Modelling and realizing the Tunnelling Ball Device in UniTi and CλaSH

Peter Lebbing

June 5, 2019

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

This report describes the modelling of the Tunnelling Ball Device using UniTi, and its implementation using CλaSH. The Tunnelling Ball Device is a cyber-physical system devised at the department of Electrical Engineering & Computer Science of the University of California, Berkeley. It consists of a box housing a spinning disc and a tower structure on top of the box. The disc has two holes at opposite ends. Small steel balls are dropped from the top of the tower, above the spinning disc. Sensors mounted on the tower measure the motion of the steel balls, and the disc is controlled in such a way that the ball passes through a hole regardless of when it is dropped. [Jen10] introduces the system and models it.

The Tunnelling Ball Device that is described in [Jen10] contains a micro-controller to control the device. Conversely, our Tunnelling Ball Device is designed around an FPGA. The FPGA interprets the sensors and controls the motor. An FPGA (Field Programmable Gate Array) is a logic device that can be programmed to perform a specific function, and is an example of a fine-grain reconfigurable platform. It consists of a large array of configurable logic blocks (CLBs)* which can be interconnected by a massive interconnect structure. Each CLB contains several slices, and each slice performs a very simple logic task, such as implementing a binary function with six inputs and one output, optionally followed by a flip-flop. A slice can be configured to perform several different tasks, but all of the same basic complexity level. Furthermore, an FPGA contains a small amount of RAM and several specialized functional units, such as multipliers, digital signal processing (DSP) functions and clock generation logic to provide a multitude of clock signals. As such, an FPGA is programmed at a much more basic level than a processor, and in fact, an FPGA can be programmed to behave like a processor. Implementing a design on an FPGA can also be a first step towards the production of a new integrated circuit, an Application-Specific Integrated Circuit (ASIC). This ASIC can then be mass-produced and perform only the one function it was designed for, unlike the FPGA where the functionality was first designed and tested.

An FPGA is programmed through the use of a hardware description language (HDL). Examples of much used HDL's are VHDL and Verilog. Modern FPGA's provide massive amounts of programmable logic, and as such, an HDL needs to provide ways for the designer to cope with the large complexity of programming those massive amounts of logic.

CλaSH [Baa15] is a functional hardware description language that borrows both its syntax and semantics from the functional programming language Haskell [Cla]. There is a clear relation between the semantics of a functional programming language and the operation of hardware. Digital synchronous hardware can be seen as a mathematical function that maps inputs, and possibly state, to outputs and a new state. Haskell, as a functional programming language, is well suited to work with such mathematical functions, and allows function composition and higher order functions to be used as the basic abstraction mechanisms in describing synchronous hardware. Higher order functions in particular encourage the designer to make a proper decomposition of the problem at hand. With hardware designs getting ever more complex, proper abstraction is highly

* The terminology is manufacturer-specific. A Xilinx FPGA was used for the Tunnelling Ball Device, so the Xilinx terminology is used here.

desirable to allow the designers to keep working with, and properly comprehend such large designs.

UniTi [Rov11] is a design flow and modelling environment for model-based design of embedded systems that interface with the physical world. It accurately simulates different notions of time by allowing multiple time domains, the continuous-time (CT), discrete-time (DT) and dataflow (DF) domains, in a single model. Models are built from components that are signal transformations, and therefore mathematical functions. These signal transformations can accurately capture the continuous-time domain, without introducing discretization errors. UniTi is implemented in Haskell, and UniTi models and components are written in Haskell code. The result is an ordinary Haskell program that can be executed to simulate the model.

Given that both UniTi and CλaSH are based on regular Haskell code, the UniTi code of the simulation is also the basis for the implementation. As the code that is the implementation is also the code used to verify the system in simulation, this improves faith in the correctness and reduces development work.

Additionally, I have found that type inference with type-level natural numbers, combined with Template Haskell, allows for some very nice parameterizations and compile-time computations. This enhances function reusability and genericity.

A common application area for UniTi and CλaSH is stream processing. But they are by no means limited to that. The Tunnelling Ball Device is a nice opportunity to use UniTi and CλaSH in a different setting. That is the ultimate goal of this project: as a testing ground for these two technologies, to see what works well and what does not, and to find bugs or missing functionality.

[Jen10] is used as a basis for the project, avoiding unnecessary duplication of work. The physical realization of the device in [Jen10] is only used as a basis for a redesign. The device is shrunk physically, so it can be made more cheaply and is easier to take to conferences.

1.1 Device overview

An overview of the basic elements of the Tunnelling Ball Device is schematically represented in figure 1. Two sensors (b) and (c) are mounted above a spinning disc (e). Balls (d) are dropped from above the top sensor through a release platform (a). When a ball passes through a sensor, the ball obstructs a beam of light, which is registered by the sensor. The disc contains two holes, and is driven by a motor (f). A rotary encoder mounted on the motor reports the angle of the motor shaft.

When a ball is dropped, it will first pass through sensor (b) and subsequently through sensor (c). The distance between the two sensors has been precisely measured, and the sensors register the time it takes for the ball to fall from sensor (b) to sensor (c). This allows the speed of the ball to be calculated as soon as it passes sensor (c). Subsequently, the speed of the disc is varied in such a way that a hole is presented at precisely the time the ball has dropped down to the disc. This allows the ball to fall through the disc.

The sensors are connected to an FPGA, and the FPGA outputs a signal to a current controller. The current controller is responsible for sending an electrical current through the windings of the motor, which is a basic permanent-magnet DC motor. The amount of current that runs through the windings of the motor is directly proportional to the signal sent by the FPGA. The rotary encoder that encodes the angular position of the motor axle is also connected to the FPGA. The CλaSH design running on the FPGA is responsible for interpreting all input from the sensors and for varying the current through the motor windings in such a way that the position of the disc is precisely controlled.

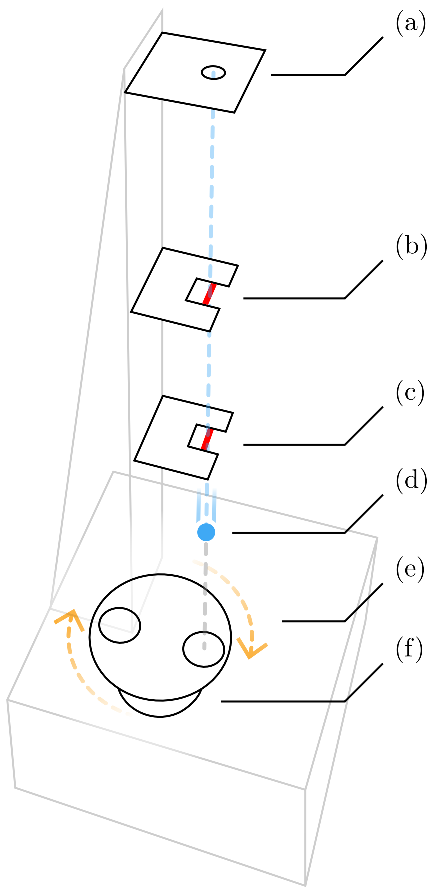


Figure 1 – Basic elements of the Tunnelling Ball Device. (a) Release platform. (b) Top ball sensor. (c) Bottom ball sensor. (d) Ball. (e) Disc. (f) Motor with rotary encoder.

2 Related work on DC motor control

Searching for literature on trajectory control of a permanent magnet DC motor such as used in the Tunnelling Ball Device, the article [RAH97] is on using an artificial neural network (ANN) to control a motor. This appears to have several appealing properties for use in the TBD. It can be trained with data obtained from experiments, measuring the response of the disc to applied stimuli, without needing a proper model of the physical system. Also, the structure of an artificial neural network can be expressed using higher order functions, making it a good fit for UniTi and CλaSH. Finally, the non-linear excitation function of a neural node is commonly a tan sigmoid or a log sigmoid, which are both functions that can be computed in an FPGA with Xilinx IP blocks. So I focused my literature study on ANN-based control, with [RAH97] as central piece.

The principle of ANN control as proposed in [RAH97] is learning the controlling ANN the inverse of the transfer function of the motor, such that the combination of both has the output (actual speed) equal to the input (reference speed). The analysed transfer function is that of a DC motor system with constant inertia but resistance as an unknown non-linear function of the speed. Analysis shows that the inverse transfer function expressed as a difference equation is a non-linear function of three consecutive speed samples. Since an ANN can learn a non-linear function by example, and it is known that the desired function is a function of three consecutive speed samples, it follows that those should be the inputs to the ANN.

[WES91] offers an alternative, by noting that the contribution of the applied terminal voltage in the resultant speed change is linear: all non-linearities arise from other variables than the terminal voltage. Moreover, this relation can be computed from the motor parameters and the total inertia, which are often available. This means that the ANN can be trained to learn and account for just the remaining non-linearities, which is a simpler function and thus easier to learn.

The major difference between [RAH97] and [WES91] is that [RAH97] has the ANN learning even while it is deployed and operating. [WES91] just trains the ANN beforehand. With the on-line learning, the ANN is even able to learn to control the system when some of its parameters have changed, like the inertia or a contributing factor in the resistance function. Note that because a changing inertia is part of the scenario, the alternative control function suggested by [WES91] would not be appropriate, as the inertia is part of the constant that is computed rather than learned.

[RAH97] analyses the case where the output of the controller and the input of the motor is a voltage to be applied across the terminals of the motor. In my setup, the output is a current through the motor; it is left to an off-the-shelf device to realize this current. This should not change the principle, just the actual function to be learned by the ANN. It is still a second order system, so the inputs to the ANN remain the same.

Another difference is the time scale and speed range of the desired control. It can be seen in the graphs of [RAH97] that the reference speed is changed every 2 s in the experiment, and that the speed changes are on the order of $200\,\mathrm{rad\,s^{-1}}$ or more. The control achieves these changes in about a second for the largest changes. On the other hand, in my setup, the changes in speed are more on the order of $30\,\mathrm{rad\,s^{-1}}$, but it needs to be achieved within about 100 ms. The shorter period available for achieving the change in reference speed would suggest that perhaps the operating frequency of the neural network might have to be increased. However, it would seem that [RAH97] operated its neural network control on a frequency of 10 kHz. This seems incredibly high; in fact, the chosen off-the-shelf current controller for the motor in my project cannot go above 5 kHz. But more importantly, I suspect there might not be any useful feedback when operating on such a timescale. The feedback from the motor comes in the form of a rotary encoder. It is a discrete output whose signal is even quite noisy. If there is no appreciable information content in a single sample of the feedback signal at a rate of 10 kHz, it seems futile to base any computations on it.

It appears that figure 2.(a) of [RAH97] contains a rather important mistake. It gives the overall structure of the control system, but the two ANN's are most likely swapped. Such as it is, the reference speed is not even a factor in the inputs to the ANN that controls the motor. Instead, the reference speed is used in the computation of an input to the ANN that is used to evaluate the performance of the controller. It is unlikely this will work at all. The desired trajectory would need to be achieved by training the ANN not just to learn the system dynamics, but to also learn the desired trajectory. However, if we compare figure 2.(a) of [RAH97] to figure 9. of [WES91], it turns out that the two ANN's of [RAH97] are most likely accidentally swapped. If figure 2.(a) of [RAH97] is modified such that the other ANN is connected to the D/A converter, it becomes almost identical to figure 9. of [WES91], at least for the part of the system they have in common. The remaining difference is that [RAH97] has all the delays before the ANN's and compares the outputs of the two ANN's directly, whereas [WES91] moves one unit delay past the controlling ANN, comparing the previous value of the controlling ANN with the current value of the reference ANN. This latter arrangement is more logical, as it directly compares prediction and outcome for each time instant.

[RAH97] says on the choice of [EKAGMS94] to use the reference (target) speed as one of the inputs to the ANN: "[Unlike the inverse dynamic model], the reference speed is arbitrarily taken as one of the inputs of the ANN. As a result, the drive system suffered from the problem of instability."

The choice of a reference model from which to derive the input sequence to the ANN is further described in [WES91]. The model is not only chosen for being asymptotically stable, but also for being achievable by the DC motor. This seems intuitive: the ANN has only been trained with achievable trajectories, so asking to do the unachievable takes it outside what it has learned.

The reference model's output is constructed to be the desired speed trajectory. The input sequence computed for the reference model is also the input to the controlling ANN. The asymptotic stability of the reference model ensures that at any time, an error in the initial conditions of the system will not cause the system to become unstable.

But the reference model might in a different way have a negative impact on the control in the TBD. In the TBD, the actual speed of the disc is of secondary importance. What matters is the position of the disc. Unfortunately, the position cannot readily be used as an input to the ANN, so the ANN should still be steered by a reference speed trajectory. Precise position control is only relevant in the correction trajectory while a ball is dropping; this is only about an eighth of a second. Hopefully, the accumulated error in the position of the disc during the correction interval is small enough that the disc is still in a good position.

But with the reference model of [RAH97], there is more of a disconnect between the reference speed trajectory and the actual trajectory than when the reference speed trajectory is directly an input to the ANN. So when it turns out that the accumulated error in the position is too large, it is an option to try feeding the reference speed to the ANN directly before anything else.

As the use of an ANN requires that the reference speed trajectory is realizable, the reference speed trajectory in the TBD is to be modified to use a sigmoid curve instead of a step to go from one speed to another. The current design with a step change in speed is not realizable in reality: no system with mass can instantly change speed, it can only gradually accelerate or decelerate. Using an appropriate sigmoid curve ensures all speed changes are gradual and lie within the limits of the motor.

Also, even though [RAH97] is the basis for an implementation of ANN control in the TBD, the ANN might actually perform well with just an off-line training beforehand, because the TBD does not have changes in the dynamic behaviour of the system. Implementing the on-line learning is non-trivial, and if it functions well without it, it is superfluous.

3 UniTi

3.1 Overview

UniTi is an environment to model multi-domain cyber-physical systems, and simulate the resulting model. UniTi is described in the thesis of Rovers [Rov11]. The model is composed of components; however, the composition of two or more components is again a component, and as such, the model as a whole is also a component itself.

A UniTi component has *state*, an *input*, an *output* and a *view*, and all of these are optional. Stateless components are very common. The model as a whole has no input, which means the first component in the model has no input unless it happens to be part of a feedback loop. An outputless component is uncommon. It would probably be something like a plotter or oscilloscope, some observer.

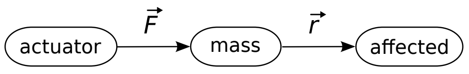


Figure 2 – Example components: a mass.

A good example of a component would be one representing an inertial mass* (figure 2). Its state would be its location and its momentum or velocity, the input would be a vector representing the net force on the mass ($\vec{F}$), and the output would be coordinates representing its current location ($\vec{r}$). It is here assumed that the velocity is not of interest to the model, so it is not part of the output. The actuator that exerts a force on the mass would have its output connected to the input of the mass, and the component that is affected by the location of the mass has its input connected to the mass's output.

Even though the velocity might not be relevant for the model, the designer might still be interested in it. This is where the view comes in: it allows one to directly communicate simulation data to the simulation environment. The model as a whole can also have an output, which is data available to the simulation environment, but the view allows one to extract data from any inner component. That way, the outputs are only used for signals relevant to the operation of the model. The previously mentioned outputless component would interpret its input and deliver data in the view.

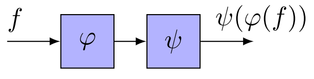
Unfortunately, for the model of the Tunnelling Ball Device, views were unusable in the version of UniTi that was used in this project. This means that all model data that the designer wants to extract and visualize needs to be passed to the output of the model as a whole. Every component that has interesting data needs a "wire" in parallel to any components that follow it, to guide the data to the exit at the end. This means a lot of clutter in the composition of the components, and a lot of components that do nothing but reorganize the wires in different combinations. Views are a very desirable feature to have.

* In Newtonian physics, a mass has two different aspects. It resists a change in velocity: this is an inertial mass. And it interacts with other masses through gravitational pull: this is a gravitational mass.

3.2 Composition in UniTi

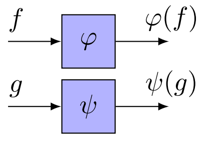
Every component in UniTi is a signal transformation. A component can be defined directly as a Haskell function, but components can also be composed of other components. This hierarchical composition is fundamental to modelling in UniTi. Before elaborating on components, we will introduce the composition operators.

The basic component composition operators are *sequential* (`>>>`), *parallel* (`||`) and *feedback* (`>@>`). Figure 3 depicts them graphically and shows how to use them in code.



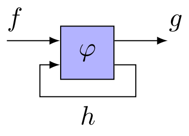
Code: ϕ `>>>` ψ

(a) Sequential composition.



Code: ϕ `||` ψ

(b) Parallel composition.



Code: `>@>` ϕ

(c) Feedback composition.

Figure 3 – Composition operators in UniTi.

Components can have any number of inputs and outputs. Components splitting and combining signals are trivial, and UniTi's standard library of components includes such functions.

3.3 Components

$$\begin{aligned} \mathit{Time} &= \mathbb{R} \\ \mathit{Sig}_{CT} &= \mathit{Time} \to \mathbb{R} \\ \mathit{Component}_{CT} &= \mathit{Sig}^{n}_{CT} \to \mathit{Sig}^{m}_{CT} \end{aligned}$$

Figure 4 – Formal definition of CT component.

While all components are signal transformations, it depends on the domain what a signal is. In the Continuous Time (CT) domain, signals are functions of time, and the signal transformation is a function transformation, i.e., a higher order function. A CT component can alter the time reference; a signal could be delayed, sped up, or even reversed if desired. Generally, a CT component can completely change the time at which the input function is evaluated, in addition to transforming the function in different ways. These characteristics enable to succinctly and accurately model physical processes, and can lead to models that intuitively match the physical process they model.

Figure 4 shows the formal definition of a CT component as in [Rov11]. Here, signals strictly go from time to a value in $\mathbb{R}$ (components can transform n input signals to m output signals). While this simplification makes it easier to discuss matter, it should be noted that it is still a simplification. For example, if we take our physical inertial mass from earlier, it is seen that this component does not output a value in $\mathbb{R}$, but rather a space vector, expressible as $\mathbb{R}^3$. It would still be possible to express this as three signals of $\mathbb{R}$, but this makes things needlessly complex. Therefore, while we will continue to discuss signals as having a result in $\mathbb{R}$, it should be noted that the codomain of a function can in practice be more general.

$$\begin{aligned} \mathit{Sig}_{DT} &= \mathbb{R} \\ \mathit{Component}_{DT} &= \mathit{Sig}_{DT}^{n} \rightarrow \mathit{Sig}_{DT}^{m} \end{aligned}$$

Figure 5 – Formal definition of DT component.

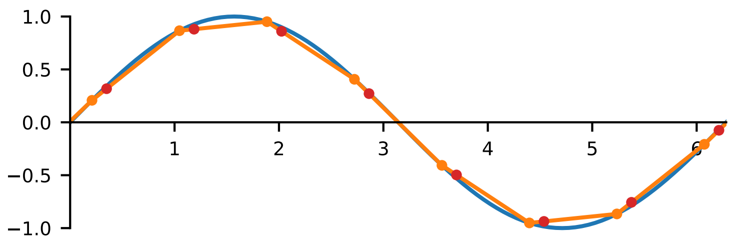
In the Discrete Time (DT) domain, signals are values at a discrete moment in time, as formalized in figure 5. The signal transformation is therefore a value transformation only. Signals are essentially piecewise linear functions of time that only change at discrete intervals; they are samples. However, time is no longer an explicit part of the signal, and a DT component can only access one signal value at a time. If it wishes to retain older values, it will need to do so explicitly. Moreover, it cannot access samples from the future. These limitations correspond to computations done in a discrete implementation (hardware or software).

[Rov11] discusses the DT domain in the context of hardware only. We will use DT components that end up being synthesizable FPGA code as well (which the thesis considers software).

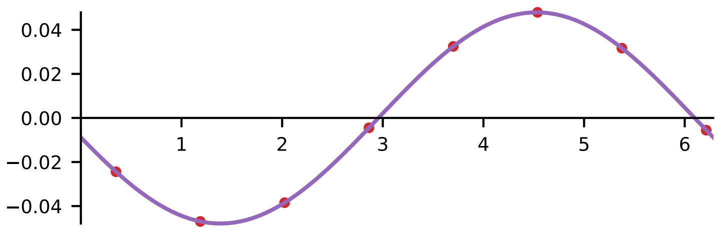
In the Dataflow (DF) domain, signals represent token updates to channels. Components are processes that have firing rules determining when to consume input and how long a computation takes. Through component composition, processes can communicate tokens to other processes. This formalization is used to represent computation in a more general fashion than the DT domain does.

3.4 Accurate continuous time

As has been mentioned, UniTi allows to accurately model continuous-time behaviour that is often the source of discretization errors in other simulation toolkits. A good example is time delays that are not a multiple of the discretization interval of the simulation. Suppose we want to model a system involving some wave propagating through a medium, like between a radio transmitter and several receiver antennas. There are applications where it is necessary to accurately model the time difference between the wave arriving at two or more



(a) The input signal.



(b) The discretization error.

Figure 6 – Linear interpolation of signal between samples.

antennas. It is possible to model the behaviour of the propagation of the wave through the medium by using time delays. The time delays model the different times of arrival at antennas. A time delay in a model merely delays a signal by a fixed amount of time. In a model of a system with several antennas, this represents the different lengths of time it takes for the radio wave to arrive at the antennas.

However, the system that is modelled is a continuous-time system as it deals with physical processes. Since the computer running the simulation is a discrete device, at some point continuous concepts need to be discretized, including time. Many simulation toolkits therefore globally discretize the continuous-time portion of the model, which means that all continuous-time processes are simulated at one common discretization interval. However, the time delays in the model will in general not be a multiple of this discretization interval. A common approach to simulate such a time delay is to linearly interpolate between two discretized samples. Due to the discretization, the only samples available are not at the desired point in time. The desired value is then linearly interpolated from the neighbouring available samples. The effects of the discretization error can be seen in figures 6a and 6b. The signal and the sampling are chosen to give a good illustration of the problematic effect. The blue sine signal of 6a is sampled at the orange marks, and the linear interpolation of these points results in the orange line. Due to a delay element, the actually relevant signal values are at the red marks, and the linear interpolation leads to a discretization error: the red marks are not on the blue signal line. Figure 6b shows the error made at each sample, and fits a sine wave on this error signal. This sine wave has the same frequency as the sampled signal, but a different phase. When we look at

Operator	Meaning
`>>>`	Sequential composition
`>>>*`	Sequential; replicate signal as needed
`||`	Parallel composition
`<^>`	Lift a function to a component
`>@>`	Feedback loop

Table 1 – Some common UniTi operators.

```
mySine () t = sin t
myDelay f t = f (t - 0.14)
myDelayedSine = (<^>) mySine >>> (<^>) myDelay
```

Listing 1 – Delayed sine wave in UniTi.

the signal in the frequency domain, the result is that the discretization error causes a phase shift in the sampled signal, in addition to a decrease in amplitude. Unfortunately, both phase shift and attenuation are frequency dependent. If we want to use the simulation to verify the operation of a new type of signal processing application, we observe small phase shifts and signal attenuations in the results. But we cannot distinguish whether this is merely a side effect of simulation errors, whether there is a mistake in the application, or both. The result is that there are models and situations that cannot be faithfully simulated.

We now discuss how this problem is avoided in UniTi. Since this will be illustrated through the use of code, table 1 provides a quick reference to some common UniTi operators, so the following code can be better understood.

Listing 1 shows how a delayed sinus wave could be written in UniTi*. Every continuous-time component gets as its arguments an input function and a point in time to be evaluated. The `mySine` function, though, is inputless, signified by the parentheses (an empty argument). It simply produces a sine wave based on its time argument. The `myDelay` function gets an input function f, and evaluates that function at an earlier moment in time. In other words, when `myDelay` itself is evaluated at a time t, the input function of `myDelay` is evaluated at a time $t - 0.14$ exactly. There is no need to interpolate, and hence no discretization error.

The component `myDelayedSine` is a sequential composition of `mySine` and `myDelay`, and in effect evaluates the function $\sin(t - 0.14)$. A more complex situation might contain two delay elements both fed from the sine wave, such

*This example code is written to give a sense of the concept; it is not idiomatic UniTi code, though it is strictly equivalent to it.

```
twoDelays = sine 1 1
            >>>*
            (delay 0.14 || delay 0.23)
            >>>
            (adc 0.1 || adc 0.1)
```

Listing 2 – Sine wave with two delays in UniTi.

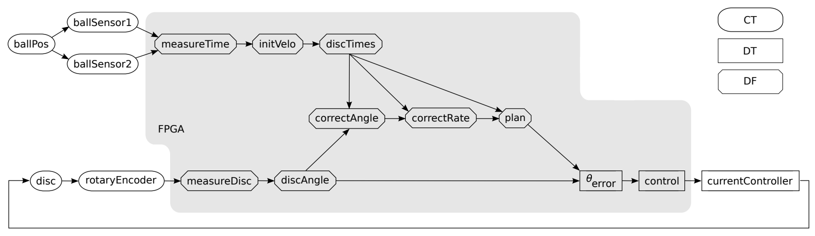


Figure 7 – All components of the UniTi model.

as in listing 2. This example uses the `sine` component from the library; its arguments are the amplitude A and angular frequency ω, both set to one. The sine is fed into two delay elements, and the `adc` components perform sampling of the signal with a time interval of 0.1 s.

Here a related property of the way UniTi deals with continuous time becomes apparent. If continuous time were discretized globally, and a single discretization interval would have had to be chosen to evaluate both delays without error, it would have to be at most 0.01 s, even though a sample is only needed every 0.1 s. But no such discretization is needed to evaluate the model, and the sine wave is only evaluated twice for each sampling interval of 0.1 s (once for each `adc`).

It is also perfectly possible to model varying time delays, where the delay is not a fixed length of time. It would not be possible to choose any single correct discretization interval for a varying time delay. This means that any simulation system employing a global discretization interval will necessarily be forced to interpolate when the time delay is not constant.

3.5 Composition difficulties

The composition operators used to compose components in UniTi were not very pleasant to work with. This is especially the case because, without working views, every component with interesting parameters needed a wire to the end of the graph. Good thought should be given to some alternate way of composition. The structure can become unwieldy and difficult to interpret for the designer with even a moderate amount of components and connections. I worked out my composition graphically with pen and paper, laying it out such that it could be translated to code easily. I was incapable of producing the code without resorting to pen and paper, and small modifications to the composition took a lot of mental effort. The feedback from the GHC compiler when one makes a mistake in the composition can be incredibly difficult to comprehend. This is often due to type inference going the wrong way. The compiler's type inference would infer the type of a definition from the one place where the actual mistake in the code was, and would subsequently complain about all the places where the definition was used correctly. It would, however, not complain about that one place where the mistake actually was. Often, it seems easier to just observe that *a* mistake was made and revert some changes until it type-checks again!

The way you can compose in CλaSH is much easier to work with. Figure 7 shows all components of the Tunnelling Ball Device model and how they are connected. The shape of the component box indicates in which of the three modelling domains they fall: a round box indicates continuous time, a box with

```
model =
    ballPos [(0, 460), (0.35,450)]
    >>>* (ballSensor1 || ballSensor2)
    >>>
    (   measureTime
        >>>
        initVelo
    )
    >>>
    (>@>)
    (   (   id
            ||
            (   disc
                >>>
                rotaryEncoder
                >>>
                measureDisc
                >>>
                discAngle
                >>>
                dup))
        >>>
        (compute || id)
        >>>
        thError
        >>>
        control
        >>>
        currentController
        >>>
        dup)

compute =
        (   (discTimes >>> repC3)
            ||
            id)
        >>>
        (id || id || correctAngle)
        >>>
        (id || correctRate)
        >>>
        plan)
```

(a) UniTi composition.

```
model = motorI
   where
      ballH    = ballPos
                   [ (0, 460)
                   , (0.35, 450)]
      bSens1   = ballSensor1 ballH
      bSens2   = ballSensor2 ballH
      dropT    = measureTime
                   (bSens1, bSens2)
      v0       = initVelo dropT
      discTh   = disc motorI
      rot0     = rotaryEncoder discTh
      mDisc0   = measureDisc rot0
      dAngle0  = discAngle mDisc0
      targetTh = compute (v0, dAngle0)
      errorTh  = thError
                   (targetTh, dAngle0)
      controlI = control errorTh
      motorI   = currentController
                   controlI

compute (v0, dAngle0) = targetTh
    where
      discTs   = discTimes v0
      corrTh   = correctAngle
                   (discTs, dAngle0)
      corrOm   = correctRate
                   (discTs, corrTh)
      targetTh = plan (discTs, corrOm)
```

(b) CλaSH style composition.

Listing 3 – Different ways of composing components.

straight corners indicates discrete time, and a box with angled corners indicates a dataflow component. Listing 3 compares the approaches of UniTi and CλaSH regarding how, ideally, the Tunnelling Ball Device would be composed of its constituent parts. For a quick overview of UniTi operators, see table 1. The `>>>*` operator in listing 3a replicates the single output of `ballPos` such that it is the input of both ball sensors. The `(>@>)` function creates a feedback loop in its argument. The effect of `(>@>)` is such that in this model, the output from `initVelo` is fed to the first `id` component and henceforth `compute`, and the output of `currentController` is connected to the input of `disc`; the output of `currentController` is also duplicated to serve as the output of the whole model. The code in 3a is already rather difficult to interpret. The actual code for the UniTi composition is pretty much incomprehensible, because of the views not working.

The CλaSH compositional syntax cannot be copied verbatim to the UniTi world. Furthermore, as evidenced by the recent, provisional dataflow operators in CλaSH which look just like the UniTi compositional operators, the solution falls short in the case where a connection has a bidirectional nature, such as backpressure. But the defining property of regular composition in CλaSH is that of *named signals*: every component is instantiated using variable names for its inputs and outputs, and connections are made by using the variable of some output as one of the input variables. This style makes a large composition more easily comprehensible, even though it hides the actual structure of the composition. The structure of the composition is better expressed graphically rather than in code, and a graphical editor would be another alternative for entry of compositions.

4 Modelling the device in UniTi

The whole Tunnelling Ball Device was first modelled in UniTi. In practice, this meant extending UniTi itself as well, especially in the area of data visualization. Graphical user interfaces are somewhat of a weak point in Haskell, and we encountered some typical issues. We could not get wxHaskell to work properly under the Linux OS that was used at the time, and similarly we could not get GTK+ to work properly under OS X. So both toolkits were used, providing either alternative.

4.1 Model overview

Figure 7 shows all components of the Tunnelling Ball Device model and how they are connected. All components that are in the shaded area are components that will be implemented in the FPGA controlling the device. Components outside the shaded area model parts of the physical world that surrounds the FPGA. All these components model the physical components of the Tunnelling Ball Device, including the balls.

`ballPos` This component simulates the movement of the balls. At any time, it outputs the current height of a dropping ball. When there is no ball currently in flight, it outputs the special value `Nothing`.

`ballSensor` These two components simulate the optical sensors that signal a ball passing through the fork of the sensor. The input to the components is the height of the ball, the output is a signal that is active when a ball is passing through the fork.

`measureTime` This component measures how much time passes between a ball passing the top and the bottom sensors. This measurement completely determines the path of the ball, with the assumptions done on that movement.

`initVelo` Based on the measurement done, the velocity with which the ball passed the bottom sensor is calculated. Implicitly, this also defines the moment in time when the ball passed the bottom sensor.

`discTimes` This component computes when the ball will be vertically centred within the disc, and when the ball has completely fallen through the disc. It follows that at the time the ball is vertically centred, one of the holes of the disc should be presented such that the ball goes through the hole.

`correctAngle` This component computes the total amount of correction that is necessary to position a hole at the correct place when the ball is going through the disc, based on the current locations of the holes in the disc.

`correctRate` This component computes the change in velocity that will effect the needed correction in the available amount of time before the ball will fall through the disc.

`plan` This component computes the desired trajectory of the disc. If a ball is dropping towards the disc, it will use the velocity computed by `correctRate`, until the ball has completed its fall through the disc as determined by discTimes. At any other time, it will use the default rotation speed as specified by the human operator. The output is the desired current angle of rotation of the disc.

`disc` This component is a model of the whole assembly of the motor and the disc. The input to the component is the electric current that runs through the windings of the motor. This current is (assumed to be) directly proportional to a torque produced by the motor. In the mechanical domain, the rigidly connected whole of the rotating parts, including motor rotor, is modelled as a rotating inertial mass experiencing this torque and a certain amount of friction. The output is the current angle of rotation of the disc.

`rotaryEncoder` The rotary encoder is a sensor mounted on the motor that encodes the angle of rotation of the motor axle as a digital signal. Since the disc is rigidly mounted on the motor axle, it effectively also senses the angle of the disc. The UniTi component is a model of that sensor. Its input is the angle of rotation of the disc, and its output is a signal that corresponds to the three digital signals of the rotary encoder.

`measureDisc` This UniTi component takes the raw digital signal from the rotary encoder, and interprets it to produce a series of events that describe information obtained from the sensor. Each full revolution of the disc produces 1000 of these events equally spaced along the revolution.

`discAngle` The rest of the model needs to know the position of the disc at any time, not just at the events `measureDisc` produces. This UniTi component interpolates the position in between events, for an estimation of the actual position at any time.

$\theta_{\texttt{error}}$ This component (called `thError` in the code) takes the current position of the disc and the desired position of the disc, and subtracts them to get the error in the position.

`control` This component computes the current that needs to flow through the motor windings such that the error signal eventually goes to zero. When the error is zero, the disc is in the desired position. When the error is near zero, it is still good enough. A large error at the moment the ball would tunnel through the disc would mean the ball hits the disc instead. The input of the `control` component is the current error in the position of the disc, and the output is a desired electric current to run through the motor windings.

`currentController` This component models the device that drives the motor. It is an off-the-shelf motor controller that is used as a current source. The FPGA generates a PWM signal that is in direct relation to the desired current (computed by `control`), and the `currentController` interprets the PWM signal and will achieve the desired current through the motor windings.

4.2 Incremental refinement

The model-based design approach in UniTi allows to start out with just the essence of the project, and to progressively refine the model until all the detail is

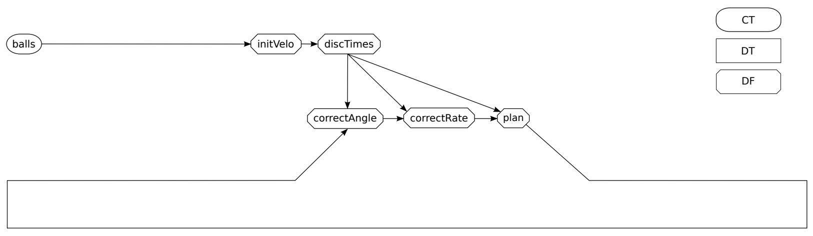


Figure 8 – The model of figure 7 reduced to the bare necessities for verifying the algorithm.

fleshed out. During all the steps, simulation can be used to check if the behaviour is still as expected.

For the Tunnelling Ball Device, a good start is to only verify the basic strategy for aligning the disc with the falling ball. This is depicted in figure 8: relative to figure 7 only the components related to computing the desired trajectory are left in, and the `ballPos` component is slightly modified.* In the eventual model, it only outputs the height of a ball. In this modification, it also outputs the velocity of the ball; data that was internally available anyway, but not communicated by the component. The `initVelo` simply outputs the reported velocity the moment the ball drops to the level of where the bottom sensor is located.

The final twist in the design is that the *actual* position of the disc is the *desired* position of the disc, linking `plan` directly to `correctAngle`. All components in between can be introduced at a later time.

In this basic form, the ball should tunnel if the algorithm is correct. From here, finer detail can be added. For instance, the models of the sensors can be added. Based on those sensors, the components that will end up in the FPGA and interpret the sensor data can be added and verified.

A step that can be done separately from the other parts is the design of the disc model and a controller to accurately steer this modelled disc. In fact, it makes sense to make a model of the system both with `rotaryEncoder`, `measureDisc` and `discAngle` and a model without those components. The sensors and the interpolation add some unwanted but unavoidable behaviour to the feedback loop, and a proper controller needs to be resistant to this in order to work in practice.

To test the behaviour under various circumstances, the disc model and controller were given completely different reference trajectories rather than those produced by `plan`. Especially reference trajectories that correspond to the most difficult trajectories `plan` could produce provide useful data without having to come up with ways to drop simulated balls at precisely the worst times for the control loop. In this way, it was also possible to determine the minimum height difference between the disc and the bottom sensor, which defines the time available to perform a full correction of the trajectory.

* While this way of phrasing it comes natural because in this report, figure 7 was presented first, in reality the model of figure 8 came first, and the model of figure 7 was only produced later on in the project.

4.3 Ball drop sensors

Due to the nature of modelling in UniTi, modelling a sensor is very intuitive. The photoelectric sensors that detect a dropping ball can in their most basic form be modelled as delivering a digital electric signal that is high when a ball is obstructing the light beam, and low when there is no ball obstructing the light beam. As input for the model of a photoelectric sensor, one takes the current height of the ball. The photoelectric sensor itself has a parameterized height it is mounted at as well. The radius of the ball is a known constant in the model. The output of the UniTi sensor component is simply 1 when the height of the ball is such that it obstructs the light beam, and 0 otherwise. Should it be necessary for accuracy, the component could later be refined to have an analogue output signal with rise and fall times; however, this was not necessary for this model.

Of course, at any time there might not be a ball that is currently falling. This is captured by making the height of "the ball" of the Haskell `Maybe Height` type, which assumes the value of `Nothing` when there is no ball currently falling. `Height` is just a type synonym for `Double`, used as a type wherever the variable signifies a height relative to what is defined as the ground, in millimetres.

4.4 Disc angle sensor

The rotary encoder is the sensor that is mounted on the axle of the motor, responsible for sensing the angle of the axle. Since the disc is mounted rigidly to the motor, the rotary encoder effectively senses the angle of the disc.

4.4.1 Abstract model of the sensor

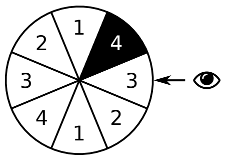


Figure 9
Mental model of rotary encoder with 2 counts per turn and an index signal in the black sector.

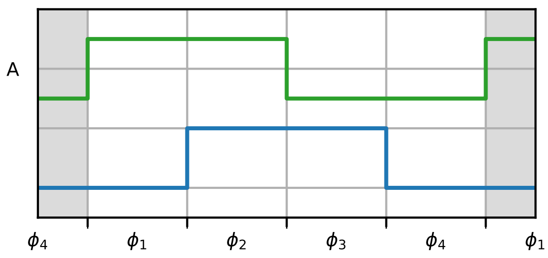


Figure 10 – Output signal of the incremental outputs of the rotary encoder.

At a conceptual level, the rotary encoder contains a circular disc mounted on the axle, and this disc is divided into 500 equal parts, called *counts*. Each count is further subdivided into four equal parts, giving rise to a total of 2000 equally sized circular sectors. Figure 9 is an illustration of the concept for a rotary encoder with 2 counts per turn instead of the 500 of the actually used sensor. Two sensor outputs of the encoder are incremental outputs: they signify the disc rotating from one sector into the next. To this end, the 500 counts are subdivided into four *phases*. As the disc rotates clockwise, the four phases pass in the order $1, 2, 3, 4, 1, 2, \ldots$; that is, in ascending order. If the disc rotates anticlockwise, they pass in reverse order: $4, 3, 2, 1, 4, 3, \ldots$. Each phase corresponds to a specific combination of low and high electrical levels on the two incremental outputs. As the disc rotates to the next phase, the incremental outputs change. In figure 9, the eye corresponds to the sensor output, and it currently sees phase number 3. Figure 10 shows the incremental outputs and their correspondence to the phases. Since a steadily rotating disc outputs two square waves which are 90° out of phase, and since such signals are said to be in quadrature, this type of rotary encoder is also referred to as a quadrature encoder.

4.4.2 UniTi model

To model the rotary encoder in UniTi, the process is again very straightforward. The input to the UniTi component is the current angle of the disc. The input angle is not required to lie in the range 0 to 2π; the computations in the UniTi rotary encoder component will treat the angle as wrapping around at that point. The 2π rad of a full circle are split into 2000 equal parts, denoted as *sectors* in the code. The input angle is converted to a number in the range of 0 to 1999. To get the incremental outputs, the modulo operation further reduces this to a repeated pattern of 0 1 2 3, directly corresponding to one of the four phases of the output signal. This directly maps to an output signal for the incremental outputs.

There is one more signal on the rotary encoder: the *index* signal. The two incremental outputs only register change, but not absolute position. To remedy this, one of the 2000 equal circular sectors in addition produces an electric pulse on the third output signal of the rotary encoder, the index output. In other words, once every full revolution, the index output is electrically high for as long as the angle is within that specific angular sector. In figure 9, the index sector is indicated in black. The index output is electrically high when the black sector passes in front of the eye.

This index signal is the reason why the previously mentioned computation produces a sector, rather than immediately reducing it to the four phases. At a specific sector, the index output is high.

The UniTi component that models the rotary encoder is a very simple one-to-one relation to how one visualizes the operation of the actual sensor. It is very straightforward to model, giving the designer confidence in the correctness of the model.

4.4.3 Phase shift

The UniTi component conforms to a conceptual model of the rotary encoder. In reality, the sensor is built somewhat differently. In the idealized model just described, the A and B outputs are exactly 90° out of phase. In reality, manufacturing tolerances cause this phase shift to vary a bit. The index output however is synchronized to either the A or the B output; so it is exactly in phase with one of the two. Either the datasheet of the rotary encoder or experimentation should make clear which incremental output it is synchronized to.

Since it is easily seen that the computation done on the signal is actually insensitive to this phase shift, it was not modelled. Modelling the phase shift would be as easy as computing A and B separately and shifting the input angle of one of the two.

4.4.4 Location of the zero point

There is some debouncing for noise in the output of the rotary encoder in the code that interprets the sensor output, which shifts the computed zero point of the disc. Additionally, the location of the index pulse with regard to the phase of the incremental outputs is important for correct operation. This is also noted in the comments in the code of the function `Model.Measure.measureDisc`. If the index pulse is not in ϕ_4, the code needs to be changed.

To compensate for the shift in the computed zero point, the input angle is used with a small offset in the actual code. The deliberately added offset in the UniTi component that models the sensor causes the $0°$ point to be the same both in the UniTi model of the motor and in the computations done with regard to the interpreted sensor output. This way, a particular angle means the same to all parts of the model; it is an arbitrary point in any case. The $0°$ point is put at the point that the index sensor signal is recognized, which is slightly after it is generated. Note that the $0°$ point is therefore recognized at different disc angles depending on the direction of rotation of the disc; it is defined as $0°$ for a clockwise rotation. In anticlockwise rotation, it needs to be corrected by a constant factor to account for the delay, since, unfortunately, delays in processing generally do not change into an ability to look into the future when you change direction. The factor is a constant angle of one count of the rotary encoder.

4.4.5 Modelling noise

The noisiness of the sensor outputs was not modelled in UniTi, but doing so is probably best done not by changing the existing model, but rather adding to it. The outputs of the basic model as it is now could be passed through a second function that adds random amounts of bouncing on signal transitions. If necessary for accuracy, the length of the bounces could be based on the angle input to the sensor, concentrating the bounces in a small sector around the transitions. The large benefit of adding to the existing model rather than changing it, is that it is easy to verify that the basic operation is not modified, and different effects are handled separately.

4.4.6 Room for experimentation

It was noted before that the angle that is the input of the model of the rotary encoder need not be in the range 0 to 2π. Effectively, the model is liberal in what it accepts. This turned out to be a beneficial programming tactic. Small changes in models of other components sometimes caused the angle to be slightly outside the range, but still correct. It was very helpful that the model as a whole did not break because of it. In the final, completed model, it makes sense that all signals are in a well-defined range, but while one is working, refining, and especially experimenting, it is useful that preconditions on data are not overly strict, as long as the data is still well-defined.

4.5 Disc model

The whole of the motor and the disc is modelled as one rotational system, with the current through the motor windings as its input. This current is assumed to

Property	Symbol	Value	Unit
Torque constant	K_τ	2.58×10^{-2}	$\mathrm{N\,m\,A^{-1}}$
Inertia	J	2.23×10^{-5}	$\mathrm{kg\,m^2}$
Coulomb friction	R_C	3.02×10^{-5}	$\mathrm{N\,m}$
Viscous friction	R_v	2.44×10^{-6}	$\mathrm{N\,m\,rad^{-1}\,s}$

Table 2 – Disc model parameters.

correspond linearly to a torque produced by the motor, with the linear relation expressed by the parameter K_τ. The mechanical model is one of an inertial mass with friction. For system identification, a constant current was applied to the motor windings and the system was allowed to reach equilibrium. The stabilization process had the shape of exponential decay.

The parameters of the disc model are summed up in Table 2.

Since the torque constant of the motor is given in the datasheet, the constant current corresponds to a known constant torque. In equilibrium, the torque applied by the motor is equal in size to the torque caused by the friction. The system was observed for several different constant currents. This revealed that the way friction was modelled in [Jen10] was insufficient for our purposes. The friction model in [Jen10] is a linear relation between velocity and the friction as a torque. Drawing the points obtained the way just described in a graph of friction and velocity, a straight line fitting the points did not go through the origin of the graph.

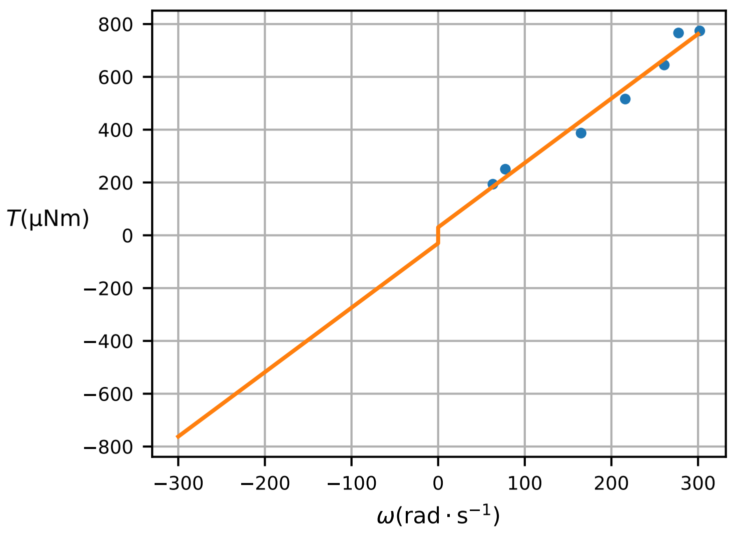
Instead, the model chosen is one with Coulomb friction (R_C) as well as viscous friction (R_v). That is, a certain, constant dry friction has to be overcome before the disc will start spinning from standstill, and additionally there is a viscous component that is linear in the velocity of the disc. The friction function is thus:

$$T(\omega) = R_C \cdot \mathrm{sgn}(\omega) + R_v\omega \tag{1}$$

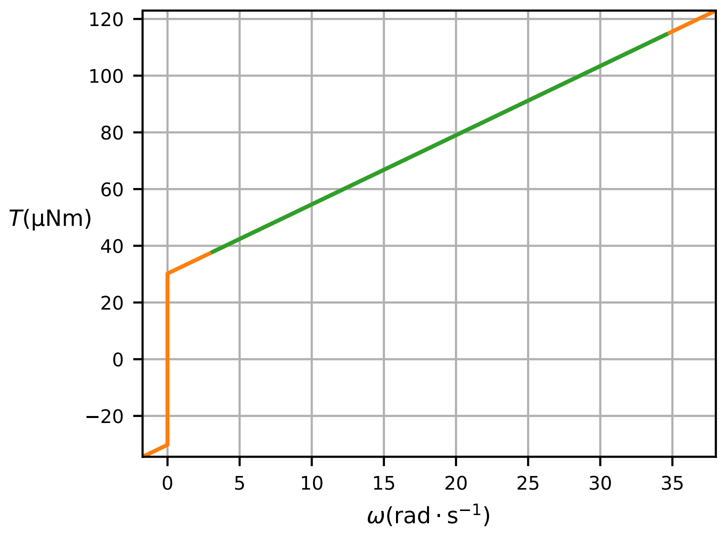
The result can be seen in Figure 11a. Figure 11b shows the modelled friction for the velocities involved in a realistic situation. When the disc is running at 3 rotations per second and corrections need to be performed in 100 ms, the green part of the relation is the range of velocities used by the system. The Coulomb friction is a large part of the total friction.

The inertia of the system could be computed, as all necessary data is already available or, in the case of weights, easily measured. But the system identification that was used to determine the friction also gave an easy way to measure the inertia directly. Figure 12 shows the velocity as a function of time. The first part of the blue line is the response when a constant current of 29.7 mA[*] is applied to the motor windings, starting from standstill at $t \approx -54\,\mathrm{s}$. Equilibrium is reached at a velocity of $277.5\,\mathrm{rad\,s^{-1}}$. At $t = 0\,\mathrm{s}$, the current is lowered to 9.7 mA while the disc is running at $277.5\,\mathrm{rad\,s^{-1}}$. When linearity is assumed for the friction in this velocity range, the blue line should have the shape of an exponential function with a time constant τ from $t = 0\,\mathrm{s}$ onwards. An exponential function

[*]The intention was to use 30 mA, but there was a problem with a deviation in the control signal.



(a) Full range.



(b) Close-up.

Figure 11 – Friction model of the disc. The blue dots are the measured datapoints. The green part is the used velocity range at 3 rps.

appears to be a reasonable approximation, and $\tau = 8.65\,\mathrm{s}$. Because the inertia J can be calculated through $J = \tau R_v$, the net inertia is $2.23 \times 10^{-5}\,\mathrm{kg\,m^2}$.

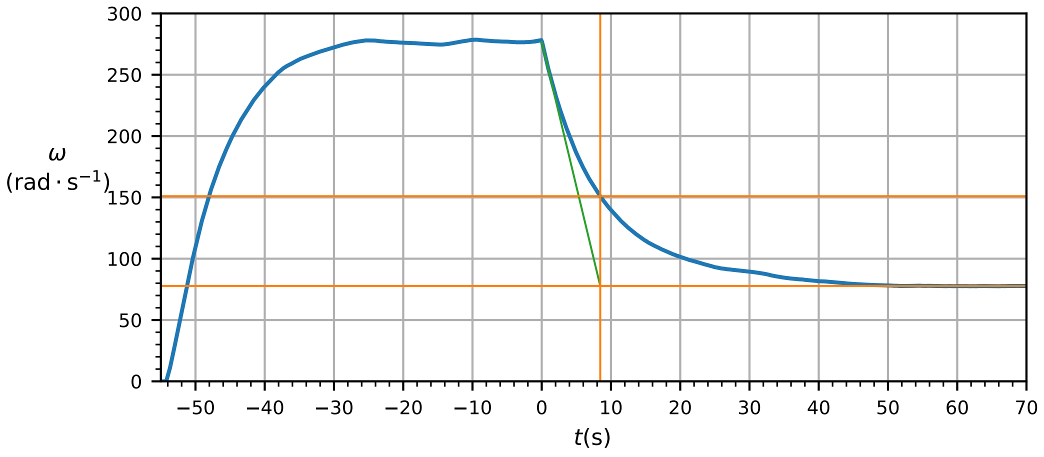


Figure 12 – System identification. System response to a constant current.

5 Motor control

As mentioned before, in the model of the combination of disc and motor, the input is a current. The off-the-shelf motor controller that is used is configured to accept as its input a PWM signal that has a direct relation to a desired motor current. The motor controller is capable of delivering 4 A. However, the PWM signal is quantized by the motor controller; from observation, it would seem that it quantizes the 0–100% PWM signal range to a 10-bit value. Additionally, while most of the time the signal is relatively stable, quite often the signal fluctuates significantly even though the input signal is kept the same. What it comes down to is that the practical resolution of the signal is far from the full 10 bits.

As per the documentation, the valid input range for the signal is 10–90% PWM, further limiting the resolution. As shown by experimentation, trying to use the whole range of current that can be delivered by the controller results in really imprecise current control, so the range of current is limited to $-0.5\,\mathrm{A}$ to $0.5\,\mathrm{A}$. It turns out even the largest corrections that occur are still possible with the range being limited as such.

In the FPGA an algorithm computes the desired current through the motor windings to realize the desired trajectory. Currently, the only input to this algorithm is the positional error of the disc with respect to the desired trajectory. The desired trajectory is an idealized trajectory that cannot be physically realized: it produces a step change in the rotational velocity, which is physically impossible with an inertial mass and finite current. As such, the positional error will always become non-zero when such a step change in velocity occurs, and the algorithm will strive to reduce this error back towards zero.

The current algorithm was written as a placeholder until a properly designed algorithm was developed. The algorithm turned out to be able to keep the disc under control and achieve the tunnelling of the ball. Even at speeds of $100\,\mathrm{rad\,s^{-1}}$, it is able to perform the largest possible positional correction of $\frac{\pi}{2}\,\mathrm{rad}$ in time for the ball to go through. In the end, developing a properly designed algorithm was dropped from the project, and the placeholder is kept, but not documented further.

If a replacement control algorithm needs a realizable reference trajectory as input, this can be achieved with relatively minor modifications to the surrounding code. For instance, the velocity changes could be made to follow a sigmoid curve, which is a class of curves having a characteristic smooth “S”-shaped curve.

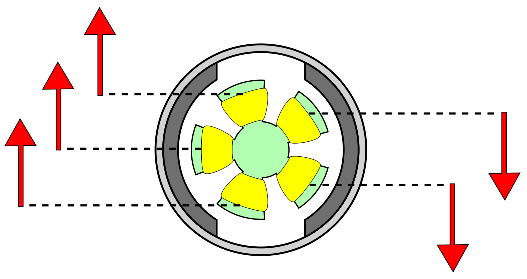


Figure 13 – Forces from field interaction in PM DC motor.

6 Choice of motor

6.1 Brushed DC motor

A *permanent-magnet brushed DC motor*, or PM DC motor for short, is a very basic motor design that used to be very common in a lot of applications, including toys [Eleb], due to its low cost and ease of operation. In many applications, it is gradually being replaced by the brushless motor, but it is still a common design.

Forces between permanent magnets and electromagnets propel the rotating axle of the motor. Figure 13 gives a schematic example of a possible configuration for a PM DC motor.

In a PM DC motor, the permanent magnets are in the *stator* of the motor: the part that doesn't rotate. The stator is rigidly connected to the motor's mounting flange (where the motor housing is connected to the device it is a part of). In figure 13, the stator is depicted as a dark grey magnet with a light grey housing. The *rotor*, which contains the rotating axle, has windings that form the electromagnet [Elea]. In figure 13, the rotor is depicted in green, with yellow windings. Electric current through the windings will interact with the magnetic field of the permanent magnets. This will cause a force to be exerted on the rotor. By controlling which of the windings have current, and by controlling the direction of that current, the forces can be made such that they work to propel the axle in the desired direction. In the figure, a possible example of the resulting forces on each of the yellow windings is depicted by a red arrow. In the example, all windings are powered and propelling the rotor to turn clockwise in the picture. The force exerted on a winding is always perpendicular to the magnetic field [Elea].

In more advanced motor designs, this selection of which windings to power and which direction the current flows is often handled by electronics. In a brushed DC motor, the selection is done completely mechanically. The direction of current flow is determined relative to the direction of current flow through the two connecting terminals of the motor; it is still possible to reverse all currents through the windings by reversing the current through the terminals, meaning the torque exerted on the axle will also reverse as a whole.

Since the windings on the rotor rotate themselves, it would be impossible to just connect electrical wire from the outside to the windings, since then the windings would no longer be able to rotate. Instead, there is a mechanical *commutator* that consists of a stationary part with brushes and a rotating part on the axle. A schematic view of a commutator can be seen in figure 14.

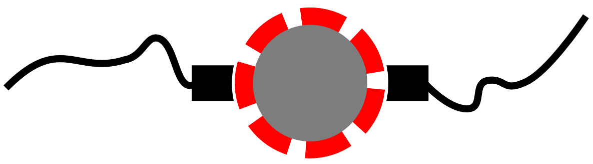


Figure 14 – Schematic view of a commutator. Black: brushes with connecting wire. Red: commutator bars. Grey: axle.

The stationary brushes are connected to the electrical terminals of the motor and slide over the rotating part of the commutator. They allow electrical current to flow to the rotating part of the commutator.

The rotating part of the commutator is split into segments, and these segments individually connect to one of the two brushes at specific parts of a full rotation of the axle. One segment is called a *commutator bar* and is indicated in red in figure 14. By wiring up the multitude of windings on the rotor to the correct segments of the commutator, the forces between the permanent magnets and any powered rotor windings are always in the desired direction of rotation. The direction of the forces can be flipped by reversing the current that is fed through the motor as a whole.

To a large extent, the transduction behaviour of the permanent-magnet brushed DC motor is nicely linear: electrical current is transduced into mechanical torque by a linear relation,

$$\tau(t) = K_\tau i(t)$$

where $\tau(t)$ is the torque produced by the motor in newton-metre and $i(t)$ the current fed through the motor in ampere. K_τ is a constant, the *torque constant*, and is determined by the construction of the motor. This linear relation has desirable properties for control. The complete characteristics of a DC motor have a lot more components, though, for instance winding resistance, mechanical friction and *back electromotive force* (back-EMF). Back electromotive force describes the phenomenon that a rotating motor induces a voltage in its windings because of Faraday's law of induction. With increasing back-EMF (i.e., increasing speed), keeping the current through the windings constant requires increasing the voltage across the motor terminals. Thus while the relation between current and torque is almost ideal over a large range of operation, the relation between voltage and current or torque is more complicated.

The rotor of brushed DC motors can be designed in several ways. Conventional design is to have the windings wound on a shaped iron core. The iron core, which is also the basis of the rigidity of the winding assembly, is magnetized by the windings and functions as part of the electromagnet. The example in figure 13 has this construction, and the iron core is indicated in green. Because independently of the magnetization by the windings, the iron itself also interacts magnetically with the permanent magnets on the stator, an undesired phenomenon called *cogging* occurs [Max12]. This is a torque ripple that occurs as the rotor rotates, leading to vibration, noise, a tendency to stop at preferential positions, and more in general non-linear disturbances in the control. By orienting the slots cut into the iron diagonally, cogging can be reduced, but not eliminated [Max12].

An additional drawback of the iron core is that it increases the inertia of the rotor due to its relatively high weight and diameter.

As an alternative, it is possible to construct the windings such that no core is needed at all, with the windings supporting themselves [Max12]. This does not have the aforementioned drawbacks that the iron core poses; on the other hand, it will heat up quicker and is less resistant to high temperature than the solid iron core, limiting the duration and magnitude at which the motor can be operated above its continuous maximum operating conditions.

The coreless motor even exceeds the conventional model with core regarding the linearity of its transduction behaviour, as saturation effects create non-linearities in the core at high currents.

A drawback of the brushed DC motor compared to other motor types is that the making and breaking of a current path between the brushes and the contacts on the commutator leads to sparking and electromagnetic emissions. The windings on the rotor have a non-negligible inductance which will oppose the breaking of the current flow, causing a sharp rise in voltage across the winding when the current is broken off. Thus sparks can form. Additionally, the brushes will wear down with use, limiting the time a motor can go without servicing. Because wear and sparking greatly increase with high speeds, it also effectively limits the maximum speed the motor can be operated at [Max11].

Variants without permanent magnet

Using a permanent magnet for the stator can also be limiting, as one is restricted to the available magnetic materials and their intrinsic properties. Beside the permanent-magnet brushed DC motor, there are also brushed DC motors where the magnetic field is not provided by a permanent magnet but by an electromagnet. This electromagnet in the stator is called the *field* electromagnet.

Separately excited motor. When the field electromagnet is not electrically connected to the windings of the rotor (the *armature* windings) but has its own terminals, it is called a separately excited motor. In this case, the torque produced by the motor can not only be controlled through the current flowing through the rotor, but also by varying the flux generated by the field windings. Both are proportional to the torque, provided the other is kept constant [Elee].

Series wound motor. There are several ways of connecting together field windings and armature windings such that only two motor terminals are needed. When they are connected in series, the motor is a series wound motor. In this type of motor, the current that flows through the field windings is equal to the current through the armature windings, where in differently constructed DC motors the current through the field windings is generally lower. This results in a higher heat generation in the series wound motor, but it also enables very high torque generation due to the high flux generated by the field windings. Therefore, this motor is generally used when short but powerful bursts of torque are needed, like in starter motors [Elec].

Shunt wound motor. When field and armature windings are connected in parallel, the motor is a shunt wound motor. The desirable property that emerges from this construction is a good *speed regulation*: with a constant voltage applied to the motor terminals, the motor is relatively insensitive to load changes. Due to the parallel connection, the flux generated by the field windings remains constant, which means there is no change in the rotation speed due to flux changes. On the other hand, a heavier load decreasing the speed will also decrease back-EMF in the armature windings, which means the current through the armature windings increases. This increase in current partially compensates the decreasing speed. Speed regulation is beneficial when the control of the motor is fairly basic and there are no advanced speed control mechanisms to compensate variations. In the series wound motor, the properties of constant flux and increasing current do not hold, which makes the shunt wound motor a better choice where speed regulation is relevant [Eled].

References

[Baa15] Christiaan Pieter Rudolf Baaij. *Digital circuit in CλaSH: functional specifications and type-directed synthesis.* PhD thesis, University of Twente, Netherlands, 2015.

[Cla] Clash-lang.org. Clash: Home. URL: `https://clash-lang.org/` [cited September 8, 2015].

[EKAGMS94] FM El-Khouly, AS Abdel-Ghaffar, AA Mohammed, and AM Sharaf. Artificial intelligent speed control strategies for permanent magnet dc motor drives. In *Industry Applications Society Annual Meeting, 1994., Conference Record of the 1994 IEEE*, pages 379–385. IEEE, 1994.

[Elea] Electrical4u.com. *DC Motor or Direct Current Motor.* URL: `http://www.electrical4u.com/dc-motor-or-direct-current-motor/` [cited May 3, 2016].

[Eleb] Electrical4u.com. *Permanent Magnet DC Motor or PMDC Motor — Working Principle Construction.* URL: `http://www.electrical4u.com/permanent-magnet-dc-motor-or-pmdc-motor/` [cited April 29, 2016].

[Elec] Electrical4u.com. *Series Wound DC Motor or DC Series Motor.* URL: `http://www.electrical4u.com/series-wound-dc-motor-or-dc-series-motor/` [cited May 3, 2016].

[Eled] Electrical4u.com. *Shunt Wound DC Motor — DC Shunt Motor.* URL: `http://www.electrical4u.com/shunt-wound-dc-motor-dc-shunt-motor/` [cited May 3, 2016].

[Elee] Electrical4u.com. *Types of DC Motor Separately Excited Shunt Series Compound DC Motor.* URL: `http://www.electrical4u.com/types-of-dc-motor-separately-excited-shunt-series-compound-dc-motor/` [cited May 3, 2016].

[Jen10] Jeff C. Jensen. Elements of model-based design. Master's thesis, EECS Department, University of California, Berkeley, Feb 2010. URL: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-19.html`.

[Max11] Maxon motor ag. *Maxon DC motor and Maxon EC motor — Key information*, April 2011.

[Max12] Maxon motor ag. *Maxon Academy - DC motor*, 2012.

[RAH97] M Azizur Rahman and M Ashraful Hoque. Online self-tuning ann-based speed control of a pm dc motor. *Mechatronics, IEEE/ASME Transactions on*, 2(3):169–178, 1997.

[Rov11] Kenneth Christian Rovers. *Functional model-based design of embedded systems with UniTi.* PhD thesis, University of Twente, Netherlands, 2011.

[WES91] Siri Weerasooriya and MA El-Sharkawi. Identification and control of a dc motor using back-propagation neural networks. *Energy Conversion, IEEE Transactions on*, 6(4):663–669, 1991.