Microsoft

Device Calibration Playbook

Project Socrates

Published: December, 2018

For the latest information, please see
bons.ai

Introduction

Calibration is the process of adjusting parameters on a physical device to ensure it functions properly or improve its behavior. The usual tuning procedure involves commanding the device to execute a defined target position or trajectory, extracting the error between the target and the actual positions, then adjusting the parameters to reduce the error. The challenge is that in many real-world mechanical devices, the differential equations or dynamics that relate input and output of the device are not readily available without proper system identification, so finding good calibration parameters requires search, often guided by a human technician's experience and intuition.

Thousands of devices are calibrated around the world every day. Difficult calibration problems can take many hours or days when done manually, bringing equipment offline or delaying new devices from being shipped to customers. There are three main reasons for delays: getting an available technician to the device, which may require flying them out from some central location, each trial of calibration parameters can take a long time, and many such trials can be required to get good parameters. Moreover, when the number of parameters to be tuned is high, substantial book-keeping of changes or prior experience of parameter sensitivities customized to machine under consideration is required to tune faster. Automating calibration can:

1. Avoid the need to fly out technicians and can reduce the number of trials needed, greatly reducing the overall time.
2. Increase overall precision by more frequent automated calibration

Traditional methods of calibration work well to get parameters for a particular scenario but are limited in generalization. Machine teaching and reinforcement learning enables robust automated calibration: making a calibrator that can quickly calibrate a range of devices for a variety of scenarios.

This playbook focuses on ways to use machine teaching and reinforcement learning to create automatic and robust calibrators for fast calibration of existing controllers. Reinforcement learning and machine teaching can also be used to adaptively re-calibrate controllers continuously during device operation, to augment the output of existing controllers, or to entire replace traditional controllers with learned controllers. These approaches can provide great results in some applications but are out of scope for this playbook.

Example Calibration Problem

There are many variants of calibration problems. Here are several illustrative examples. We will refer to these later.

- Calibrating the friction compensator on a CNC machine to reduce friction error to below 1 micron.
- Calibrate an adaptive controller for the steering angle in an autonomous vehicle to provide robust, responsive, and stable control.
- Tuning a set of PID controllers in a bulldozer to obtain flat cuts across the ground. The needed parameters vary based on driving speed and material being cut.
- Calibrate a PID controller for engine throttle control.
- Configure a reactor for making plastics. The needed parameters can depend on humidity, temperature, and many other factors.

Variants of Calibration Problems

The simplest definition of the calibration problem is to find a set of calibration parameters *p* that make a device perform a specified task with sufficiently low error, for an appropriate definition of error. Depending on the application, "error" can include the maximum deviation from a specified target, cumulative deviation over time, amount of overshoot, controller response time, etc. If we define the error to be a function *e(p)* of the calibration parameters, the problem becomes

$$p^* = argmin_p\, e(p)$$

This definition can be refined based on the specifics of the calibration problem to be solved. This section describes dimensions in which calibration problems can vary, to establish a vocabulary we'll use in describing ways to solve these variants. It is important to understand which variant applies to your problem to select the right machine teaching approach.

Device Variation

The first dimension is the variability in the *devices* that must be calibrated. This variability usually stems from hard-to-observe differences in physical properties of the device: slight differences in component sizes, masses, viscosities of lubricants, friction between components, imperfections in sensors, etc. We will call such parameters, which can be modelled in simulation, but not easily measured in real devices, *hidden parameters*.

Single Device

One case is where you need to calibrate a single device – for example, a custom manufacturing machine. Even a single device may need periodic re-calibration because its hidden parameters and thus its behavior changes over time, e.g. due to wear and tear.

Multiple Devices

The more general case is that there are many devices, that are similar to each other, but have variation in hidden parameters arising from manufacturing tolerances, slight differences in assembly, and wear and tear. Most mass-manufactured devices fall in this category.

Multiple device classes

Even more general is having several *device classes*. For example, you may want to calibrate different models of excavators, or different models of CNC machines. Each model defines a device class, and there may be significant variation in behavior within a single class as well as across classes. The reason to distinguish device classes from the intra-class variability is that device class is usually known, and is available as a potential input into the calibration process.

Scenario Variation

The second dimension is the range of *scenarios* the devices need to be calibrated for. This encompasses changes in the external environment and differences in the task to be performed with the device.

The simple case is when the device only does one particular task and is not subject to significant external variation. An example would be a piece of manufacturing equipment that only makes one kind of part at a time.

In some cases, there is a range of scenarios, but it is known that if the device works well for one of them, it will work adequately for others as well. This is the case in many applications. For example, in CNC milling, the friction compensation component is calibrated with a circle test – if the machine can cut a close-to-perfect circle, it will also cut more complex shapes accurately. Many other applications use a step function test, where the device is commanded to immediately move from one state to another (e.g. an engine going from 2000 rpm to 3000 rpm), and the controller is tuned to do this quickly without overshooting or oscillating.

The more complex situation is where there are many scenarios that are significantly different, and some parameters may work well in one scenario but poorly in others. In our excavator example, an excavator must be able to work in a range of different materials in different conditions and at different speeds – quick cuts in dry sand, slow cuts in wet mud, when it's very warm or very cold, etc. It may not be possible to find a single set of parameters that works well enough across all these scenarios. This leads to our next dimension of variation…

Scope of Calibration

In some applications, the goal is to find parameters that work well across a range of devices and scenarios. In others, there is the need and ability to calibrate individual devices, even for specific scenarios. This section describes these options, since the first part of machine teaching for calibration problems is defining the range of calibration situations the BRAIN needs to learn to handle, and the desired domain of applicability of each set of parameters found.

Shared parameters

Here, we want to find a single set of parameters that works well across devices and scenarios. Mathematically, this can be stated as reducing the max error across device and scenario variation:

$$p^* = \ argmin_p \max_i e_i(p)$$

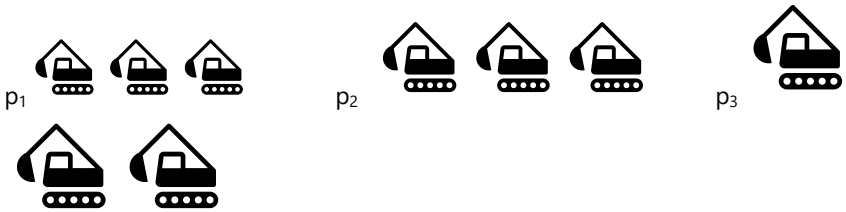
Per-device parameters

Here, there is the opportunity and need to find separate parameters for each individual device, and devices may require periodic recalibration as the device changes through wear and tear. Mathematically, we try to minimize the error of the particular device, rather than maximum error across devices as above:

$$p_i^* = argmin_p\ e_i(p)$$

For example, in CNC machine example mentioned above, each CNC machine is slightly different from others of the same model, so a single set of parameters cannot provide adequate accuracy.

Per-device-class parameters



Here, there are several classes of device, and we need a single parameter for each class of devices, where the individual variation is small enough to not require fine-tuning.

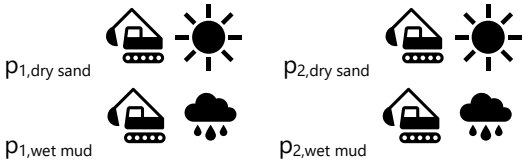
$$p_c^* = argmin_p \max_i e_{i,c}(p)$$

Per-scenario parameters

Here, we define separate parameters for different scenarios, across a class of devices. An example is calibrating a particular model of bulldozer to work at different speeds, or in different materials.

$$p_s^* = argmin_p \max_i e_{i,s}(p)$$

Per-scenario, per-device



The most specific case is doing separate calibration per-device, per-scenario.

$$p_{i,s}^* = argmin_p\ e_{i,s}(p)$$

An example here is fine-tuning the per-scenario bulldozer parameters described in the previous case to specific bulldozers in the field, to account for differences in wear and tear, assembly, and so forth.

Accuracy Requirements and Cost Per Calibration Step

The final aspect of variability among calibration problems is the desired tradeoff between calibration speed and final accuracy. In some cases, doing a calibration iteration is cheap – it is possible to automatically set and evaluate parameters quickly, so it is practical to do more iterations to get improved precision. In other problems, doing an iteration is very slow and expensive – for example, if some of the calibration parameters define physical size or shape of parts, and new prototypes must be manufactured when those parameters change. This variation is important to define up front, to know what how many calibration iterations are acceptable before the BRAIN must reach satisfactory performance.

Reinforcement Learning for Calibration

Reinforcement Learning (RL) is an AI technique for learning from feedback. In the generic RL formulation, an *agent* in an *environment* observes *states* and has to use those observations to take *actions* to maximize *reward* over time.
When applied to calibration, the RL approach works as follows: the agent, called a BRAIN in the Bonsai platform, is repeatedly tasked with calibrating different devices from the target device class(es). It is given the current calibration error and some additional information, and has to choose what parameters to try next. The BRAIN is rewarded based on how well it reduces the error, and gets to try again. Over many thousands of such calibration episodes, the BRAIN learns how to quickly calibrate any device in the target class. To make this practical, training is usually performed using a simulator, and the trained BRAIN is used to calibrate real devices.
To use the Bonsai platform for training, you will need:

- A simulation of the full range of target devices and scenarios, including the ability to vary the hidden parameters that define which device and scenario to attempt calibration for at any time.
- A way to run many of these simulators at scale, likely in a cloud environment like Azure
- Best practices on how to use machine teaching to solve the problem. This document describes our recommended machine teaching approaches for calibration.

Definitions

To help orient the reader to how RL can be used for calibration, this section defines terms linking Control, RL, and calibration problems. The next section uses these terms in describing the machine teaching best practices for teaching BRAINs to calibrate.

Problem specifications

Device: Device is the machine under operation which is being calibrated. Device is sometimes interchangeably called "plant" or "system" in control terminology. Because the goal is to build a generic calibrator that works for a range of devices and scenarios, we will sometimes say "target device or scenario" to define the particular situation the BRAIN is solving at a given time.
Reference: Reference, $ref(t)$, is the instructed position or trajectory that the device output should follow during the calibration process. Reference is sometimes called the desired signal, again in control terminology.
Measurement: Measurement, $m(t)$, is the device output, and the value which is expected to match the reference.
Error: The error is the difference between reference and measurement, meaning $e(t) = ref(t) - m(t)$. This may be a multi-dimensional signal and may be summarized in problem-specific ways.
Calibration Parameters: The parameters that must be tuned. For instance, in a simple PID controller, these are K_p, K_i, and K_d.
Error Tolerance: The maximum allowed error, e_{max}, for which a device as considered calibrated. This is not strictly necessary: the goal may be simply to minimize error as much as possible within some time limit.
Hidden parameters: Parameters of a device that affect its behavior but are not easily observable. Examples include friction between components, variations in hydraulic system performance due to leaks, small differences in positions of mounted sensors, etc. Hidden parameters can be modelled in simulation and used to generate a variety of scenarios for the BRAIN to learn from.
Generalization: the ability of a trained BRAIN to calibrate a variety of devices and scenarios. The range of generalization needed for your application affects the machine teaching approach.

Reinforcement learning

BRAIN: The RL agent that learns to calibrate devices by tuning the calibration parameters.
Iteration: Iteration, t, is a single step of a calibration attempt by the BRAIN: generating calibration parameters to try, evaluate them, observing the result, giving the BRAIN a reward, and deciding whether to continue.
State: State, $s(t)$, is the information given to the BRAIN at a given time-step t. It should describe the current state of calibration and may also include information about previous steps.
Action: Action, $a(t)$, is what the BRAIN produces in each time step, and defines the calibration parameters to try next.
Reward: Reward, $r(t)$, is the signal given to the BRAIN as feedback on how well it is calibrating a device.
Terminal: Terminal conditions define when an episode ends. Typically, there is a success terminal for getting to a sufficiently low error, and failure terminals for running out of time or taking invalid or undesirable actions. condition involves certain fixed numbers such as error tolerance, or maximum number of iterations T_{max} that an episode can last.
Episode: An episode is an attempt to find good parameters over a series of iterations. We pick a particular device from the target class, start with some initial state, and the BRAINs repeatedly chooses new parameters and gets rewards until it either succeeds in lower error to an acceptable error, or we choose to terminate the episode.
Training: The process by which a BRAIN learns how to act to maximize reward. Training consists of many episodes where the BRAIN tries different actions and learns which work best.
Prediction: Using a BRAIN to test or deploy. In prediction mode, the BRAIN no longer trains or takes exploratory actions. In prediction, BRAINs can be queried with a state and responds with the corresponding action.
Policy: the function learned by the BRAIN, mapping from all possible states to corresponding actions. A good policy for calibration is one that quickly gets any target device from any sensible initial state to a well-calibrated state.

Machine Teaching Best Practices

Machine teaching is the process of using your domain expertise about the problem to set up the learning problem for the BRAIN in a way that enables the BRAIN to learn to solve the problem correctly, efficiently, and robustly. We'll go through the main elements you have to define.

Config

The first thing to specify is the range of devices and scenarios that the BRAIN needs to be able to calibrate. The simulator configuration defines this via the allowable variation in scenario parameters and hidden device parameters. During training, the system will select a random value of each configuration parameter.
The appropriate way to vary the config depends on your problem variant.

Per-device calibration

- Each episode: pick an individual device with parameters drawn from the entire range.

Per-device-class calibration

- The easy case is if there's a representative device, where a policy that works on that device will work well across all the devices in the class. In this case, pick a class at random and pick a representative device from that class.
- If there's a lot of variation within a class, either:
    - pick a random device in the class for each episode, or, likely better:
    - pick a representative set of devices and evaluate all of them at each step. This increases state size unless there's a concise summary of the errors, but reduces noise – the BRAIN can consistently see effect of its choices on the real problem.

Per-device, across-scenarios

Here, we want to calibrate individual devices such that the parameters work well across a range of scenarios. This is similar to the above, with class replaced by scenario:

- The easy case is if there's a representative scenario, where a policy that works on that scenario will work well across all the target scenarios. In this case, pick a random device, and configure for the representative scenario.
- If there's a lot of variation across scenarios, either:
    - pick a random scenario for each episode, or, likely better:
    - pick a representative set of scenarios and evaluate all of them at each step. This increases state size unless there's a concise summary of the errors, but reduces noise – the BRAIN can consistently see effect of its choices on a range of scenarios.

Per-device, per-scenario

- pick a device, scenario pair for each episode.

Non-independent hidden variables

What if independently selecting each hidden variable doesn't make sense in my application?
e.g. one variable is the size of some device element, and it's correlated with another variable like friction. To implement this, define intermediate config variables that get translated into the actual hidden parameters in the simulator.

Validate your config

Run a series of episodes, log the resulting device configs, and ensure the distribution matches what you expect the BRAIN to see in deployment.

Action

The action of the BRAIN specifies what controller parameters to try next. This can be specified two ways: **absolute actions**, where we directly output the parameter values, and **delta actions**, where the action specifies a delta from the previous set of parameters. Let's compare the two approaches.
Directly specifying the parameters to try next with absolute actions can be simpler: with this approach, you can specify the valid range of parameters in Inkling, ensuring that the BRAIN will not go out of those bounds. When the action is a parameter delta, you need to actively teach the BRAIN to stay in bounds. Do this by including the distance from each parameter's upper and lower limits in the state, and add a reward penalty and terminate the episode if the BRAIN tries to go out of bounds.
The advantage of delta actions is that they can be more robust to device variation: while different devices will have different calibration parameters when properly tuned, the direction to change the parameters given a particular error profile may still be consistent. If this is the case, giving a version of the current error profile as the primary state and having the system choose which way to adjust the parameters can solve the problem. See the CNC tuning calibration example below for an illustration of this approach.

To accomplish the same kind of generalization while directly specifying the parameters to use requires providing the BRAIN more history in the state. If it can see the system's behavior with several previous sets of parameters, the BRAIN can learn to extract the relevant patterns and pick a good parameter value.

A note on stability: Ideally, you already know what ranges of parameters ensure the system remains stable, either from theory or experience. If you can restrict the BRAIN's action space to such a range, you can let the BRAIN focus on minimizing error without concern about stability. If this is not the case, additional concern need to be taken to avoid or safely detect instability, especially during non-simulated testing.

Discrete vs continuous action spaces: calibration parameters are typically continuous, in which case the most straightforward way to specify actions is to give the valid range if using absolute ranges:

```
type CalibrationAction {
    # Absolute actions – specify valid ranges
    param1: number<-1.0 .. 1.0>,
    param2: number <0 .. 3.7>
}
```

or to specify a step range for if using delta actions:

```
type CalibrationAction {
    # Relative/delta actions – specify adjustments to
current values
    param1: number<-0.2 .. 0.2>,
    param2: number <-0.1 .. 0.1>
}
```

For delta actions, there is a tradeoff between allowing very large steps so that a well-trained BRAIN can quickly get to the right value, and bounding the step size, so that the BRAIN can more easily learn to solve the problem, without going out of bounds or overshooting the right set of parameters. If unsure, get it working with a relatively small maximum step size, then try to improve calibration speed by retraining with increased step sizes.

For discrete and ordered parameters (e.g. valid values are 1, 2, or 3), use continuous ranges and round. For discrete and unordered parameters (e.g. use mode "fast" or "slow"), use discrete values.

Defining State

The state is the information given to the BRAIN as a basis for its decision on the next parameters to try. The state must include enough info to make a good decision about what parameters to try next, no matter which device and scenario it is faced with. Note that the state is the *only* information the BRAIN can use to make its decision. The BRAIN does not have any built-in memory of what happened in previous iterations, nor any common-sense understanding of what the state elements mean and how they relate to each other. (e.g. BRAINs don't know physics, and wouldn't infer anything about two state variables called 'position' and 'velocity').

Typically, the state for a calibration problem includes several components: info about the target device and scenario, information about the current error, historical information based on previous iterations, and, if using delta actions, information about where the current parameters are in the valid range (as described above). Expanding on the first three components:

Device and scenario info

If the BRAIN is intended to calibrate a variety of devices, or for a variety of scenarios, it may be helpful to include info about the particular device or scenario to target in the episode. This allows the BRAIN to learn a policy that behaves differently for different devices or scenarios.

Only include information available at deployment! Including the values of the hidden parameters will make it much easier for the BRAIN to learn, but since they are only available in simulation, you will be unable to use the trained BRAIN for real devices.

Current error

This can be passed as a summary of the factors used by a human expert (e.g. number of peaks, peak error(s), cumulative error, peak timing(s), rise time, decay times, etc), or as a raw error signal. Learning from a good summary is easier than learning to extract the appropriate features from raw signals, so if you are confident that a summary is sufficient to solve the problem, prefer that. If you know certain summary factors that are relevant, but they turn out to be insufficient to get good enough performance, including both the raw signal and helpful summary metrics is an option.

Historical information

As alluded to in the action section above, including information about past iterations can help the BRAIN differentiate between devices or scenarios where the same current error should lead to different next steps. For example:

- Consider two devices with the same error profile with current parameters: a "sensitive" device where small parameter adjustments have large effects and a "sticky" device where large parameter adjustments are needed to have much effect. Even if the direction of the correct adjustment is the same in both cases, the size of the needed step may be very different. The best the BRAIN would be able to do without a way to distinguish these cases is to either take steps that overshoot on "sensitive" devices, or to always take the smallest step that's ever warranted, taking many more steps than needed on "sticky" devices.
- Two devices with similar sensitivity, but a different "base": e.g. the two get similar error profiles when their parameters are offset by a constant. With delta actions, this is not a problem because the direction of adjustment is consistent. However, when using absolute action, the error profile by itself isn't enough info for the BRAIN to pick the right values to try.

Of course, both kinds of variation can be present in the same problem. The most general way to provide historical information is to simply include several pairs of previous parameters with their resulting error. If you have insight into the types of variation the BRAIN will have to deal with, you can make learning easier by summarizing the historical information. For example, with absolute actions and devices of different sensitivity, it may be enough to include only the most recent set of parameters tried and an estimate of sensitivity calculated from the reaction of the device to the parameters in previous iterations.

Initial values for historical information: at the start of an episode, you will not have historical information. To avoid confusing the BRAIN, it is best to include "neutral" default values. If the history repeats previously used parameters and error, simply repeat the first iteration values. If using a summarized history such as the sensitivity example suggested above, include an average value across the target scenarios.

Assessing a proposed state definition

A good test for whether a proposed state definition is sufficient is to generate a set of target device and scenario configurations, pick non-optimal calibration parameters for each, and verify that a human expert can pick an error-reducing set of parameters given only the state variable values, without access to the device configuration or any other info.

Terminal Conditions

Terminal conditions define when to stop an episode. For calibration, there are typically two or three types of terminals:

- Success: the calibration error is sufficiently low.

- Running out of time: typically, it is a useful to have a limit on the number of steps the BRAIN has to calibrate a device. This ensures that during training, the BRAIN doesn't get stuck for a very long time on any particular scenario. Make sure to set these timeout limits high enough to give the BRAIN a chance to learn — if the limit is so short that perfect decision making is required to succeed before running out of time, the BRAIN may take longer to train.
- Out of bounds: When using delta actions, terminate (with a reward penalty) if the BRAIN tries to go out of bounds.

Reward Functions

The reward function gives the BRAIN feedback on whether the actions it takes are good. The BRAIN tries to learn to take actions that maximize cumulative reward – for example, it can learn to take an action that gives low immediate reward, but leads to a state where high rewards become possible. In calibration problems, the feedback is usually fairly simple: we want to reward the BRAIN for reducing calibration error, and encourage it to do so quickly. The simplest such reward is the negative error: $-e(t)$. Using negative reward values incentivizes the BRAIN to finish an episode as quickly as possible, and smaller errors result in larger (less negative) rewards. If the goal isn't to minimize error, but rather to get it to be below some threshold, it may be helpful to make the reward a non-linear function of error, e.g. $-e(t)^2$, and to provide an additional bonus for reaching the threshold,. This helps clearly distinguish the value of states just above and just below the threshold.
As described above, if using a delta action approach, there should be an additional negative penalty if the BRAIN tries to step out of the valid range for a parameter. This penalty should be significant, to quickly discourage such behavior. As a rule of thumb, start with 10 times the largest error reward.

Using Lessons for Calibration Problems

Lessons can help the BRAIN learn by presenting simpler or less varying variants of the problem first, and introducing additional complexity over time. For calibration problems, the main place lessons can help is gradually increasing the amount of hidden parameter and scenario variation the BRAIN is faced with, rather than introducing it all at once. If you find that you system learns well with a small range of configurations, but learns very slowly if trained with the full range needed, it may be helpful to start with the small range, and expand once the policy is performing well.
For lessons to work, a single policy must be able to work across the full set of configuration variability. If the state does not provide enough information to make the right decision in all scenarios, a multi-concept approach that splits the problem may be better.

Splitting the Problem

If the same policy won't work for different device classes, split the problem. The simplest way to do so is to pick some observable element of the scenario that significantly changes device behavior and may benefit from a different policy (e.g. machine size, device model, the scenario being tuned for), and train separate BRAINs for different values or ranges of that parameter. Once you have separate brains, a simple switch can select the appropriate one to use for each scenario.
Another way to split the problem is to only tune a subset of parameters at one time. Train separate sub-BRAINs to tune each parameter subset while keeping the others unchanged, then either write a heuristic to select which to apply when, or train a *selector* BRAIN to automatically choose which sub-BRAIN to invoke based on the current state.

Diagnosing Problems

This section describes some issues you may run into, and what to do.
Training is not converging:

- Ensure human can solve problem given info in the state
- Ensure reward gives useful info – should get more reward for quickly going to good parameters than for any other sequence of actions.

The policy works in some cases, but doesn't generalize to all the target devices and scenarios:

- Ensure human expert can solve problem given info in the state, across full range of configs.
- If the policy works in one range of hidden params, but doesn't when you try to expand the range, consider breaking up the problem and training separate concepts / BRAINs for different ranges.

Works in simulation, but doesn't work in real life:

- Can you manually config the simulator into a scenario that's close to the device you're testing on and check that it works in the sim?
- Check software integration – are the right states and actions flowing back and forth, are they being applied properly?
- Validate that your simulator covers enough variation in scenarios during training.
- Are you tuning part of a larger system? Is the component you're tuning interacting with other components in new ways?

Examples

In this section, we provide examples of how to solve real-world calibration problems with machine teaching and reinforcement learning.

CNC Calibration

Problem description

Computer Numerical Control (CNC) machines are used for precision manufacturing across variety of industries. The machine executes a set of digital instructions and moves a cutting tool along a specified trajectory. The deviation between instructed and actual executed position of a CNC machine, which limits the tolerance of machined parts, is mainly due to friction between mechanical components when the tool head changes direction. Advanced CNC machines include a friction compensation controller that adjusts the applied forces to maintain precise control. To calibrate the friction compensator, we command the machine to cut a perfect circle, measure the deviation from the instructed trajectory, and adjust parameters until the error is below a target maximum, e_{max} (e.g. $e_{max} = 2\ \mu m$). The friction compensator has three parameters:

- The injection time τ_1, which defines when to start the compensation.
- A decay time τ_2, that affects how quickly the compensation is reduced.
- The amplitude of the compensation, amp.

Each parameter has bounds for minimums and maximums given by a domain expert:

$$\tau_1(t) \in [\tau_{1_{min}}, \tau_{1_{max}}],$$
$$\tau_2(t) \in [\tau_{2_{min}}, \tau_{2_{max}}],$$
$$amp(t) \in [a_{min}, a_{max}],$$

The friction which needs to be compensated for can be modeled as a combination of viscous and coulomb friction forces for the main components of the machine: the motor, screw, and slide. These change between machines, and over time for a specific machine, so for maximum precision, each individual CNC machine must be periodically re-calibrated. With human

calibrators, this is expensive and requires taking the machine offline for up to several hours. An automated solution would need to be able calibrate any machine where the friction parameters fall within some range, e.g. -50% to +100% of their nominal values. The nominal values and ranges come from measurements and the domain expert's experience.
Here, we describe a way to use machine teaching to learn an automatic calibrator that can calibrate a range of CNC machines of varying sizes.

Machine Teaching Approach

Here we describe the machine teaching approach that solved this problem. Note that it differs somewhat from the suggestions above – this is what worked in this instance, and we have not tried all variants.

- Ensure the training configuration includes the full range of variation in the devices that we want to support
- Use delta-actions, specifying Δamp, $\Delta\tau_1$, and $\Delta\tau_2$ at each iteration t
- The state includes:
    - A summary of the current error: the peak error, the time of peak error, and the total area under the (absolute value) of the error curve.
    - Historical information: the current parameter values
    - Delta-action support: each parameter's distance from its min and max.
- Use a reward that encourages low error, with a bonus for reaching the error goal.
- Terminate episodes on success, after a limit of $T_{max} = 50$ iterations, or on going out of bounds.

Reward function at each iteration is negatively proportional to the magnitude of the peak $|a(t)|$, plus additional offset terms:

$$r_{base}(t) = -|a(t)| + o(t),$$
$$o(t) = \mathcal{N}(a(t); 0, \sigma) + k\ ,$$

where $o(t)$ is an offset term, $\mathcal{N}(.)$ is a Gaussian function and k is a bias term, which are added to avoid negative reward as much as possible. There exists an additional bonus reward which is maximum iterations minus current iteration ($T_{max} - t$) to encourage faster tuning:

$$r_{bonus}(t) = o(t)(T_{max} - t)$$

Each tuning attempt or episode takes maximum iterations of $T_{max} = 50$. The episode is successful if target error is reached and is declared as failed if iterations maxed out before ever reaching the target error.

Table 1. Setup details for CNC Calibration.

STAR	Definition		Notes
State	$(a - a_{\min})$, $(a - a_{\max})$, $(\tau_1 - \tau_{1_{min}})$, $(\tau_1 - \tau_{1_{max}})$, $(\tau_2 - \tau_{2_{min}})$, $(\tau_2 - \tau_{2_{max}})$, peak, time of the peak, area under error curve, peak - 0.1 * initial_peak.		(peak - 0.1 * initial_peak) is a state to teach the agent reduce the peak by 90%. The ratio 0.1 is device specific, depending or the initial and expected error ranges.
Action	Δa, $\Delta\tau_1$, $\Delta\tau_2$		
Terminal	Success:	$\lvert amp\rvert < 2\mu m$	$T_{max} = 50$
	Failure:	I) $t > T_{max}$	

		II) amp, τ_1, τ_2 out of bound	
Reward	Regular:	$\max(r_{base}, 0)$	bias $k = 7$, Gaussian's $\sigma = 5\mu m$.
	Success:	$r_{base}(t) + r_{bonus}(t)$	
	Failure:	I) $r_{base}(t)$ II) 0	

Sample Inkling

```
inkling "2.0"
using Number
const AMPL_STEP: Number.Float32 = 0.001
const T_STEP: Number.Float32 = 0.01

type FrictionState {
   # Error summary
   peak: Number.Float32,
   time_peak: Number.Float32,
   integral: Number.Float32,
   # Supporting delta actions
   ampl_minus_min: Number.Float32,
   max_minus_ampl: Number.Float32,
   t1_minus_min: Number.Float32,
   max_minus_t1: Number.Float32,
   t2_minus_min: Number.Float32,
   max_minus_t2: Number.Float32,
   # Historical info
   dist_to_orig_peak: Number.Float32
}

type FrictionAction {
   delta_ampl: Number.Float32<-AMPL_STEP .. AMPL_STEP>,
   delta_t_rise: Number.Float32<-T_STEP .. T_STEP>,
   delta_t_fall: Number.Float32<-T_STEP .. T_STEP>
}

type FrictionConfig {
   # percent below nominal (since we're using the same range
   # for all friction params, can use a single config value)
   friction_range_low: Number.Float32,
   # percent above nominal
   friction_range_high: Number.Float32
}


simulator simulink_friction_sim(action: FrictionAction, config: FrictionConfig):
FrictionState {
}

graph (input: FrictionState): FrictionAction {
```

```
    concept compensate(input): FrictionAction {
        curriculum {
            source simulink_friction_sim
            lesson my_first_lesson {
                constraint {
                    friction_range_low: -50,
                    friction_range_high: 100
                }
            }
        }
    }
    output compensate
}
```

Results

The resulting BRAIN can reliably calibrate CNC machines across the entire desired range of variability.

Conclusion

This document described how machine teaching and reinforcement learning can be used to solve device calibration problems. We present a set of best practices to guide the creation of calibration solutions.