Causal Block Diagrams Assignment

October 21, 2015

1 Discrete Time Causal Block Diagrams

Discrete time Causal Block Diagrams (CBDs), as the name implies, are used to describe processes that evolve intermittently, or that can be abstract as such.

The purpose of this assignment is that you build and test a Discrete time CBD simulator by completing the code already provided as a starting point.

1.1 Tasks

1.1.1 Discrete Time CBD simulator

Implement a CBD simulator for Discrete time CBDs by filling in what is missing in the provided source files. Take the time to go through the folder structure and the code to understand the general architecture of the simulator. The python script that you have to fill in is the `CBD.py`, under the `Source` folder.

The main classes `CBD.py` are:

- `BaseBlock` class – An abstract class that represents a Block in a CBD. Specific blocks such as `ConstantBlock` extend this class.
- Specific block classes – A set of classes, each extending `BaseBlock` and representing a specific Block in a CBD.
- `Clock` – A class whose instance is used to keep track of the simulated time in a simulation.
- `CBD` – A class representing an entire CBD diagram.
- `DepGraph` – Represents a dependency graph. This dependency graph is used to determine the order of evaluation of the blocks at each simulation step.
- `DepNode` – Represents a node of the dependency graph.

The file `EvenNumbersCBD.py` shows how to build, draw, simulate and plot the hierarchical CBD shown in Figure 1, using the bokeh plotting library. You are not required to use any specific plotting library.

For this task, look for the `TO IMPLEMENT` comments and implement the following functions:

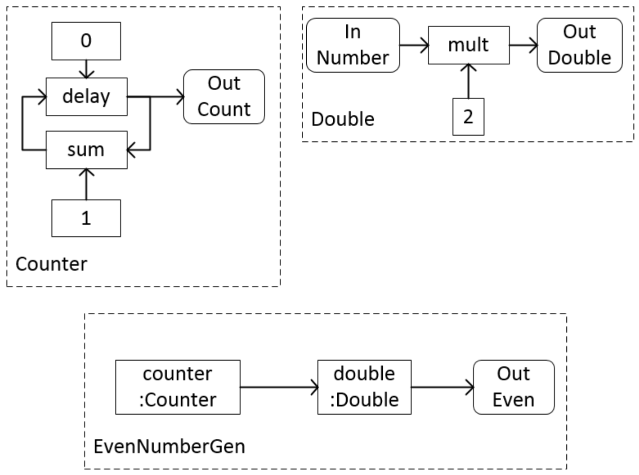

Figure 1: Example of a hierarchical CBD (EvenNumberGen) that computes the sequence of even numbers and makes use of other two CBDs (Counter and Double).

1. `getDependencies` function of the `BaseBlock`. This function should return the list of blocks on which the current block depends on. It will be helpful when building the dependency graph.

2. `compute` function of several blocks. These functions should compute the output value of the block at the iteration `curIteration`. Look at the code of the `BaseBlock` class to understand how the inputs and outputs are represented.

3. `getDependencies` function of the `DelayBlock` class. This function should override the `getDependencies` function of the `BaseBlock` class because delay blocks have different dependencies depending on the current iteration.

4. `__createDepGraph` of the `CBD` class. This function should create a dependency graph for the current CBD, at the iteration `curIteration`.

5. `__isLinear` of the `CBD` class. This function should detect whether a strong component - representing an algebraic loop - is a linear algebraic loop or not.

6. Do not worry about implementing the `DerivatorBlock` and `IntegratorBlock` classes. For now.

1.1.2 Test the CBD Simulator

Use the supplied unit tests to test your implementation and learn how to build your CBD models by instantiating the blocks and connecting them together. Then build your own CBD model that must include the following features:

- At least one inner CBD.
- At least one linear algebraic loop.
- At least one Delay block.

Simulate your model for some time and plot the results. You can use the matplotlib python library[1] or dump the results and copy/paste them to a spreadsheet program. Here is an example of how to plot the results in a single window: `http://msdl.cs.mcgill.ca/people/hv/teaching/MoSIS/201415/assignments/CBD/plot.py`

1.1.3 Document

Write a small report containing the tasks that you have completed and how you have completed them. Include all the plots that you have made from the simulations. The CBD model that you have built in the previous tasks should be displayed graphically in the report. You can use the `draw` function, defined in the `CBDDraw.py` file to export your model as a DOT (Graph Description Language)[2] model. You can then use the generated DOT file to draw the graph with the GraphViz tool or online: `http://graphs.grevian.org/`.

2 Continuous Time Causal Block Diagrams (CBDs)

CBDs can be used to model causal continuous-time systems [4, 1, 2]. It is implemented in tools such as Simulink® to model systems that evolve continuously over time, or that can be abstracted as such.

In this assignment you will continue were you left with the simulator that you have built in the previous assignment. You will extend this simulator to allow it to simulate Continuous Time CBDs.

2.1 Tasks

2.1.1 Integral and Derivative Blocks

Complete the implementation of the `IntegratorBlock` and `DerivatorBlock` classes. The implementation should be made using other blocks, as was taught in the theory lectures.

[1] `http://matplotlib.org/1.4.0/faq/installing_faq.html`
[2] `https://en.wikipedia.org/wiki/DOT_(graph_description_language)`

2.1.2 Test Approximations

A simple harmonic oscillator[3] with no friction exhibits a behavior that can be modelled by the following Second Order Ordinary Differential Equation:

$$\frac{d^2x}{dt^2} = -x$$

where $x(0) = 0$ and $\frac{dx}{dt}(0) = 1$.

For this task you need to:

a) Build a Continuous Time CBD that models the harmonic oscillator using integral blocks – let us call it CBD A;

b) Simulate CBD A and plot the value of x over time.

c) Build a Continuous Time CBD that models the harmonic oscillator using derivative blocks – let us call it CBD B;

d) Simulate CBD B and plot the value of x over time.

2.1.3 Measure Error

This task builds on the previous one. You need to:

a) For CBD A, include blocks to measure the accumulated error being made in the value of x against the real (analytical) solution of the harmonic oscillator $sin(t)$. The accumulated error is then given by $e(t) = \int |sin(t) - x(t)|dt$. You will need to implement your own `SinBlock` to compute $sin(t)$ and the absolute value of its input. (Hint: use the `self.getClock().getTime()` instruction to get the current time for the `SinBlock`).

b) Plot the error being made in the simulation of CBD A with at least two different step sizes, for instance, with step size 0.1 and 0.001.

c) Repeat the previous two tasks for the CBD B.

2.1.4 Document

Write about the tasks that you have accomplished and document your findings. Include drawings of the CBDs and all the plots. Discuss the error measured in task “Measure Error”. What happens to the error when the step size decreases? And if, for the same time step, you compare the measured error in CBD A with the measured error in the CBD B which one seems to yield a smaller error in a long running simulation?

[3] `https://en.wikipedia.org/wiki/Harmonic_oscillator`

3 Controller Design and Tuning

In this assignment, you will learn the main challenges involved in creating a simple controller for a complex continuous system. Furthermore, you will learn about abstraction and its important role in modeling. The CBD simulator that you built in the first part of this assignment will be required to simulate all models created in this part.

3.1 Background

A control system is a system whose purpose is to command, direct or regulate, itself or another system[4].

Control systems abound. A human is a control system. Suppose that you want to grab an object. Your eyes allow your brain to estimate the distance between the object and your hand; your brain commands the muscles in your arm and hand to contract so as to move you hand closer to the object; the closer your hand is (distance given by the eyes) the slower your arm and hand move to grab the object. In this example, your brain is acting as the controller, the muscles in your hand and arm are actuators and your eyes are sensors.

There are two kinds of control systems: closed-loop and open-loop systems. The detailed distinction is not important here because we will be focusing in closed-loop control systems, but keep in mind that your brain would act as a open control system in the previous example, if you were to grab the object with your eyes closed.

3.1.1 Closed-Loop Control Systems

As an example of a closed-loop control system, we will build a cruise control system for a car. The cruise control system is responsible for maintaining the car at an ideal velocity v_i. To achieve this, the controller continuously measures the car's velocity v, compares it with the ideal velocity v_i and accelerates or decelerates the car accordingly.

In order to be able to simulate the control system, and because we do not have access to a real car, we need a model of a car. The car model, in control systems' jargon is called the Plant. For this, let us consider a car with mass M, under the influence of some force $F_{traction}$, to be controlled by the control system, and air drag F_{drag}. The $F_{traction}$ force is what makes the car accelerate or decelerate (if $F_{traction}$ is negative) so you can see it as the acceleration force. Disregarding the gravity force, we can abstract the car as shown in Figure 2. The traction force $F_{traction}$ pushes the car forward and the drag force F_{drag} resists the traction force by pushing the car in the opposite direction. The resulting force F_{res} is the sum of these two forces and it gives the total acceleration of the car:

$$F_{res} = F_{traction} + F_{drag} \tag{1}$$

[4] `https://en.wikipedia.org/wiki/Control_system`

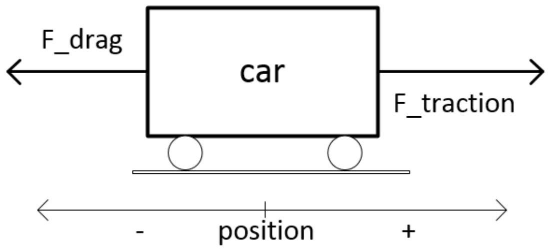

Figure 2: Car system under drag and traction forces.

The drag force is given by the equation[5]:

$$F_{drag} = -\frac{1}{2} \cdot p \cdot v^2 \cdot C_D \cdot A \tag{2}$$

where p is the air (fluid) density, v the speed of the car relative to the air, A the cross sectional area and C_D the drag coefficient. The drag force is negative because we are assuming that any motion forward, to the right in Figure 2, is positive and any motion to the left is negative. The drag force pushes the card to the left, hence it is negative.

Newton's second law of motion states that the resulting force on an object is equal to its mass times its acceleration:

$$F_{res} = M \cdot a \tag{3}$$

Merging equations 1 and 2 yields:

$$M \cdot a = F_{traction} - \frac{1}{2} \cdot p \cdot v^2 \cdot C_D \cdot A \tag{4}$$

At seal level and $15°C$, the air density p is approximately $1.225kg/m^3$ [6]. For a Lamborghini Diablo, the product $C_D \cdot A$ is $0.573m^2$ [7] and its mass M is $1576kg$ [8]. Notice that these constants are obtained empirically.

In Equation 4 the acceleration represents the rate of change of the velocity, so:

$$a = \frac{dv}{dt} \tag{5}$$

Using Equation 5 to replace a in Equation 4, plugging the Lamborghini Diablo and air density parameters, and rewriting in terms of the rate of change of velocity yields the following first order Ordinary Differential Equation:

$$\frac{dv}{dt} = \frac{1}{1576}\left(F_{traction} - \frac{1}{2} \cdot 1.225 \cdot v^2 \cdot 0.573\right) \;\; ; \;\; v(0) = 0 \tag{6}$$

[5] https://en.wikipedia.org/wiki/Drag_(physics)
[6] https://en.wikipedia.org/wiki/Density_of_air
[7] https://en.wikipedia.org/wiki/Automobile_drag_coefficient
[8] https://en.wikipedia.org/wiki/Lamborghini_Diablo

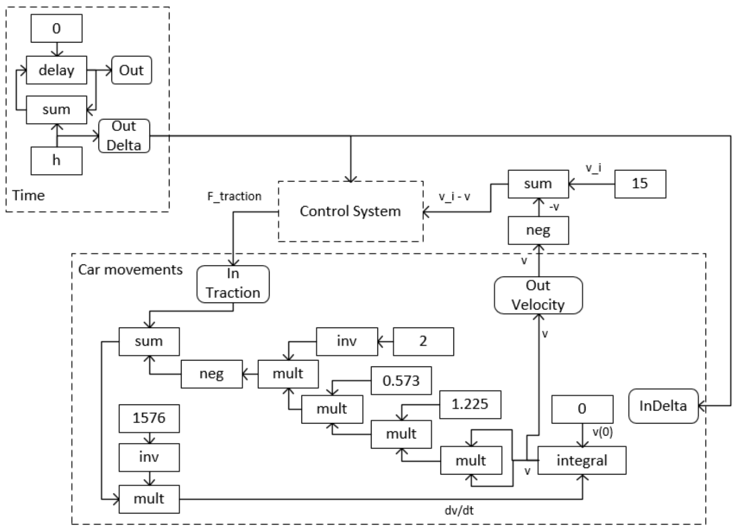

Figure 3: Car with the cruise control system. The ideal velocity is 15 m/s. The control system should ensure that the car is kept at that velocity, despite the air drag.

The control system decides the value of the $F_{traction}$ based upon the velocity v of the car and some ideal velocity v_i.

Coupling the control system with the car as one CBD model, we obtain the continuous time CBD shown in Figure 3.

Without going into the internals of the control system, you can already guess what its output – the traction force $F_{traction}$ – should be, based on its input – the difference between the ideal velocity v_i and the actual car's velocity v: if the difference $v_i - v$ is small, the car is close to the ideal velocity and the output $F_{traction}$ should be small. Otherwise, if $v_i - v$ is large, then $F_{traction}$ should be large too in order to speed the rate at which the car reaches the ideal velocity.

There are many possibilities to realize the cruise control system of the car. The simplest one is the so-called Bang-bang control: we set two thresholds d_{min} and d_{max}; if the difference $v_i - v$ exceeds d_{max}, then the controller sets $F_{traction}$ to some positive value c; otherwise, if $v_i - v$ gets smaller than d_{min}, then the controller sets $F_{traction}$ to 0. The real world effect on this is that the passenger in the car would feel these sudden accelerations whenever the velocity drops below a certain level. Hence the name "Bang-bang".

A slightly more sophisticated controller is the Proportional-Integral-Derivative (PID) [9] controller: instead of setting thresholds, this controller relates the value of the $F_{traction}$ to the difference $v_i - v$. This relation is established by a sum of three different controllers:

[9] `https://en.wikipedia.org/wiki/PID_controller`

Proportional Controller – For some constant value K_p, this controller sets $F_{traction} = K_p \cdot (v_i - v)$;

Integral Controller – For some constant value K_i, this controller sets $F_{traction} = K_i \cdot \int (v_i - v)dt$;

Derivative Controller – For some constant value K_d, this controller sets $F_{traction} = K_d \cdot \frac{d(v_i - v)}{dt}$;

In total, the PID Controller sets

$$F_{traction} = K_p \cdot (v_i - v) + K_i \cdot \int (v_i - v)dt + K_d \cdot \frac{d(v_i - v)}{dt}$$

Intuitively, the derivative component of the PID controller analyzes the speed at which the difference $v_i - v$ evolves. If it is decreasing faster, the derivative $\frac{d(v_i - v)}{dt}$ will be negative and the contribution of the derivative component will be negative. In a sense, the derivative component is predicting the state of the system in the following instants and trying to smooth the $F_{traction}$ accordingly.

The integral component on the other hand, accumulates the difference $v_i - v$ over time. The bigger that difference, the greater its contribution to $F_{traction}$. This component, if not properly tuned with the proportional and derivative components, causes the velocity of the car to overshoot the ideal velocity.

Tuning the PID controller is the act of finding the best constants K_p, K_i and K_d such that the cruise control behaves according to the requirements. These requirements can vary. For the car example, a requirement might be to never overshoot the ideal velocity and to reach it as fast as possible.

Figure 4 shows the CBD with the PID controller.

3.1.2 Mass-spring-damper System

For this assignment, we will need a slightly more complicated model to represent a moving train with people in it. A mass-spring-damper is a good abstraction for a wide range of physical systems. Multiple interacting mass-spring-damper systems are used to model all sorts of rigid bodies and even human bodies [3]! We will introduce you to this system and its governing equations and then, in Section 3.5.1 we will show you how it can used in practice to model a more complex system.

Figure 5 shows a representation of the mass-spring-damper system. It is called a mass-spring-damper system because it has a mass (the rectangle in the figure), a spring (the zig-zag drawn in the upper left part of the figure) and a damper (the zig-zag enclosed in a rectangle in the lower left part of the figure).

For a spring, the intensity of the force of which the spring opposes compression is proportional to how compressed the spring is. So, in Figure 5, if there is an external force pushing the mass to the left, displacing it by $-x$ (note that anything to the left is negative), the spring will exert a force in the mass in the opposite direction: $F_{spring} = -k(-x)$, where k is a constant.

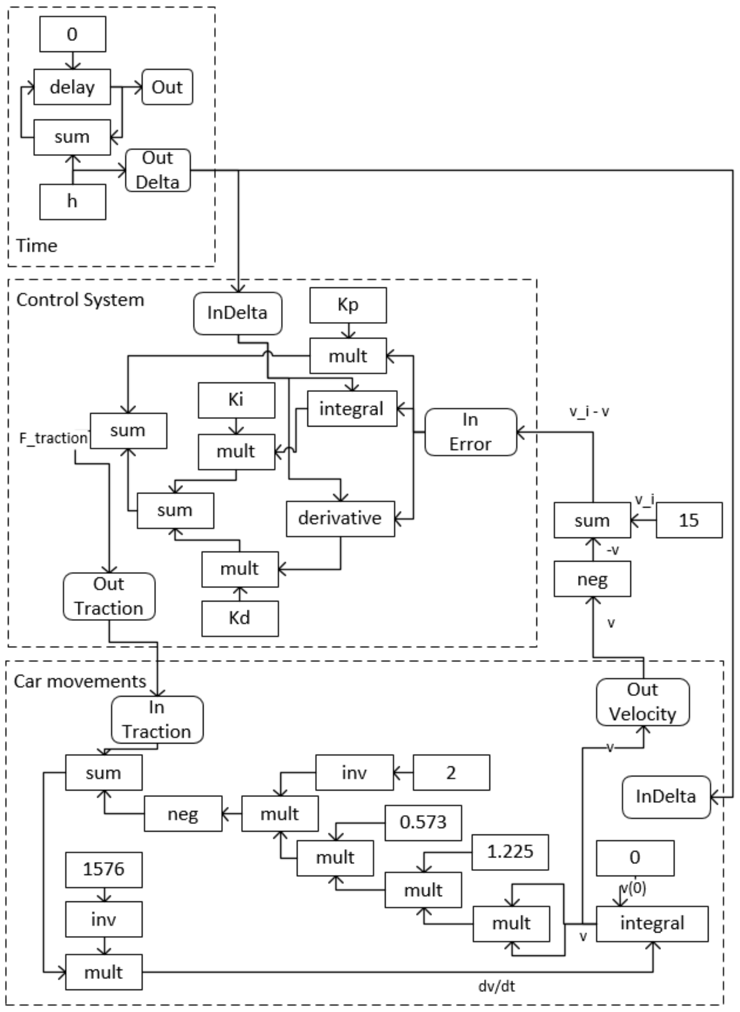

Figure 4: Cruise control system implemented with a PID controller.

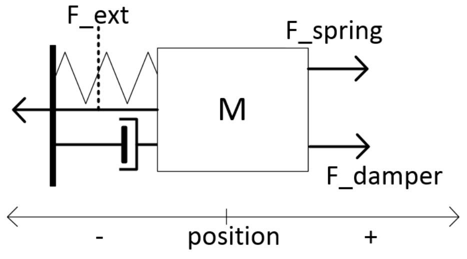

Figure 5: Mass-Spring-Damper system with forces applied to the mass body.

A damper, on the other hand, opposes compression in a different way: A *slow* compression of the damper will not be opposed with significant force. However, a *fast* compression will. The behaviour is independent of the amount of compression. So, in Figure 5, if there is an external force applied to the mass, pushing it to the left with a velocity of $-v$, the damper will exert an opposite force in the mass: $F_{damper} = -c(-v)$, where c is a constant.

Applying the second law of Newton, the behaviour of the system shown in Figure 5, under the influence of an external force $F_{ext} = f$, yields the following equations:

$$\begin{cases} F_{ext} & = -f \\ F_{spring} & = -k(-x) \\ F_{damper} & = -c(-v) \\ M \cdot a & = F_{ext} + F_{spring} + F_{damper} \\ \frac{dv}{dt} & = a \\ \frac{dx}{dt} & = v \end{cases} \tag{7}$$

Equation 7 can be converted to the following first order ODE:

$$\begin{cases} \frac{dv}{dt} & = \frac{1}{M}(-f + k \cdot x + c \cdot v) \\ \frac{dx}{dt} & = v \end{cases} \tag{8}$$

Which can be converted into a CBD using integrators or derivatives, just like you did in the previous assignments.

3.2 Driver-less Train

We wish to create a control system for a driverless train. The control system will replace a human driver.

3.3 Scenario

We know that, to maximize space, and because most commuters travel short distances, there are no seats in the train. There are many poles across the train so that commuters can secure themselves while standing. Figure 6 shows one carriage of this train with one passenger.

In the typical operation, the train communicates constantly with a central computer, which receives the train position by GPS and sends the recommended velocity to the control system of the train, which we are responsible for designing.

The control system replaces a human driver by constantly monitoring the velocity of the train and accelerating or braking/decelerating the train, according to the velocity ordered by the computer. Notice that, just as a human driver would do and because people are standing, the control system cannot accelerate too much as people might fall down.

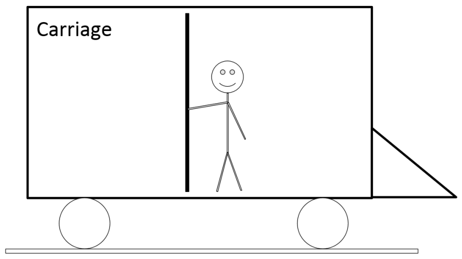

Figure 6: Diver-less train.

3.4 Experiment

For the purposes of this experiment, you may assume that the train moves along one dimension (the rail is flat and straight) and that it has only one carriage with only one passenger inside, as we shown in Figure 6.

The computer system is just a pre-defined script, as shown in Figure 7. The velocities given by the computer are in meters per second (m/s); they are positive naturals (including zero) and they cannot differ by more than $10m/s^2$. This means the computer will never tell the train to stop (velocity zero) and, one second later, to go at $20m/s$. In the worst case, it will tell the train to go at $10m/s$.

The control system, for the ideal velocity example shown in Figure 7, must ensure the a *correct* and *optimal* operation of the train.

A *correct* operation of the train means that:

- the train, given enough time, will attain the ideal velocity and
- no passenger in the train will fall due to acceleration/deceleration.
- the train will not go backwards (negative velocity) unless the computer orders it to.

An *optimal* operation of the train is:

- a *correct* operation, and
- the train attains, as fast as possible, the velocity ordered by the computer.

Notice that the train can overshoot/undershoot the desired velocity.

3.5 Plant Model

In order to create the control system, and because we do not have access to a real train with real people and the computer system, we need to create a model of them. This is called the plant model.

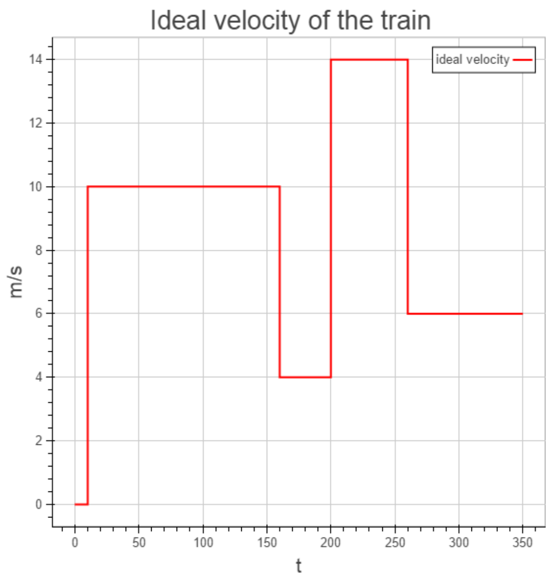

Figure 7: Example of the velocities given by the central computer system over a period of 350 seconds.

3.5.1 Abstractions

For the system and the experiment described in the previous sections, there are many possible ways to create a model of the driver-less train, each with different degrees of fidelity. A high fidelity model mimics the real system in a more accurate way than a lower fidelity model.

For our experiment we are only interested in whether a passenger, subjected to the inertial force of the train accelerating, can fall. For this it is sufficient to measure horizontal displacement of that passenger.

A passenger that is grabbing the pole in the train can be modelled as a single mass that is connected to a spring and a damper to the pole, as is shown in Figure 8. This is a low fidelity model. If we were experimenting with a passenger walking or running in the train, we would need a higher fidelity model such as a set of interconnected masses, spring and dampers. There are innumerous ways to model the mechanics of the human body. If you are interested in those, please read a good review in [3].

In the next section you will see why Figure 8 is a good abstraction.

The utility of the model is related to the experiment that you want to perform. By consequence, the fidelity of the model should the appropriate to the experiment. “A model should be as simple as possible, but no simpler”, Einstein.

As for the rest of the plant, there are also many possible decisions about which abstractions to use. We decided to keep things simple and assume that there is no friction between the train and the rail, no friction between the pas-

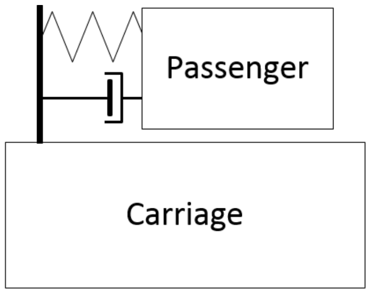

Figure 8: Plant abstraction.

senger and the train and no air drag.

3.5.2 Equations

After settling on the abstractions to use, we need to come up with the equations that model the behaviour of the abstraction that we have introduced.

Figure 9 shows the free body diagram of the passenger and the train. We have separated the elements to make it easier to understand the forces acting in each body but notice that they are still connected. An intuitive way to come up with these forces is to imagine what would happen if an external force $F_{traction}$ pushed the train forward (in the right direction), accelerating it. The train, having no friction with the rail and no air drag, would move forward. The passenger inside the train, would immediately feel the acceleration of the train, as a force $F_{inertia}$ acting upon the passenger, opposite to the direction of the movement of the train. Depending on the passenger's mass, this force can be really strong or not. For a really thin passenger (think of an ant), this force would be really weak, almost imperceptible but for a really big passenger, it would be stronger. So this fictitious force[10] is proportional to the mass of the passenger:

$$F_{inertia} = -m_{passger} * a_{inertia}$$

The force is negative because it pushes the passenger in the negative (left) direction.

As soon as the passenger start to move in the left direction, the passenger will exert a force in the spring and a force in the damper (these are not represented in the free body diagram). The reactions to these two forces are the forces exerted in the passenger by the spring F_{spring} and by the damper F_{damper}, in the right direction.

When the passenger exerts a force in the spring (in the negative/left direction), the spring will be displaced from its normal position. The result is that

[10] `https://en.wikipedia.org/wiki/Fictitious_force#Acceleration_in_a_straight_line`

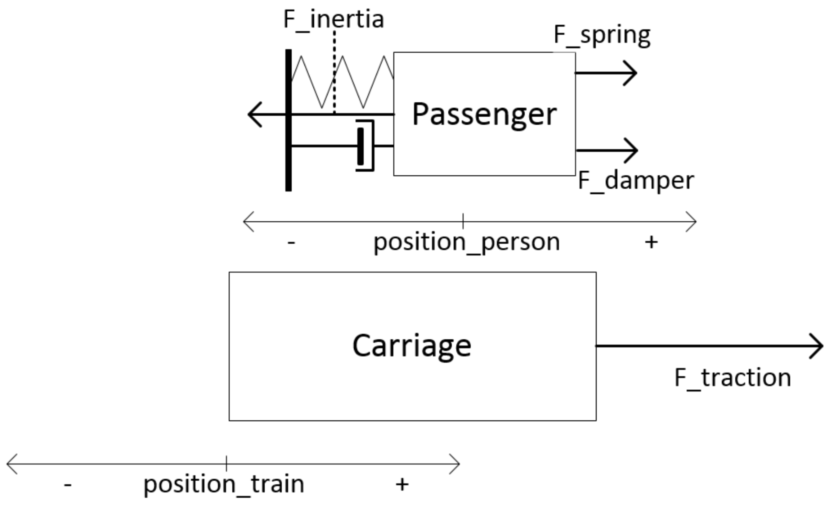

Figure 9: Free body diagram of the train and the passenger.

the spring will exert a positive force in the passenger that is proportional to its displacement:

$$F_{spring} = k(-x_{passger})$$

Notice that F_{spring} is positive because it pushes the passenger in the right direction. The $-x_{passger}$ is positive because the passenger is displaced in the left direction.

Similarly, when the passenger exerts a force in the damper, it will be displaced but in this case, the force that is exerted in the passenger is proportional to the velocity of the displacement of the damper:

$$F_{damper} = c(-v_{passger})$$

Notice that $v_{passger}$ is the derivative of $x_{passger}$.

The resulting force in the passenger is the sum of all the forces acting upon it:

$$F_{passger} = F_{inertia} + F_{spring} + F_{damper}$$

Applying to Newton's second law to the passenger, we get

$$F_{passger} = m_{passger} * a_{passger} = F_{spring} + F_{damper} + F_{inertia} \Leftrightarrow$$
$$m_{passger} * a_{passger} = k(-x_{passger}) + c(-v_{passger}) - m_{passger} * a_{inertia}$$

Doing the same thing to the train, we get:

$$F_{train} = (m_{train} + m_{passger}) * a_{train} = F_{traction}$$

This is because the only force acting in the train is the traction force that pushes it forward. Notice that the total mass of the train includes the mass of the passenger.

We still do not know $a_{inertia}$. This is the acceleration felt by the passenger when the train accelerates forward. It is of the same intensity in absolute value as the acceleration of the train $a_{train} = a_{inertia}$.

In summary, we have the following system:

$$\begin{cases} m_{passger} * a_{passger} & = k(-x_{passger}) + c(-v_{passger}) - m_{passger} * a_{train} \\ (m_{train} + m_{passger}) * a_{train} & = F_{traction} \\ a_{passger} & = \frac{dv_{passger}}{dt} \\ v_{passger} & = \frac{dx_{passger}}{dt} \\ a_{train} & = \frac{dv_{train}}{dt} \\ v_{train} & = \frac{dx_{train}}{dt} \end{cases}$$

We can convert it to first order Ordinary Differential Equation form by replacing the accelerations by the derivatives of the velocities in the right hand of the equations and replacing the a_{train} term in the left hand by its solution $\frac{F_{traction}}{m_{train}}$:

$$\begin{cases} \frac{dv_{passger}}{dt} & = \frac{k(-x_{passger})+c(-v_{passger})-m_{passger}*\frac{F_{traction}}{(m_{train}+m_{passger})}}{m_{passger}} \\ \frac{dv_{train}}{dt} & = \frac{F_{traction}}{(m_{train}+m_{passger})} \\ \frac{dx_{passger}}{dt} & = v_{passger} \\ \frac{dx_{train}}{dt} & = v_{train} \end{cases} \tag{9}$$

The values for the other constants that give somewhat realistic values are:

$$\begin{cases} m_{passger} & = 73kg \\ m_{train} & = 6000kg \\ k & = 300 \\ c & = 150 \end{cases}$$

You will build a CBD from the Equation 9 just like in the previous assignments.

$F_{traction}$ is currently unknown because it will be the output of the control system. However, for testing purposes, we can assume it has a predefined trajectory. Whenever $F_{traction}$ has a positive intensity, the train accelerates the people are displaced, as shown in the blue trajectory of Figure 10. Notice that in the figure, we do not have included the $F_{traction}$ because it is proportional to the acceleration of the train, in red. $F_{traction}$ is a predefined trajectory with the exact same shape as the acceleration of the train but with much larger intensities.

You can also use Figure 10 to convince yourself that the model is appropriate for our experiment. When the train suddenly accelerates, the passenger gets displaced by more than 2 meters. This is a huge displacement and of course means that the passenger has fallen but this happens because we have predefined $F_{traction}$ values, without a proper control system. So, ignoring the absolute values of the displacement and observing the general trajectory we see

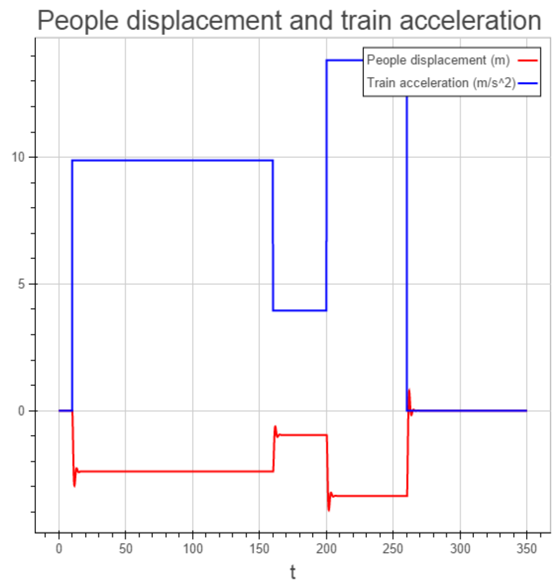

Figure 10: Trajectories of the train acceleration and the passenger's reaction when the train acceleration is not controlled.

that the passenger is initially taken by surprise with the sudden acceleration. However, after a while, the passenger is stabilized and gets used to (or stabilizes) the acceleration. Of course, the passenger is still displaced because the train is still accelerating. When the train no longer accelerates (from 250 seconds onwards), the passenger is again taken by surprise but after a while is able to steady itself.

3.5.3 Controller Requirements

Having a working model of the plant, we can now come up with proper requirements for our controller.

The experiment done previously (see Figure 10) shows how sudden changes in the acceleration of the train can have a huge impact on the displacement of the passenger. So one of the requirements is that the controller never applies a force $F_{traction}$ strong enough to make the passenger fall. We also provide a concrete condition to detect whether the passenger has fallen: whenever the absolute value of the displacement of the passenger exceeds $0.4m$ then the passenger is considered to have fallen.

The control system should be implemented with a PID Controller. The input to the control system is the difference between the ideal velocity v_i and the velocity of the train v_{train}; and the output is the $F_{traction}$ force. The controller, the computer system and the plant are shown as CBDs in Figure 11. The

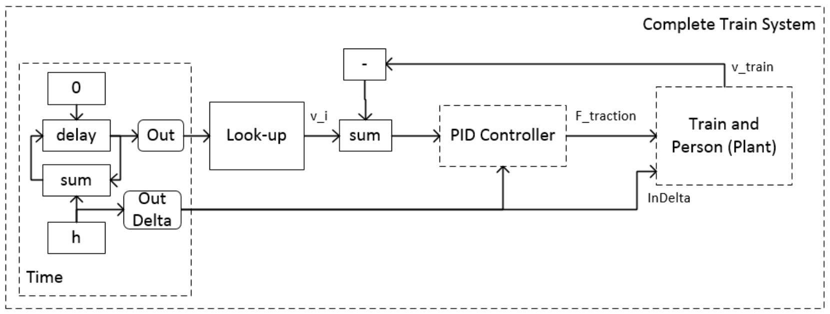

Figure 11: Top-level of the complete train system.

computer system is abstracted as the predefined trajectory shown previously in Figure 7. Section 3.6.1 shows how to model it.

3.6 Tasks

3.6.1 Implement a Look-up Block

A look-up block is a CBD block that implements a predefined trajectory over time such as the one shown in Figure 7. It is useful to model the computer system, as shown in Figure 11.

Create a new class `ComputerBlock`, extending from the `BaseBlock` class. This block has no inputs and one output. Its output, at any point in time, is given as summarized in Table 1. You can obtain the current simulated time using `self.getClock().getTime()`.

Table 1: Look-up Block table that implements the ideal velocity trajectory shown in Figure 7.

time	**output**
<10	0
<160	10
<200	4
<260	14
>260	6

3.6.2 Build and simulate the Driver-less train CBD Model

This task consists of building and simulating the complete CBD model shown in Figure 11. The equations for the train and the passenger are in Equation 9 and the computer is modelled with the Look-up block created in the previous task.

For this task, use the following values for the PID controller parameters:

$$\begin{cases} K_p & = 200 \\ K_i & = 0 \\ K_d & = 0 \end{cases}$$

Use the figures 13 and 14 to validate your model. Your results should match the ones shown in the figures. It also is clear that the initial parameters of the controller ensure that the passenger never falls, that is, the controller is *correct* (recall Section 3.5.3) but it certainly is not optimal.

3.6.3 Tune the PID controller

If you recall Section 3.5.3, our purpose is tune the PID Controller such that it is *correct* and *optimal.* Tuning the PID controller boils down to finding good values for the constants K_p, K_i and K_d. This can be done automatically or manually. Either way, it is a trial and error process:

1. Given some initial values for K_p, K_i and K_d;
2. Simulate the CBD;
3. Evaluate the results (using a cost model);
4. Set new values for K_p, K_i and K_d;
5. Repeat steps 2-4 until the results are satisfactory.

Manually or automatically, this is an optimization task. We need to define our evaluation (or cost) model so that we know when we have improved or not. For this task, a custom cost model is provided as a form of a CBD Block in the file `TrainCostModelBlock.py`. You can use the custom cost model block as shown in Figure 12. Notice that the cost model block needs the time step delta to be given as an input, just like the integrator block.

The output of the cost model block is the current cost of the model, which is an accumulation of the past values of the cost. The final cost is the last value, at the last simulation step, outputted by the block.

Your task is to find proper constants such that the cost model is as low as possible, while satisfying the correctness requirements specified in Section 3.5.3.

An optimal set of parameters will minimize the time it takes for the train to cross the desired velocity. It can overshoot but it cannot become a negative velocity. If some of the correctness requirements is not satisfied, the cost model block will automatically halt the simulation with a `SimulationException`.

There are two levels of sophistication, that will be reflected in your grade, for the realization of this task:

Manual Tuning – You perform the optimization manually.

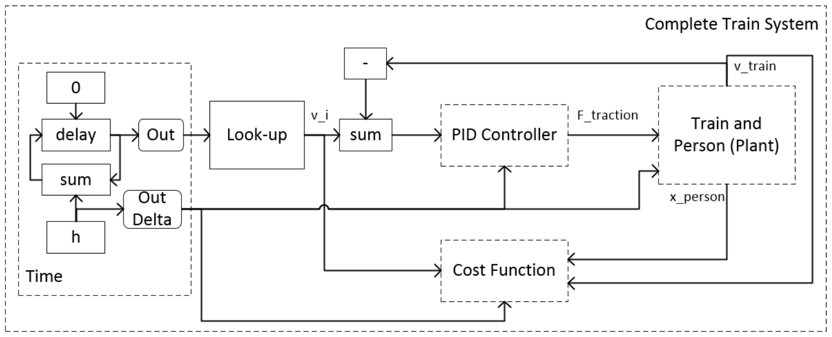

Figure 12: Top-level of the complete train system with the custom cost model block.

Automatic Tuning in Python – You perform the optimization automatically by building an optimization loop that calls the CBD simulator at each iteration, evaluates the cost model and changes the constants. How you change the constants is your choice. In the limit, you can simply increment/decrement them by 1, compute the cost for each combination of values for the constants, store them and later pick the combination yielded the minimum cost. Make sure you discard combinations which cause the passenger to fall.

3.6.4 Document

Write a small report containing the tasks that you have completed and how you have completed them. Include all the plots that you have made from the simulations. The CBD model that you have built should be displayed graphically in the report. You can use the `draw` function, defined in the `CBDDraw.py` file to export your model as a DOT (Graph Description Language)[11] model. You can then use the generated DOT file to draw the graph with the GraphViz tool or online: `http://graphs.grevian.org/`.

3.7 Results

Figures 13 and 14 show my results. Use these to validate your model.

References

[1] Karl J Aström and Bjorn Wittenmark. *Computer-controlled systems: theory and design.* Courier Corporation, 2011.

[11] `https://en.wikipedia.org/wiki/DOT_(graph_description_language)`

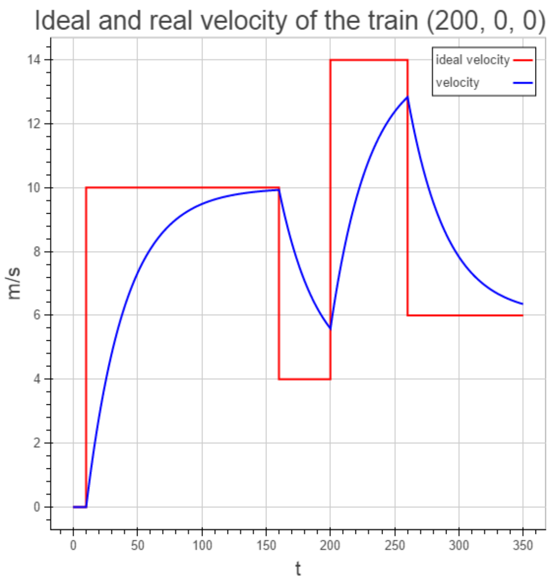

Figure 13: Plot of the train's velocity and the recommended velocity using the initial parameters given in Section 3.6.2.

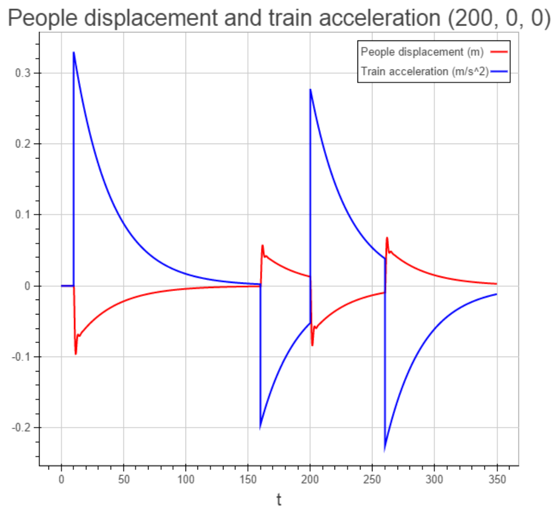

Figure 14: Plot of the passenger's displacement and the train's acceleration using the initial parameters given in Section 3.6.2.

[2] Chi-Tsong Chen. *Linear System Theory and Design*. Oxford University Press, Inc., New York, NY, USA, 2nd edition, 1995.

[3] Ali A. Nikooyan and Amir A. Zadpoor. Mass-spring-damper modeling of the human body to study running and hopping : an overview. *Proceedings of the institution of mechanical energineers part H-journal of engineering in medicine*, pages 1121 – 1135, 2011.

[4] Ernesto Posse, Juan De Lara, and Hans Vangheluwe. Processing Causal Block Diagrams with graphgrammars in AToM^3. In *European Joint Conference on Theory and Practice of Software (ETAPS), Workshop on Applied Graph Transformation (AGT)*, pages 23–34, 2002.