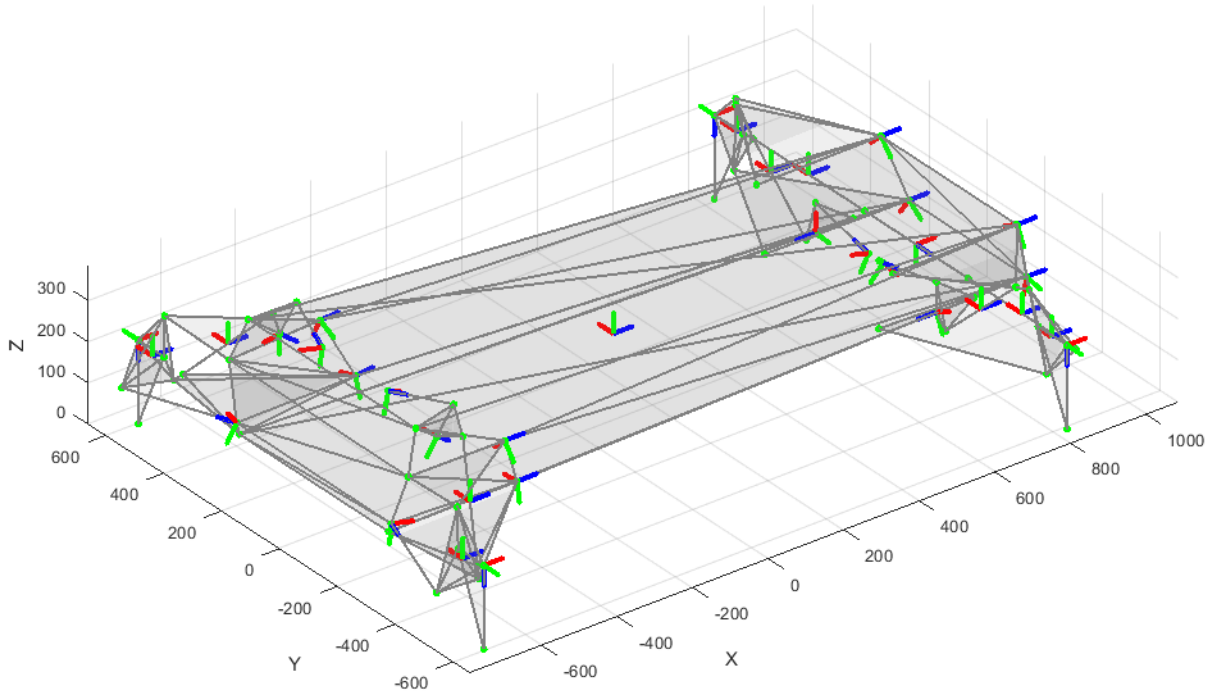
Mech 410D Term Project Report

Modelling Vehicle Suspension Kinematics of a Formula SAE Race Car

Graham Rowe

December 2020



Preface

This report outlines my efforts to model the suspension of a race car. The scope of this report differs slightly from what was stated in the interim report. Originally I had planned to create a dynamic model of a vehicle, which took into account the effects of inertias, springs, dampers, tires, etc. to predict the state of a vehicle as a function of time given initial conditions and driver inputs. This turned out to be infeasible given the amount of time and my understanding of the subject matter.

I settled on creating a Matlab class that could model the kinematics of a suspension given the state of several key variables. Namely:

- 4× shock positions
- Steering position
- The vehicles position in space.
  - 3× positions
  - 3× rotations

Given these inputs it is possible to predict the position and orientation of all parts of the vehicle.

Some snippets of Matlab code are included in the body of this report for explanation purposes. These are not necessarily exactly the same as the code being executed in the model, but have been simplified for brevity. A full copy of the model code is included as an appendix.

1 Introduction and Problem Statement

1.1 Background

The design of a race car is obviously extremely complex, and requires the use of many design tools to develop a well performing car. The suspension system is of particular interest to the subject of this course, as there are over 30 rigid bodies interacting in different ways to ultimately allow the vehicle to roll down the track.

The Formula UBC Racing team has used multiple software packages and design techniques to help develop the suspension kinematics of their vehicles. This includes but is not limited to:

1. Spreadsheets and hand calculations:
    - Useful to develop high level design targets
    - A good first step in any design, but is limited in the fidelity that can be achieved:
    - Assume parameters to be constant
        - This makes it difficult to properly model the effects of variable rate motion ratios, non-linear springs, etc. all of which can be useful to extract the maximum amount of performance from a vehicle.
    - only calculate single configurations at once
        - Although it is theoretically possible to create a spreadsheet or similar tool to calculate every permutation of vehicle setup, it is infeasible for our application given the number of parameters that can be varied even on a simple vehicle.
2. Team developed software tools (mainly written in Matlab)
    - A simplified quarter car model
    - A quazi-steady state lap-time simulation
    - Tire modelling package
3. Dedicated vehicle kinematic modelling packages (Optimum Kinematics):
    - Being a dedicated package, it obviously serves its purpose very well of modelling the motion of the suspension. However it can be challenging to integrate the results effectively into other design tools.
4. Cad modelling packages (Solidworks, NX, etc.):
    - Required when developing individual components and assessing packaging.

1.2 Goals

The tool developed as a part of this project will aim to replace item 3 in the previous list. The software used currently costs the team hundreds of dollars each year, and developing our own software will allow for greater control and understanding of the underlying calculations. Specifically the goals of this project are:

1. Model the motion of suspension components in 3D space.
2. Allow the user to easily input geometry

3. Be written in a general enough manner that it is possible to model different suspension layouts. (eg. double wishbone, multi-link, solid axle, etc.) This can be done by either
   - Altering the inputs given to the tool
   - Making minor alterations to the source code.
     "Minor" = less than 20 lines.
4. Model forward kinematics
   - Inputs are 11 parameters mentioned in preface.
5. Visualize the vehicle to the user
   - This is required to ensure that the configuration and inputs are as expected.
6. Provide a framework for calculating desired suspension output values

2 Initialization

2.1 Layout

The majority of the effort in this project was put into writing `classdef SuspensionSimulation` which will be referred to as "the model."

A spreadsheet is used to input suspension configuration data

Another matlab script can call `SuspensionSimulation` in order to perform analysis on the suspension.

2.2 Storage

I elected to store the configuration data in a spreadsheet, that will then be read by the model to create the vehicle model object in Matlab. FUBC has used a spreadsheet to store suspension configuration for other analysis in the past, and in the future it should be possible to write a Solidworks macro to extract information directly from a CAD model.
The following is the data that is included. Headings are used to distinguish one section from another.

1. A list of bodies that make up the suspension

### Components ###	
Element	Points
Chassis	LFUF LFUA LFLF LFLA LFRP LFRV LFSC RFUF RFUA RFLF RFLA RFRP RFR
LF Upright	LFUU LFLU LFTU LFWC LFWA
RF Upright	RFUU RFLU RFTU RFWC RFWA
LR Upright	LRUU LRLU LRTU LRWC LRWA

   - And a list of connected points for each body.

2. A list of points in 3D that define the positions of everything when the car is at ride height

### Hard Points ###			
Point Nam	X	Y	Z
LFUU	508	298.45	809.244
LFUF	234.95	269.88	985.27
LFUA	234.95	266.7	655.32
LFLU	539.75	101.6	829.31

3. User options

### User Options ###	
Units, Len	mm
Units, Ang	deg
Units, For	N
Coordinat	[[0 0 1] [1 0 0] [0 1 0]]

   - Units (length, angle, force)
   - Coordinate system
     - This is so that the list of points can be given in the coordinate system of a CAD package, and then rendered using the default Matlab coordinate system of $Z : up$.

4. The position of each section of information

7	### User Options ###	D1:E5
8	### Components ###	H2:S36
9	### Hard Points ###	U1:X88
10	### Wheels ###	Z1:AA5
11	### Steering ###	AC1:AD4
12	### Damping ###	AF1:AG11

   - This can be calculated programatically using excel to ensure that we are getting exactly the dat required.

2.3 Loading

In Matlab, meta data from the spreadsheet is read by the following:

```
meta = readtable(spreadsheet_name, 'Range', 'A:B', 'ReadRowNames',
   true, 'VariableNamingRule', 'preserve');
```

I can then get each section of data based on the range strings stored in that "meta" table by repeating the following code for each heading of interest.

Doing this also allows the user to put other information in the spreadsheet outside of these ranges that will not affect this model, as only the required range is read.

```
range = string(meta.(1)(heading_string))
data = readtable(spreadsheet_name, 'Range', range, 'ReadRowNames',
   true, 'VariableNamingRule', 'preserve');
```

From here, all that's required is to copy the data into properties of the `SuspensionSimulation` object and perform some array manipulation to get everything in the correct format for calculations.

The following outlines the key variables stored as object properties:

- `points`:
  Array of point positions in 3D space. The fourth row is all 1s, as the transformations are performed using homogeneous transformations. (To be further explained in a later section.) Each point that is given in the spreadsheet is listed twice in this array; once per body that contains that point.

$$points = \begin{bmatrix} X_1 & X_2 & X_3 & & X_n \\ Y_1 & Y_2 & Y_3 & \dots & Y_n \\ Z_1 & Z_2 & Z_3 & & Z_n \\ 1 & 1 & 1 & & 1 \end{bmatrix} \tag{1}$$

- `body_names`
  Cell array of strings. A direct copy of the data from the *Element* column in the spreadsheet.

- `connection_matrix`:
  Array that lists which points in the master points list are connected.

$$connections = \begin{bmatrix} C_1 & C_2 & C_3 & \dots & C_n \end{bmatrix} \tag{2}$$

  For example:
  if

$$C_{25} = C_{10} = a$$

  then

$$\begin{bmatrix} X_{10} \\ Y_{10} \\ Z_{10} \end{bmatrix} - \begin{bmatrix} X_{25} \\ Y_{25} \\ Z_{25} \end{bmatrix} \overset{\text{should}}{=} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{3}$$

  and P_{10} and P_{25} are connected to 2 separate bodies.

- `attached_pts_truth`:
  Boolean array. Each row of this array corresponds to a body in the model. Each column corresponds to a point in the master points list. $ATP = \begin{bmatrix} T_{1,1} & T_{1,2} & T_{1,3} & & T_{1,n} \\ T_{2,1} & T_{2,2} & T_{2,3} & \dots & T_{2,n} \\ T_{3,1} & T_{3,2} & T_{3,3} & & T_{3,n} \\ & \vdots & & \ddots & \vdots \\ T_{m,1} & T_{m,2} & T_{m,3} & \dots & T_{m,n} \end{bmatrix}$
  This makes for easy indexing of the points array when manipulating the model later on. The point positions and identifiers attached to a given body, m, are:

```
point_positions = points(attached_pts_truth(m,:));
point_ids = connection_matrix(attached_pts_truth(m,:));
```

- `body_frames`:
  A cell array that contains the frames of each body represented in homogeneous form in a 4×4 matrix. Each frame represents the position and orientation of it's associated boy in the world frame.

$$frames = \left\{ \begin{bmatrix} \underline{\underline{R}}_1 & \underline{P}_1 \\ \underline{0} & 1 \end{bmatrix} \quad \begin{bmatrix} \underline{\underline{R}}_2 & \underline{P}_2 \\ \underline{0} & 1 \end{bmatrix} \quad \dots \quad \begin{bmatrix} \underline{\underline{R}}_m & \underline{P}_m \\ \underline{0} & 1 \end{bmatrix} \right\}$$

Each of the variables listed above is copied to a "static" version of itself. This is to reset everything before a transformation is applied so that any transformation is absolute, instead of moving relative to the previous position. Relative moves are still possible by simply recording the current state and calculating the relative state from there, so this shouldn't pose a problem.

The next step is to find indices of key components in the `body_names` array. This allows for easy identification of components when manipulating the model. The following code snipping finds the index of `name` in the array of `body_names`.

```
idx = sum([1:num_bodies]' .* contains(body_names, name,
    'IgnoreCase',true));
```

This is done for:

- The chassis
- 4× dampers
- The steering rack
- 4× tires

3 Kinematic Calculations

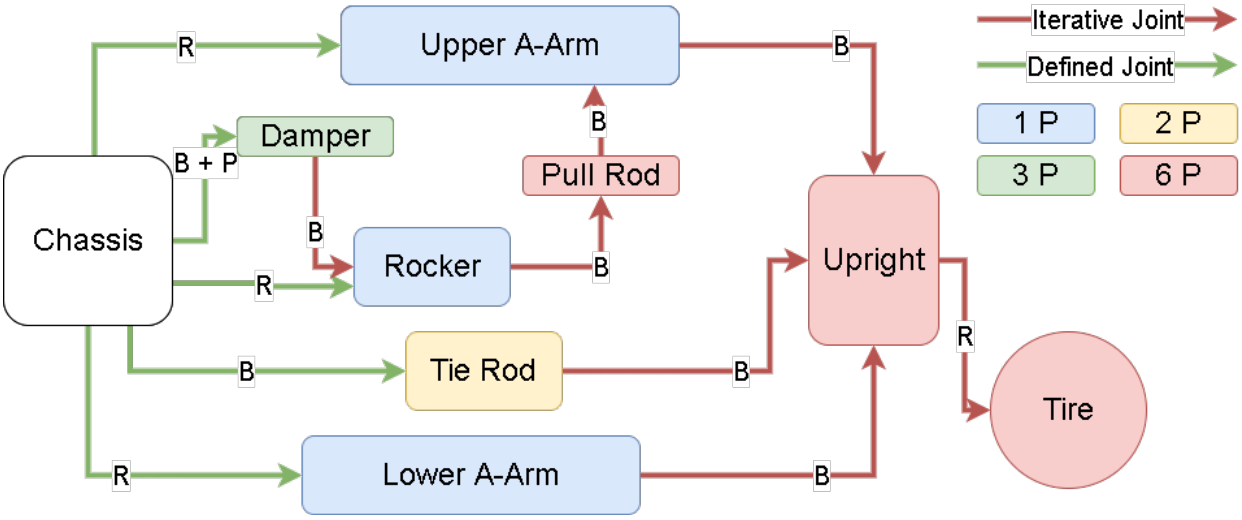


Figure 1: Connections between bodies on one corner of a suspension system.

Figure 1 shows the layout of a single corner of a vehicle suspension. It will be referred to throughout this section to help explain the calculations being performed.

3.1 Theory

3.1.1 Frames

A frame of reference is required in order to perform any kinematic calculations. In 3D cartesian coordinates, a frame is typically represented by 3 mutually perpendicular unit vectors $(\hat{i}, \hat{j}, \hat{k})$ indicating the 3 coordinate direction, and an origin point $(\underline{p})$ relative so some world frame. In homogeneous form a frame is represented by a 4×4 matrix.

$$F_{sa} = \begin{bmatrix} \hat{i} & \hat{j} & \hat{k} & \underline{p} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

The frame F_{sa} is the coordinate system a expressed in s frame. To get s expressed in a frame, take the inverse of the matrix F_{sa}:

$$F_{as} = F_{sa}^{-1} \tag{5}$$

A frame that moves with a rigid body can be used to keep track of that body's motion through space, as it is much more efficient to perform calculations on a single matrix, rather that comput position and orientation from a list of points. Additionally, pre-multiplying some vector expressed in s frame by F_{sa} will give that vector expressed in a frame.

$$r_s = F_{sa} r_a \tag{6}$$

3.1.2 Transformations

General rigid body motion can be expressed as a rotation and linear translation in 3 independent axes. One method to represent this is homogeneous transformations. A homogeneous transformation of point $\underline{p}$ is:

$$\begin{bmatrix} \underline{r} \\ 1 \end{bmatrix} = \underline{\underline{T}} \begin{bmatrix} \underline{p} \\ 1 \end{bmatrix} \tag{7}$$

$$\underline{\underline{T}} = \begin{bmatrix} \underline{\underline{R}} & \underline{P} \\ \underline{0} & 1 \end{bmatrix} \tag{8}$$

$$\underline{\underline{R}} = \underline{\underline{R}}(\omega, \theta) = \underline{\underline{I}} + sin(\theta \left[\hat{\omega}\right]) + (1 - cos(\theta)) \left[\hat{\omega}\right]^2 \tag{9}$$

$$\left[\hat{\omega}\right] = \begin{bmatrix} 0 & -\hat{\omega}_Z & \hat{\omega}_Y \\ \hat{\omega}_Z & 0 & -\hat{\omega}_X \\ -\hat{\omega}_Y & \hat{\omega}_X & 0 \end{bmatrix} \tag{10}$$

Where:

- $\underline{p}$ is the point to transform (3×1)
- $\underline{r}$ is the new transformed point (3×1)
- The transformation matrix $\underline{\underline{T}}$ is a is a 4×4 matrix combining a rotation matrix and translation vector and is defined by (8).
- $\underline{P}$ is the linear translation
- $\left[\hat{\omega}\right]$ is called the screw symmetric form of the vector $\hat{\omega}$ and is the axis around which the point $\underline{p}$ is rotated around by the angle θ. It is defined by (10).

Note that is also possible to describe motion as movement of a 4×4 reference frame, in which

case the vector $\left(\begin{bmatrix} p \\ 1 \end{bmatrix}\right)$ can be replaced with $\left(\begin{bmatrix} \hat{i} & \hat{j} & \hat{k} & O \\ \underline{0} & & & 1 \end{bmatrix}\right)$

3.1.3 Screw Theory

To create the transformation matrix, we can use screw axis theory, also known as Chasles-Mozzi theorem, which expresses any rigid body motion as a combination of linear motion along an axis and rotation about that axis. This axis is called the "screw axis" and is denoted by s. s is the normalized twist, $\underline{\nu}$. Both are 6×1 vectors.

$$\underline{\nu} = \begin{bmatrix} \underline{\omega} \\ \underline{v} \end{bmatrix} = \begin{bmatrix} \hat{s}\dot{\theta} \\ -\hat{s}\dot{\theta} \times \underline{q} + h\hat{s}\dot{\theta}` \end{bmatrix} \tag{11}$$

Where:

- $\hat{s}$ is the unit vector defining the axis direction
- $\dot{\theta}$ is some angular velocity
- $\underline{q}$ is the position of the body expressed in the world frame
  (or whatever frame of reference is convenient for the transformation)
- $h = \dfrac{||\underline{v}||}{||\underline{\omega}||}$ is the "pitch"

$$\text{Therefore:} \quad \underline{s} = \frac{\underline{\nu}}{||\underline{\nu}||} \tag{12a}$$

$$\text{Which for pure rotation is:} \quad \underline{s} = \begin{bmatrix} \dfrac{\underline{\omega}}{||\underline{\omega}||} \\ \\ \dfrac{\underline{v}}{||\underline{\omega}||} \end{bmatrix} \tag{12b}$$

$$\text{And for pure translation is:} \quad \underline{s} = \begin{bmatrix} \underline{0} \\ \\ \dfrac{\underline{v}}{||\underline{v}||} \end{bmatrix} \tag{12c}$$

Finally, using exponential coordinates:

$$e^{\left[\underline{s}\right]\theta} = \underline{\underline{T}} = \begin{bmatrix} \underline{\underline{R}} & \underline{\underline{I}}\theta + (1 - cos(\theta))\left[\hat{\omega}\right] + (\theta - sin(\theta))\left[\hat{\omega}\right]^2 \underline{v} \\ \underline{0} & 1 \end{bmatrix} \tag{13}$$

Note that θ is now being used as some generalized coordinate, not exclusively for rotation. The screw axis defines the track that the object will follow through space, θ defines how far along that track the object moves.

To represent multiple joints, simply calculate $\underline{\underline{T}}$ for each joint ($\underline{s}$, θ combination) and multiply the results to get the final transformation functio. The model creates functions for these transformation functions and uses a numerical solver to determine the constrained parameters. More on this next.

3.2 Implimentation[1]

In Matlab, calculating $\underline{\underline{T}}$ given $\underline{s}$ and θ can be done with `e_SxT(s, theta)`

Listing 1: Calculating transformation matrix

```
function [e_SxTmat] = e_SxT(S,theta)
    omega=S(1:3);
    v=S(4:6);

    ax=omega; ang=theta; Rmat = rot_mat(ax,ang); % Find the
       rotation matrix

    somega = vectoskew(omega);  % Skew symmetric form of omega

    if(norm(omega)~=0)  % rotation
        e_SxTmat =[Rmat
           (eye(3,3)*theta+(1-cos(theta))*somega+(theta-sin(theta))*somega^2)*v
           zeros(1,3) 1];%zeros(size(Rmat)) eye(size(somega))]
    else % Pure translation
        e_SxTmat =[eye(3,3) v*theta;zeros(1,3) 1];

    end
end
```

Listing 2: Calculating rotation matrix

```
function [Rmat] = rot_mat(ax,ang)
    % Rodrigues' formula
    Sax=[0 -ax(3) ax(2);ax(3) 0 -ax(1);-ax(2) ax(1) 0];
    Rmat=eye(3,3)+sin(ang)*Sax+(1-cos(ang))*Sax^2;
end
```

Listing 3: Converting a vector to skew symmetric form

```
function [Svec] = vectoskew(vec)
    % Skew symmetric form of a vector
    Svec=[0 -vec(3) vec(2);vec(3) 0 -vec(1);-vec(2) vec(1) 0];
end
```

Since θ is a parameter, and $\underline{s}$ is constant, it makes more sense to write a function that only takes in θ and returns a transformation matrix. Additionally, some of the calculations above are only required once when the suspension is initialized, and it would be expensive to perform all of them every time a body is moved. Instead, the model calculates the screw axis (more on this next) and then uses the Matlab symbolic toolbox to define the parameter variable, θ. From there it gets the transformation matrix from `e_SxT(s, theta)` and passes that into a builtin Matlab function, `matlabFunction(T, 'Vars', theta)`. This converts the

[1]Throughout this section I will use superscripts (e.g.$^{\text{L452}}$) to point to a relevant line in the source code contained in Appendix A.

symbolic matrix into a function handle. This function handle will return the appropriate transformation matrix for any given θ and can be stored in a cell array for later use.

An example is below:

Listing 4: Calculating transformation function

```
% s is some screw axis
syms theta real

T = e_SxT(s,theta);
func = matlabFunction(T, 'Vars', theta);
```

The body frame and screw axis is calculated in different ways for different joints and bodies. All transformations performed on the suspension bodies are done so in the chassis frame. From there the entire vehicle is moved to where it needs to be in space.

Referring to figure 1, all of the components connected directly to the chassis (through the green arrows) can easily manipulated by defining a screw axis. There are several scenarios that all need to be handled differently. When initializing, the model loops through each body that makes up the suspension and creates the body frame and transformation functions. The different scenarios are listed in table 1. The model loops over each body, using `m` as a loop counter.

Table 1: Frames and Screw Axis Definitions

Condition(s)	Joint(s) Description	Matlab Implimentation

Dampers	1× Universal joint with both revolute axes intersecting at the shared point. 1× Prismatic joint along it's own axis	

```
%% Frame
attached_pts =
   obj.points(1:3,obj.attached_pts_truth(m,:));
% Z axis is along the axis of the
   damper
Z = attached_pts(:,1) -
   attached_pts(:,2);
Z = Z./norm(Z);
T = rand(3,1);
X = cross(T,Z);
X = X ./ norm(X);
Y = cross(X, Z);
R = [X Y Z];
O = shared_pt(:,1);

%% Screw Axis
% calculate the 2 screw axes for
   rotation
s1 = [X ; cross(-X, shared_pt)];
s2 = [Y ; cross(-Y, shared_pt)];
% calculate the screw axis for
   prismatic joint
s3 = [[0 0 0]'; Z];
```

Steering Rack	1× Prismatic joint defines by the 2 end points of the rack	`%% Frame`
		`attached_pts = obj.points(1:3,obj.attached_pts_truth(m,:));`
		`% Z axis aligned with axis of rack`
		`Z = attached_pts(:,1) - attached_pts(:,2);`
		`Z = Z./norm(Z);`
		`T = [0 1 0]';`
		`X = cross(T,Z);`
		`X = X ./ norm(X);`
		`Y = cross(X, Z);`
		`R = [X Y Z];`
		`% origin is at center`
		`O = (attached_pts(:,1) + attached_pts(:,2))./2;`
		
		`%% Screw Axis`
		`s1 = [zeros(3,1) ; Z]; % prismatic joint`

Connected to the chassis via 2 points	1× revolute joint defined by the 2 shared points.	`%% Frame`
		`% Z axis is coincident with rotation axis Z = shared_pts(:,1) - shared_pts(:,2);`
		`Z = Z/norm(Z); T = rand(3,1);`
		`X = cross(T,Z);`
		`X = X ./ norm(X);`
		`Y = cross(X, Z);`
		`R = [X Y Z];`
		`O = shared_pts(:,1);`
		
		`%% Screw Axis`
		`s_h = Z;`
		`% screw axis. In this case because it's pure rotation, the pitch = 0`
		`screw_ax = [s_h;cross(-s_h, shared_pts(:,1))];`

Connected to the chassis via 1 point and has exactly 2 points total	1× Universal joint: both revolute axes intersecting at the shared point.	`%% Frame`
		`attached_pts = obj.points(1:3,obj.attached_pts_truth(m,:));`
		`% Z axis is along axis of link`
		`Z = attached_pts(:,1) - attached_pts(:,2);`
		`Z = Z./norm(Z);`
		`T = rand(3,1);`
		`X = cross(T,Z);`
		`X = X ./ norm(X);`
		`Y = cross(X, Z);`
		`R = [X Y Z];`
		`O = shared_pt;`
		
		`%% Screw Axis s1 = [X ; cross(-X, shared_pt)];`
		`s2 = [Y ; cross(-Y, shared_pt)];`

Connected to the chassis via 1 point and has at least 3 points total	1× Spherical Joint: 3× revolute joints intersecting at the shared point	`%% Frame`
		`attached_pts = obj.points(1:3,obj.attached_pts_truth(m,:));`
		`% Z axis points towards the center of mass of body`
		`Z = attached_pts(:,1) - sum(attached_pts, 2) ./ sum(obj.attached_pts_truth(m,:));`
		`Z = Z./norm(Z);`
		`T = rand(3,1);`
		`X = cross(T,Z);`
		`X = X ./ norm(X);`
		`, Y = cross(X, Z);`
		`R = [X Y Z];`
		`O = shared_pt;`
		
		`%% Screw Axis`
		`% calculate the 3 revolute screw axes`
		`s1 = [X ; cross(-X, shared_pt)];`
		`s2 = [Y ; cross(-Y, shared_pt)];`
		`s3 = [Z ; cross(-Z, shared_pt)];`

Everything Else	All other bodies are given 6 degrees of freedom. The solver will be able to move them to any point in space to satisfy the constraints.	

```
%% Frame
attached_pts =
   obj.points(1:3,obj.attached_pts_truth(m,:));
R = eye(3);
% origin is ~at center of mass of
   object
O = sum(attached_pts, 2) ./
   sum(obj.attached_pts_truth(m,:));

%% Screw Axis
% Rotations
s1 = [[1 0 0]' ; cross(-[1 0 0]', O)];
s2 = [[0 1 0]' ; cross(-[0 1 0]', O)];
s3 = [[0 0 1]' ; cross(-[0 0 1]', O)];
% Translations
s4 = [zeros(3,1) ; [1 0 0]'];
s5 = [zeros(3,1) ; [0 1 0]'];
s6 = [zeros(3,1) ; [0 0 1]'];
```

After each iteration of the loop, the model excecutes the code in listing 11 to store the transformation function and frame. Note: there may be up to 6 transformation functions, but only 2 are shown here.

Listing 11: Saving the frame and transformation function

```
obj.body_frames{m} = [R, O; zeros(1,3), 1];

T_1 = obj.e_SxT(s1, theta1);
T_2 = obj.e_SxT(s2, theta2);

T = T_1 * T_2;
func = matlabFunction(T_cb, 'Vars', {[a b x y z]});
obj.trans_funcs{m} = func;
```

3.3 Move[L758-813]

Because all movements of the suspension are calculated in absolute deviation from a "home" position, before performing any move calculation, the points and body frames are reset to their static positions.[L764-765] The model then loops from 1 to the total number of bodies.

1. If m corresponds to the chassis, then skip to the next instance of the loop. [L769-773] The chassis transformations are applied to everything at the end.
2. Retrieve the transformation function for the body.[L775]
3. Retrieve the parameters from the parameter list.[L776-777]

4. Calculate the transformation matrix using the parameters and transformation function.[L779]
5. calculate the new position of the body's points and frame by multiplying by the transformation matrix.[L781-782]

Once that loop is complete, steps 2 to 5 are repeated, this time for the chassis. Now, however, the transformation is applied to all points and frames.[L785-796]

3.4 Solver

The previous sections describe how the model is able to move bodies around in 3D space based on parameters. The solver function aims to solve the parameters subject to the constraints of the model. As was described in 2.3, the points list contains duplicates of each points, and these points should be in the same position in space. The solver will vary the parameters in order to minimize the error function.

3.4.1 Error Function [L701-728]

The error function is equation 3 applied to each par of points. For any general scenario where the parameters are not necessarily abiding by the constraints of the model, for two points a and b that are supposed to be coincident, 3 becomes

$$\begin{bmatrix} X_a \\ Y_a \\ Z_a \end{bmatrix} - \begin{bmatrix} X_b \\ Y_b \\ Z_b \end{bmatrix} = \begin{bmatrix} E_x \\ E_y \\ E_z \end{bmatrix} \tag{14}$$

Where $\underline{E}$ is the error. Notice that the Euclidean distance is never calculated, which avoids the use of relatively expensive squares and roots. Since Matlab's `fsolve()` solver is able to accept an array from the function to be minimized, this makes the error function relatively simple and fast:

1. The move command is called to ensure that the current state of all points and frames is representative of the guessed parameters. [L704-705]
2. For each unique point number in the connection matrix:
   (a) Find the point ideces[2] to compare using `find(obj.connection_matrix==i)`.[L715]
   (b) Calculate the error using from equation 14.[L716-719]
3. flatten the error array and return the result.[L724]

3.4.2 Setting the Vehicle State[L880-895]

The user assigns the input parameters by calling the `setState(inputs)` function. This function parses the 11×1 input vector and assigns the values to the appropriate parameters. It then adds the indeces of those parameters to a list of locked parameters. At the beginning of the error function, only the unlocked parameters are taken from the solver's guess, and the locked parameters are carried over from the user's inputs. [L704]

[2] Although up until this point I have said that there are only ever 2 points that are connected, I've written this portion so that in theory there could be an infinite number of points that are coincident. In the design of real race car this would be nearly impossible to acheive, but it's benefits may pose an interesting design question in the future.

3.4.3 Solve Function[L671-699]

In essence, the solve function just attempts to minimize the error function. However, the solver choice and options makes a significant difference in solution time and accuracy. Several tests were run to confirm that the solution would converge consistently, and do so in a reasonable amount of time. There are several options that were investigated.

1. `fsolve()` [3]
2. `fminsearch()` [4]

A random set of input parameters were generated. These parameters were created from the `rand()` fuction and scalled to the appropriate scale. Thus, consecutive inputs are not necessarily close together, which is a worst case scenario as the changes to the parameter list will need to be relatively large. Each solver method was then called to solve for the given vehicle state of each set of input parameters. The typical solve time for 20 different input cases are shown for each of the two methods in figure 2. Note the order of magnitude difference shown on the Y axis.

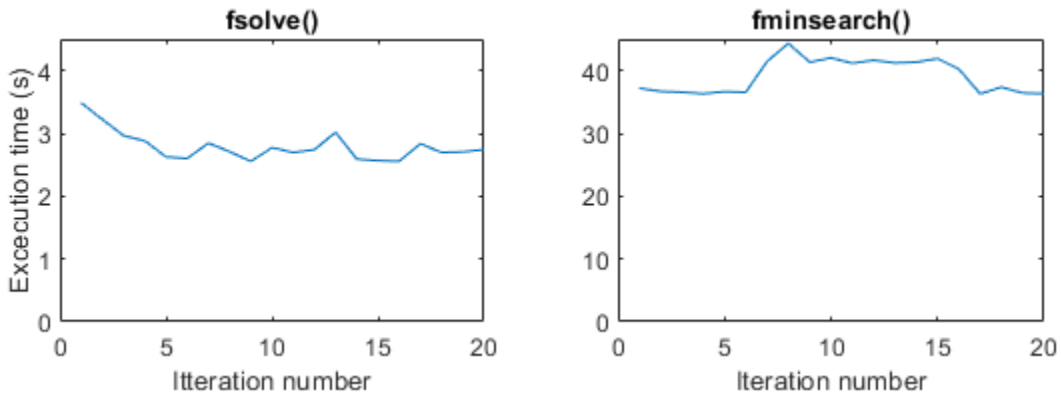


Figure 2: Solve time of fminsearch and fsolve

In general, `fsolve()` vastly outperformed `fminsearch()`. `fminsearch()` also did not correctly solve every state. The hump in the middle of the right graph shows instances where `fminsearch()` reached the maximum iteration limit. `fsolve()` on the other hand converged every time.

Within fsolve there is also several choice of algorithms.

1. 'trust-region-dogleg'
2. 'trust-region'
3. 'levenberg-marquardt'

'levenberg-marquardt' solves in every condition, however it is significantly slower for this problem than either of the trust-region algorithms (>3.5s vs <2.5s). An additional benefit of the 'trust-region' algorithm is the ability to pass in a jacobian pattern matrix. This significantly reduces the number of computations required, as the algorithm does not need

[3] https://www.mathworks.com/help/optim/ug/fsolve.html
[4] https://www.mathworks.com/help/matlab/ref/fminsearch.html

to check the partial derivative of every single output parameter to continue. The jacobian pattern is calculate when the model is first loaded. [L656-669]

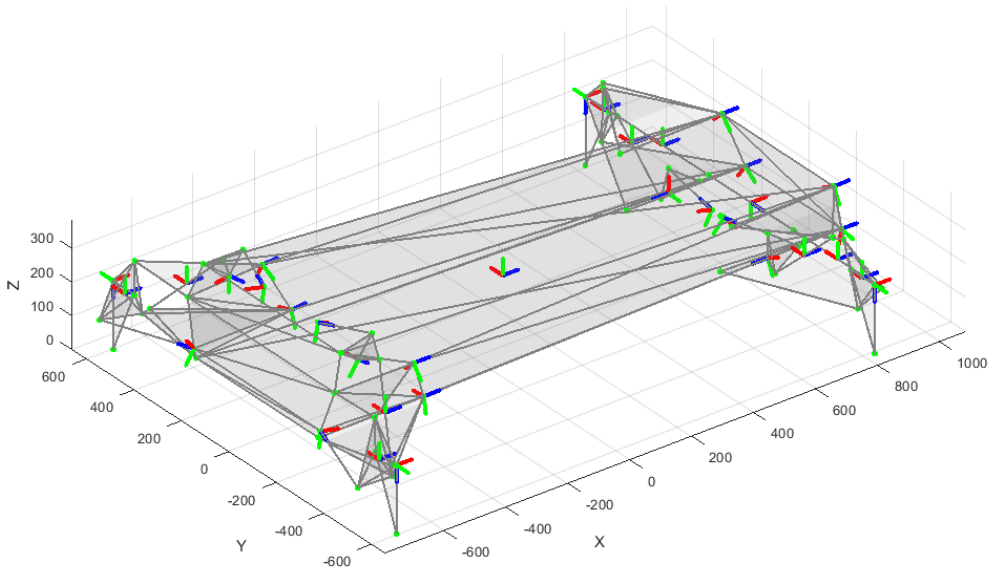
'trust-region' requires the system to be square, meaning that there are the same number of inputs as there are outputs. Since the system in this model is not necessarily square, the parameters list is padded with zeros when the model is loaded.[L659] This addition of "dummy" parameters does not affect the calculations of any of the other functions in the model, and the speed benefits of using the 'trust-region' algorithm make it worth while.

The trust region algorithms, however, will only converge when the assigned state is relatively close to the home state. When the state is far from home it will find a condition that is close, but does not satisfy the conditions. In order to get the best of both algorithms, the model first runs `fsolve()` with the 'trust-region' algorithm, and if the exitflag indicates that the solution is not satisfactory, `fsolve()` is called again, this time using 'levenberg-marquardt'. This seems to yield the best results, as when possible the solution is found quickly, and otherwise the more accurate method is used.

4 Visualization[L1147-1328]

The model uses the builtin `plot3` to visualize the vehicle's position in space. Each body is rendered as a tessellated surface. Each point is rendered as a point. The color of the point corresponds to whether or not that constraint is being satisfied. A red point means that the 2 points are further apart that the allowable tolerance (default is 0.01 mm), whereas a green point means the points are coincident.

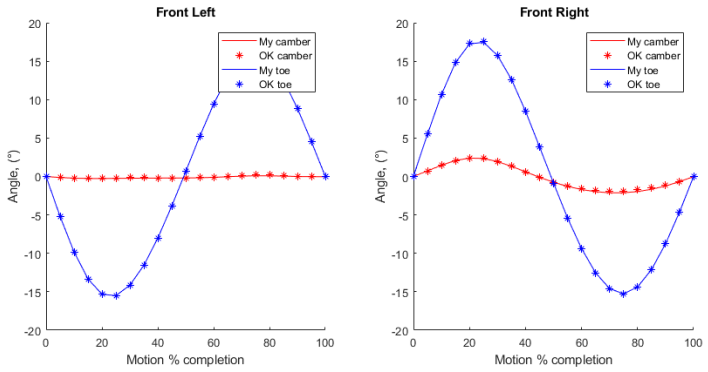
Figure 3: Visualization of the suspension



5 Validation

Optimum Kinematics was used to calculate the camber and toe of the front tires over an arbitrary motion of the vehicle. The exact same scenario was simulated with this model, and results compared. These can be seen in figure 4.[5] The maximum deviation between the two software packages is less than 0.02°. This is more than acceptable error for the Formula UBC Team's use case.

Figure 4: results comparison of optimum K and this model



Several issues were identified during this simulation.

1. The sign convention and units of the two softwares is not the same, meaning that some post processing needed to be done in order for the results to match. This is an area where the model could be improved in the future. Currently the "units" section of the user settings in the excel file is not used. An input/output processor function could easily be written to deal with multiple units and sign conventions, based on what the user needs.
2. The Speed of this model is significantly slower than the off the shelf software. This is to be expected, as Optimum G (the makers of Optimum Kinematics) have poured considerable resources into making it a fast, reliable software.

6 Conclusion

This model replicates the functionality of the off the shelf kinematic modelling package to a limited extend. All of the primary goals were acheived, but there is still a lot of fine tuning and improvements that will be made. The model is able to effectively display the position of suspension bodies given a set of input parameters. Because the suspension is defined by bodies connected through individual points, the user can easily alter the geometry and layout of the suspension by editing the associated excel file. The only goal not fully realized is 6. Although it is possible to calculate any desired parameter, the function must be written by the user after the simulation has run. In short, this model serves the same purpose of the off the shelf software, but allows for further flexibility and adjustments in the future.

[5]The code to recreate this plot is in Appendix B

Appendices

A Source Code

Source code removed for this version of the report

B Test Code

Code removed for this version of report