well as some other general information about the suspension. Figure 1, below, shows a representative (front) SLA suspension and the associated coordinate system. The origin (0, 0, 0) is at half-track (halfway between the contact patches of the left and right tires) on the ground plane. X is positive towards the left of the vehicle, Y is positive upwards, and Z is positive towards the front of the vehicle. The program was designed to take input in units of inches. Although this sign convention differs from the SAE J670 standard, the users are most likely unaware of this sign convention. This sign convention was chosen because it seemed to be the simplest and easiest to use.

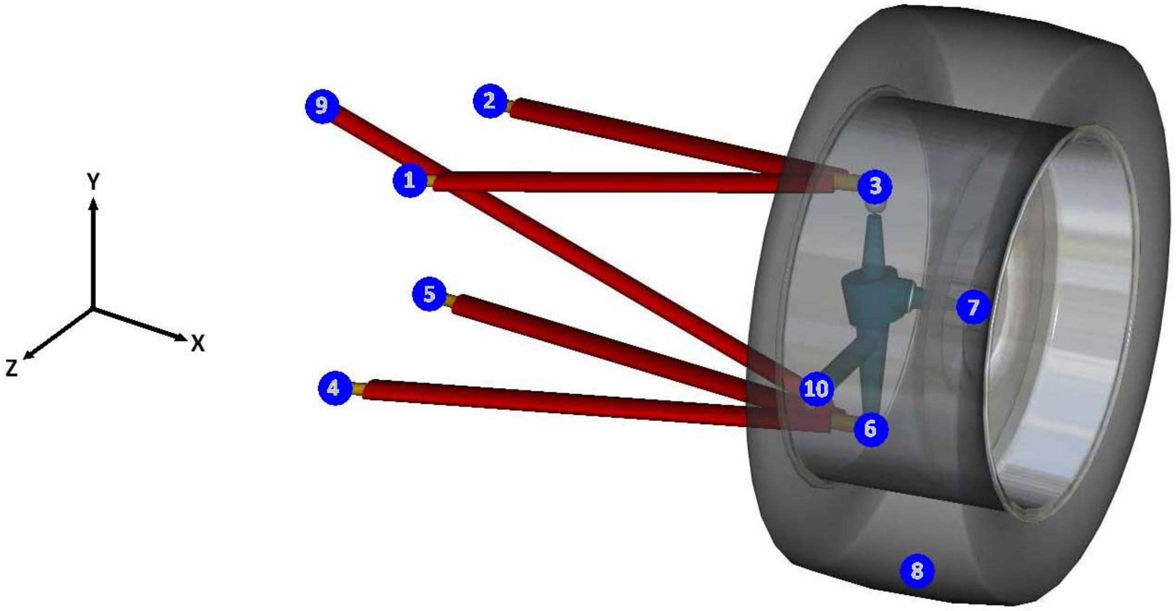

Figure 1: Coordinate system and input points (adapted from Timmins, (5))

As seen from Figure 1, there are a total of ten points that the program needs in order to calculate the kinematics of the suspension. Table 2, below, shows the coordinates that are entered by the user. There are also several other items entered by the user, summarized in Table 3, below.

Table 2: Input coordinates

Point	Member	Description
1	Upper A-arm	Front inner ball joint
2	Upper A-arm	Rear inner ball joint
3	Upper A-arm	Outer ball joint
4	Lower A-arm	Front inner ball joint
5	Lower A-arm	Rear inner ball joint
6	Lower A-arm	Outer ball joint
7	Wheel	Center
8	Wheel	Center, ground plane
9	Tie rod	Inner ball joint
10	Tie rod	Outer ball joint

Table 3: Other user input

Description	Type
Travel	Distance
Brake bias (front)	Percentage
Outboard brakes	Yes/no
Front suspension	Yes/no
Wheelbase length	Distance
CG height	Distance

The coordinates are, again, simply XYZ locations (in units of inches). Note that “Middle of tire, Ground Plane” is requested rather than contact patch. The exact location of the contact patch may not be known (it may, for instance, lie closer to the inside of the tire), and kinematically it is irrelevant. Nevertheless, this point may be referred to as the contact patch, both in this document and the code, because it is relatively close. Travel is needed in order to know how far the suspension should be ‘moved’ by the program. Brake bias, wheelbase length, and CG (center of gravity) height are used when calculating the magnitude of suspension ‘anti-’ effects (and can be replaced with reasonable estimates if not explicitly known). Brake location and suspension type (front or rear) are both used to determine which (if any) suspension ‘anti-’ effects to calculate. The GUI used to input data is shown below in Figure 2. After the user presses the ‘RUN’ button, the input data is placed in memory and the program executes.

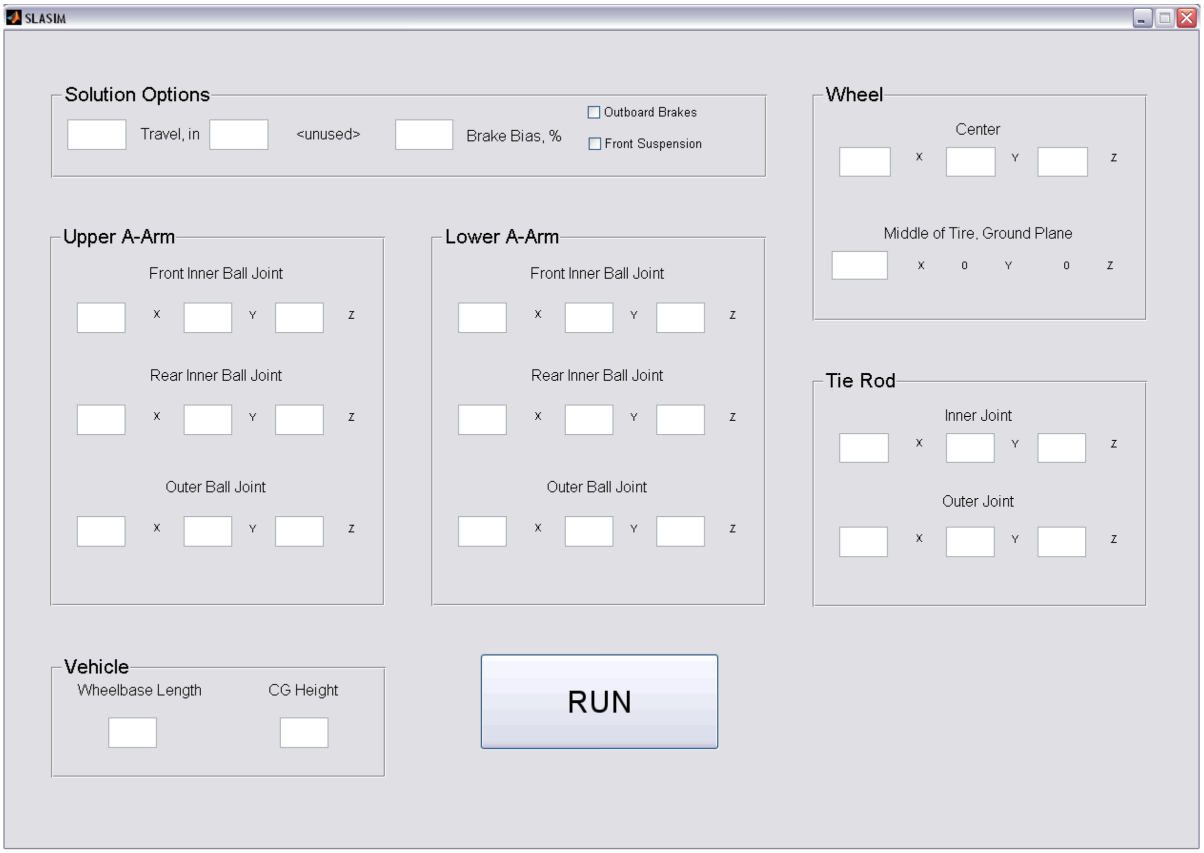

Figure 2: SLASIM GUI

Program Overview

SLASIM consists of five files, which accomplish various tasks. **SLASIM.fig** is the figure file which contains the GUI (Figure 2). **SLASIM.m** is an M-file (script file) which contains instructions for the GUI. It was largely generated by MATLAB, and serves to place the user's input data into matrices. When the 'RUN' button is pressed, **SLASIM_Call.m** is executed. This M-file (also a simple script file) contains program logic and also serves to return static parameters and create plots. **Kinematics.m** (a function file) calculates the suspension positions (in jounce or rebound) based on the travel and the initial positions. **Geometry.m** (also a function file) uses the suspension positions to calculate parameters of interest at different locations. All of the M-files are contained in Appendix A. It is recommended to refer to the code often. The overall program logic is shown below in Figure 3.

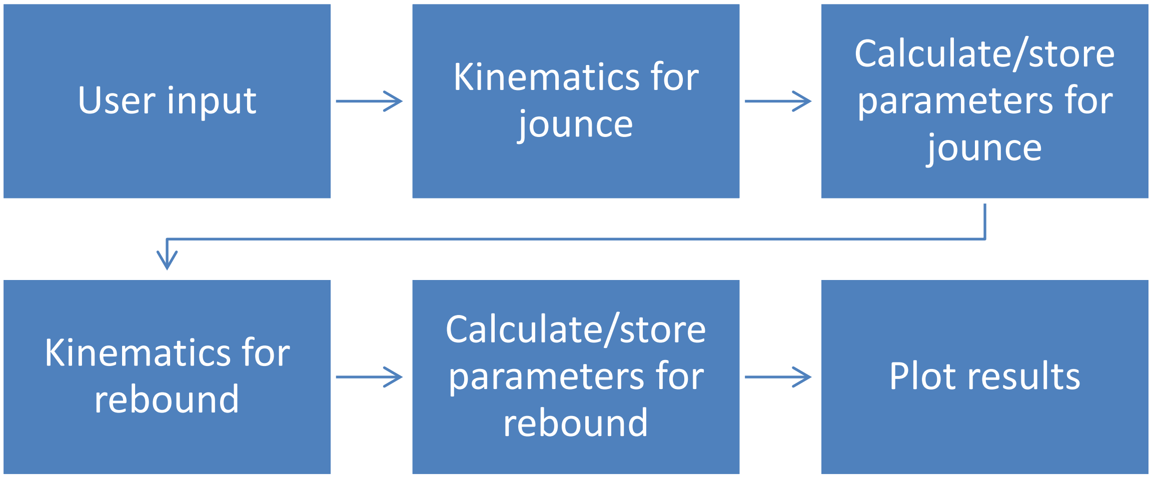
User input
Kinematics for jounce
Calculate/store parameters for jounce
Kinematics for rebound
Calculate/store parameters for rebound
Plot results

Figure 3: Overall program logic

This is the process contained within the **SLASIM_Call.m** file. The details of how each M-file functions will be presented in-depth next.

SLASIM_Call.m

In the **SLASIM.m** script file (the one that contains instructions pertaining to the GUI) the input data is placed in global variables (matrices). If the user leaves a field blank, the variable *FLAG* (note: variable names are italicized, and input variable names are all capital letters) is set equal to one.

At the beginning of the **SLASIM_Call.m** file, the same global variables (*UPPER, LOWER, SOLN, WHEEL,* and *TIEROD*) are initialized so they can be accessed. The script file also has a place to manually input data. To save time while iterating, the user may not wish to use the GUI. This provides a place where variables can be changed without inadvertently altering the program. The user can change their input in the Editor window and then use MATLAB's run button.

In this input portion of the file, the user also has the ability to change two important parameters: *n_step* and *n_points*. *N_step* sets the number of suspension locations (the number of 'steps') to be calculated in each direction. By default, it is 25, which means the program will calculate 25 suspension positions in jounce (from ride height up to half of the total travel) and 25 suspension positions in rebound. The default value of 25 steps is enough to provide a smooth curve for plotted data (such as camber angle) without unduly increasing program runtime. It is important to note that increasing *n_step* does not increase the accuracy of the calculated parameters. It simply causes the program to calculate parameters at more points.

N_points, on the other hand, does affect accuracy. Within the **Kinematics.m**, there are several numerical solutions to position problems. This is discussed in depth later, but essentially the program is trying to find the intersection of a circle and sphere in 3D space. The sphere is defined by its distance and origin. However, defining a circle in 3-space requires the use of parametric equations. *N_points* instructs the program on how many points to generate along these circles. This value is set to 10,000 by default.

Next, the program initializes global variables for the output of the **Kinematics.m** and **Geometry.m** function files. Global variables are used here in lieu of function returns. The program then initializes variables to store both the suspension positions and parameters at each step. At this time, the program checks to see if the missing data flag has been tripped. If so, the program warns the user that data is missing and that unexpected output may result.

After this, the program calls the **Kinematics.m** function. The arguments passed to it are *UPPER, LOWER, SOLN, WHEEL, TIEROD, n_step, n_points,* and *direction. Direction* tells the function whether the suspension is supposed to move in jounce (direction 1) or in rebound (direction -1).

After the **Kinematics.m** function has finished for the increasing case (jounce, direction 1), the suspension parameters need to be calculated for each position. The **Geometry.m** function is written so that it takes in a single suspension position and calculates the parameters at that position. In order to calculate this for all the positions in jounce, it is called iteratively (in a ‘for’ loop). These parameters are then stored in results matrices.

While most of the parameters stored in the results matrices are calculated by **Geometry.m**, the steer angle at each suspension position is calculated within **SLASIM_Call.m**. It is assumed that at ride height the steer angle is zero (i.e., zero static toe). In order to calculate the steer angle, the heading of the steering knuckle at ride height is subtracted from the current heading of the steering knuckle. Steer angle is taken to be positive when steered inwards towards the centerline of the vehicle. While static toe may not actually be zero, the change in toe as a function of suspension travel is the important parameter.

At this time, the front view (XY plane) suspension positions are also plotted. Since plotting 25 (or whatever the value of *n_step* is) suspension positions in each direction would create a cluttered plot, only five, evenly spaced positions are plotted (ride height, maximum jounce, halfway between maximum jounce and ride height, maximum rebound, and halfway between maximum rebound and ride height).

The caster angle, mechanical trail, kingpin angle (inclination), scrub, and spindle length are displayed to the user at this point using the **fprintf()** function in MATLAB.

Next, the **Kinematics.m** function is called for the decreasing case (rebound, direction -1). The same (looping) process takes place: position data is formatted for input, **Geometry.m** is called for each suspension position in rebound, front view suspension positions are plotted, and parameters are stored in results matrices.

Now that all the parameters have been tabulated, the program can begin producing plots. First, the program checks to see if any suspension ‘anti-’ effects are present. For front-suspensions, this is limited to front anti-dive. For rear suspensions, this includes anti-squat and anti-lift. If no suspension ‘anti-’ effects are present, the portions of the results matrices which store the values will be filled with either zeros or ‘NaN’ (not a number). To check this, first the maximum value of the ‘anti-’ effect is calculated using the **max()** function in MATLAB. If the maximum value is zero, there is no ‘anti-’ effect present. Also, if the maximum value is not less than or equal to infinity, no ‘anti-’ effect is present (note: this would evaluate to true for any actual number – the program is checking for NaN). If there are any ‘anti-’ effects present, the program plots them as a function of suspension position.

After plotting ‘anti-’ effects, the program plots the camber curve, bump steer curve, and roll center height. Here, roll center height is plotted for symmetric suspension movements (moving together in jounce or rebound). Since data is only entered for one suspension, the suspension is assumed to be symmetrical and thus the roll center is necessarily located at half-track.

Finally, the roll center location during cornering is calculated. Again, since data is only entered for one suspension, the design is assumed to be symmetrical left/right. It is also assumed that during steady-state cornering, the movement of the suspension is opposite and equal. That is, if the outside wheel goes two inches into jounce, the inside wheel is assumed to go two inches into rebound.

To calculate the location, the following approach is taken (note that this is done iteratively, a total of *n_step* times). First, a total of four points are extracted from the results matrices. Point 1 is the location of the center of the wheel at the ground plane (roughly, the contact patch). Point 2 is roll center height. Both of these are taken to be 2D points in the front view (XY plane). Points are extracted from the increasing (jounce) and decreasing (rebound) cases. Again, the magnitude of jounce for one pair of points is the same as the magnitude of rebound for the other set of points.

Each of these pairs of points defines a line from the bottom of the wheel through what the roll center would be if the suspension was moving symmetrically. However, the roll center is not the reason this line is drawn as it is. The following procedure details how the roll center can be found graphically:

1) In the front view, find the intersection point of the A-arms for one side of the suspension. This defines the instant center of the wheel on that side of the vehicle.
2) Repeat for the opposite suspension. Again, this intersection point is the instant center of the steering knuckle and wheel.
3) Draw lines connecting the instant centers with the contact patch of their respective tires.
4) The intersection of the lines from (3) is the roll center. (Note that if the suspension were symmetrical, this procedure dictates that the roll center is located at half-track.)

So, the lines defined by the pairs of points from the results matrix also go through the instant centers and contact patches of their respective sides. From step (4) above, it is clear that the intersection of these two lines (defined by the pairs of points taken from the results matrix) defines the roll center location. Since each line is a linear equation, the roll center location is found by simple matrix algebra. This location is then plotted for the user, along with the direction of the turn (i.e., the direction of the turn that would produce these suspension movements) relative to the plot.

Kinematics.m

The purpose of the **Kinematics.m** function is to take in the arguments *UPPER, LOWER, SOLN, WHEEL, TIEROD, n_step, n_points,* and *direction* and return all the relevant suspension positions in a given direction. It begins by initializing global variables used to communicate the results, namely *Q_3D, P_3D, tierod_obj, close_point, wheel_center,* and *contact_patch*. *Q_3D* and *P_3D* correspond to the upper A-arm outer ball joint and the lower A-arm outer ball joint locations, respectively. *Tierod_obj* is, naturally, the location of the tie rod outer ball joint. *Close_point* is a variable used for debugging purposes. *Wheel_center* and *contact_patch* refer to the locations of the wheel center and contact patch. These five locations (the A-arm and tie rod outer ball joints, wheel center, and contact patch) are the only points which move, the other five input locations (A-arm and tie rod inner ball joints) are fixed.

After initializing variables, the function finds the locations along the A-arm instant axes closest to the outer ball joint (*cp_upper* and *cp_lower*, where 'cp' stands for 'close point'). This is a point, along the line from the rear inner ball joint to the front inner ball joint, which is closest to the outer ball joint of the same A-arm. To find this, first the unit vector, *u_unit*, from the rear inner ball joint to the front inner ball joint is calculated. Then a vector, *v*, from the rear inner ball joint to the outer ball joint is created. This point, *cp_upper* or *cp_lower*, can be thought of as being offset from the rear inner ball joint by some scalar value, in the direction of *u_unit*. The magnitude of this scalar offset is given by the dot product of *v* and *u_unit*. So, *cp_lower* is equal to the sum of the location of the lower A-arm rear inner ball joint location and the scalar offset multiplied by *u_unit*. At this time, the effective lengths of the A-arms are also calculated, using the 2-norm of the vector from the close point to the outer ball joint.

Next, the step size is calculated, based on the amount of travel and the number of steps (*n_step*). The distance between the upper A-arm outer ball joint and the lower A-arm outer ball joint is also calculated. This distance is fixed by the steering knuckle geometry, and will be used to later find the upper A-arm outer ball joint location.

After the *cp_upper* and *cp_lower* are known, these points are used to generate circles which describe the locus of possible outer ball joint locations. The equation of a circle in 3-space given below in Equation 1.

$$point_i = center + \vec{a} * \cos\theta_i + \vec{b} * \sin\theta_i$$

Equation 1: Circle in 3-space

Note that $point_i$ refers to some point along the circle which corresponds to a certain θ_i. A *theta* vector is created, which ranges from zero to 2π, in steps of 2π/*n_points* (note that the length of this vector is *n_points* + 1). The vectors *a* and *b* are any two vectors that are orthogonal to the *u_unit* vector and each other. These are found using the **null()** function, which computes the null space of a matrix (in this case, the argument is the *u_unit* vector). This equation is computed iteratively in a **for** loop, and the resulting 'circles' are stored as *lower_circ* and *upper_circ*. These circles will be used when finding the upper A-arm outer ball joint location.

The steering knuckle axis is represented by a unit vector (*j_unit*) which extends from the lower A-arm outer ball joint towards the upper A-arm outer ball joint. This vector is the axis about which the steering knuckle rotates. The program calculates the point along *j_unit* nearest to the tie rod outer ball joint. Note that unlike *cp_upper* or *cp_lower*, this point will change as the suspension moves. This first point, at ride height, is termed *first_cp*. The distance from *first_cp* to the tie rod outer ball joint is computed; again using the 2-norm of the vector connecting them (this distance is termed *r_circ*). This value is fixed, for all suspension positions. If the tie rod was removed, and there were no interferences, the possible locations for the tie rod outer ball joint could be described by a three dimensional circle having the following properties: origin at *first_cp*, radius *r_circ*, and normal vector *j_unit*. This circle is directly analogous to *upper_circ* and *lower_circ*, and will be used in much the same way (i.e., it will be used to find the tie rod outer ball joint location). After this, variables are initialized to speed the program as it begins looping.

Since the initial (ride height) position is known, **Kinematics.m** loops from 2 to *n_step*. It begins by rotating the lower A-arm by some amount. To do this, it first decides what the desired new position is. Assuming *direction* is 1 (jounce, or moving upwards), this will be a point which has a Y-value equal to the previous (for the iteration of the loop, 'previous' refers to the initial condition), plus half the travel divided by *n_step*. However, this poses a slight problem. The locus of points with the desired Y-value is a plane, and the intersection of this plane and *lower_circ* will be two points (mathematically, the intersection of a circle and plane can be zero, one, two, or infinitely many points – for realistic suspensions, it will be two points). In order to decide which point to use, it checks the X and Z values. Since the suspension should be moving primarily upwards, it checks to see that the X and Z values have not changed by more than threshold value (which is fairly large – half the effective length of the A-arm). The point chosen is the one on the correct half of the circle with the smallest difference between the desired Y position and the actual Y position. Note that this difference is not in fact a source of error, because the point selected really does lie on *lower_circ*.

Next, the upper A-arm outer ball joint location is found. Two facts are known about its location. First, it must lie somewhere on *upper_circ*. Secondly, there is a fixed distance between the A-arm outer ball joints (termed *dist_obj*). This fixed distance, couple with an origin at the lower A-arm outer ball joint, describes a circle. Possible solutions are those which satisfy both facts about the suspension. Mathematically, this can be zero, one, two, or infinitely many points. Again, two points will exist for practical suspension designs. These two points correspond to the two possible assembly modes of the suspension (the extraneous solution would almost certainly be impossible to physically assemble, though). In order to find these two points, there is a **for** loop which checks each possible point on *upper_circ* to see if it is a solution. To do this, the program defines an error term equal to the distance between a given point on *upper_circ* and the lower A-arm outer ball joint, minus *dist_obj* (i.e., what the distance had ought to be). Ideally, there should be two points with zero error. Note that the term is not an absolute value – in some cases, the error will be positive (the point is too far away), and in some cases the error will be negative. Since the **for** loop is stepping through values of theta, the program is checking the points in a single direction around the circle. This provides an easy method for finding the two solutions. In one case, the error will be transitioning from negative to positive. The point with the smallest, increasing error is saved as *point_inc*. The point with the smallest, decreasing (transitioning from positive to negative) error is saved as *point_dec*. To determine which solution is correct, the two points are compared to the previous location of the upper A-arm outer ball joint. Since the program is moving the suspension in small steps, whichever location is closest to the previous position is the correct one.

The program then finds the location of the tie rod outer ball joint. Although this location should generally not move very much (with regard to the steering knuckle) as the suspension increases, it usually will move slightly. This causes the wheel to turn (steer) as the suspension moves. The tie rod outer ball joint location is found in exactly the same way as the upper A-arm outer ball joint. First, the axis of the steering knuckle is found. Although this was done previously for the initial position, it must be computed upon each iteration because it changes as the A-arm outer ball joint locations change. However, the close point (the point along the steering knuckle axis closest to the tie rod outer ball joint location) cannot be directly computed in this instance, because the location of the tie rod outer ball joint is not known. Since the steering knuckle is a rigid body, the offset of the close point from the lower A-arm outer ball joint is constant, and this can be used to find the close point. Using the close point as the origin, a circle with *n_points* is generated around the steering knuckle that describes the possible locations of the tie rod outer ball joint for this particular suspension position. Then, using the same methodology as before in the case of the upper A-arm outer ball joint, the tie rod outer ball joint location is found (this time, the 'sphere' has an origin at the tie rod inner ball joint and a radius equal to the length of the tie rod).

Finally, **Kinematics.m** finds the last two points of interest: the wheel center and contact patch. Unlike the upper A-arm outer ball joint, or the tie rod outer ball joint, the wheel center and contact patch are fixed to a single rigid body. The upper A-arm outer ball joint is a member of both the upper A-arm and the steering knuckle, which prompted the use of numerical intersection solutions (the tie rod outer ball joint, of course, is a member of both the tie rod and the steering knuckle). The wheel center and contact patch can be considered members of the steering knuckle rigid body: they are always located in precisely the same position with respect to the steering knuckle. Since the steering knuckle location and orientation are known (three points are known), transformation matrices can be used to analytically find the wheel center and contact patch locations.

The first step in using transformation matrices is to establish the origin and coordinate system of the steering knuckle (the world origin and coordinate system have already been established, of course) with respect to the global coordinate system. This will change upon each iteration. First, it is computed for the initial position. The origin is *first_cp*, which is the close point (to the tie rod outer ball joint) on the steering knuckle. The Y axis is the same as the steering knuckle axis. The Z axis extends from *first_cp* to the tie rod outer ball joint, which makes it necessarily orthogonal to Y (the shortest line from any point non-coincident with a line to that line will necessarily be orthogonal). The cross product of the Y and Z axes then gives the X axis.

Now that the axes and origin of the knuckle coordinate system are known, the transformation matrix is assembled from these components. The locations of the wheel center and contact patch with respect to the steering knuckle are determined next. The product of the steering knuckle transformation matrix and the location with respect to the steering knuckle gives the location with respect to the world coordinate system. This means that the inverse of the transformation matrix of the knuckle multiplied by the location with respect to the world coordinate system gives the location with respect to the knuckle coordinate system, which is constant (rigid body).

Now that the locations of the wheel center and contact patch with respect to the knuckle coordinate system are known, the locations with respect to the world coordinate system at each suspension position can be easily calculated. During each iteration, the transformation matrix is computed based on the outer ball joint locations. It is then simply multiplied by the previously determined locations with respect to the steering knuckle coordinate system, and the result is stored.

Kinematics.m also plots the 3D suspension positions in jounce. This is only done in jounce because the program only calculates the kinematics for one direction at a time. The initial position is shown in green, and other positions are shown in blue.

Geometry.m

Geometry.m is a function that takes arguments *upper, lower, soln, wheel,* and *tierod*. These correspond to some suspension position, not necessarily the initial position (the function is called once for each suspension position). This program returns, via global variables, *camber, caster, trail, kingpin, scrub, spindle, fv_ic, sv_ic, RCH, camber_change_rate, front_dive, rear_squat,* and *rear_lift* (not all of these are used, see the **SLASIM_Call.m** section for specifics of how results are used).

After formatting the input data for easy manipulation, the program begins by computing camber angle. Camber is computed by using the angle, from the vertical, in the front view (XY plane), of the line connecting the contact patch and the wheel center. It is positive when the wheel leans outward away from the vehicle.

Next, the caster angle and trail are found. The caster angle is computed by using the angle, from the vertical, in the side view (YZ plane), of the line from the lower A-arm outer ball joint to the upper A-arm outer ball joint. This angle is positive when the lower A-arm outer ball joint is forward of the upper A-arm outer ball joint. This information is also used to find the amount of mechanical trail, by finding the point where the steer crosses the ground plane. Mechanical trail is positive if this crossing point is forward of the contact patch.

After that, the function calculates the kingpin inclination, spindle length, and scrub radius. The kingpin inclination is computed using the angle, from the vertical, in the front view (XY plane), of the line connecting the outer ball joints of the upper and lower A-arms. This angle is positive when leaning inwards toward the vehicle. Spindle length is the distance between the wheel center location and the steering axis, in the X direction. Spindle length is positive when the wheel center is further outboard than the steering axis. Scrub radius is the distance between the point where the kingpin axis (steering axis) crosses the ground plane and the contact patch location, in the X direction. Scrub radius is positive when the contact patch is further outboard than the point where the steering axis crosses the ground plane.

Next, the function finds the instant centers in the front and side views. First, the normal vectors of the A-arm planes are calculated. The cross product of these normal vectors then gives the instant axis of the steering knuckle. To find the front view instant center, it is necessary to next compute the point at which this instant axis passes through the front view plane (Z = 0). Although the instant axis is known in the sense that its orientation has been computed, no points lying on this axis are known. The front view instant center location is then given by Equation 2, Equation 3, and Equation 4. Note that this is a three plane intersection problem.

$$Front\ view\ instant\ center_X = \frac{-\left(\overrightarrow{Lower\ Normal} * \overrightarrow{Lower\ FIBJ}\right)Upper\ Normal_Y + \left(\overrightarrow{Upper\ Normal} * \overrightarrow{Upper\ FIBJ}\right)Lower\ Normal_Y}{Instant\ Axis_Z}$$

Equation 2: Front view instant center X location

$$Front\ view\ instant\ center_Y = \frac{-\left(\overrightarrow{Upper\ Normal} * \overrightarrow{Upper\ FIBJ}\right)Lower\ Normal_X + \left(\overrightarrow{Lower\ Normal} * \overrightarrow{Lower\ FIBJ}\right)Upper\ Normal_X}{Instant\ Axis_Z}$$

Equation 3: Front view instant center Y location

$$Front\ view\ instant\ center_Z = 0$$

Equation 4: Front view instant center Z location

Where:

- Upper/lower Normal is the normal vector of the respective A-arm plane
- Upper/lower FIBJ is the front inner ball joint location on the respective A-arm
- Upper/lower Normal$_X$ is the X (or Y, if labeled as such) component of the respective A-arm normal vector
- Instant Axis$_Z$ is the Z component of the steering knuckle instant axis
- Vector multiplication, denoted by an asterisk, represents the dot product

Once the front view and instant axis are known, this information can be used to easily compute the side view instant center. Since a line in 3-space is defined by parametric equations, the parameter, *t*, is first computed. Because the side view instant center lies in the plane of X = *contact_patch(1)* (the X value of the contact patch), *t* is equal to the difference between the X value of the contact patch and front view instant center X location, divided by the X component of the instant axis. Y and Z values of the side view instant center are found by beginning with the corresponding front view instant center value and adding the product of the *t* parameter and the corresponding component of the instant axis.

The roll center location can also be computed once the front view instant center is known. Since data is only entered for one suspension (e.g. left front), symmetry is assumed, and the roll center must lie at half-track. The roll center will lie, then, at the point where a line connecting the front view instant center and the contact patch crosses half-track. The slope of this line is computed, and then the roll center height is found by solving for the Y-intercept of the line (X = 0 is half-track by definition).

Next, the front view swing arm length is calculated. Although this is not reported to the user, it is computed to check that it is positive. If the front view swing arm length is negative, a warning is printed in the Command Window. If positive, no results are reported. The front view swing arm length can also be used to compute camber change rate, but this is not reported, since the camber curve gives this information already.

Finally, the function calculates suspension 'anti-' effects. It first checks that outboard brakes are present, and if they are not, it reminds the user that 'anti-' effects require wheel torque (either driving or braking). Then, the 'anti-' effects themselves are calculated. Regardless of whether the suspension is front or rear, the angle, *theta*, is first computed. *Theta* is the angle in the side view between a horizontal line at the wheel center and a line extending from the wheel center to the side view instant center. *Theta* is positive when the instant center lies above this horizontal line. If the data is for a front suspension, *theta* and the braking bias are then used to calculate front anti-dive. Since rear drive is assumed (typical of Formula SAE and Baja SAE vehicles), no front anti-lift is calculated. If the data is for a rear suspension, rear anti-lift (which depends on braking bias) and rear anti-squat (independent of braking bias) are calculated.

Test Case

In order to examine the usefulness of SLASIM, the program was used to analyze and then improve upon an actual suspension design. Data was taken a Formula SAE open-wheel race car, specifically the front suspension of the 2009-2010 Formula Buckeyes competition vehicle. Suspension coordinates were extracted from the vehicle's CAD model. Wheelbase length and CG height were estimated (only their ratio is important, and this is only used when calculating suspension 'anti-' effects).

Baseline (Initial Design)

Table 4 and Table 5, below, show the baseline values used for the test case. Again, these represent suspension coordinates (at ride height) and other vehicle information for the front suspension of a Formula SAE open-wheel race car. All coordinates are in units of inches. ‘FIBJ’ stands for ‘front inner ball joint,’ ‘RIBJ’ stands for ‘rear inner ball joint,’ ‘OBJ’ stands for ‘outer ball joint,’ and ‘IBJ’ stands for ‘inner ball joint.’ Note that from the definition of the coordinate system, WHEEL, GROUND must have values of zero for both Y and Z.

Table 4: Baseline suspension coordinates

MEMBER	POINT	X	Y	Z
UPPER	FIBJ	8.27	11.75	3.39
	RIBJ	8.27	11.75	-5.86
	OBJ	23.40	14.85	-0.92
LOWER	FIBJ	4.62	4.92	5.00
	RIBJ	4.62	4.92	-6.84
	OBJ	24.07	5.00	0.00
WHEEL	CENTER	25.00	9.75	0.00
	GROUND	25.00	0.00	0.00
TIEROD	IBJ	6.30	4.92	0.20
	OBJ	23.68	6.58	3.14

Table 5: Baseline SOLN values

SOLN					
TRAVEL	BRAKE BIAS	OUTBOARD BRAKES	FRONT SUSPENSION	WHEELBASE	CG HEIGHT
4 IN	60%	YES	YES	75 IN	12 IN

Executing the program using this input data produces the following output on the Command Window:

```
The caster angle is 5.336 degrees.

The mechanical trail is 0.467 inches.

The kingpin angle is 3.891 degrees.
The scrub is 0.590 inches.

The spindle length is 1.253 inches.

Saving to: matlab.mat
```

Running the program with the initial design data also produces Figure 4, Figure 5, Figure 6, Figure 7, Figure 8, and Figure 9, below. Note that no ‘anti-’ effects are included. This geometry is a special case: the A-arm instant axes (i.e., the lines connecting the front and rear inner ball joints) are normal to the front view. As a result, the A-arm outer ball joints do not move in the Z direction, which means the side view instant center is at infinity. This drives *theta*, the angle used when computing suspension ‘anti-’ effects, to zero, which means all ‘anti-’ effects are equal to zero.

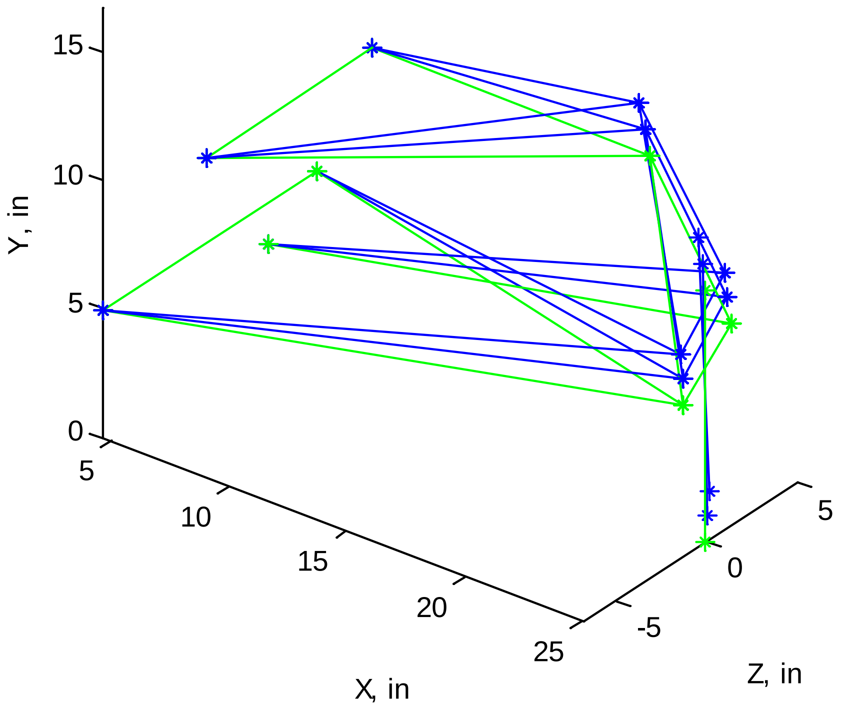

Figure 4: Suspension in jounce, baseline

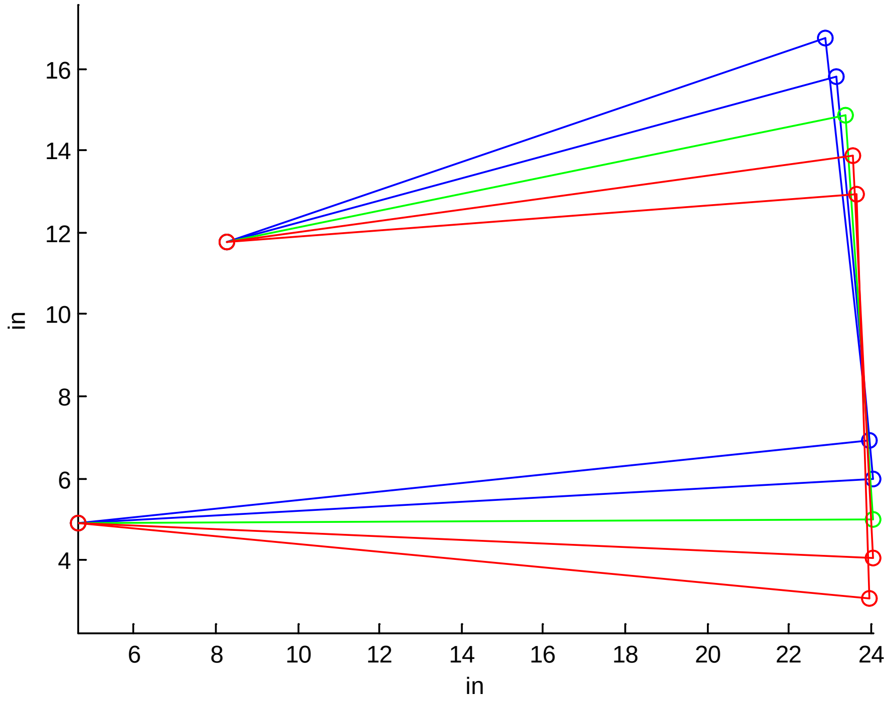

Figure 5: Suspension positions throughout travel, front view, baseline

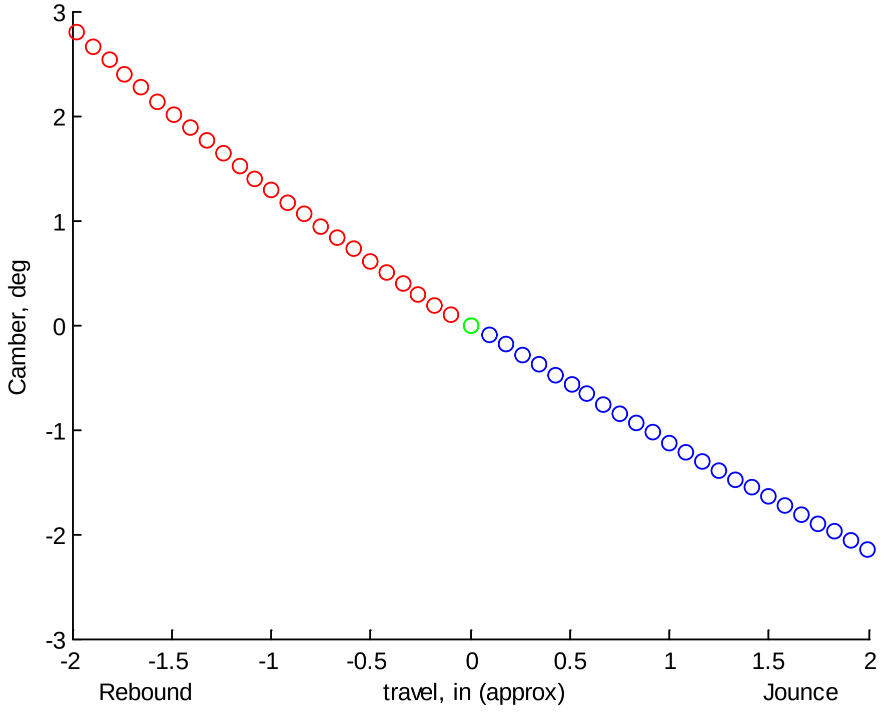

Figure 6: Camber curve, baseline

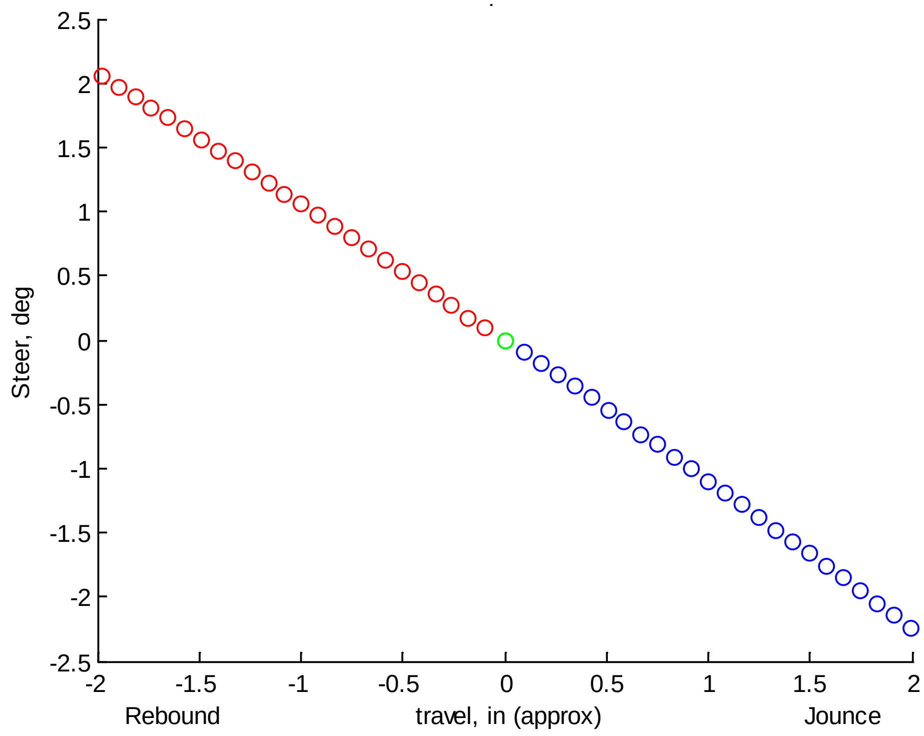

Figure 7: Bump steer, baseline

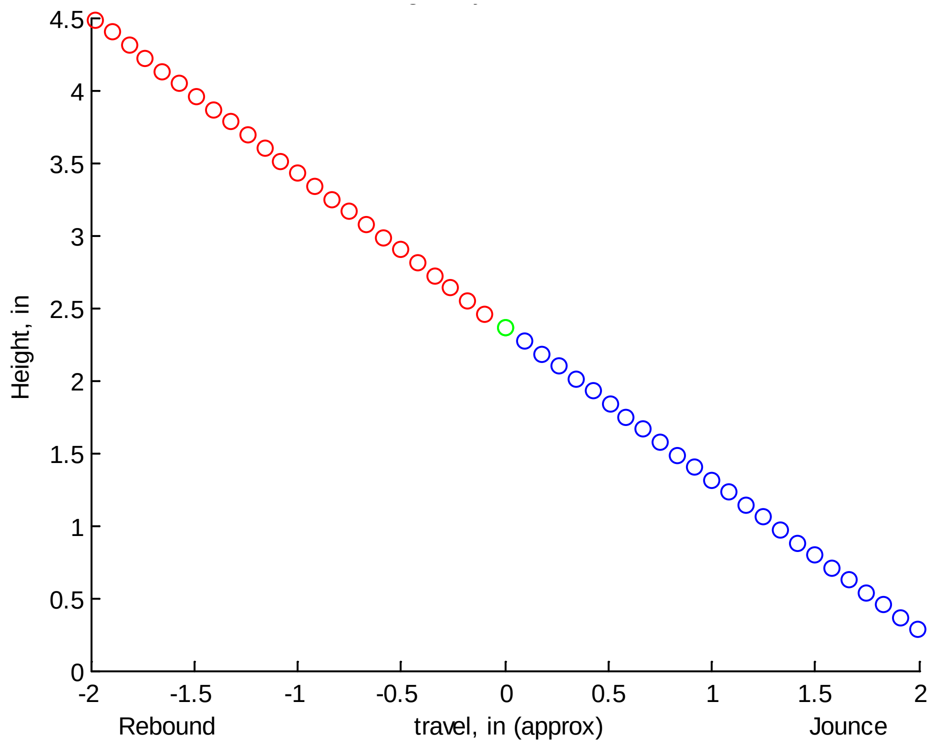

Figure 8: Roll center height, symmetric jounce/rebound, baseline

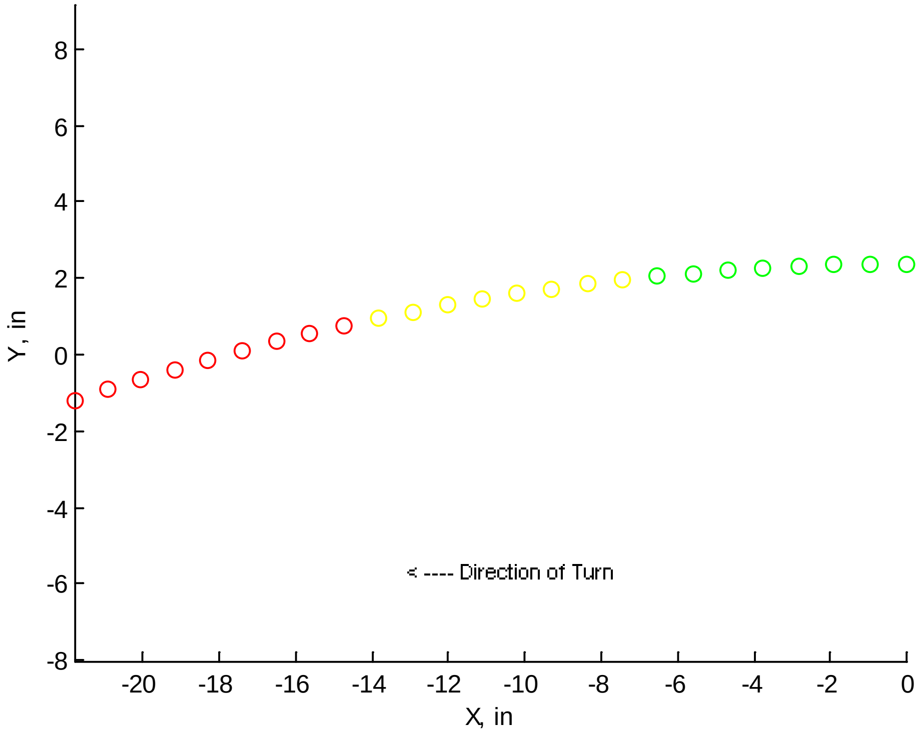

Figure 9: Roll center location, cornering, baseline

Next, these results need to be interpreted in some manner. The static parameters are appropriate (small, positive values for all). Figure 4 and Figure 5 do not say anything meaningful about the design, but do give a 'reasonable' appearance. Figure 6, the camber curve, is more interesting. It shows that the vehicle has zero static camber and a camber gain of approximately -1.2 deg/in of jounce. While the negative camber gain is certainly a beneficial trait, some initial negative camber is recommended for improved turn-in (3 p. 406). Static negative camber will cause some camber thrust, which can be counteracted by adding static toe-out.

Figure 7, the bump steer curve, shows that the vehicle will exhibit roll understeer (front outside wheel will turn to the outside of the turn). The rate of steer is approximately -1 deg/in. Introducing roll understeer can be beneficial as a means of combating lateral force compliance steer, or any undesired oversteer effects (3 p. 723). However, this value is likely too large – since this is a Formula SAE vehicle, racing tires are used. This means small steer angles result in relatively high side forces, and so it is disadvantageous for tracking to induce steer under one-wheel bump conditions (3 p. 406). Fortunately, the bump steer curve is linear (and not significantly curved).

A roll center height between the ground plane and CG height, as exhibited by this design, is typical. Although data is not available for this vehicle, it will be assumed for analysis purposes that the rear roll center height is 4 inches. For best acceleration performance out of a corner with a rear drive vehicle, it is beneficial for the front suspension to react as much of the roll couple as possible. This means the roll center on the front should be higher than that of the rear (3 p. 403). This will in turn create an anti-roll effect on the front, which will reduce the amount of suspension travel for a given amount of lateral acceleration (3 p. 403). This reduced suspension travel reduces the amount of camber change for a given lateral acceleration, which suggests that if roll center height is increased, the negative camber gain should also be increased.

Improved Design

The initial design is fairly good in that it exhibits some of the basic features desired, such as negative camber gain, linearity of camber gain and bump steer, etc. However, there are several targets for ‘improvement,’ listed below. Incorporating these changes may not actually result in improved performance, and it is very unlikely that these targets would prove to be optimal (tire data, performance targets, packaging constraints, etc., are unavailable or unknown), but they do represent a possible improvement that the program could be used to design.

- -1.5 degree static camber
- -2.0 deg/in camber gain
- Zero (or near-zero) bump steer
- Roll center height of 5 in
- 30% front anti-dive

After a few iterations based on reasonable guesses, suspension coordinates were found that satisfy (approximately) the goals of the improved design. While this design may not in exceed the performance of the initial design, the exercise shows the capability of the program to solve problems for the user. Student project teams often, even when building an entirely new car, rely on old designs as guidelines for new designs. This exercise shows the ability of SLASIM to rapidly improve upon an old design. Table 6, below, lists the coordinates for improved suspension design. Table 7 on page 31 lists the results of the improved design. Note that the upper A-arm outer ball joint has moved up two inches from its original location. Depending on steering knuckle geometry, this may not be physically possible.

Table 6: Improved suspension coordinates

MEMBER	POINT	X	Y	Z
UPPER	FIBJ	8.27	10.50	3.39
	RIBJ	8.27	9.00	-5.86
	OBJ	23.40	16.85	-0.92
LOWER	FIBJ	4.62	5.50	5.00
	RIBJ	4.62	5.50	-6.84
	OBJ	24.07	5.50	0.00
WHEEL	CENTER	25.00	9.75	0.00
	GROUND	25.25	0.00	0.00
TIEROD	IBJ	4.62	5.95	3.14
	OBJ	23.68	6.45	3.14

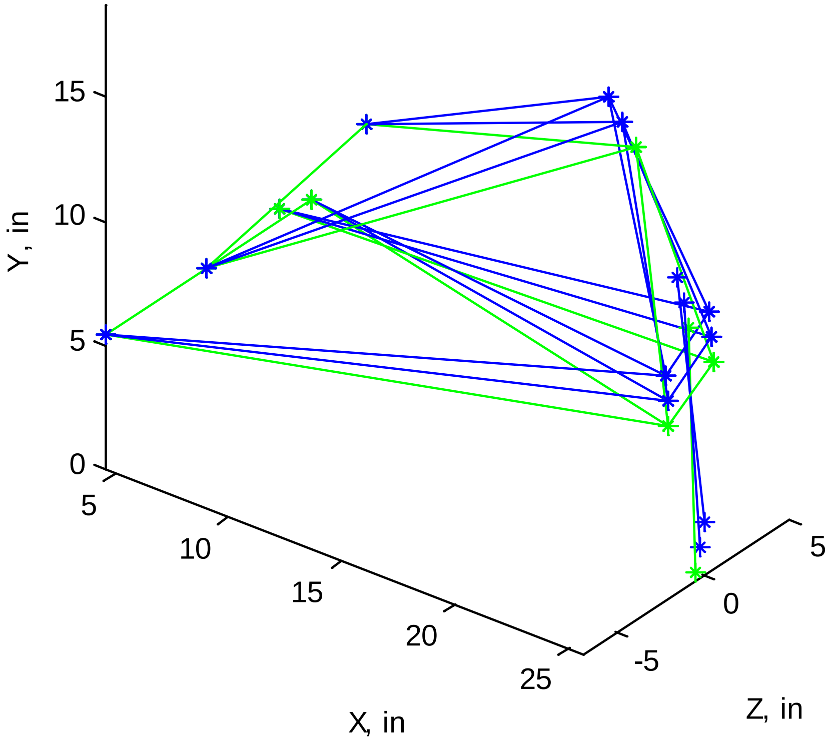

Figure 10: Suspension positions in jounce, improved

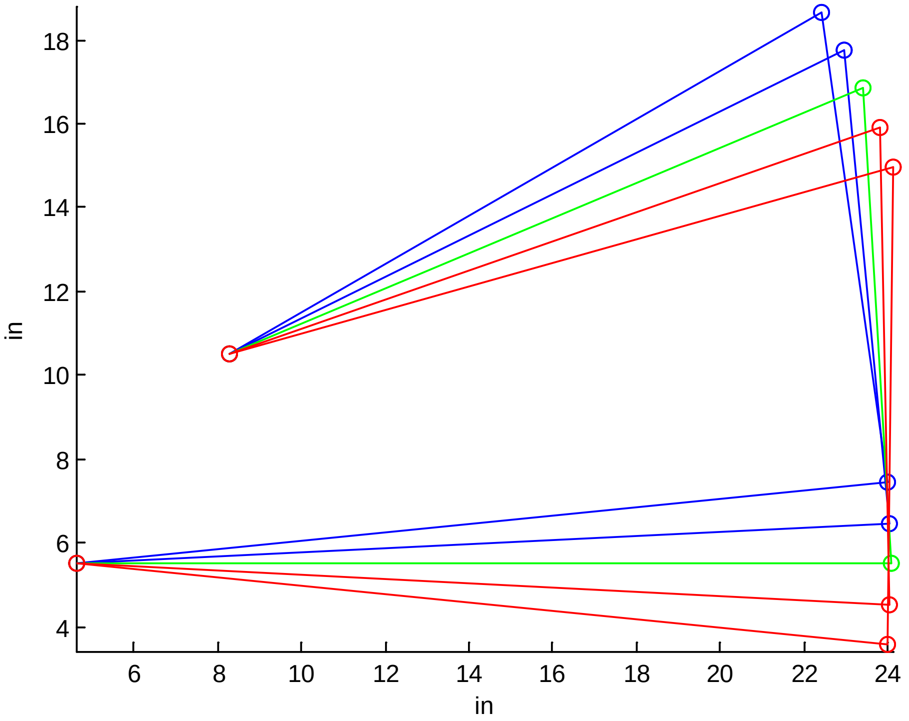

Figure 11: Suspension positions throughout travel, front view, improved

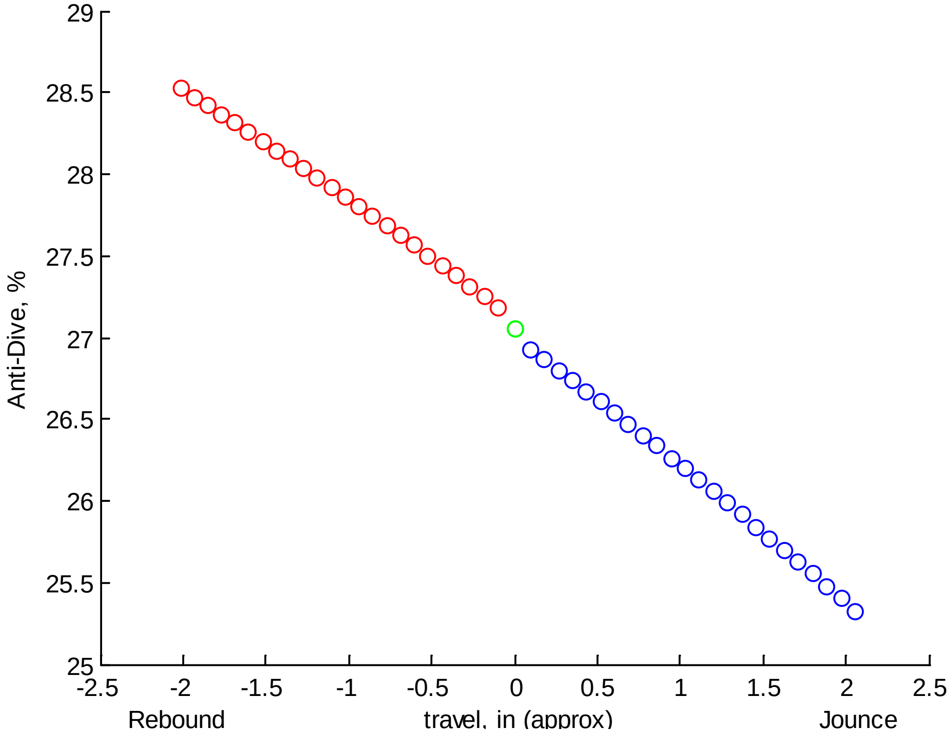

Figure 12: Front anti-dive, improved

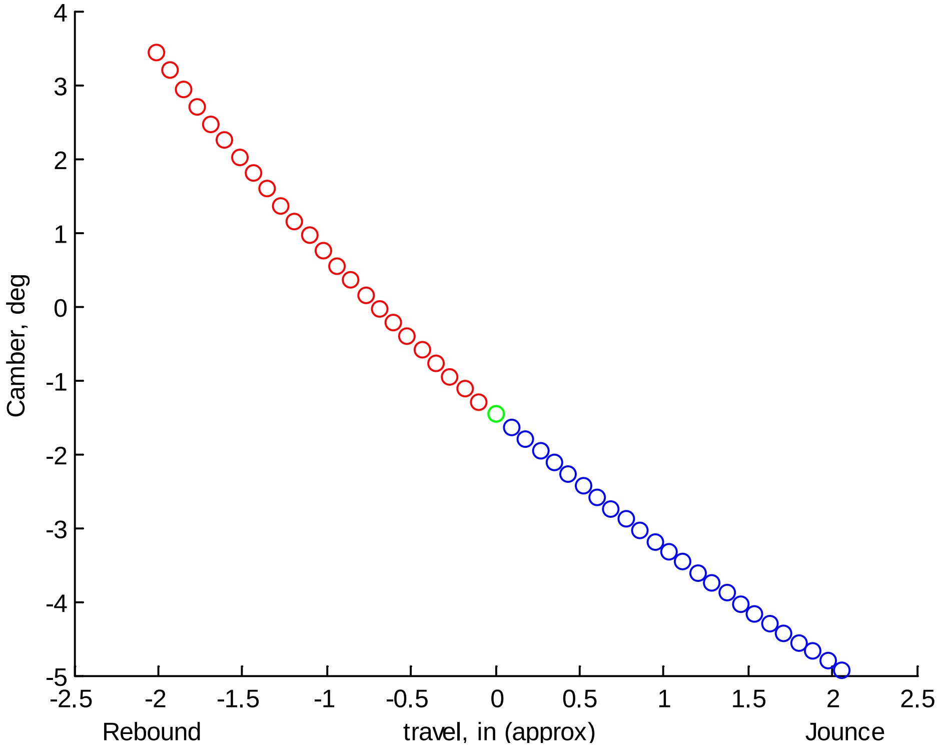

Figure 13: Camber curve, improved

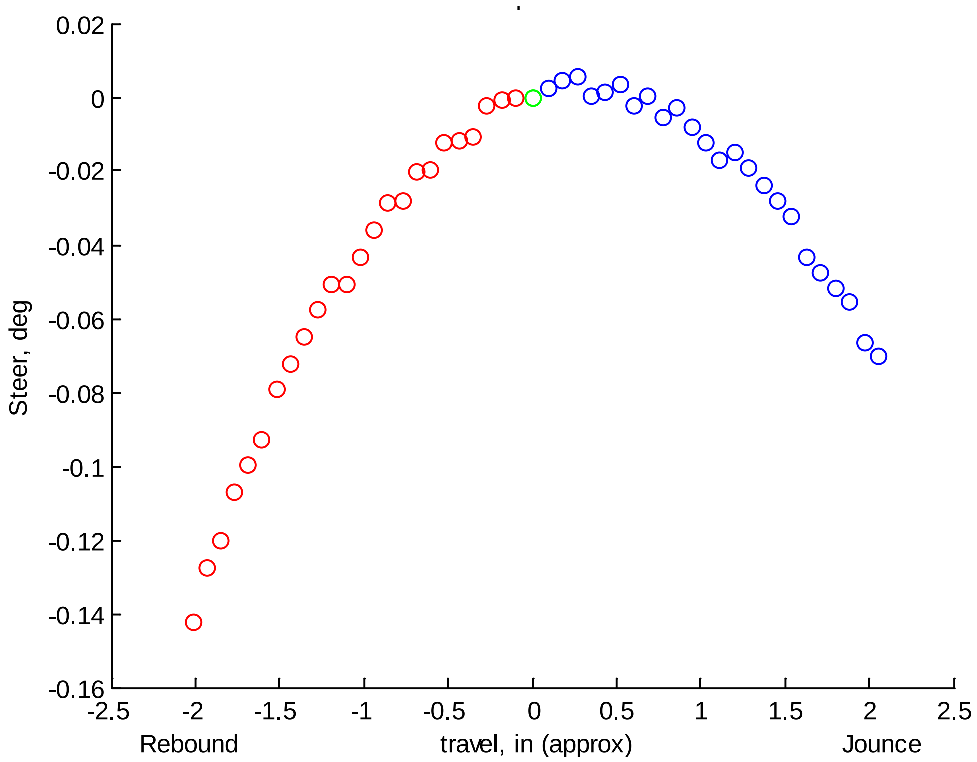

Figure 14: Bump steer, improved

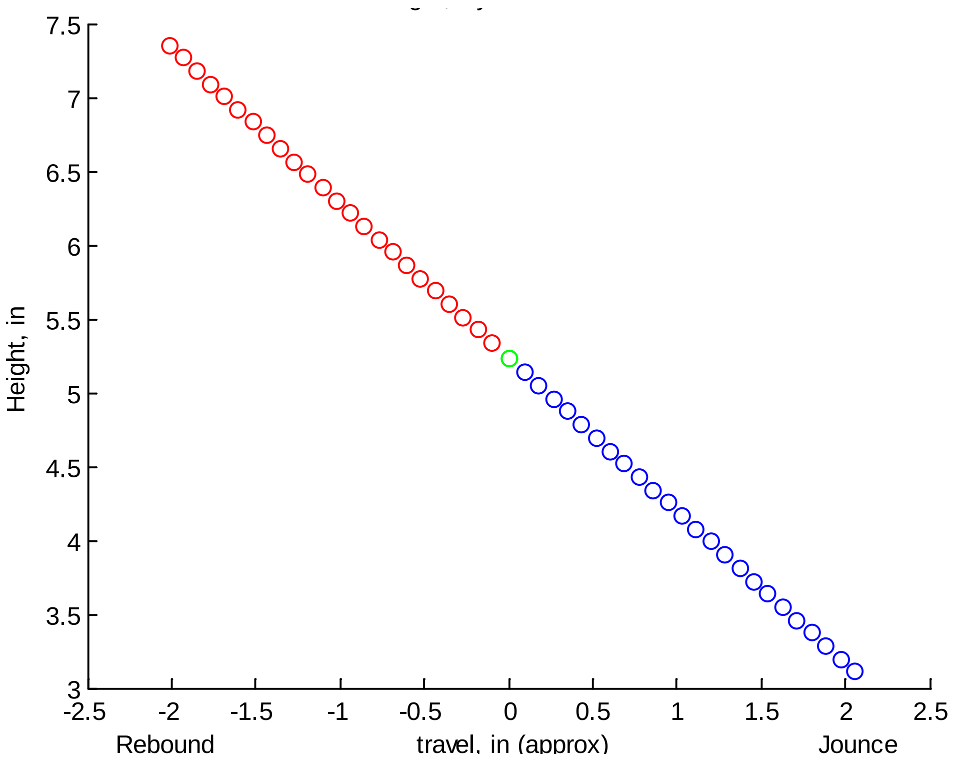

Figure 15: Roll center height, symmetrical jounce/rebound, improved

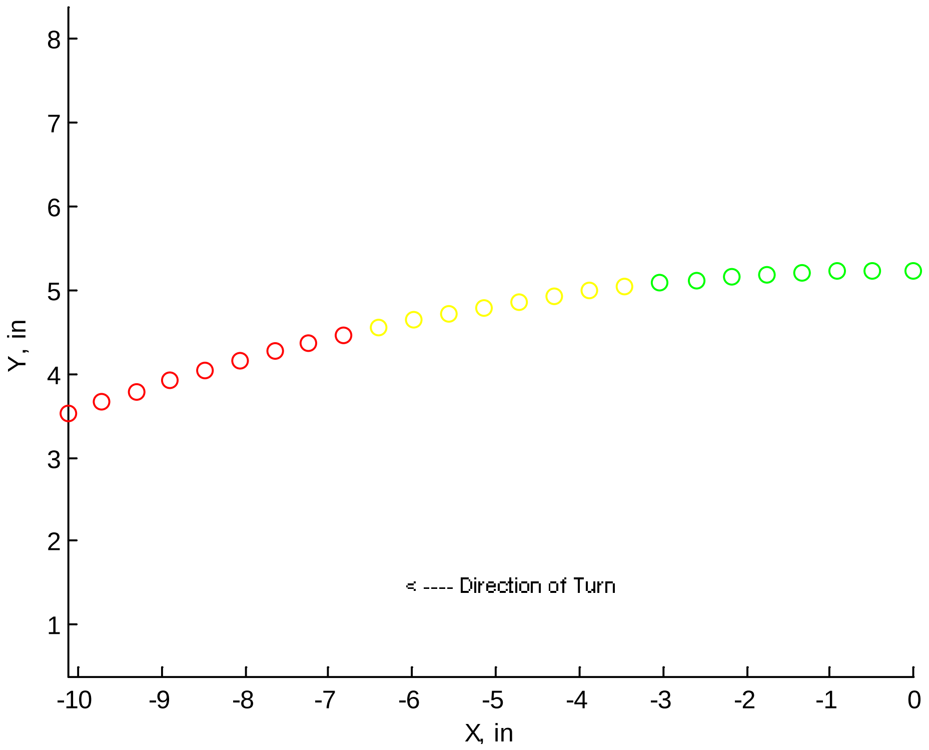

Figure 16: Roll center location, cornering, improved

Table 7: Results of suspension improvement

Parameter	Desired	Actual
Static camber	-1.5 deg	-1.47 deg
Camber gain	-2.0 deg/in	-2.05 deg/in
Bump steer	Zero	0.07 deg/in
Roll center height	5 in	5.23 in
Front anti-dive	30%	27%

Note that the coordinates in Table 4 on page 21, the initial design, are very close to those in Table 6 on page 27, the improved design. The coordinates of the A-arms only changed in the Y direction (up and down). The tie rod inner ball joint location was significantly changed to meet the bump steer target.

The static camber and camber gain values were both very close to the desired value. The character of the bump steer plot looks very poor (note: the choppy appearance is believed to be due to discretization errors in the tie rod outer ball joint position). In a turn, the outside wheel would understeer, while the inside wheel would oversteer. Both wheels would toe out, inducing undesired friction. However, the actual magnitude of the bump steer is very low. At reasonable suspension deflections, the steer angle is less than one tenth of a degree. It is unlikely that this would adversely affect vehicle performance. While the roll center height is 0.23 inches higher than desired, the difference in the magnitude of the roll couple is only about 3.3% (the CG height is 12 inches). The front anti-dive is 27% at ride height, and decreases only slightly with jounce (25% at 2 inches of jounce). To reiterate, this design may truly be an improvement, or it may not. However, it shows that SLASIM provides a user-friendly tool for suspension design (and is especially helpful for minor redesigns or optimization).

Although not addressed in this redesign, some static toe-out should be added to counteract the thrust caused by the static camber. Initial steer angle is not an input to the program, though, and the steer angles calculated (the bump steer plot) are simply deviations from the initial position.

Limitations

Error

Because the program relies heavily on numerical solutions, there are several errors present. Within the **Kinematics.m** function, the positions of both the tie rod outer ball joint and the upper A-arm outer ball joint are based on numerical solutions. The locus of possible upper A-arm outer ball joint locations is known precisely. This describes a circle, and the circle is represented by 10,000 points. This means that the spacing between points is equal to the circumference divided by the number of points. With a 20 inch long A-arm (the radius of the circle) and 10,000 points, the spacing is 0.0126 inches. The largest error will occur if the 'correct' point is in between two points, which means the largest discretization error (under these conditions) will be 0.0063 inches (approximately the thickness of two pieces of copier paper).

While this error is insignificant (surely much smaller than the error incurred while building the suspension), it does affect the error of the tie rod outer ball joint location. This location is defined by the location of three other points: the lower A-arm outer ball joint, the upper A-arm outer ball joint, and the tie rod inner ball joint. While the possible lower A-arm outer ball joint locations are also represented by discrete points, when new suspension coordinates are calculated, this value is chosen, and thus does not contribute any error. The tie rod inner ball joint location is fixed. So the error in upper A-arm outer ball joint location results in an error in the inclination of the steering axis, which is used to calculate the tie rod outer ball joint location. However, this will be a relatively small effect. If the distance between the A-arm outer ball joints is incorrect (but the inclination of the steering axis is correct), this will not result in any additional error in the tie rod outer ball joint location. This is because the tie rod outer ball joint location is calculated using a known offset from the lower A-arm outer ball joint. However, the change in inclination will induce error in the tie rod outer ball joint location. Since the upper A-arm outer ball joint is moving primarily in the Y (up and down) direction, this effect should be relatively small.

The wheel center and contact patch locations are found analytically using transformation matrices. While this solution method does not introduce error, the transformation matrix is constructed using the steering knuckle location and orientation. Since this is based on numerical solutions, error in the wheel center and contact patch locations is on the order of that present in the tie rod outer ball joint location.

Capability

SLASIM is strictly a kinematics program, and all members are considered to be rigid bodies. When designing a real suspension, it is necessary to be aware of the effect deformations can have, which can be significant (e.g., steering rack compliance).

Additionally, these kinematic linkages are treated simply as points and lines. The program does not know the size or shape of the parts themselves. As a result, it is possible for the user to design a suspension that physically will not work due to interference. The User's Guide recommends assembling the design in a CAD package to ensure it is a sound design.

Since the user only inputs one suspension (e.g. front left), the suspension on the other side of the vehicle is assumed to be the same. This does not change most results, but this assumption is used when calculating the roll center height. In fact, this is why roll center height is calculated, instead of roll center location. With a symmetric suspension, the roll center location must be at half-track.

In addition to assuming a symmetric suspension, symmetric (but opposite) suspension movements are assumed. That is, if the right suspension goes an inch into jounce, it is assumed that the left suspension goes an inch into rebound. Although dampers react differently to jounce and rebound, this should be a good assumption for steady-state cornering (when no damping force is exerted on the suspension), since coil springs are typically very linear.

SLASIM assumes that the data taken in is reasonable. For instance, if the user specified ten feet of suspension travel, the program would attempt to find the corresponding solution. There is no algorithm in place to deal with unexpected or unrealistic input, and no error messages are displayed to alert the user if problems arise with finding a solution. However, since the suspension positions are plotted, the user should be able to see problems using common sense and good judgment.

Conclusion

SLASIM was designed to be a simple, user-friendly suspension analysis program. Its goal is to bridge the gap between graphical analysis techniques and multibody kinematics packages, allowing a second or third year undergraduate engineering student to easily and intelligently design an SLA suspension for a high-performance race vehicle.

The program will be disseminated using Formula SAE and Baja SAE message boards. Hopefully, the program will prove to be beneficial to students when designing or re-designing their suspensions. Although SLASIM calculates the kinematics and a variety of features, it makes no judgment about the fitness of the design for any particular application. Accordingly, the experience gained by students using the program is no less valuable – they are still tasked with the most important part of design, exercising engineering judgment.

Bibliography

1. **Goodyear.** Goodyear Careers || Co-op and Internship Programs. *Goodyear Corporate Website.* [Online] [Cited: April 15, 2010.] http://www.goodyear.com/careers/campus/coops.html.

2. **Free Software Foundation.** The GNU General Public License. *GNU Website.* [Online] June 29, 2007. [Cited: May 1, 2010.] http://www.gnu.org/licenses/gpl.html.

3. **Milliken, William F. Milliken & Douglas L.** *Race Car Vehicle Dynamics.* Warrendale, PA : SAE, Inc., 1995.

4. **Guenther, Denny and Chrstos, Jeff.** *Initial Project Overview.* October 13, 2009.

5. **Timmins, Steven.** Race Car Vehicle Dynamics Course Website. *Univeristy of Delware Website.* [Online] Spring 2004. [Cited: April 7, 2010.] http://www.me.udel.edu/meeg467/Vehicles/index.html.

Appendix A

SLASIM MATLAB Code

Table of Contents

SLASIM.m

```
function varargout = SLASIM(varargin)
% SLASIM is a simple suspension analysis program which calculates
% suspension parameters for an SLA (short long arm) suspension.
%
% To run the program, simply type 'SLASIM' (no quotes) into the MATLAB
% Command Window (the program and associated files must be in the Current
% Directory). This launches the SLASIM GUI. Input the requested data
% into the GUI and press "RUN."
%
% It is highly recommended to read the user's guide which accompanies this
% program.
%

% Edit the above text to modify the response to help SLASIM

% Last Modified by GUIDE v2.5 17-Dec-2009 18:15:50

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;

gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @SLASIM_OpeningFcn, ...
                   'gui_OutputFcn',  @SLASIM_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT




% --- Executes just before SLASIM is made visible.
function SLASIM_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to SLASIM (see VARARGIN)

% Choose default command line output for SLASIM
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);
```

```
% UIWAIT makes SLASIM wait for user response (see UIRESUME)
% uiwait(handles.figure1);


%Create zero-valued matrices to store input data
global UPPER LOWER SOLN WHEEL TIEROD FLAG
UPPER = zeros(3);
LOWER = zeros(3);
SOLN = zeros(1,7);
WHEEL = zeros(2,3);
TIEROD = zeros(2,3);
FLAG = 0; %Missing data flag




% --- Outputs from this function are returned to the command line.
function varargout = SLASIM_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;




function outboard_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
SOLN(4) = get(hObject,'Value');
guidata(hObject, handles);




function upper_front_x_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    UPPER(1,1) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    UPPER(1,1) = temp;
end
guidata(hObject, handles);


function upper_front_x_CreateFcn(hObject, eventdata, handles)
```

```
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function upper_front_z_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    UPPER(1,3) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    UPPER(1,3) = temp;
end
guidata(hObject, handles);


function upper_front_z_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function upper_front_y_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    UPPER(1,2) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    UPPER(1,2) = temp;
end
guidata(hObject, handles);


function upper_front_y_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function lower_front_x_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG
```

```
%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    LOWER(1,1) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    LOWER(1,1) = temp;
end
guidata(hObject, handles);


function lower_front_x_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function lower_front_z_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    LOWER(1,3) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    LOWER(1,3) = temp;
end
guidata(hObject, handles);


function lower_front_z_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function lower_front_y_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    LOWER(1,2) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    LOWER(1,2) = temp;
end
guidata(hObject, handles);
```

```
function lower_front_y_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function lower_rear_x_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    LOWER(2,1) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    LOWER(2,1) = temp;
end
guidata(hObject, handles);


function lower_rear_x_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function lower_rear_z_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    LOWER(2,3) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    LOWER(2,3) = temp;
end
guidata(hObject, handles);


function lower_rear_z_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function lower_rear_y_Callback(hObject, eventdata, handles)
```

```
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    LOWER(2,2) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    LOWER(2,2) = temp;
end
guidata(hObject, handles);


function lower_rear_y_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function lower_outer_x_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    LOWER(3,1) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    LOWER(3,1) = temp;
end
guidata(hObject, handles);


function lower_outer_x_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function lower_outer_z_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    LOWER(3,3) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    LOWER(3,3) = temp;
end
```

```
guidata(hObject, handles);


function lower_outer_z_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function lower_outer_y_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    LOWER(3,2) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    LOWER(3,2) = temp;
end
guidata(hObject, handles);


function lower_outer_y_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function upper_rear_x_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    UPPER(2,1) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    UPPER(2,1) = temp;
end
guidata(hObject, handles);


function upper_rear_x_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

```
function upper_rear_z_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    UPPER(2,3) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    UPPER(2,3) = temp;
end
guidata(hObject, handles);


function upper_rear_z_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function upper_rear_y_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    UPPER(2,2) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    UPPER(2,2) = temp;
end
guidata(hObject, handles);


function upper_rear_y_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function upper_outer_x_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    UPPER(3,1) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
```

```
    UPPER(3,1) = temp;
end
guidata(hObject, handles);


function upper_outer_x_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function upper_outer_z_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    UPPER(3,3) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    UPPER(3,3) = temp;
end
guidata(hObject, handles);


function upper_outer_z_CreateFcn(hObject, eventdata, handles)
% hObject    handle to upper_outer_z (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function upper_outer_y_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    UPPER(3,2) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    UPPER(3,2) = temp;
end
guidata(hObject, handles);
```

```
function upper_outer_y_CreateFcn(hObject, eventdata, handles)
% hObject    handle to upper_outer_y (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function travel_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    SOLN(1) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    SOLN(1) = temp;
end
guidata(hObject, handles);


function travel_CreateFcn(hObject, eventdata, handles)
% hObject    handle to travel (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function steer_angle_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    SOLN(2) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    SOLN(2) = temp;
end
```

```
guidata(hObject, handles);


function steer_angle_CreateFcn(hObject, eventdata, handles)
% hObject    handle to steer_angle (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function brake_bias_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    SOLN(3) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    SOLN(3) = temp;
end
guidata(hObject, handles);


function brake_bias_CreateFcn(hObject, eventdata, handles)
% hObject    handle to brake_bias (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function wheel_center_x_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    WHEEL(1,1) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
```

```
    WHEEL(1,1) = temp;
end
guidata(hObject, handles);


function wheel_center_x_CreateFcn(hObject, eventdata, handles)
% hObject    handle to wheel_center_x (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function wheel_center_z_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    WHEEL(1,3) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    WHEEL(1,3) = temp;
end
guidata(hObject, handles);


function wheel_center_z_CreateFcn(hObject, eventdata, handles)
% hObject    handle to wheel_center_z (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function wheel_center_y_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    WHEEL(1,2) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    WHEEL(1,2) = temp;
end
guidata(hObject, handles);


function wheel_center_y_CreateFcn(hObject, eventdata, handles)
% hObject    handle to wheel_center_y (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function wheel_contact_x_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    WHEEL(2,1) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    WHEEL(2,1) = temp;
end
guidata(hObject, handles);


function wheel_contact_x_CreateFcn(hObject, eventdata, handles)
% hObject    handle to wheel_contact_x (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function tierod_inner_x_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
```

```
if (isempty(temp))
    TIEROD(1,1) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    TIEROD(1,1) = temp;
end
guidata(hObject, handles);


function tierod_inner_x_CreateFcn(hObject, eventdata, handles)
% hObject    handle to tierod_inner_x (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end




function tierod_inner_z_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    TIEROD(1,3) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    TIEROD(1,3) = temp;
end
guidata(hObject, handles);


function tierod_inner_z_CreateFcn(hObject, eventdata, handles)
% hObject    handle to tierod_inner_z (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end




function tierod_inner_y_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG
```

```
%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    TIEROD(1,2) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    TIEROD(1,2) = temp;
end
guidata(hObject, handles);


function tierod_inner_y_CreateFcn(hObject, eventdata, handles)
% hObject    handle to tierod_inner_y (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function tierod_outer_x_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    TIEROD(2,1) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    TIEROD(2,1) = temp;
end
guidata(hObject, handles);


function tierod_outer_x_CreateFcn(hObject, eventdata, handles)
% hObject    handle to tierod_outer_x (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function tierod_outer_z_Callback(hObject, eventdata, handles)
```

```
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    TIEROD(2,3) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    TIEROD(2,3) = temp;
end
guidata(hObject, handles);


function tierod_outer_z_CreateFcn(hObject, eventdata, handles)
% hObject    handle to tierod_outer_z (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function tierod_outer_y_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    TIEROD(2,2) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    TIEROD(2,2) = temp;
end
guidata(hObject, handles);


function tierod_outer_y_CreateFcn(hObject, eventdata, handles)
% hObject    handle to tierod_outer_y (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

```
% --- Executes on button press in runbutton.
function runbutton_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG


SLASIM_Call

%Inform user program is running
%set(handles.runbutton,'String','RUNNING...');
guidata(hObject, handles);




function wheelbase_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    SOLN(6) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    SOLN(6) = temp;
end
guidata(hObject, handles);




function wheelbase_CreateFcn(hObject, eventdata, handles)
% hObject    handle to wheelbase (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end




function cgheight_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
temp = str2double(get(hObject,'String')); %Cannot assign blanks to matrix
if (isempty(temp))
    SOLN(7) = 0;
    FLAG = 1; %If empty, set missing data flag to 1
else
    SOLN(7) = temp;
end
guidata(hObject, handles);
```

```
function cgheight_CreateFcn(hObject, eventdata, handles)
% hObject    handle to cgheight (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function front_Callback(hObject, eventdata, handles)
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%Storing variable in proper place
SOLN(5) = get(hObject,'Value');
guidata(hObject, handles);
```

SLASIM_Call.m

```
%SLASIM_Call -- this is the .m-file called when the user presses the run
%button. It coordinates the activities of the other subroutines.

%Global input variables -- taken from the GUI
global UPPER LOWER SOLN WHEEL TIEROD FLAG

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% ONLY CHANGE CODE INSIDE THIS BLOCK, UNLESS YOU ARE SURE OF WHAT YOU ARE
% DOING. YOU CAN USE THIS SPACE TO ENTER DATA MANUALLY RATHER THAN USING
% THE GUI. TO DO THIS, MODIFY THE VALUES FOR 'UPPER,' 'LOWER,' 'SOLN,'
% 'WHEEL,' AND 'TIEROD.' AFTER YOU HAVE ENTERED YOUR DATA, PRESS THE RUN
% BUTTON. NOTE THAT THIS ELMINATES THE FUNCTIONALITY OF THE MISSING DATA
% FLAG.

% DATA FORMAT:
% UPPER = [FIBJ_X FIBJ_Y FIBJ_Z ; RIBJ_X RIBJ_Y RIBJ_Z ; OBJ_X OBJ_Y OBJ_Z]
% LOWER = [FIBJ_X FIBJ_Y FIBJ_Z ; RIBJ_X RIBJ_Y RIBJ_Z ; OBJ_X OBJ_Y OBJ_Z]
% SOLN = [TRAVEL UNUSED BRAKE_BIAS OUTBOARD FRONT WHEELBASE CG_HEIGHT]
% WHEEL = [CENTER_X CENTER_Y CENTER_Z ; GROUND_X 0 0]
% TIEROD = [IBJ_X IBJ_Y IBJ_Z ; OBJ_X OBJ_Y OBJ_Z]

%Test values -- from a Formula SAE vehicle
% Initial
% UPPER = [8.27 11.75 3.39 ; 8.27 11.75 -5.86 ; 23.4 14.85 -.92];
% LOWER = [4.62 4.92 5 ; 4.62 4.92 -6.84 ; 24.07 5 0];
% SOLN = [4 10 60 1 1 75 12];
% WHEEL = [25 9.75 0 ; 25 0 0];
% TIEROD = [6.3 4.92 .2 ; 23.68 6.58 3.14];
% Improved
% UPPER = [8.27 10.5 3.39 ; 8.27 9.0 -5.86 ; 23.4 16.85 -.92];
% LOWER = [4.62 5.5 5 ; 4.62 5.5 -6.84 ; 24.07 5.5 0];
% SOLN = [4 10 60 1 1 75 12];
% WHEEL = [25 9.75 0 ; 25.25 0 0];
% TIEROD = [4.62 5.95 3.14 ; 23.68 6.45 3.14];
% FLAG = 0;

%Solution parameters -- see User's Guide
n_step = 25;
n_points = 50000;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Global output variables -- eliminates need for function returns

%The first are from the Geometry function
global camber caster trail kingpin scrub spindle fv_ic sv_ic RCH
camber_change_rate front_dive rear_squat rear_lift

%These are from the Kinematics function
global Q_3D P_3D tierod_obj close_point wheel_center contact_patch
```

```
%Set up results matrices

%There are 13 parameters of interest (camber, RCH, etc)
%A parameter may be single valued (e.g. camber) or have three dimensions
%(e.g. fv_ic)
n_parameters = 13;
results_inc = zeros(n_parameters, n_step);
results_dec = zeros(n_parameters, n_step);

%Results matrices for Kinematics results
upper_inc = zeros(3,3,n_step);
upper_dec = zeros(3,3,n_step);
lower_inc = zeros(3,3,n_step);
lower_dec = zeros(3,3,n_step);
tierod_inc = zeros(2,3,n_step);
tierod_dec = zeros(2,3,n_step);
wheel_inc = zeros(2,3,n_step);
wheel_dec = zeros(2,3,n_step);



if FLAG == 1
    fprintf('\rWARNING: Input data missing. Unexpected output may
result.\r\r')
end



%Call to Kinematics for increasing case
direction = 1;
Kinematics(UPPER, LOWER, SOLN, WHEEL, TIEROD, n_step, n_points, direction);



%Figure is for the plot of the suspension positions in the front view
figure
hold on
for i = 1:n_step
    %Setting up input for Geometry
    upper_inc(1:2,:,i) = UPPER(1:2,:); %Fixed points
    lower_inc(1:2,:,i) = LOWER(1:2,:);
    tierod_inc(1,:,i) = TIEROD(1,:);
    upper_inc(3,:,i) = Q_3D(i,:);
    lower_inc(3,:,i) = P_3D(i,:);
    tierod_inc(2,:,i) = tierod_obj(i,:);
    wheel_inc(1,:,i) = wheel_center(i,:);
    wheel_inc(2,:,i) = contact_patch(i,:);

    if (i == n_step || i == n_step/2 || abs(i - (n_step/2 + .5)) < .0001)
        %Plotting kinematic results
        plot([UPPER(1,1),Q_3D(i,1)],[UPPER(1,2),Q_3D(i,2)],'-o');
        plot([LOWER(1,1),P_3D(i,1)],[LOWER(1,2),P_3D(i,2)],'-o');
        plot([Q_3D(i,1),P_3D(i,1)],[Q_3D(i,2),P_3D(i,2)],'-o');
    end

    Geometry(upper_inc(:,:,i), lower_inc(:,:,i), SOLN, wheel_inc(:,:,i),
tierod_inc(:,:,i));
```

```
    %Storing results
    results_inc(1,i) = camber;
    results_inc(2,i) = caster;
    results_inc(3,i) = trail;
    results_inc(4,i) = kingpin;
    results_inc(5,i) = scrub;
    results_inc(6,i) = spindle;
    results_inc(7,i) = RCH;
    results_inc(8,i) = camber_change_rate;
    results_inc(9,i) = front_dive;
    results_inc(10,i) = rear_squat;
    results_inc(11,i) = rear_lift;
    results_inc(12,i) = atand((tierod_obj(i,1) -
close_point(i,1))/(tierod_obj(i,3) - close_point(i,3)))-
atand((tierod_obj(1,1) - close_point(1,1))/(tierod_obj(1,3) -
close_point(1,3)));
    results_inc(13,i) = contact_patch(i,2);

end

%Caster
fprintf('\n\nThe caster angle is %.3f degrees.\n\n',results_inc(2,1));

%Trail
fprintf('\n\nThe mechanical trail is %.3f inches.\n\n',results_inc(3,1));

%Kingpin
fprintf('\n\nThe kingpin angle is %.3f degrees.\n\n',results_inc(4,1));

%Scrub
fprintf('\n\nThe scrub is %.3f inches.\n\n',results_inc(5,1));

%Spindle
fprintf('\n\nThe spindle length is %.3f inches.\n\n',results_inc(6,1))

%Call to Kinematics for decreasing case
direction = -1;
Kinematics(UPPER, LOWER, SOLN, WHEEL, TIEROD, n_step, n_points, direction);

for i = 1:n_step
    %Setting up input for Geometry
    upper_dec(1:2,:,i) = UPPER(1:2,:); %Fixed points
    lower_dec(1:2,:,i) = LOWER(1:2,:);
    tierod_dec(1,:,i) = TIEROD(1,:);
    upper_dec(3,:,i) = Q_3D(i,:);
    lower_dec(3,:,i) = P_3D(i,:);
    tierod_dec(2,:,i) = tierod_obj(i,:);
    wheel_dec(1,:,i) = wheel_center(i,:);
    wheel_dec(2,:,i) = contact_patch(i,:);

    if (i == 1)
        %Plotting kinematic results
        plot([UPPER(1,1),Q_3D(i,1)],[UPPER(1,2),Q_3D(i,2)],'-og');
        plot([LOWER(1,1),P_3D(i,1)],[LOWER(1,2),P_3D(i,2)],'-og');
        plot([Q_3D(i,1),P_3D(i,1)],[Q_3D(i,2),P_3D(i,2)],'-og');
    end

    if (i == n_step || i == n_step/2 || abs(i - (n_step/2 + .5)) < .0001)
        %Plotting kinematic results
        plot([UPPER(1,1),Q_3D(i,1)],[UPPER(1,2),Q_3D(i,2)],'-or');
        plot([LOWER(1,1),P_3D(i,1)],[LOWER(1,2),P_3D(i,2)],'-or');
        plot([Q_3D(i,1),P_3D(i,1)],[Q_3D(i,2),P_3D(i,2)],'-or');
    end

    Geometry(upper_dec(:,:,i), lower_dec(:,:,i), SOLN, wheel_dec(:,:,i),
tierod_dec(:,:,i));

    %Storing results
    results_dec(1,i) = camber;
    results_dec(2,i) = caster;
    results_dec(3,i) = trail;
    results_dec(4,i) = kingpin;
    results_dec(5,i) = scrub;
    results_dec(6,i) = spindle;
    results_dec(7,i) = RCH;
    results_dec(8,i) = camber_change_rate;
    results_dec(9,i) = front_dive;
    results_dec(10,i) = rear_squat;
    results_dec(11,i) = rear_lift;
    results_dec(12,i) = atand((tierod_obj(i,1) -
close_point(i,1))/(tierod_obj(i,3) - close_point(i,3)))-
atand((tierod_obj(1,1) - close_point(1,1))/(tierod_obj(1,3) -
close_point(1,3)));
    results_dec(13,i) = contact_patch(i,2);

end
title('Suspension Positions Throughout Travel, Front View')
xlabel('in');ylabel('in');
axis equal;
hold off

%Check to see if suspension "anti-" features exist. If they don't exist,
%they'll either be zero or NaN in the workspace. If they do exist, plot
%them.
front_dive_flag = 0;
rear_squat_flag = 0;
rear_lift_flag = 0;

%Front anti-dive
max_front_dive = max([max(results_inc(9,:)) max(results_dec(9,:))]);
if max_front_dive == 0 || ~(max_front_dive <= Inf)
    front_dive_flag = 1;
end

if front_dive_flag == 0
    figure
    hold on
    for i = 1:n_step
        if i == 1
```

```
            plot((results_inc(13,i)-results_inc(13,1)),results_inc(9,i),'go')
            plot((results_dec(13,i)-results_dec(13,1)),results_dec(9,i),'go')
        else
            plot((results_inc(13,i)-results_inc(13,1)),results_inc(9,i),'o')
            plot((results_dec(13,i)-results_dec(13,1)),results_dec(9,i),'ro')
        end
    end
    title('Front Anti-Dive');
    xlabel('Rebound                          travel, in (approx)
Jounce');
    ylabel('Anti-Dive, %');
    hold off
end


%Rear anti-squat
max_rear_squat = max([max(results_inc(10,:)) max(results_dec(10,:))]);
if max_rear_squat == 0 || ~(max_rear_squat <= Inf)
    rear_squat_flag = 1;
end

if rear_squat_flag == 0
    figure
    hold on
    for i = 1:n_step
        if i == 1
            plot((results_inc(13,i)-
results_inc(13,1)),results_inc(10,i),'go')
            plot((results_dec(13,i)-
results_dec(13,1)),results_dec(10,i),'go')
        else
            plot((results_inc(13,i)-results_inc(13,1)),results_inc(10,i),'o')
            plot((results_dec(13,i)-
results_dec(13,1)),results_dec(10,i),'ro')
        end
    end
    title('Rear Anti-Squat');
    xlabel('Rebound                          travel, in (approx)
Jounce');
    ylabel('Anti-Squat, %');
    hold off
end


%Rear anti-lift
max_rear_lift = max([max(results_inc(11,:)) max(results_dec(11,:))]);
if max_rear_lift == 0 || ~(max_rear_lift <= Inf)
    rear_lift_flag = 1;
end

if rear_lift_flag == 0
    figure
    hold on
    for i = 1:n_step
        if i == 1
```

```
            plot((results_inc(13,i)-
results_inc(13,1)),results_inc(11,i),'go')
            plot((results_dec(13,i)-
results_dec(13,1)),results_dec(11,i),'go')
        else
            plot((results_inc(13,i)-results_inc(13,1)),results_inc(11,i),'o')
            plot((results_dec(13,i)-
results_dec(13,1)),results_dec(11,i),'ro')
        end
    end
    title('Rear Anti-Lift');
    xlabel('Rebound                          travel, in (approx)
Jounce');
    ylabel('Anti-Lift, %');
    hold off
end

%Camber curve
figure
hold on
for i = 1:n_step
    if i == 1
        plot((results_inc(13,i)-results_inc(13,1)),results_inc(1,i),'go')
        plot((results_dec(13,i)-results_dec(13,1)),results_dec(1,i),'go')
    else
        plot((results_inc(13,i)-results_inc(13,1)),results_inc(1,i),'o')
        plot((results_dec(13,i)-results_dec(13,1)),results_dec(1,i),'ro')
    end
end
title('Camber Curve');
xlabel('Rebound                          travel, in (approx)
Jounce');
ylabel('Camber, deg');
hold off

%Bump steer
figure
hold on
for i = 1:n_step
    if i == 1
        plot((results_inc(13,i)-results_inc(13,1)),results_inc(12,i),'go')
        plot((results_dec(13,i)-results_dec(13,1)),results_dec(12,i),'go')
    else
        plot((results_inc(13,i)-results_inc(13,1)),results_inc(12,i),'o')
        plot((results_dec(13,i)-results_dec(13,1)),results_dec(12,i),'ro')
    end
end
title('Bump Steer');
xlabel('Rebound                          travel, in (approx)
Jounce');
ylabel('Steer, deg');
hold off

%Roll center height
figure
hold on
```

```
for i = 1:n_step
    if i == 1
        plot((results_inc(13,i)-results_inc(13,1)),results_inc(7,i),'go')
        plot((results_dec(13,i)-results_dec(13,1)),results_dec(7,i),'go')
    else
        plot((results_inc(13,i)-results_inc(13,1)),results_inc(7,i),'o')
        plot((results_dec(13,i)-results_dec(13,1)),results_dec(7,i),'ro')
    end
end
title('Roll Center Height, Symmetric Jounce/Rebound');
xlabel('Rebound                          travel, in (approx)
Jounce');
ylabel('Height, in');
hold off

%Find roll center locations for the case of cornering. This is done by
%assuming that the jounce in one side is equal to the rebound in the other
%side. p1 is the contact patch and p2 is the roll center height when
%assuming symmetric suspension movements. These points define two separate
%lines for the jounce and droop cases. Their intersection point is the roll
%center when cornering (given the aforementioned assumption).
p2_inc = zeros(1,2);
p2_dec = zeros(1,2);
a = zeros(2);
b = zeros(2,1);
figure
hold on
for i = 1:n_step
    p2_inc(2)= results_inc(7,i);
    p2_dec(2)= results_dec(7,i);
    p1_inc = wheel_inc(2,1:2,i);
    p1_dec = wheel_dec(2,1:2,i);
    p1_dec(1) = -p1_dec(1);

    m_inc = (p2_inc(2)-p1_inc(2))/(p2_inc(1)-p1_inc(1));
    m_dec = (p2_dec(2)-p1_dec(2))/(p2_dec(1)-p1_dec(1));

    a(1,1) = m_inc;
    a(2,1) = m_dec;
    a(1,2) = -1;
    a(2,2) = -1;

    b(1) = -p2_inc(2);
    b(2) = -p2_dec(2);

    roll_center = a\b;

    if i < n_step/3
        plot(roll_center(1),roll_center(2),'og');
    elseif i < 2*n_step/3
        plot(roll_center(1),roll_center(2),'oy');
    else
        plot(roll_center(1),roll_center(2),'or');
    end
end
title('Roll Center Location, Cornering');
```

```
xlabel('X, in');
ylabel('Y, in');
axis equal
textbox = uicontrol('style','text');
set(textbox, 'String', '< ---- Direction of Turn');
textbox_pos = get(textbox,'Position');
textbox_pos(1) = textbox_pos(1) + 150;
textbox_pos(2) = textbox_pos(2) + 50;
textbox_pos(3) = textbox_pos(3) + 200;
textbox_pos(4) = textbox_pos(4) + 10;
set(textbox,'Position',textbox_pos)
set(textbox,'BackgroundColor',[1 1 1])

hold off

save;
```

Kinematics.m

```
function [] = Kinematics(UPPER, LOWER, SOLN, WHEEL, TIEROD, n_step, n_points,
direction)
%This M-file is responsible for finding the positions of suspension points
%as the suspension travels (jounce and rebound).
%The variables in capital letters are the suspension hard points. The other
%variables are solution parameters.

%For reference, please see Kinematics, Dynamics, and Design of Machinery by
%K. Waldron and G. Kinzel

%Global variables used in lieu of function return
global Q_3D P_3D tierod_obj close_point wheel_center contact_patch

%Take data from matrices and place into vectors containing single 3D
%points.
upper_fibj = UPPER(1,:);
upper_ribj = UPPER(2,:);
upper_obj = UPPER(3,:);

lower_fibj = LOWER(1,:);
lower_ribj = LOWER(2,:);
lower_obj = LOWER(3,:);

wheel_center_init = WHEEL(1,:);
contact_patch_init = WHEEL(2,:);

tierod_inner = TIEROD(1,:);
tierod_outer = TIEROD(2,:);

travel = SOLN(1);
asteer = SOLN(2);
bias = SOLN(3);
outboard = SOLN(4);
front_susp = SOLN(5);
wheelbase = SOLN(6);
cg_height = SOLN(7);

%Find the points on the A-arm instant axes closest to the outer ball joints
%and the vector connecting them to the
%Upper
g = upper_fibj - upper_ribj;
g_unit = g/(sqrt(sum(g.^2)));
v = upper_obj - upper_ribj;
cp_upper_offset = dot(g_unit,v);
cp_upper = upper_ribj + cp_upper_offset*g_unit;
upper_vector = upper_obj - cp_upper;
upper_length = norm(upper_vector, 2);
%Lower
u = lower_fibj - lower_ribj;
u_unit = u/(sqrt(sum(u.^2)));
v = lower_obj - lower_ribj;
cp_lower_offset = dot(u_unit,v);
cp_lower = lower_ribj + cp_lower_offset*u_unit;
```

```
lower_vector = lower_obj - cp_lower;
lower_length = norm(lower_vector, 2);



%Find the step size needed (approximately), assume n_step wanted in each
%direction
step = direction*(travel/2)/n_step;

%Find distance between OBJs
w = upper_obj - lower_obj;
dist_obj = norm(w,2);


%Numerically construct a circle around the A-arms
theta = 0:(2*pi/n_points):2*pi;
%Need two arbitrary vectors normal to the instant axis of the A-arms
temp = null(u_unit);
a = temp(:,1)';
b = temp(:,2)';
clear temp

temp = null(g_unit);
c = temp(:,1)';
d = temp(:,2)';
clear temp

lower_circ = zeros(n_points + 1, 3);
upper_circ = zeros(n_points + 1, 3);

% hold on
% plot3(cp_upper(1),cp_upper(2),cp_upper(3),'g*')
% plot3(upper_obj(1),upper_obj(2),upper_obj(3),'g*')
% plot3(cp_lower(1),cp_lower(2),cp_lower(3),'r*')
% plot3(lower_obj(1),lower_obj(2),lower_obj(3),'r*')
for k = 1:(n_points + 1)
    lower_circ(k,:) = cp_lower + a*cos(theta(k))*lower_length +
b*sin(theta(k))*lower_length;
    upper_circ(k,:) = cp_upper + c*cos(theta(k))*upper_length +
d*sin(theta(k))*upper_length;
%     plot3(lower_circ(k,1),lower_circ(k,2),lower_circ(k,3),'r')
%     plot3(upper_circ(k,1),upper_circ(k,2),upper_circ(k,3),'g')
end
% axis equal

%Upper OBJ
Q_3D = zeros(n_step,3);
Q_3D(1,:) = upper_obj;

%Lower OBJ
P_3D = zeros(n_step,3);
P_3D(1,:) = lower_obj;
desired_pos = zeros(1,3);
```

```
%Finding where along the steering knuckle normal vector is closest to the
%tierod outer ball joint
close_point = zeros(n_step,3);
j = upper_obj - lower_obj;
j_unit = j./norm(j,2);
h = tierod_outer-lower_obj;
close_point_offset = dot(j_unit,h);
first_cp = P_3D(1,:) + close_point_offset*j_unit;
close_point(1,:) = first_cp;
r_circ = norm((tierod_outer - first_cp),2);
ltierod = norm((tierod_outer-tierod_inner),2);



%Set up variables
best_err = 100;
threshold1 = lower_length/2;
threshold2 = step;
err2 = zeros(1,(n_points + 1));
min_inc = 100;
min_dec = 100;
min_index_inc = -1;
min_index_dec = -1;
circle = zeros(n_points + 1, 3);
dist_err = zeros(n_points + 1, 1);
tierod_obj = zeros(n_step,3);
tierod_obj(1,:) = tierod_outer;
wheel_center = zeros(n_step,3);
contact_patch = zeros(n_step,3);

for i = 2:n_step
    %Calculate desired position (Y value is the important one)
    desired_pos(1) = P_3D((i-1),1);
    desired_pos(3) = P_3D((i-1),3);
    desired_pos(2) = P_3D((i-1),2) + step;

    %Find point on lower_circ closest to desired position
    for k = 1:(n_points + 1)
        err_y = abs(desired_pos(2) - lower_circ(k,2));
        err_x = abs(desired_pos(1) - lower_circ(k,1));
        err_z = abs(desired_pos(3) - lower_circ(k,3));
        check = direction*(lower_circ(k,2) - P_3D(i-1,2));

        %Look for lowest Y-error. Check X and Z to determine point is on
        %correct half of circle. Threshold2 checks that the point moved the
        %proper direction, and that it moved far enough.
        if err_y < best_err && err_x < threshold1 && err_z < threshold1 &&
check > threshold2
            best_err = err_y;
            index = k;
        end
    end


    %Lower OBJ location is the point with the lowest distance error
    P_3D(i,:) = lower_circ(index,:);


    %Find the intersection of the circle described by the upper A-arm OBJ
    %with the sphere defined by the distance between the lower A-arm OBJ at
    %its center and a radius equal to the length between the ball joints.
    %Since this will generally have two solutions, the index is found for
    %both cases.
    for k = 1:(n_points + 1)
        err2(k) = norm((upper_circ(k,:)-P_3D(i,:)),2) - dist_obj;
%         check = direction*(lower_circ(k,2) - Q_3D(i-1,2));
        if k ~= 1
            if abs(err2(k)) < min_inc && err2(k) > err2(k-1)
                min_inc = abs(err2(k));
                min_index_inc = k;
            end
            if abs(err2(k)) < min_dec && err2(k) < err2(k-1)
                min_dec = abs(err2(k));
                min_index_dec = k;
            end
        end
    end


    %In order to find the 'correct' intersection, the program looks to the
    %last solution found. Whichever intersection point is closer to the
    %last solution will be the correct one.
    point_inc = upper_circ(min_index_inc,:);
    point_dec = upper_circ(min_index_dec,:);
    dist_inc = norm((point_inc - Q_3D(i-1,:)),2);
    dist_dec = norm((point_dec - Q_3D(i-1,:)),2);
    if dist_inc < dist_dec
        Q_3D(i,:) = point_inc;
    else
        Q_3D(i,:) = point_dec;
    end


    min_inc = 100;
    min_dec = 100;
    min_index_inc = -1;
    min_index_dec = -1;

    %Finding tie rod outer ball joint location (i.e., coupler rotation)

    %Normalized (unit) vector of steering knuckle
    q = Q_3D(i,:) - P_3D(i,:);
    unit = q./sqrt(sum(q.^2));

    %Closest point on knuckle normal vector to the tierod obj
    close_point(i,:) = P_3D(i,:) + close_point_offset*unit;


    %Find two vectors orthogonal to 'unit' (and each other) in order to
    %construct the circle
    temp = null(unit);
    e = temp(:,1)';
    f = temp(:,2)';
    clear temp

    %Construct the circle numerically (i.e., generate n_points on the
    %circle)
    for k=1:(n_points+1)
        circle(k,:) = close_point(i,:) + e.*cos(theta(k))*r_circ +
f.*sin(theta(k))*r_circ;
        %Distance between sphere defined by tierod and circle defined above
        dist_err(k) = norm((circle(k,:) - tierod_inner),2) - ltierod;
    end

    %Finding indices of minimum dist_err. Since the intersection of a
    %sphere and circle is generally two points, there will be two minima
    for k = 2:(n_points+1)
        if abs(dist_err(k)) < min_inc && dist_err(k) > dist_err(k-1)
            min_inc = abs(dist_err(k));
            min_index_inc = k;
        end
        if abs(dist_err(k)) < min_dec && dist_err(k) < dist_err(k-1)
            min_dec = abs(dist_err(k));
            min_index_dec = k;
        end
    end

    %Determine which of the two solutions is correct by checking which
    %defines a point closer to the last value
    tobj_inc = circle(min_index_inc(1),:,1);
    tobj_dec = circle(min_index_dec(1),:,1);
    dist_inc = norm((tobj_inc - tierod_obj(i-1,:)),2);
    dist_dec = norm((tobj_dec - tierod_obj(i-1,:)),2);
    if dist_inc < dist_dec
        tierod_obj(i,:) = circle(min_index_inc,:);
    else
        tierod_obj(i,:) = circle(min_index_dec,:);
    end


    best_err = 100;
    min_inc = 100;
    min_dec = 100;
    min_index_inc = -1;
    min_index_dec = -1;



    %Next, use transformation matrices to find the location of the wheel
    %center and contact patch. Since the wheel center and contact patch are
    %fixed w.r.t. the steering knuckle, the transformation matrix along
    %with the orientation of the knuckle uniquely describe the contact
    %patch and wheel center locations.

    o = close_point(i,:);
    y = Q_3D(i,:)-o;
    y = y./sqrt(sum(y.^2));
    z = tierod_obj(i,:)-o;
    z = z./sqrt(sum(z.^2));
    x = cross(y,z);
    T_knuckle = zeros(4);
    T_knuckle(1:3,1) = x';
    T_knuckle(1:3,2) = y';
    T_knuckle(1:3,3) = z';
    T_knuckle(1:3,4) = o';
    T_knuckle(4,4) = 1;

    %The first time through, need to find the position of the wheel center
    %and contact patch w.r.t. the knuckle coordinate system. Since this
    %does not change, it only needs to be calculated once.
    if i == 2

        o_first = first_cp;
        y_first = Q_3D(1,:) - o_first;
        y_first = y_first./sqrt(sum(y_first.^2));
        z_first = tierod_obj(i,:) - o_first;
        z_first = z_first./sqrt(sum(z_first.^2));
        x_first = cross(y_first,z_first);
        T_knuckle_first = zeros(4);
        T_knuckle_first(1:3,1) = x_first';
        T_knuckle_first(1:3,2) = y_first';
        T_knuckle_first(1:3,3) = z_first';
        T_knuckle_first(1:3,4) = o_first';
        T_knuckle_first(4,4) = 1;

        wheel_center_init(4) = 1;
        wc_kcs = T_knuckle_first\wheel_center_init';
        wheel_center(1,:) = wheel_center_init(1:3);

        contact_patch_init(4) = 1;
        cp_kcs = T_knuckle_first\contact_patch_init';
        contact_patch(1,:) = contact_patch_init(1:3);

    end

    %Solve for wheel center positions
    temp = T_knuckle * wc_kcs;
    wheel_center(i,:) = temp(1:3)';
    clear temp

    %Solve for contact patch positions
    temp = T_knuckle * cp_kcs;
    contact_patch(i,:) = temp(1:3)';
    clear temp

%     if i == 2 || i == 3 || i == 4 || i == 5 || i == 6
%         fprintf('\n\n  close_point = %f, %f, %f\n  tobj_inc = %f, %f, %f\n
tobj_dec = %f, %f,
%f\n',close_point(1),close_point(2),close_point(3),tobj_inc(1),tobj_inc(2),to
bj_inc(3),tobj_dec(1),tobj_dec(2),tobj_dec(3));
%         if i == 4
%             hold on
%             plot3(circle(:,1),circle(:,2),circle(:,3),'r')
%             plot3(P_3D(i,1),P_3D(i,2),P_3D(i,3),'g*')
%             plot3(Q_3D(i,1),Q_3D(i,2),Q_3D(i,3),'g*')
%             plot3(close_point(1),close_point(2),close_point(3),'r*')
%             plot3(tierod_obj(i,1),tierod_obj(i,2),tierod_obj(i,3),'b*')
%             plot3(tierod_outer(1),tierod_outer(2),tierod_outer(3),'k*')
%             hold off
%             axis equal
%             save
%         end
end

clear x y z

%Plot kinematics
if direction == 1
    figure
    hold on
    for i = 1:n_step
        if i == 1

            x = [lower_ribj(1) lower_fibj(1)];
            z = [lower_ribj(2) lower_fibj(2)];
            y = [lower_ribj(3) lower_fibj(3)];
            plot3(x,y,z, 'g-*')

            x = [lower_ribj(1) lower_obj(1)];
            z = [lower_ribj(2) lower_obj(2)];
            y = [lower_ribj(3) lower_obj(3)];
            plot3(x,y,z, 'g-*')

            x = [lower_obj(1) lower_fibj(1)];
            z = [lower_obj(2) lower_fibj(2)];
            y = [lower_obj(3) lower_fibj(3)];
            plot3(x,y,z, 'g-*')

            x = [upper_ribj(1) upper_fibj(1)];
            z = [upper_ribj(2) upper_fibj(2)];
            y = [upper_ribj(3) upper_fibj(3)];
            plot3(x,y,z, 'g-*')

            x = [upper_ribj(1) upper_obj(1)];
            z = [upper_ribj(2) upper_obj(2)];
            y = [upper_ribj(3) upper_obj(3)];
            plot3(x,y,z, 'g-*')

            x = [upper_fibj(1) upper_obj(1)];
            z = [upper_fibj(2) upper_obj(2)];
            y = [upper_fibj(3) upper_obj(3)];
            plot3(x,y,z, 'g-*')

            x = [lower_obj(1) upper_obj(1)];
            z = [lower_obj(2) upper_obj(2)];
            y = [lower_obj(3) upper_obj(3)];
            plot3(x,y,z, 'g-*')

            x = [lower_obj(1) tierod_outer(1)];
            z = [lower_obj(2) tierod_outer(2)];
            y = [lower_obj(3) tierod_outer(3)];
            plot3(x,y,z, 'g-*')

            x = [tierod_outer(1) upper_obj(1)];
            z = [tierod_outer(2) upper_obj(2)];
            y = [tierod_outer(3) upper_obj(3)];
            plot3(x,y,z, 'g-*')

            x = [tierod_outer(1) tierod_inner(1)];
            z = [tierod_outer(2) tierod_inner(2)];
            y = [tierod_outer(3) tierod_inner(3)];
            plot3(x,y,z, 'g-*')

            x = [contact_patch(1,1) wheel_center(1,1)];
            z = [contact_patch(1,2) wheel_center(1,2)];
            y = [contact_patch(1,3) wheel_center(1,3)];
            plot3(x,y,z, 'g-*')

        end

        if (i == n_step || i == n_step/2 || abs(i - (n_step/2 + .5)) < .0001)

            x = [lower_ribj(1) P_3D(i,1)];
            z = [lower_ribj(2) P_3D(i,2)];
            y = [lower_ribj(3) P_3D(i,3)];
            plot3(x,y,z, 'b-*')

            x = [P_3D(i,1) lower_fibj(1)];
            z = [P_3D(i,2) lower_fibj(2)];
            y = [P_3D(i,3) lower_fibj(3)];
            plot3(x,y,z, 'b-*')

            x = [upper_ribj(1) Q_3D(i,1)];
            z = [upper_ribj(2) Q_3D(i,2)];
            y = [upper_ribj(3) Q_3D(i,3)];
            plot3(x,y,z, 'b-*')

            x = [upper_fibj(1) Q_3D(i,1)];
            z = [upper_fibj(2) Q_3D(i,2)];
            y = [upper_fibj(3) Q_3D(i,3)];
            plot3(x,y,z, 'b-*')

            x = [P_3D(i,1) Q_3D(i,1)];
            z = [P_3D(i,2) Q_3D(i,2)];
            y = [P_3D(i,3) Q_3D(i,3)];
            plot3(x,y,z, 'b-*')

            x = [P_3D(i,1) tierod_obj(i,1)];
            z = [P_3D(i,2) tierod_obj(i,2)];
            y = [P_3D(i,3) tierod_obj(i,3)];
```

```
            plot3(x,y,z, 'b-*')

            x = [tierod_obj(i,1) Q_3D(i,1)];
            z = [tierod_obj(i,2) Q_3D(i,2)];
            y = [tierod_obj(i,3) Q_3D(i,3)];
            plot3(x,y,z, 'b-*')

            x = [tierod_obj(i,1) tierod_inner(1)];
            z = [tierod_obj(i,2) tierod_inner(2)];
            y = [tierod_obj(i,3) tierod_inner(3)];
            plot3(x,y,z, 'b-*')

            x = [wheel_center(i,1) contact_patch(i,1)];
            z = [wheel_center(i,2) contact_patch(i,2)];
            y = [wheel_center(i,3) contact_patch(i,3)];
            plot3(x,y,z, 'b-*')

        end
    end
    xlabel('X, in')
    ylabel('Z, in')
    zlabel('Y, in')
    title('Suspension in Jounce')
    axis image
    view(37.5,30)
    hold off
end
end
```

Geometry.m

```
function [] = Geometry(upper, lower, soln, wheel, tierod)
%Geometry
%This M-file is called by the SLASIM_Call function.
%When called, it takes in data points and calculates the parameters.

%For reference, please see Chapter 17 of "Race Car Vehicle Dynamics," by
%Milliken & Milliken.

global camber caster trail kingpin scrub spindle fv_ic sv_ic RCH
camber_change_rate front_dive rear_squat rear_lift


%Take data from matrices and place into vectors containing single 3D
%points. Although unnecessary, this makes the calculations more intuitive.

upper_fibj = upper(1,:);
upper_ribj = upper(2,:);
upper_obj = upper(3,:);

lower_fibj = lower(1,:);
lower_ribj = lower(2,:);
lower_obj = lower(3,:);

wheel_center = wheel(1,:);
contact_patch = wheel(2,:);

tierod_inner = tierod(1,:);
tierod_outer = tierod(2,:);

travel = soln(1);
asteer = soln(2);
bias = soln(3);
outboard = soln(4);
front_susp = soln(5);
wheelbase = soln(6);
cg_height = soln(7);


%Find camber
camber_x = wheel_center(1) - contact_patch(1);
camber_y = wheel_center(2) - 0;
camber = atand(camber_x/camber_y);


%Find caster angle and mechanical trail
caster_y = upper_obj(2) - lower_obj(2);
caster_z = lower_obj(3) - upper_obj(3);
caster = atand(caster_z/caster_y);
m_caster = -caster_y/caster_z;
b_caster = upper_obj(2) - m_caster*upper_obj(3);
trail = -b_caster/m_caster;
```

```
%Find kingping inclination, spindle length, and scrub radius
kp_x = upper_obj(1) - lower_obj(1);
kp_y = upper_obj(2) - lower_obj(2);
kingpin = atand((-1*kp_x)/kp_y);
m_kp = kp_y/kp_x;
b_kp = upper_obj(2) - m_kp*upper_obj(1);
scrub = contact_patch(1) - (-b_kp/m_kp);
spindle = wheel_center(1) - (wheel_center(2)-b_kp)/m_kp;


%Return normal vectors of A-arm planes
upper_normal = cross((upper_fibj - upper_ribj),(upper_obj - upper_ribj));
lower_normal = cross((lower_fibj - lower_ribj),(lower_obj - lower_ribj));


%Find normal vector of instant axis
instant_axis_normal = cross(upper_normal, lower_normal);


%Find point corresponding to front view instant center (i.e., Z = 0 on the
%instant axis)
dot_upper = -dot(upper_normal,upper_fibj);
dot_lower = -dot(lower_normal,lower_fibj);
fv_ic = [0;0;0];
fv_ic(1) = (dot_lower*upper_normal(2) -
dot_upper*lower_normal(2))/instant_axis_normal(3);
fv_ic(2) = (dot_upper*lower_normal(1) -
dot_lower*upper_normal(1))/instant_axis_normal(3);
fv_ic(3) = 0;


%Use this information to find the side view instant center
t = (contact_patch(1) - fv_ic(1))/instant_axis_normal(1);
sv_ic(1) = contact_patch(1);
sv_ic(2) = fv_ic(2) + t*instant_axis_normal(2);
sv_ic(3) = fv_ic(3) + t*instant_axis_normal(3);
% fprintf('\nSV_IC = %f %f %f\n',sv_ic(1),sv_ic(2),sv_ic(3));

%We are assuming we have a symmetrical suspension. To find the roll center
%height, we need to find the intersection of a 2D line (that goes between
%the contact patch and the front view instant center) with the origin. The
%first step finds the slope of this line, and the second finds the RCH (in
%this case, the y-intercept of a 2D line in slope-intercept form).
m_rch = (fv_ic(2) - contact_patch(2))/(fv_ic(1) - contact_patch(1));
RCH = -m_rch*contact_patch(1);


%Use FVSA length to find camber change rate. Results are in degrees/inch of
%travel.
fvsa_length = contact_patch(1) - fv_ic(1);
camber_change_rate = atand(1/fvsa_length);

if fvsa_length < 0
```

```
    fprintf('\n\nWarning!: front-view swing-arm length is negative!\n\n');
end


%Next, calculate suspension anti's. Note that brake bias is defined as the
%percent front brake bias, so (100 - % brake bias) = % rear brake bias.
%Determine front/rear suspension and calculate appropriate anti's.
if soln(4) == 0
    fprintf('\n\nNote: suspension anti features require application of torque
(driving or braking)\n\n');
end



if front_susp == 1
    theta = atand((sv_ic(2)-contact_patch(2))/(-1*(sv_ic(3)-
contact_patch(3))));

    front_dive = bias*tand(theta)*wheelbase/cg_height;
    rear_squat = 0;
    rear_lift = 0;
else
    theta = atand((sv_ic(2)-contact_patch(2))/(sv_ic(3)-contact_patch(3)));
    rear_squat = 100*tand(theta)/(cg_height/wheelbase);
    rear_lift = (100-bias)*tand(theta)*wheelbase/cg_height;
    front_dive = 0;
end
```

Appendix B

SLASIM User's Guide

SLASIM

User's Guide

Sage Wolfe
The Ohio State University :: College of Mechanical Engineering
4/7/2010

Table of Contents

License

Purpose

This program is designed to take short-long arm suspension design data (hard points, ball joint locations, etc.) and return suspension parameters of interest, such as camber curves, bump steer, and so on.

These same suspension parameters can also be calculated in other ways, including graphical methods and multibody kinematics packages. However, there are problems with both. Graphical methods, while simple, are time consuming and thus discourage design iteration. Kinematics packages, while powerful, require experience and expensive licensing.

SLASIM is intended for use by undergraduate students working on student project teams (though it may be helpful to others, such as hobbyists). It allows designers to quickly (and easily) see the effects of changing their design, which aids in creating a good design. The program was designed primarily for Formula SAE and Baja SAE teams.

System Requirements

SLASIM was written using MATLAB R2009a and Windows XP Pro. It has not been tested in other versions of MATLAB or different operating systems, although it is assumed to work. Again, there is no guarantee or warranty of any kind.

Because many of the solution steps are numerical, it is suggested to run the program on a relatively fast computer if possible. For reference, the program has a run time of approximately 8 seconds on a PC which has an Intel E5345 2.33 GHz dual core and 3 GB of RAM.

Suggested Reference Texts

Race Car Vehicle Dynamics by Milliken & Milliken (ISBN 1560915269) is highly suggested as a general vehicle dynamics reference. While this program outputs suspension parameters, it does not judge whether they are acceptable or desirable. This decision still falls on the designer, and Milliken is a great help to that end. (Milliken, 1995)

Kinematics, Dynamics, and Design of Machinery by Waldron & Kinzel (ISBN 0471244171) was used in the development of this program, and may be useful if questions arise regarding kinematics. (Waldron, 2003)

User Input

In order to analyze the suspension, the user must input the position of the suspension at ride height, as well as some other information about the suspension. Figure 1, below, shows a representative (front) SLA suspension and the associated coordinate system. The origin (0, 0, 0) is at half-track (halfway between the contact patches of the left and right tires) on the ground plane. X is positive towards the left of the vehicle, Y is positive upwards, and Z is positive towards the front of the vehicle. The program was designed to take input in units of inches.

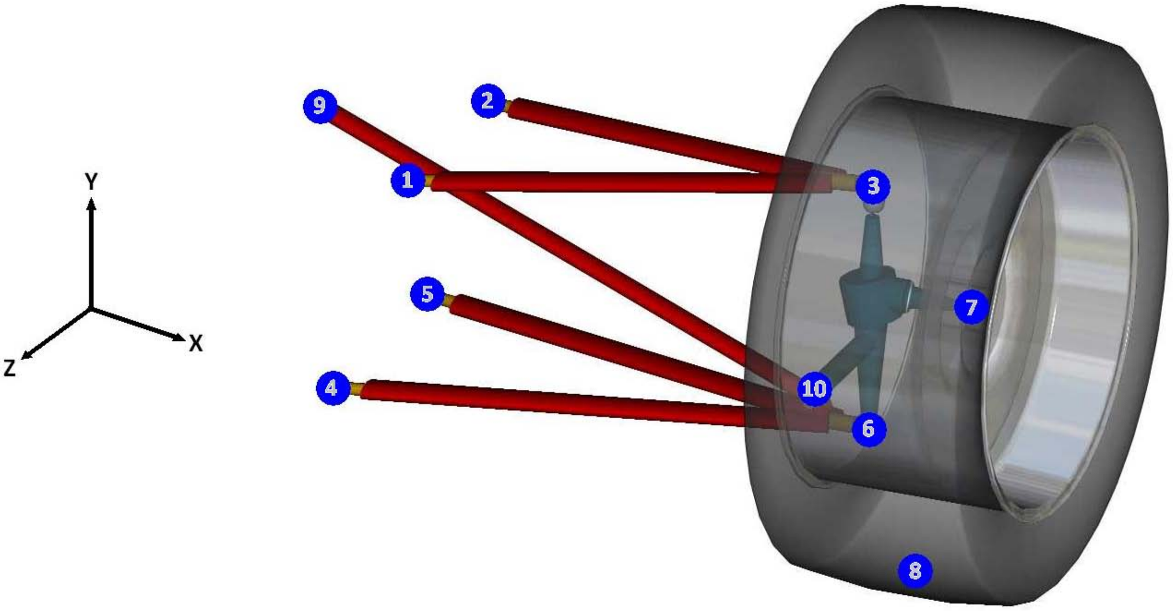

Figure 1: Coordinate system and input points [adapted from (Timmins, 2004)]

Suspension Geometry

1) Upper A-arm: front inner ball joint
2) Upper A-arm: rear inner ball joint
3) Upper A-arm: outer ball joint
4) Lower A-arm: front inner ball joint
5) Lower A-arm: rear inner ball joint
6) Lower A-arm: outer ball joint
7) Wheel: center (halfway between each tire sidewall)
8) Wheel: middle of tire, ground plane (note that this will necessarily have a zero value for Y and Z)
9) Tie rod: inner ball joint (attachment point to steering linkage)
10) Tie rod: outer ball joint

Note that the suspension to be analyzed may not have precisely the same configuration or hardware. For instance, the inner ball joints are often revolute joints instead of ball (spherical) joints. However, this is kinematically equivalent. The inner ball joints form an axis about which the A-arm rotates, which is completely equivalent to two revolute joints which are coaxial.

Also, for rear suspensions, the suspension may be of the "H-arm" type in which the outer ball joints are revolute joints (i.e., they do not allow rotation of the spindle). If this is the case, the program can still be used, but some additional steps need to be taken. The non-existent tie rod needs to be included, but in such a way as to prevent the spindle from rotating. In order to accomplish this, the tie rod must be coplanar with either the upper or lower A-arm. The suggested method is this:

1) Set tie rod inner ball joint location equal to either upper or lower front inner ball joint location
2) Find a point a few inches in front of the upper or lower (whichever was chosen in step one) which is still coplanar with the chosen A-arm
3) Enter this location for the outer ball joint

If step two is too difficult, try guessing at the point. A correct (or close enough) point has been found when the bump steer results are zero for all travel (because this means there is no spindle rotation throughout jounce/rebound).

Solution Options

Travel – enter the total travel (jounce and rebound). Travel is assumed to be symmetric. The program uses this to determine how far to move the suspension when calculating the kinematics.

Brake bias – this is the braking bias applied towards the front (0-100%). So, a value of 60% means that 60% of the braking force is on the front wheels, and the remaining 40% is on the rear. This is used when calculating suspension "anti-" features.

Outboard brakes – whether or not the vehicle has outboard brakes on the portion of the suspension entered. Because torque (either driving or braking) is required for suspension "anti-" features, outboard brakes are required to generate braking torque at the wheel (the torque generated by inboard brakes does not actuate "anti-" features).

Front suspension – this is used to decide which "anti-" features to calculate (e.g., anti-dive for a front suspension is analogous to anti-lift for a rear suspension). Note that it is assumed that the vehicle is rear drive (the program will not calculate anti-lift for a front suspension), because this is true of the vast majority of Formula SAE and Baja SAE vehicles).

Wheelbase length – this is used when calculating "anti-" features.

CG height – this is used when calculating "anti-" features.

When iterating many times, it may be advantageous to avoid using the GUI and instead enter data manually. If this is the case, simply enter the data at the beginning of the **SLASIM_Call.m** file. Within this file some advanced features can also be changed, including the number of steps calculated in each direction (jounce and rebound) and the relative precision of the numerical solutions. The default number of steps taken in each direction is 25, which produces a relatively smooth curve. To change this, simply change the variable **n_steps** within the **SLASIM_Call.m** file (note: increasing this number will increase runtime). Changing the number of steps will not affect the accuracy of the parameters calculated at each step, just the smoothness of the data plotted. In order to change the relative precision of the numeric solutions, **n_points** can be changed. It is not recommended to change the value below the default of 10,000 (position errors are on the order of a few thousandths of an inch when using this value), although it can be safely increased at the expense of runtime.

Program Output

After running SLASIM (either from the GUI or directly through the SLASIM_Call.m file), the program will return (single-valued) numeric parameters as well as plotted data (parameter values as a function of suspension travel). Additionally, it saves the workspace at the end of the run as "matlab.mat." This file will be overwritten on subsequent runs unless renamed.

Numeric Parameters

- Caster angle
- (Mechanical) Trail
- Kingpin angle
- Scrub
- Spindle length

Plotted Parameters

- Suspension position throughout travel (front view)
- Suspension positions in jounce (3D view – use "Rotate 3D" tool)
- Camber curve
- Bump steer curve
- Roll center height (symmetric jounce/rebound)
- Roll center location while cornering (symmetric, opposite suspension movements)
- Front anti-dive (front suspensions only)
- Rear anti-squat (rear suspensions only)
- Rear anti-lift (rear suspensions only)

Program Limitations

This program has some limitations that should be kept in mind during use. Those thought to be most important are listed below. If a particular calculation is in question, it is recommended to go to the code.

Numerical Methods

When calculating suspension kinematics, many of the points are found (for example, the location of the upper A-arm outer ball joint) numerically. In this case, two things are known about the location of the upper A-arm outer ball joint:

1) It is rigidly affixed to the upper A-arm, which means it lies somewhere along a circle described by the rotation of the upper A-arm.
2) The spindle is rigid, so the distance between the upper A-arm outer ball joint and the lower A-arm outer ball joint is constant.

From (1) and (2), we know that the upper A-arm lies at the intersection of a circle (from (1)) and a sphere (from (2), it is a sphere with origin at the lower A-arm outer ball joint and a radius equal to the distance between the two outer ball joints).

Generally, the intersection of a sphere and circle will describe two points. A circle consisting of 10,000 (by default) points is generated, and the two points which come closest to intersecting with the sphere are saved. Because the suspension is moving in small steps, the program looks for the solution which is closest to the previous solution.

Position error (using the default values of 25 steps and 10,000 points) is believed to be on the order of a few thousandths of an inch. Note that the error does not accumulate from step to step.

Other

When calculating roll center height, it is assumed that the suspension is symmetric from left to right. This is of course necessary when only taking data from one side of the vehicle. Since the suspension is assumed to be symmetric, the roll center is necessarily located at half-track.

When calculating roll center *position* during cornering, it is assumed that the vehicle has opposite, symmetric movements. That is, if the left suspension goes two inches into jounce, it is assumed that the right suspension goes two inches into rebound. Without data on weight transfer and spring rates, it is impossible to know suspension positions (and so opposite, symmetric movements are assumed).

No deformations are taken into account when the program is run. Because this is strictly a kinematics program, it is beyond its scope. However, these can be important for vehicle performance (see Milliken for reference). Physical testing or finite element analysis is recommended.

This program was tested using data from a Formula SAE vehicle at The Ohio State University (the values are included in the code, but commented out). It has only been tested using these values (or values very close). The results have not been rigorously verified. The code also may not work when presented with unrealistic values. It may also have trouble with impossible commands (e.g., 10,000 inches of suspension travel).

SLASIM assumes that the suspension can be assembled and will not bind. When the program calculates suspension positions, it does not check to see if the components would interfere. For this reason, it is recommended to assemble the suspension in a normal CAD program before fabrication.

Bibliography

Milliken, W. F. (1995). *Race Car Vehicle Dynamics.* Warrendale, PA: SAE, Inc.

Timmins, S. (2004, Spring). *Race Car Vehicle Dynamics Course Website*. Retrieved April 7, 2010, from Univeristy of Delware Website: http://www.me.udel.edu/meeg467/Vehicles/index.html

Waldron, G. L. (2003). *Kinematics, Dynamics, and Design of Machinery.* John Wiley & Sons.