CS 452 Train Control Final Project

Benjamin Zhao, Kyle Verhoog

Apr 06, 2018

Contents

Operation

The elf file is located at `/u/cs452/tftp/ARM/ktverhoo/final.elf` and can be initiated using the typical `load` command.

After `load`, wait a few seconds for tasks to initialize.

You should be presented with our interface featuring *windows*.

You can type commands described in the next section.

Commands

- `tr <tr#> <speed>` sets the given train to the given speed.
- `dr <tr#> <speed> <dest_node>` drives the specified train to the specified
- location at the specified speed.
- `dummy <offx> <offy> <width> <height>` creates a dummy task which writes to the screen in a new window.
- `cal <train> <speed> <pivot>` stopping calibration command, uses pivot as the
- `ms <train>` calibrates the inching speed using train speed 4.
- `tst <train> <node>` tests the calibrated test on a node of your choice.

Design

Train Model

The purpose of the Train Model is to hold all related data about the train including speed, acceleration and deceleration. We use the Train Model to accurately track trains running on the track which provide finer granularity on a millimetre to millimetre basis.

Calibrating the Train Model

The calibration of the train model includes live updates and offline calibrations. When doing offline calibrations, we specifically run through tests specialized to calculate the speed, stopping distance and acceleration distances of the train.

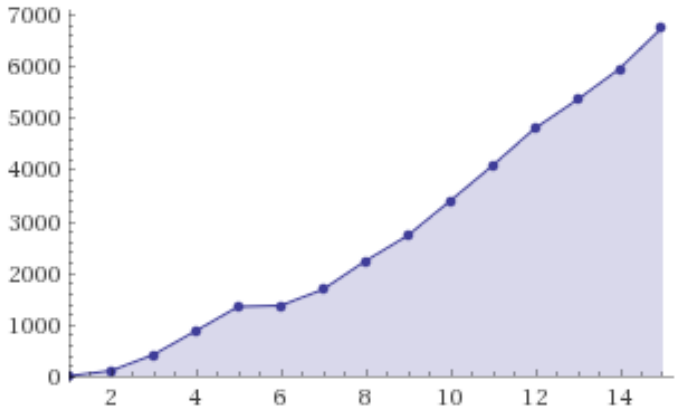
Live updates are temporary for the running project. While the project runs, it accumulates values calculated sensor to sensor which correspond to a train travelling along the track.

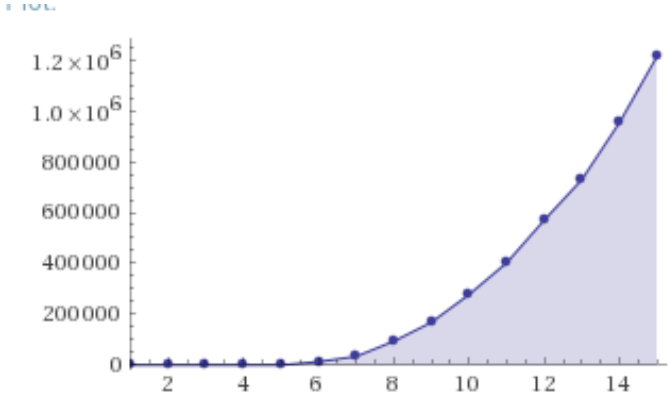
We base stopping distance and acceleration distance on a train setting we deem with "zero-time" acceleration. We measure the total distance (sensor-to-sensor) and subtract the travel distance on our zero-time gear to find the acceleration distances.

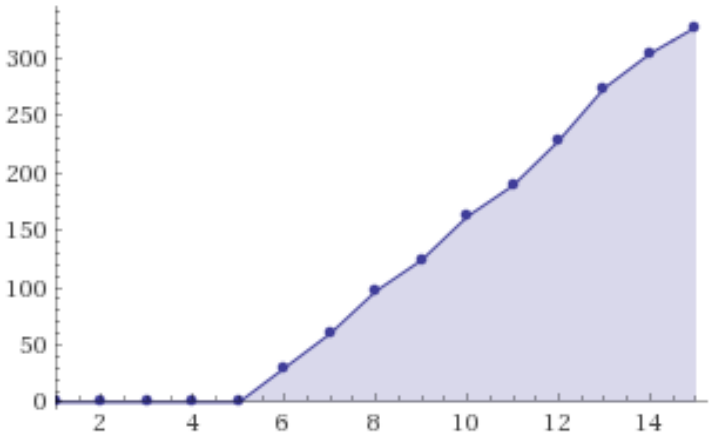
Interpolating the Data

By using the Lagrange Formula for interpolation, we can obtain an accurate continuous model which reflect the trains datapoints. Further more, we break down the train setting to a finer granularity (ie 0 - 140) which allows us to map the x, y measurements. We note that although lagrange can flucuate which may produce inaccurate results, tuning the upper and lower bounds allows us to obtain relevant information.

Furthermore, we've noticed linear trends on the acceleratio and deceleration timings which allows us to easily map between gear, velocity and acceleration without solving roots. This breakdown allows our model to behave closer to the real trains.







Alpha Updates, Estimation and Standard Deviation

The Train Model can update its value based on real data as well. We use an alpha scale where alpha ~ 0.40 weighting influences the estimated model. We also use standard_deviation (or used to) to guard our datasets from outliers.

Usage and Applications

Generally, the application uses the model to interpolate how far a train has gone into an acceleration. It also interpolates the distances travelled when there is no absolute train gear to be associated.

```
int get_lagrange_basis(TrainModel *tm, int point, int x){
    int frac = tm->y[point];
```

```
    int i;
    for(i = 0; i < TRAIN_MODEL_SIZE; i++){
        if(i != point){
            frac *= (x - tm->x[i]);
            frac /= (tm->x[point] - tm->x[i]);
        }
    }

    return frac;
}


int interpolate(TrainModel *tm, int setting){
    int val = 0;
    int i;
    for(i = 0; i < TRAIN_MODEL_SIZE; i++){
        val += get_lagrange_basis(tm, i, setting);
    }
    return val;
}

int linear_interpolate(TrainModel *tm, int setting){
    int floor, lower, upper, frac, result;
    floor = setting/10 * 10;
    frac = setting - floor; //Ensure to divide by 10

    lower = tm->y[floor/10];
    upper = tm->y[floor/10 + 1];

    result = lower + (upper - lower)*frac/10;
    // assert(result <= upper);
    return result;
}
```

The Pipeline

We spent a long time and had a number of extended conversations with each other regarding how to design a clean, scalable, testable solution to the problem of interpreting track events. Our solution was to come up with a data pipeline which provides layers of abstraction to real and virtual events.

Events

Raw

We name events that emanate from the real world "raw events". These include `tr` commmands, `sw` commands and results from sensor polls.

Virtual

We name events that are generated by our system (which are fed back into itself) virtual events. These events may correspond to raw events or may not. They help maintain program state that is asynchronous or requires timing.

Virtual events allow our program to achieve a finer grained granularity than mere sensor poll updates. This allows subscriber tasks to subscribe on events on track nodes that are not sensors.

Types

The waiting room will return different event types depending on the situation of the anticipated event. Virtual Events can pre-emptively register to the waiting room that the virtual event is coming up known as a VRE. When a raw event corresponding to the virtual event, it raises an RE. When the virtual event comes in, a VE is raised. Combining the three, we obtain the combinations (VRE VE), (VRE RE), (VRE VE RE).

`VRE VE` - Virtual Event came in, timed-out without a corresponding Raw Event
`VRE RE` - Virtual Event Registered, Raw Event came before estimated Virtual Event `VRE VE RE` - Virtual Event came, Raw Event came shortly after (within an alloted threshold)

Providers

Data Providers

The data providers are tasks which primarily focus on talking directly with the train controller to issue commands or poll on sensor data. These tasks form into three major groups.

TrainProvider

The train provider entity is a set of tasks which performs the operations of sending atomic train commands to the train controller. The train controller listens on any incoming requests made by other tasks and sends the request to the train controller. In addition, the TrainProvider raises a raw event to the pipeline signalling that an event to move a train has been issued. Subscribers on the raw event are notified when the event takes place. Currently, only the WaitingRoom task subscribes to the TrainProvider.

SwitchProvider

Similar to the TrainProvider, the SwitchProvider is a set of tasks performing sends and raising of events for when switch requests occur, The SwitchProvider listens on any incoming requests and executes them. It raises an event and notifies all subscribers that a raw event for a change on a specific switch has occured. Currently, only the WaitingRoom task subscribes to the SwitchProvider

SensorProvider

The SensorProvider is a set of tasks which manage the polling of sensor data. All round trips of sensor data are raised as a raw event for the subscribers to listen for. On a further note the WaitingRoom which subscribes to the SensorProvider sits behind a SensorDelta, which filters the raw events for only the iterations which differ from its previous.

VirtualEventProvider

Along side the raw data provider, TheVrtualEvent provider raises virtual events to subscribers. Virtual event registrations are periodically delivered to subscribers. Note that VirtualEvents are not affiliated with TrainController itself, but can provide accurate measurements and predictions for when Raw Events may actually occur.

Waiting Room / Matchmaker

The waiting room is a precursor to the interpreter. Many Virtual Events have direct correlations with a Raw Event. These virtual events will wait in the waiting room for the corresponding raw events to occur. At the worst case, raw events which do not occur (ie broken sensor) can be pseudo reflected by the virtual event and will be up to the interpreter to decide an acceptable state.

Interpreter

The Interpreter is where most of the logic is performed for figuring out, given both virtual and raw events, what is actually going on, on the track.

We took what we consider to be an interesting approach of associating trains to sensors. We make no assumptions about where and when the train will be. The interpreter assumes that anything can happen and then tries to make sense of the data it gets back.

If irregularities are detected (for example, a train appears to go down both paths of a branch) then the Interpreter invalidates the reading it got and places the

train in a `TR_LOST` state. When new events emerge, we check `TR_LOST` trains attempting to re-associate the train.

The Interpreter generates higher level events to pass on to the Representor which then goes on to distribute them to subscribers. Events the Interpreter generates include train i at position n.

Acceleration, Velocity and Distance

In order to modularize at a higher precision based on train velocity, a straight mapping between train setting and train velocity is not enough. An easy way to interpolate the velocity between a speed setting is to apply a `(n-1)`-degree polynomial interpolation (in our case we used the lagrange method). Since integrating and deriving polynomials are systematic, we can easily attribute acceleration to velocity to distance to a confident degree of accuracy.

Note that interpolation may fluctuate at higher valued results, and thus it may be worth while to apply partial interpolation instead.

Estimator

The Estimator is the follow-up rightful successor to the Interpreter. While we started out with a model trying to infer train positions given only sensor data, we determined that this would not suffice for any kind of elaborate project.

The Estimator is almost the complete opposite of the interpreter in that it relies heavily on models that we generated of the train speed. It relies heavily upon the speed and acceleration models that we measured and calculated. It works by updating a virtual model of the train every 5 clock ticks as well as encompassing sensor events generated from our events layer.

While we originally had an intuition against trusting our calculated models and depending on track data, we came around to realize that we could much more accurately and reliably track the trains.

Positioning

To keep *track* of trains we position them relative to track nodes. Essentially a train always exists on some edge of the track graph relative to some track node. For example a train could be 300mm past sensor `A1`.

Updating

The Estimator is updated every 5 ticks. During an update operation we take, for each train, its previous location, speed snapshot and the time elapsed since

the last update. With this information we update the train's position to the present location.

Track Algebra

To move trains along the track we have track algebra methods which allow us to move trains forward and backwards on the track with the ability to support blocking trains.

When an update comes in and the train has to be moved some distance we apply the following algorithm. We move the train up as far of the distance we can until we potentially hit another train. If this occurs then perhaps that train has not yet been updated. So we attempt to recursively update that train.

Thus, if we have chosen arbitrarily to update the last train in a group of 3, the last will call the second to update which will call the first.

When the updates return back we attempt to move the train the remainder of the distance. If we cannot move the train the complete distance (due to a blocking train) then we place the train immediately behind the blocking train.

This allows us to avoid mixing up train identities in the model.

Sensor Attribution

When sensor events come in we must try to attribute them to a train. We have the following algorithm.

1. We store for each sensor a list of trains that have virtually passed the sensor. That is, trains that have passed the sensor in our model but not in reality. We check this list first to associate a train. Since our track algebra maintains ordering then we will attribute the correct train.
2. If the above fails then we check the node directly in front of the sensor for trains. If there are trains, we take the one which is closest.
3. If both of these fail then we try to find a train by its crumbs.

Crumbs

Each train as it travels leaves crumbs. The crumbs contain a timestamp and position along with a speed snapshot. When a sensor event comes in we BFS backwards from the node looking for crumbs. When we find a node with crumbs, we take the most logical train according to its speed, distance, timestamp and ordering.

These crumbs are removed recursively when a train is associated to a sensor.

Representer

The Representer is a high level event provider that gives subscribing tasks an API to by notified of current events. It essentially passes on some events from the Interpreter with additional computed events. We modified the representor to encompass the Estimator.

Subscribers

The subscribers form the bottom of the pipeline. They subscribe to events provided by the Representor. These events are high level events like train `24` at node `B13`.

Some examples of subscribers are the interfaces for the track events. The sensor, switch and train interfaces each subscribe to events provided by the Representor.

Reservation Manager

The Reservation Manager provides a mechanism for trains to allocate track nodes to attempt to avoid collisions.

Our Reservation Manager is extremely conservative, we reserve the stopping distance of the train and a little more at the granularity of sensors for all possible paths ahead of us. That is, a reservation is made between two sensors. So even if the stopping distance of the train is 1mm, we will reserve all the way to the next sensor in each direction possible.

This is due to the fact that we did not have time to get a finer grained update granularity functional for this milestone.

Trains request track nodes ahead of it and return nodes that it has passed by. Due to the nature of the interpreter model and since the train can potentially be reset a node backwards we free two nodes behind us.

An API for pathing was attempted but not achieved. This API considers current reservations and attempts to path around them.

Train Driver

The Train Driver does just that, drives the train. It is in charge of attempting to allocate and free nodes using the Reservation Manager. If it cannot allocate nodes from the Reservation Manager then it sends the stop command to the train and attempts to reverse and path in the opposite direction.

The Train Driver also keeps track of the state of the train it is driving by subscribing to the Representor. It attempts to path its train to a given position. Once the train has arrived at the position, it re-routes the train to the reversed

position. For example, if we route the train to node `A15`, once the train has attempted to stop at `A15` it will re-route to `A16`.

Display Manager

We figured that it would be worth it to have a structured way to present data to the screen both for presentation as well as for debugging.

Terminal Manager

Terminal Manager manages a set of windows. Tasks can request a window from the Terminal Manager and output from the task can be redirected to either its own window or a common-to-all-tasks logging window.

There is currently support (not enabled currently) to route input to windows other than the shell, depending on the cursor focus.

The Terminal Manager attempts to smartly render the screen as to limit the amount of cursor movement byte-sequences needed.

Shell

Shell is just the first task which registers to the terminal manager which is configured to accept input. It is currently a monstrosity which handles the parsing and executing of all commands.

IOServer

Blocking PutC

We added a blocking version of the `PutC` function which is very useful in applications like the Sensor Manager for when it polls to the train controller.

TC 2

Res

After calibrating, we manually store the stopping distances. The stopping logic is relatively straightforward. We step backwards from the last node until we find the first sensor that is more than the stopping distance away from the node. Then after passing this sensor we can set a delay to stop at the correct position.

Calibrating

Method 1

Our intial method to calibrate train stopping distances was to have the train use a starting sensor as a measuring stick to try to land on a targer sensor further on down the track. We make an intial guess and check whether or not we overshoot the target. We then subtract or add to our guess and bring the train around to try again.

Obviously this is not very efficient and took quite a while to run, taking up to 5 or 6 iterations to achieve accurate results.

The plus side to this technique was that the results we get from the test are quite accurate.

Method 2

Our more efficient method was an inching strategy. Again, we start out with a guess. But this time if we overshoot the sensor we inch at a slow speed that we know to the next sensor, measuring how much time it takes. By using a slower speed we make the assumption of very little acceleration and thus we can calculate the stopping distance using the time and the speed.

This method is much more efficient and only takes a couple of iterations to get good results.

A problem, not with the method, is that moving at a slow speed means a greater chance of getting stuck, which ruins calibration results.

Pathing

Pathing is done using Dijstra's algorithm using our own heap implementation. There is nothing particularly special about the algorithm or its implementation.

It is thoroughly unit tested and we are fairly confident in its correctness.

When given a node to stop at, we calculate the path starting two sensors ahead of the train to the destination node. Then we check the switches in the path and set them in order starting from the destination node.

Resetting

We have a simple reset task which configures the switches to form a loop which we use to set ourselves up for the pathing and stopping.

Files/Hashes

The code is in the `master` branch at `https://git.uwaterloo.ca/bkcs452/kernel/commits/master`