Notes on a python-based high performance automobile suspension dynamics simulator

Sanjay St. Lawrence

April 26, 2023

Computational Physics Final Project

Introduction. There are all kinds of phenomena in nature that modern physics can accurately describe and even predict the occurrences of, most of which can be modelled by what is arguably a physicist's greatest asset: oscillatory motion. Particularly, the dynamics of any object can be described as some sort of oscillatory motion, even when it may not be optimal or necessary to do so. Being able to describe the motion of any object simplistically (as an oscillator) is ideal for modeling and simulating the motion of said object under different conditions, and one such object that modern civilization is heavily supported by – used every day in many parts of the world, in fact – is the automobile.

A part of what makes an automobile so versatile is its suspension system, which is a spring-damper-mass system, physically. Suspension controls much of how cars behave when driven, and they are usually designed around the ideal suspension setup for some given operating conditions, which vary by the automobile's application (road racing, daily driving...), terrain, even the driver's driving style. As such, what many automobile manufacturers do is model the suspension so it can be simulated for conditions the automobile will be operating in to ensure the car will behave as intended, and if that cannot be achieved for a given setup, information from sims can be used to extrapolate an ideal suspension design. Thus, my proposal is to make my own automobile suspension simulation for mildly high-level analysis of a car's behavior. The basic process behind doing so is inputting parameters to constrain the suspension, modeling the geometry based on those constraints using vectors, and finally taking that geometry and simulating its dynamics using differential equations.

I. We start with the input parameters. Specific points of each suspension assembly must be provided in order for the program to model its geometry, and those points depend on the motion of the suspension. As such, the motion of the kind of suspension we are interested in modeling, *double-wishbone* suspension, can be described with circular paths generally given by

$$P = r(cos(\theta) + sin(\theta)),$$

which is how the two wishbones in the suspension move. Since every component of the suspension is connected rigidly to these wishbones, the rest of the assembly is constrained to this motion as well.

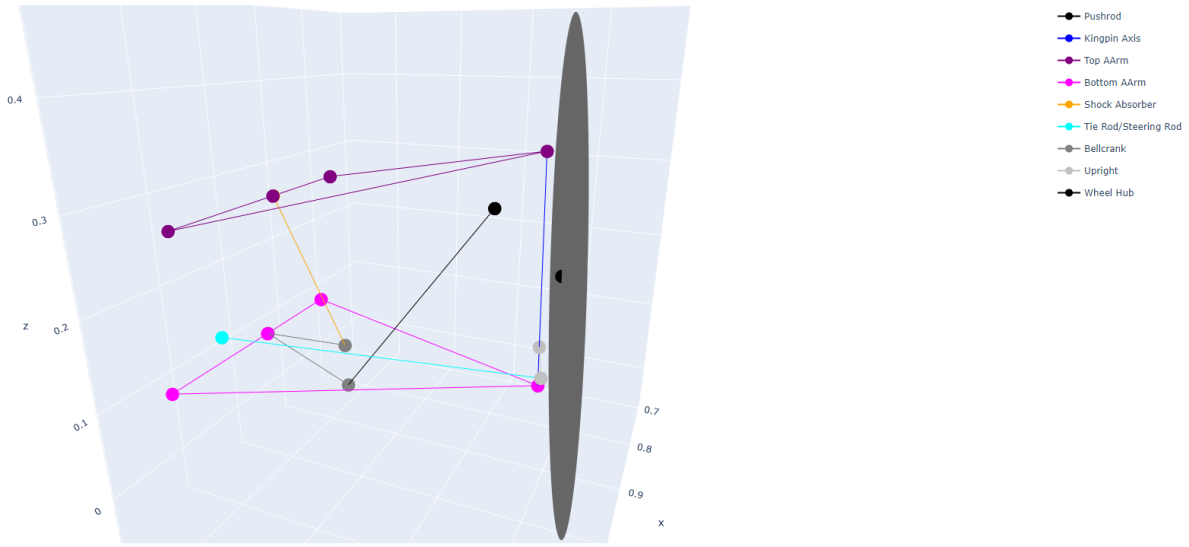

Figure 1: Example suspension model. All mentions of suspension components throughout the document can be referenced here, and otherwise will be explained.

For example, in the figure above of an example suspension model output by the suspension simulator, the motion of the two wishbones (Note the legend) is circular about their bases, which are along the x-axis. When parameterized, the path equation becomes

$$P_i = R(cos(t_i)\hat{j} + sin(t_i)\hat{k}),$$

for the ith point, P_i along the path of the wishbone, with length, R, in question. $\hat{j}$ and $\hat{k}$, for convenience, are labeled in the context of the wishbones in the figure, however they can be any pair of orthonormal vectors. Those vectors, along with another orthonormal vector defining the rotation axis, $\hat{i}$, form a set that is a basis to the space of the wishbones, centered at the point between the ends of $\hat{i}$. The vector between this point and the tip of the wishbone, the magnitude of which is R, the rotation axis, and the cross product of those two, can be used to form this basis; and, the points defining them - the ends of the rotation axis and the tip of the wishbone - must be specified to the simulator to model that geometry. This can be seen in SuspensionGeometry #6-#11, and they're used in SuspensionBellcrankSimulatorCLASSES2.bot_A_arm_arc() (Note that function_circle uses the general P expression and can be found in SuspensionBellcrankSimulatorCLASSES.function_circle()).

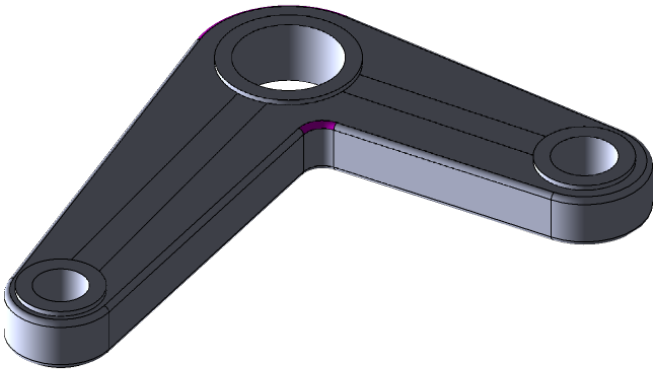

Figure 2: Example Bellcrank model. It's two arms are analogous to the gray line-segments in Fig. 1

The other suspension component that is modeled this way is something called a *bellcrank*, which is an apparatus that converts translational motion into rotational motion, and vice versa. In this case, either one of the wishbones in an assembly are connected to one end of the bellcrank, and the *shock absorber* is connected to the other end of the bellcrank (Note that the newly-mentioned components can be seen in Fig. 1). The engineering standard for any bellcrank's geometry is that it should be a simple prism, so its arms (that connect to other apparatuses) should be planar. Because of this, defining the bellcrank here is simpler than the wishbones, since we need only define a point the bellcrank is rotating about, and 2 more points - one for each arm. The simulator then makes two vectors with the center point being the tail of both vectors, and the cross product of them will be orthogonal to both, which we use as the rotation axis. The motion of the bellcrank isn't explicitly defined in the simulator, however, since it's motion is constrained to that of the wishbones, the shock absorber, and the *suspension rod*.

The suspension rod is what connects the wishbone and the bellcrank. When the suspension assembly moves, the wishbone that the suspension rod is connected to (Note that this is specified by SuspensionGeometry #12) moves the rod into the bellcrank[1], rotating it, and compresses the shock absorber. Obviously, the suspension rod is rigid, like everything else in this model. So, the way it is modeled here is with length conservation. Taking the point on the wishbone where the suspension rod is connected to it, P_{sr}, the bellcrank point on the suspension rod's side, p_{sr}, and an intended length of the suspension rod, L,

$$v = P_{sr} - p_{sr},\ ||v|| \equiv \sqrt{v \cdot v} = L' \Rightarrow L' - L = 0$$

The RHS of the arrow must hold for the entire motion of the suspension assembly. Therefore, the solution is optimization, via *scipy.optimize.root*. That is, we provide P_{isr}, which can be found in a similar fashion to the wishbone's path, $L = ||(P_{0sr} - p_{0sr})||$, and vectors defining the bellcrank to scipy.optimize.root to find angles that define all of p_{isr}.[2]

[1] This is where the specification of whether a suspension system is "pushrod" or "pullrod" is based. If the suspension rod's translational motion moves towards the center of the bellcrank, it is pushrod, and if it moves away, pullrod. There are specific implications to using either that involve rising rate suspension and falling rate suspension.

[2] Angles must be guessed instead of points because scipy.optimize.root is 1-D.

This process can be seen in SuspensionBellcrankSimulatorCLASSES.suspension_rod() initialization and SuspensionBellcrankSimulatorCLASSES.get_rod_points(), and the points provided for P_{0sr}, p_{0sr}, and the rest of the bellcrank are in SuspensionGeometry #0-#3, #5. It must also be noted that both basis sets and optimization are being used in a similar fashion to find the *steering rod* travel path, which uses SuspensionGeometry #13-#15 in SuspensionBellcrankSimulatorCLASSES2.steering_rod_turning().

The only constrained component that is not rigid is the shock absorber, obviously. Standard coil-over shock absorbers consist of a spring and a damper, which is what makes suspension systems spring-damper-mass systems, physically. Here, the compression of the shock absorber is what gives the suspension system stiffness, and it's displacement is given by

$$v = ||p_{isa} - p_{mount}|| - ||p_{0sa} - p_{mount}||,$$

where p_{isa} are the bellcrank points on the shock absorber's side, and p_{mount} is the point at which the shock absorber is mounted to the chassis of the automobile, which is specified by SuspensionGeometry #4. Note that p_{isa} is already known, as optimize root guesses angles, and

$$t_{isr} = t_{isa},$$

so we need only evaluate t_{isr} for a basis set describing that bellcrank arm's space using SuspensionBellcrankSimulatorCLASSES.function_circle(), which can be seen in SuspensionBellcrankSimulatorCLASSES.get_shock_points(theta_vector), where theta_vector is t_{isr}.
All of the geometry of each suspension component has been touched on. While there are a couple that were not mentioned, their constraints are similar to these concepts, so talking about them would be redundant.

II. For the dynamics of the system, we begin with Hooke's Law[3],

$$F = kx.$$

Now, Hooke's Law must be expanded upon, since we have a system that includes not only a spring, but also a damper, and driven force from an assortment of moments, inertia, body roll, etc. (For simplicity, I am assuming the car is at rest and force from the weight of the car is evenly split across all four wheels). In addition, in an effort to analyze the suspension system's response a little deeper, we can also consider the *tire* of the car as a spring-damper-mass:

$$F \equiv M_{total}\ddot{x_1} = k_1 x_1 + c_1 \dot{x_1} + F_{D1},$$

where F is the net force on the wheel attached for that given suspension assembly. The output of this, the motion of the wheel, is necessary to find the motion for the chassis in the shock absorber system:

$$F \equiv M_{chassis}\ddot{x_2} = kx_2 + c\dot{x_2} + F_{D2},$$

where F is the net force on the quadrant of the chassis for that given suspension assembly. The displacement of the tire is presumed to be linear (vertical) here, much like the shock absorber, for simplicity. The displacements of both require more context, as the motion of the car is actually only dependent on F_D, and changes in the *surface* the car is on for both the tire and the shock absorber.

[3]https://phys.org/news/2015-02-law.html, although seeing it should be unsurprising.

So,

$$x_1 = z_1 - U,\ x_2 = C(z_2 - z_1),$$

where U is the surface displacement (vertical), z_1 is the wheel's displacement, z_2 is the chassis' displacement, and C is called the *compression ratio* of the system, the obvious definition of which is

$$C = \frac{x_2}{(z_2 - z_1)},$$

the ratio of shock absorber displacement to suspension assembly displacement, which is always defined by the elevation of the lower wishbone in the chassis reference frame. This is necessary given that their motions are not similar whatsoever. Now, the equations for net wheel force and net chassis force become

$$M_{total}\ddot{x}_1 = k_1(z_1 - U) + c_1(\dot{z_1} - \dot{U}) + F_{D1}$$
$$M_{chassis}\ddot{x}_2 = k_2C(z_2 - z_1) + c_2C(\dot{z_2} - \dot{z_1}) + F_{D2}.$$

F_{D1} and F_{D2} will be assumed to be just the weight of the car, along with an oscillatory force in one example. Although, the transfer of force on the wheel to the force on the shock absorber is also not similar, much like the motion. So, a ratio is required for the driven force on the shock absorber as well. We can obtain this by taking the torque the shock absorber is exerting due to normal force on the bellcrank, and using that torque to find the force that the suspension rod is exerting on the bellcrank:

$$N_{sa}:\ \tau_{sa} = N_{sa} \times v_{bellcranksa} \rightarrow \tau_{sa} = F \times v_{bellcranksr}, (v_{bellcranksr} \times \tau_{sa})/||v_{bellcranksr}||^2 = F,$$

which gives the tangent (to the bellcrank arm) component of the suspension rod force. The use of this identity[4] can be seen in SuspensionBellcrankSimulatorCLASSES.get_force_magnitudes_arm(). Vector projections can be done to obtain the final force magnitude, which the methods following get_force_magnitudes_arm take care of. Once the vectors are obtained, the ratio of the vertical components of those vectors to N_{sa} can be found, which we will call Γ. So,

$$F_{D2} = \Gamma f_{D2}.$$

This system of differential equations is coupled, since $M_{chassis}\ddot{x_2} \equiv M_{chassis}\ddot{x_2}\ (z_1, \dot{z}_1)$. Plus, the ODEs are second-order, so finding a solution set analytically would be tricky, at best.
To find solutions to the system numerically, we can use *scipy.signal.lsim*[5] and *scipy.signal.lti*[6]. The lsim module simulates time-linear systems using a class object formed with the lti module, which expresses the ODE in a *state-space representation*[7].

[4] https://www.euclideanspace.com/maths/algebra/vectors/vecAlgebra/inverse/index.htm - $B = (A \times C)/||A||^2$
[5] https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.lsim.html
[6] https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.lti.html
[7] https://lpsa.swarthmore.edu/Representations/SysRepSS.html

State-space, for any given n^{th} order ODE, is given by

$$\dot{q}(t) = Aq(t) + Bu(t)$$
$$y(t) = Cq(t) + Du(t)$$

where A, B, C, D are matrices, $u(t)$ is the input vector, and $y(t)$ is the output vector, the specifics of which are in the documentation. For our case, the state vector, $q(t)$, is a 2x1 matrix, since we have two second-order ODEs:

$$q(t) = \begin{bmatrix} x \\ \dot{x} \end{bmatrix}, \dot{q}(t) = \begin{bmatrix} \dot{x} \\ \ddot{x} \end{bmatrix}.$$

So,

$$\dot{q}(t) = \begin{bmatrix} \dot{x} \\ \ddot{x} \end{bmatrix} = A \begin{bmatrix} x \\ \dot{x} \end{bmatrix} + B \begin{bmatrix} u \\ \dot{u} \\ \ddot{u} \end{bmatrix}, \text{ and}$$

$$\ddot{z_1} = \left(\tfrac{1}{M_{total}}\right) k_1(z_1 - U) + \left(\tfrac{1}{M_{total}}\right) c_1(\dot{z_1} - \dot{U}) + \left(\tfrac{1}{M_{total}}\right) F_{D1},$$
$$\ddot{z_2} = \left(\tfrac{1}{M_{chassis}}\right) k_2 C(z_2 - z_1) + \left(\tfrac{1}{M_{chassis}}\right) c_2 C(\dot{z_2} - \dot{z_1}) + \left(\tfrac{1}{M_{chassis}}\right) \Gamma f_{D2}$$

$$\longrightarrow \dot{q_1}(t) = \begin{bmatrix} \dot{z_1} \\ \ddot{z_1} \end{bmatrix} = A \begin{bmatrix} z_1 \\ \dot{z_1} \end{bmatrix} + B \begin{bmatrix} u \\ \dot{u} \\ \ddot{u} \end{bmatrix}$$

$$\equiv \begin{bmatrix} \dot{z_1} \\ \ddot{z_1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \frac{k_1}{M_{total}} & \frac{c_1}{M_{total}} \end{bmatrix} \begin{bmatrix} z_1 \\ \dot{z_1} \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ -\frac{k_1}{M_{total}} & -\frac{c_1}{M_{total}} & \frac{1}{M_{total}} \end{bmatrix} \begin{bmatrix} U \\ \dot{U} \\ F_{D1} \end{bmatrix},$$

$$\dot{q_2}(t) \equiv \begin{bmatrix} \dot{z_2} \\ \ddot{z_2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ C\frac{k_2}{M_{chassis}} & C\frac{c_2}{M_{chassis}} \end{bmatrix} \begin{bmatrix} z_2 \\ \dot{z_2} \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ -C\frac{k_2}{M_{chassis}} & -C\frac{c_2}{M_{chassis}} & \Gamma\frac{1}{M_{chassis}} \end{bmatrix} \begin{bmatrix} z_1 \\ \dot{z_1} \\ f_{D2} \end{bmatrix}, \text{ and}$$

$$y_1(t) = \begin{bmatrix} z_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} z_1 \\ \dot{z_1} \end{bmatrix},$$
$$y_2(t) = \begin{bmatrix} z_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} z_2 \\ \dot{z_2} \end{bmatrix}$$

with D_1, D_2 being the zero vector. Now that we have our system in state-space, the state, input, and output matrices can be fed to scipy.signal.lti, and that class object can be fed to scipy.signal.lsim along with some arbitrary time array to find the state vectors of the system. This entire process can be found in SuspensionBellcrankSimulatorCLASSES3; the entire script is dedicated towards setting this up. It must be noted that C and Γ are constants that are unique to each z_2, since the torque on the bellcrank, τ_{sa} is changing throughout the compression of the system due to the change in angle between the shock and the suspension rod to the bellcrank, on their respective sides. This is the reasoning behind the for loop in the dynamics script, since the matrices defining the shock absorber system must be recalculated.

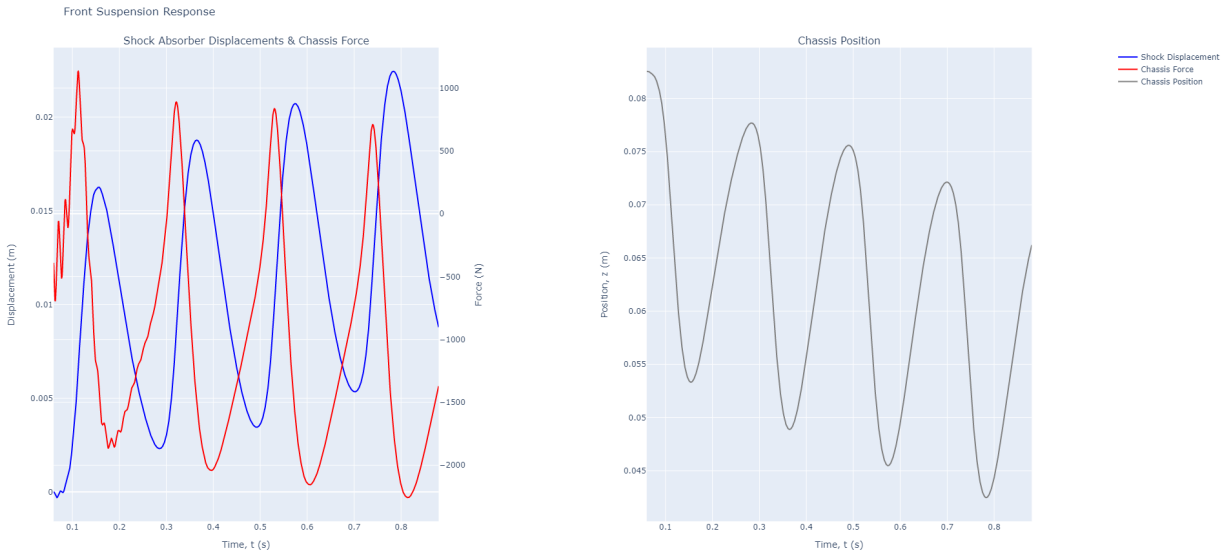

Figure 3: Motion of example suspension, front assembly

III. The big picture with doing this is to further optimize automobile suspension for a given application. Again, while most automobile companies are already doing simulations like this - at least, the serious ones - obtaining mid to low-level analysis in an easily digestible form would spread knowledge about oscillatory systems and hopefully uncover concepts that are *not* used as apart of industry standard. Some of these concepts, such as compression ratios, torque transfer through the bellcrank, and chassis dynamics, if understood well, would even be used to specify a suspension design *instead* of physical geometry in order to find ideal geometry that has those characteristics, which can then be designed around. The most significant reason why this script is closer to a high-level analysis and is far from being at that point, as-is, is because of chassis dynamics, since things like an inertia tensor for the chassis design, a continuous (or close to) weight distribution, and car performance (torque output) would be required to properly calculate F_{D1} and F_{D2}, so that the independent variables for the equations traces back to driver-input, which would be a pure simulation.

I felt that using vector calculus was an appropriate way to go about modeling the geometry, it felt the most familiar since our days of taking classical physics courses, and from things like question 1 on HW4. I also tried to round to less than 14 decimal places wherever I could to prevent rounding errors, and the interval of points that are used to calculate the circular motion, for example, is around 350, so as to simulate continuous, analytic motion. In fact, that is the one thing I would do differently if I had to improve the scripts somehow, is to use an interval of as little as 20 points or so, and interpolate the paths of the geometry model to include many more points.

Running the main script of the program, SuspensionBellcrankSimulatorPLOTS (although not aptly named...) outputs animations via plotly that can be played with and a graph of chassis position, shock displacement, and net chassis force, $M_{chassis}\ddot{z}_2$ (Fig. 3), for the animated motion. Example geometry - the same from Fig. 1 - is provided and does not need to be calculated, although that can be done by uncommenting line 3, as it says in the script.

Bibliography. Erik Cheever, Swarthmore College. State Space Representations of Linear Physical Systems, https://lpsa.swarthmore.edu/Representations/SysRepSS.html.

"Maths - Inverse Vector." Maths - Inverse Vector - Martin Baker, https://www.euclideanspace.com/maths/algebra/vectors/vecAlgebra/inverse/index.htm.

"Scipy.signal.lsim." Scipy.signal.lsim - SciPy v1.10.1 Manual, https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.lsim.html.

"Scipy.signal.lti." Scipy.signal.lti - SciPy v1.10.1 Manual, https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.lti.html.

Williams, Matt. "What Is Hooke's Law?" Phys.org, Phys.org, 16 Feb. 2015, https://phys.org/news/2015-02-law.html.