Sam Brennan
December 3th, 2019
MCEN 4123
Dr. Castro

Active Damping-Vibrational Analysis Project

Introduction

Vehicle suspension systems are essential for maintaining a smooth ride and keeping the wheels in contact with the road. Most vehicles have constant damping parameters for the front and rear struts. Some vehicles have an active damping suspension system where the damping coefficients of the struts are adjusted based on local acceleration data. Active damping suspension systems are used in vehicles to reduce body roll, pitch motion, yaw motion, and displacement oscillations for maintaining a level body under different driving conditions. Car manufacturers state that this creates a safer and more enjoyable experience for the passengers.

In this report, an active damping feedback loop was created and integrated into an open source python simulation to study the effects of active damping suspension systems **[1]**. Active damping has uses beyond the automotive industry. Active damping can benefit many different control systems prone to oscillation **[2]**.

After investigation, it was determined that current active damping suspension systems are luxury features and should not be used in lower end vehicles. The equipment needed to achieve active damping is expensive and prone to failure. In other words, the cost and maintenance required for active damping suspension systems outweighs the marketability and benefits of these systems. Active suspension systems could have a use in racing applications to maximum efficiency, dampen oscillations, and maximum road-tire contact.

Theory and Background

The Euler time step method was used in the open source simulation to compute the positions, velocities, and accelerations of various points on the vehicle. The vehicle is modeled as a four-degree or freedom system where the front and rear suspensions are modeled as equivalent spring constants and equivalent damping constants. The small angle approximation is utilized for the pitch angle and the struts are assumed to be perfectly vertical for the entire simulation. The input to this simulation is the road and two points must be described (the front and rear road-tire contact points). Both wheels are modeled as a spring and dashpot. The center of mass of the chassis is assumed to be equivalent to the center of mass of the system since the mass of the chassis is significantly greater than the mass of the wheels. There is only one drive force acting on the front wheel since the modeled vehicle is front wheel drive.

Figure 1. below shows the vehicle model used in the simulation. **Figure 2.** below shows the model used for the front and rear wheels in the simulation. **Figure 3.** below shows the system of differential equations used in the simulation. **Figure 1, Figure 2,** and **Figure 3** were supplied with the original python simulation package **[1]**.

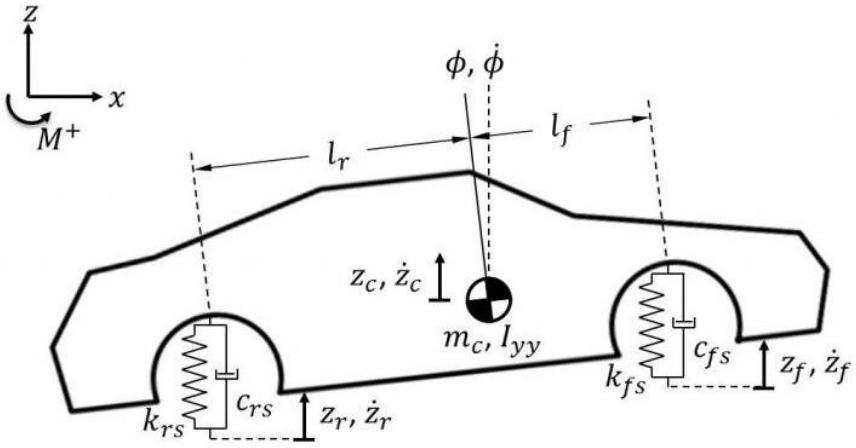
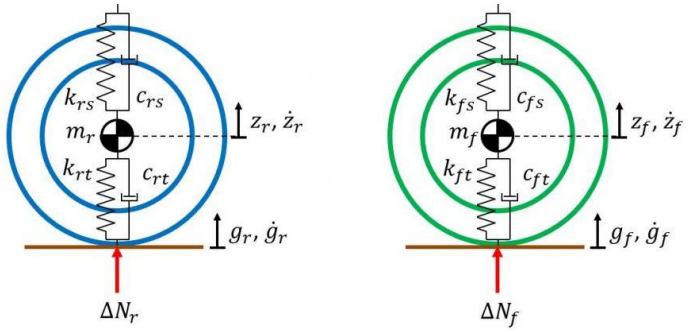


Figure 1. Vehicle and Suspension Model



$$m_f\ddot{z}_f = (l_f\phi + z_c - z_f)k_{fs} + (l_f\dot{\phi} + \dot{z}_c - \dot{z}_f)c_{fs} - (z_f - g_f)k_{ft} - (\dot{z}_f - \dot{g}_f)c_{ft} + \Delta N_f$$

$$m_r\ddot{z}_r = -(l_r\phi - z_c + z_r)k_{rs} - (l_r\dot{\phi} - \dot{z}_c + \dot{z}_r)c_{rs} - (z_r - g_r)k_{rt} - (\dot{z}_r - \dot{g}_r)c_{rt} + \Delta N_r$$

Figure 2. Front and Rear Wheel Models

$$\ddot{z}_c = \frac{1}{m_c}\left[-(l_f\phi + z_c - z_f)k_{fs} + (l_r\phi - z_c + z_r)k_{rs} - (l_f\dot{\phi} + \dot{z}_c - \dot{z}_f)c_{fs} + (l_r\dot{\phi} - \dot{z}_c + \dot{z}_r)c_{rs}\right]$$

$$\ddot{\phi} = \frac{1}{I_{yy}}\left[-l_f(l_f\phi + z_c - z_f)k_{fs} - l_r(l_r\phi - z_c + z_f)k_{rs} - l_f(l_f\dot{\phi} + \dot{z}_c - \dot{z}_r)c_{fs} - l_r(l_r\dot{\phi} - \dot{z}_c + \dot{z}_r)c_{rs}\right]$$

$$\ddot{z}_f = \frac{1}{m_f}\left[(l_f\phi + z_c - z_f)k_{fs} + (l_f\dot{\phi} + \dot{z}_c - \dot{z}_f)c_{fs} - (z_f - g_f)k_{ft} - (\dot{z}_f - \dot{g}_f)c_{ft} + \Delta N_f\right]$$

$$\ddot{z}_r = \frac{1}{m_r}\left[-(l_r\phi - z_c + z_r)k_{rs} - (l_r\dot{\phi} - \dot{z}_c + \dot{z}_r)c_{rs} - (z_r - g_r)k_{rt} - (\dot{z}_r - \dot{g}_r)c_{rt} + \Delta N_r\right]$$

Figure 3. System of differential equations

Required Equipment:

Active damping is achieved by taking an instantaneous acceleration reading and sending it through a PID controller to return a modified damping coefficient (or voltage). Several pieces of equipment are required to achieve active damping on a vehicle. The equipment includes voltage controlled variable damping struts, accelerometers, and a PID controller (generally in the ECU). Several vehicle manufacturers have created active damping suspension systems, including Toyota, BMW, Lamborghini, and more **[3]**.

Method

An open source python simulation was used to model a moving vehicle. An active damping feedback loop was created and integrated into the simulation, making two simulations. The first simulation had constant damping parameters and the other simulation had the active damping feedback loop.

In the constant damping simulation, the damping parameters of the front and rear suspension are defined in the "Car" class. This results in a constant damping matrix that the "update_state" method uses to calculate the accelerations.

In the active damping simulation, the damping parameters of the front and rear suspension are adjusted every time step in the "update_state" method in "car.py". Then, the damping matrix is recalculated and used to calculate the accelerations in the "update_state" method. The damping coefficients are adjusted based on the previous time step acceleration. If the previous acceleration (positive or negative) is outside of a range, then the damping constant is increased unless a maximum value has been reached. If the previous acceleration is inside of the range, then the damping constant is decreased unless a minimum value has been reached. Most of the changes to the simulation code are in **Appendix A**. The feedback loop that was integrated into this simulation was not a true PID controller. Rather than modifying the damping constants proportionally, with respect to the integral, and with respect to the derivative, a constant value was added or subtracted from the front or rear damping coefficient. Two switch statements (multiple ifs) were used to achieve this. The acceleration comparison range modifies the sensitivity of the feedback loop. A wider range represents a less sensitive feedback loop. The C_RS and C_FS variables were the only variables to be modified during the simulation run time. A vector of the front and rear wheel well accelerations was captured and manipulated with OCTAVE to produce **Figure 4a, 4b,** and **4c**. A vector of the front and rear wheel well positions was captured and manipulated with OCTAVE to produce **Figure 5a** and **5b**. The modified damping constant values were also added to the output video.

Results and Analysis

Video results from the active damping simulation are very similar to results from the constant damping simulation. So, it was necessary to create two simulations where the active damping could be measured easily. The braking simulation below was used to measure the effect of the active damping. In the braking simulation, the road is completely flat and the car reaches a top speed of 60-m/s. Then, the brakes are applied at 30-seconds. This creates an ideal situation for the active damping feedback loop to modify the front damping constant when the large negative acceleration is sensed.

The braking simulation results:

Figure 4a. below shows the y-acceleration of the front wheel well during the breaking simulation right after the breaks are applied for the active damping model. **Figure 4b.** below shows the y-acceleration of the front wheel well during the breaking simulation right after the breaks are applied for the constant damping model. The axis scales for **Figure 4a.** and **Figure 4b.** are the same for easy comparison. **Figure 4c.** below shows the y-acceleration of the front wheel well for the active damping model after the feedback loop sensitivity was adjusted. **Figure 4c.** has considerably less overcorrecting from about 31-seconds to 36-seconds.

The variances between these figures are subtle, but the active damping clearly has a pronounced effect. Right when the breaks are applied at 30-seconds, a large negative acceleration is experienced by both models. This occurs in the active damping model because the active damping is one time-step behind the measured acceleration value. However, the positive (rebound) acceleration experienced by the active damping model is significantly lower than the positive (rebound) acceleration experienced by the constant damping model. The lower rebound acceleration experienced by the active damping model is the result of the feedback loop telling the front suspension to increase the damping constant when the large negative acceleration was measured. In other words, the active damping quickly reduces the amplitudes of subsequent acceleration oscillations. **Figure 5a** shows that the position of the front wheel well dampens out much quicker for the active damping model compared to the constant damping model shown in **Figure 5b.**

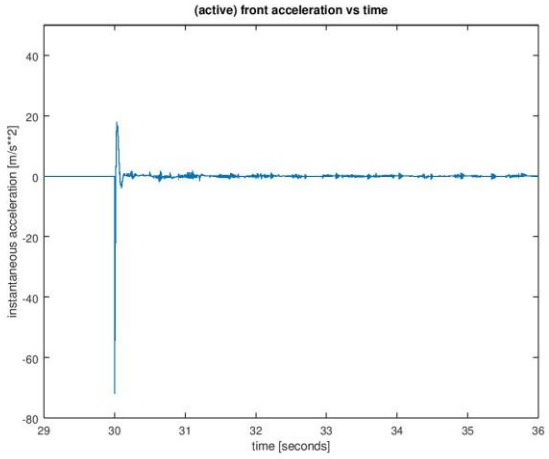


Figure 4a. Front Wheel Well Acceleration (Active Damping)

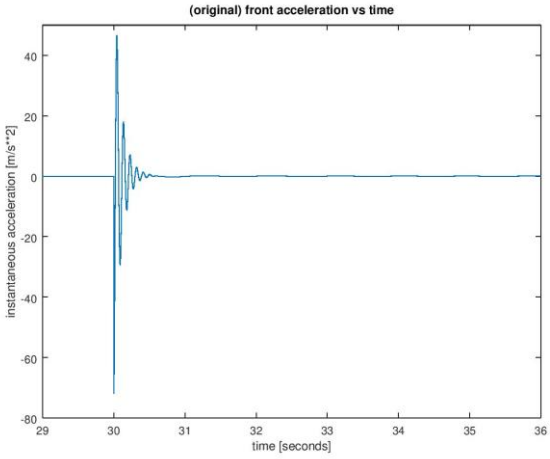


Figure 4b. Front Wheel Well Acceleration (Constant Damping)

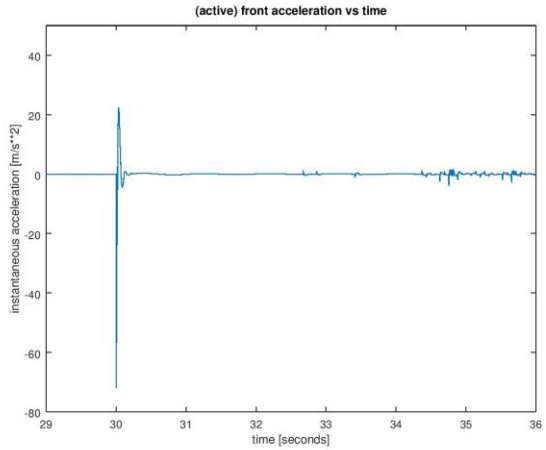


Figure 4c. Front Wheel Well Acceleration (Less Sensitivity)

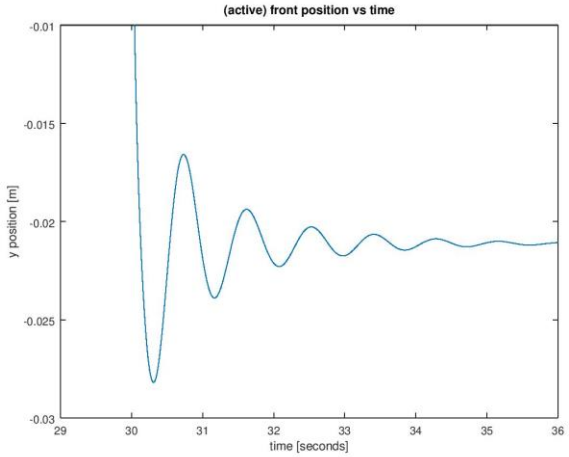


Figure 5a. Front Wheel Well Position (Active Damping)

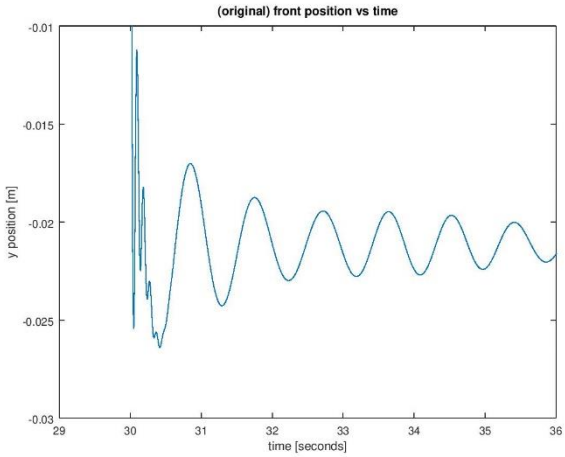


Figure 5b. Front Wheel Well Position (Constant Damping)

The redesign of the “update_state” method, where the damping matrix is recalculated every time step, had no noticeable effects on the time for the simulation to run.

Recommendations

The complexity of this code could be reduced significantly by using the state model rather than the Euler time step numerical method. This would also allow for an easier implementation of the active damping control loop. This would also eliminate the need for the road class since the input data would be grabbed from an external file or array.

There are a few hypothetical actions that could be taken to make the active damping feedback loop in this simulation significantly more effective. Rather than having the switch statements adding or subtracting a set value from the damping constants, a true PID controller could be implemented. To do this, a feedback loop would need to be created in Simulink and compiled into a C++ executable. Then, in the “update_state” method, a new function could be called that uses the subprocess python library to send the acceleration reading to stdin and receive the modified damping constants in stdout from the C++ executable. This redesign would eliminate the need for the desired acceleration range and switch statements and would potentially reduce the overcorrecting issues. This redesign would also reduce the complexity of programing a true PID controller. Another simpler action that could be taken is to use the simple-pid python library and redesign the “update_state” method with the new library. To do this, the desired acceleration and previous time step acceleration would be passed into a PID function and the function would return the modified damping value before the damping matrix and accelerations are computed.

Both of these hypothetical actions would slow down the rate of change of the damping constants when the error between accelerations is closer to zero and reduce the overcorrecting issue. To model a true active suspension system, the electrical and mechanical delay in the dampers, accelerometers, and ECU would also need to be taken into account.

References

[1] Nrsyed. “Nrsyed/Half-Car.” *GitHub*, 11 Aug. 2019, https://github.com/nrsyed/half-car.

[1] Syed, Najam. “Numerical Approach to Studying Vehicle Dynamics with a Half-Car Suspension Model.” *Najam R Syed*, 11 May 2019, https://nrsyed.com/2018/01/07/numerical-approach-to-studying-vehicle-dynamics-with-a-half-car-suspension-model/.

[2] “Active Damping.” *Active Damping*, https://deltamotion.com/support/webhelp/rmctools/Controller_Features/Control_Modes/Advanced_Control/Active_Damping.htm.

[3] “How Adaptive Suspension Works (In Depth). ”https://www.youtube.com/watch?v=weLfP22sq7g

Appendices

Appendix A: Modifications to Simulation Code

“simulation.py”

```
## GLOBAL VARIABLES
elapsed_time = 0
iteration = 0
i = 0
time = 0;
c_fs_data = [0]
c_rs_data = [0]
time_vector = [0]
fs_accel_vector = [0]
rs_accel_vector = [0]
start_time = datetime.now()
accel_vector = [0,0,0,0]
c_rs_vector = [0]
c_fs_vector = [0]
fs_pos_vector = [0]
rs_pos_vector = [0]
```

PASSING VARIABLES TO “update_state”

```
i += 1
car.update_state(time_step, i, accel_vector, c_fs_vector, c_rs_vector, start_time, time, time_vector, fs_accel_vector, rs_accel_vector, c_fs_data, c_rs_data, fs_pos_vector, rs_pos_vector)
```

“car.py”

```
def update_state(self, time_step, i, accel_vector, c_fs_vector, c_rs_vector, start_time, time, time_vector, fs_accel_vector, rs_accel_vector, c_fs_data, c_rs_data, fs_pos_vector, rs_pos_vector):
    """
    TODO
    """
    print(i)
    if i == 88000:
        print(datetime.now() - start_time)


    # Active Damping. The front and rear damping constants and damping matrix are updated every time step.
```

```
if i >= 2:
    # Mass and Inertia Properties.
    m_c = 1600
    m_f = 23
    m_r = m_f
    I_zz = 2500
    m = m_c + m_f + m_r
    mass_vector = np.array([m_c, I_zz, m_f, m_r])

    # Constant Damping Properties
    c_ft = 20
    c_rt = 20

    # length properties
    wheelbase = 2.74
    l_f = 0.4 * wheelbase
    l_r = 0.6 * wheelbase

    # Active Damping Feedback and Desired Output
    feedback_accel = accel_vector[i+2]
    feedback_accel_front = feedback_accel[2]
    feedback_accel_rear = feedback_accel[3]

    # Control the sensitivity of the active damping
    desired_accel = 0
    lower = desired_accel - 0.08
    upper = desired_accel + 0.08
    #desired_accel_lower = desired_accel - 0.01
    #desired_accel_upper = desired_accel + 0.01

    ## ACTIVE DAMPING LOGIC
    c_fs = c_fs_vector[i-1]
    c_rs = c_rs_vector[i-1]

    ## FRONT SUSPENSION
    if feedback_accel_front > upper:
        if c_fs == 2000:
            c_fs = c_fs
        else:
            c_fs += 100
    elif feedback_accel_front < lower:
        if c_fs == 2000:
            c_fs = c_fs
        else:
            c_fs += 100
    if feedback_accel_front >= lower and feedback_accel_front <= upper:
```

```
    if c_fs == 1000:
        c_fs = c_fs
    else:
        c_fs -= 100

## REAR SUSPENSION
if feedback_accel_rear > upper:
    if c_rs == 2000:
        c_rs == c_rs
    else:
        c_rs += 100
elif feedback_accel_rear < lower:
    if c_rs == 2000:
        c_rs = c_rs
    else:
        c_rs += 100
if feedback_accel_rear >= lower and feedback_accel_rear <= upper:
    if c_rs == 1000:
        c_rs = c_rs
    else:
        c_rs -= 100
```

CAPTURING DATA

```
fs_pos_vector.append(self.state["position"][2,0])
rs_pos_vector.append(self.state["position"][3,0])

time_vector.append(time)
fs_accel_vector.append(feedback_accel_front)
rs_accel_vector.append(feedback_accel_rear)
c_fs_data.append(c_fs)
c_rs_data.append(c_rs)

if i == 88000:
    with open('active_fs_accel_data.csv', 'w') as csvfile:
        fieldnames = ['time', 'accel']
        writer_fs = csv.DictWriter(csvfile, fieldnames=fieldnames)
        for j in range(1,len(time_vector)):
            writer_fs.writerow({'time': time_vector[j], 'accel': float(fs_accel_vector[j])})
    with open('active_fs_pos_data.csv', 'w') as csvfile:
        fieldnames = ['time', 'ypos']
        writer_fs = csv.DictWriter(csvfile, fieldnames=fieldnames)
        for j in range(1,len(time_vector)):
            writer_fs.writerow({'time': time_vector[j], 'ypos': float(fs_pos_vector[j])})
```

UPDATING CAR.STATE WITH NEW DAMPING CONSTANTS

```
    self.state["c_fs"] = c_fs
    self.state["c_rs"] = c_rs
    c_fs_vector.append(c_fs)
    c_rs_vector.append(c_rs)
    #print(c_fs_vector)
    #print(accel_vector[i-1])
    damping_matrix = np.array([
      [-(c_fs + c_rs), l_r * c_rs - l_f * c_fs, c_fs, c_rs],
      [-(l_f * c_fs - l_r * c_rs), -(l_f**2 * c_fs + l_r**2 * c_rs),
        l_f * c_fs, -l_r * c_rs],
      [c_fs, l_f * c_fs, -(c_fs + c_ft), 0],
      [c_rs, -l_r * c_rs, 0, -(c_rs + c_rt)] ])
    damping_matrix = damping_matrix / mass_vector[:, None]

    #acceleration for remaining time-steps
    accel = (
      (stiffness_matrix @ position)
      + (damping_matrix @ velocity)
      + (road_stiffness_matrix @ road_position)
      + (road_damping_matrix @ road_velocity)
      + (self.normal_force_vector)
      )
    #print(accel_vector[i+2])
    c_fs_vector.append(c_fs)
    c_rs_vector.append(c_rs)
    accel_vector.append(accel)
    #print(accel_vector[i])
```

##”plot_sim.py”

POSITIONING C VALUES IN VIDEO OUTPUT

```
  annotations["C_FrontSus"] = ax.annotate(
    "", xy=(0.5, 0.06), xycoords="axes fraction"
  )
  annotations["C_RearSus"] = ax.annotate(
    "", xy=(0.5, 0.02), xycoords="axes fraction"
  )
```

UPDATING ANNOTATIONS

```
  self.annotations["C_FrontSus"].set_text(
    "C_FrontSus = {:.1f} Ns/m".format(car.state["c_fs"])
  )
  self.annotations["C_RearSus"].set_text(
    "C_RearSus = {:.2f} Ns/m".format(car.state["c_rs"])
  )
```

##”road.py”

CHANGING ROAD PROPERTIES FOR BRAKING SIMULATION

```
class Road:
    def __init__(
        self, length, resolution=300, mode="bump", amplitude=0,
        frequency=0.05, x_min=None
    ):
```