# Final Milestone Report for ReMath: Reverse Engineering Functions Using Symbolic Regression

Lucca Fraser, 'Dillo' Okay, Daniel Kuehn, and Anthony Di Franco

2022-05-04

## REFUSR Final Milestone Report

The object of this report is to sum up our research on the ReMath project, under the rubric of **REFUSR** (Reverse Engineering Functions Using Symbolic Regression). In the following sections, we will provide synopses of our work on developing a "PLC whisperer" hardware interface, facilitating the observation of PLC program behaviour under a controlled set of inputs, a probabilistic property testing (PPT) module that facilitates the efficient detection of boolean function properties, a genetic programming (GP) framework for symbolic regression, and a reinforcement learning (RL) framework to assist GP-driven SR.

### Hardware Interface

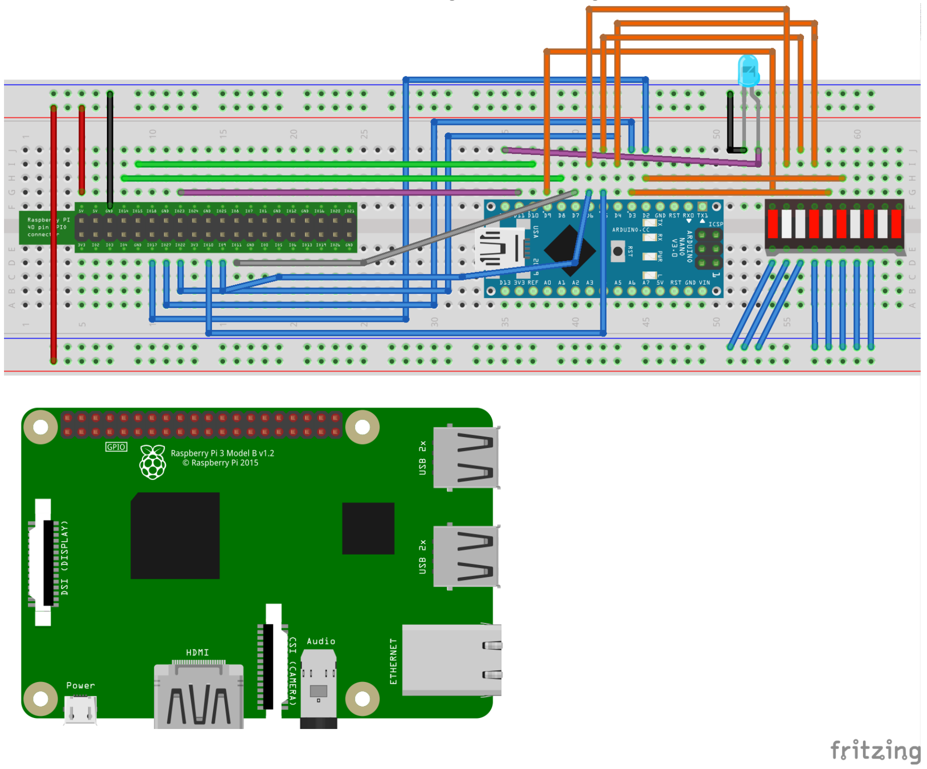
This section will initially give a recap of the hardware progress during REFUSR and discuss future work on the hardware interface and tooling at the end.

#### Recap

We started with a simple idea of using a cheap, off-the-shelf PLC called ACE from a company called Velocio. They had two entry level PLCs (ACE11 and ACE1600) that looked promising as first targets. The problem with them was that the IDE (vBuilder) for programming and the protocol to program them are proprietary. We did a little reverse engineering effort to the protocol used by the vBuilder program to program the PLCs, but we concluded that the effort to reverse engineer the protocol to the point that we could program the PLCs from our own code was too steep. Another problem was that there was no way to program them from anything than vBuilder without doing the reverse engineering effort, because Velocio does not provide a standalone compiler, nor did they have any software for anything but the Windows platform.

Together with the problems of compiling and provisioning the PLC from our own tools, there was problems with using the GPIO pins from an Arduino Nano to feed signals to the input pins of either ACE model. But then we found a platform called OpenPLC that was a project that turned various Single Board Computers (SBC) into basic PLCs. It had its own, separate, toolchain to compile various IEC 61131-3 standard languages to a C binary that it then executed to run whatever PLC code that had been compiled. It could even run on regular x86 hardware. Thus we decided to use OpenPLC to be the “PLC Target” instead of one of the ACE models.

With the decision to use OpenPLC to be our PLC we decided to create a hardware platform around it, that was dubbed “Refuduino: A PLC Busybox”. The first version of the “Refuduino” consisted of a Raspberry Pi 4 (RPi) that was connected to an Arduino Nano microcontroller (MCU) on a breadboard, with wires between the RPi’s GPIOs and the MCU’s GPIOs to be able to input and output signals between the PLC and the MCU. A 10-segment LED display was added to be able to visualize the signals running between the RPi and MCU.

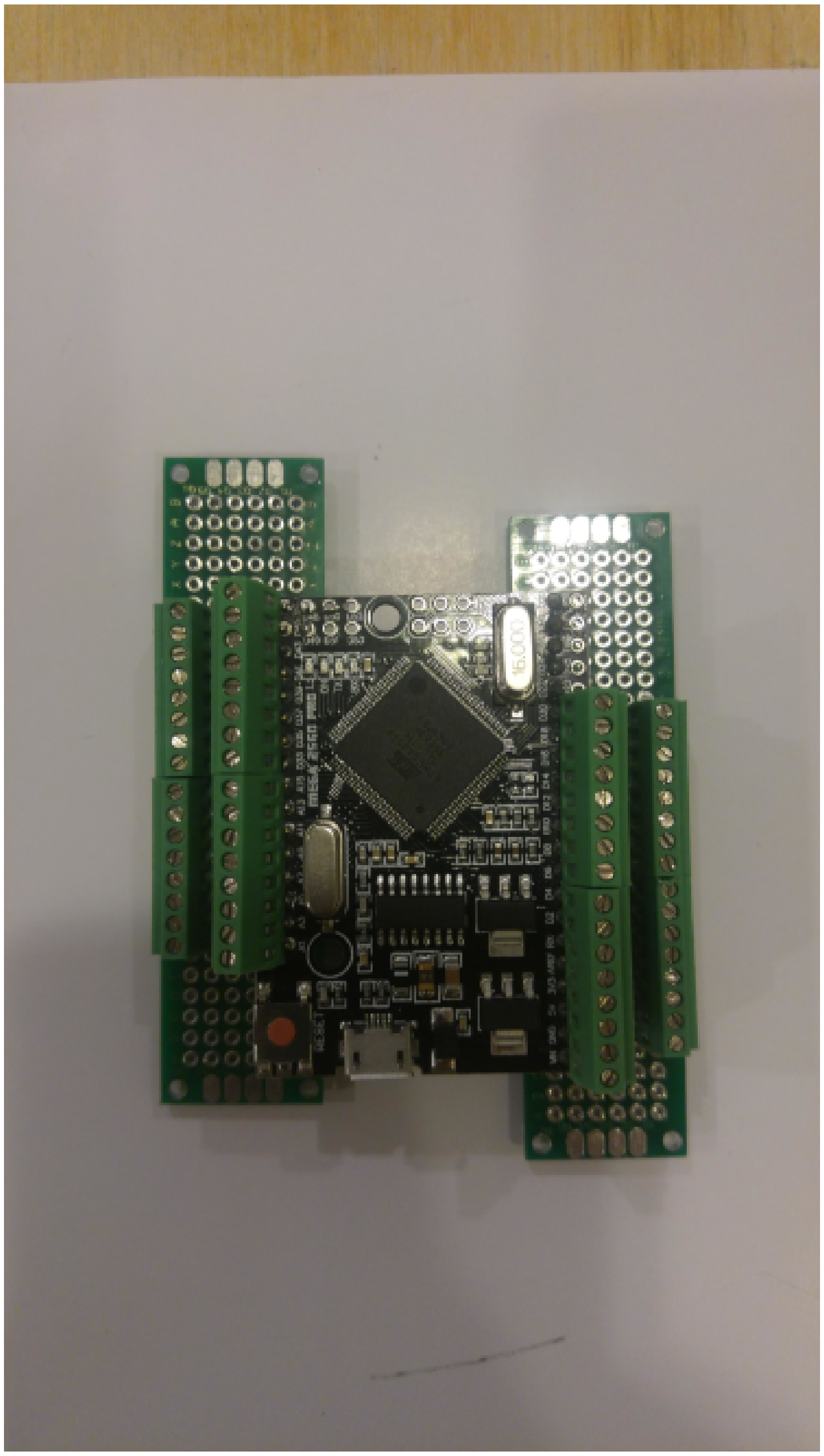


*Fritzing diagram of first Refuduino version*

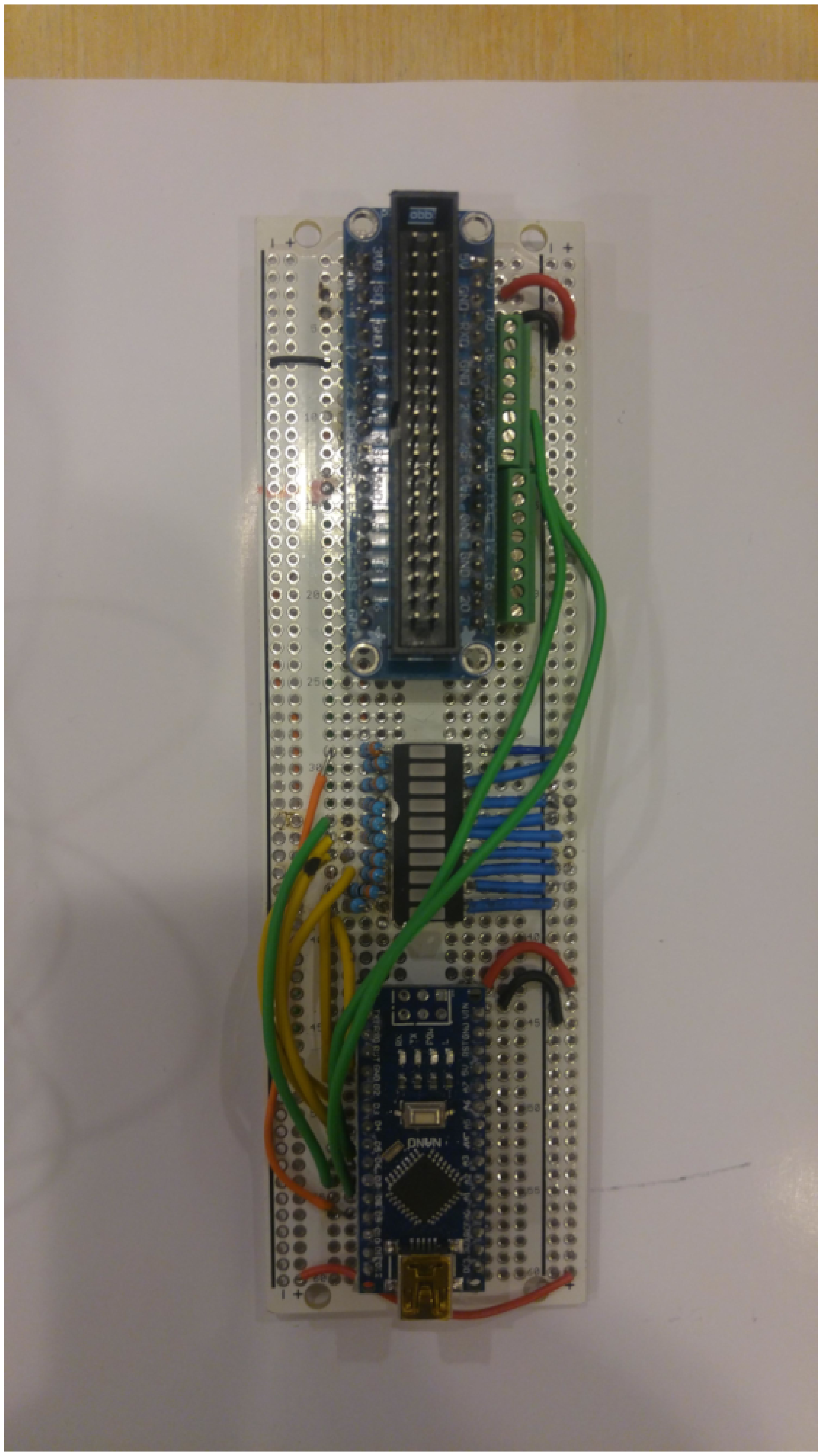
The plan was to instrument inputs to the PLC through handing them to the MCU and the MCU forwarding the inputs to the PLC. The PLC would then process the inputs and return an output to the MCU, which in turn would relay that output back to the software that gave the inputs. This would enable the “Refuduino” to be able to act like a “PLC oracle” for the GP-framework.

A few versions of the PLC/MCU interaction was trialed and tested before it was decided that the MCU was more going to act like a number station to the PLC. Initially we simply weren't that interested in what inputs it got, but that it could get inputs, process it and return some output. This also resulted in the MCU/PLC interaction becoming more akin to how PLCs are used in real world scenarios. In real word scenarios you usually have one or several "masters" that instrument several "downstream" sensors/PLCs, sometimes in more than one hierarchical layer.

Once we had the first version of the Refuduino up and running, proving that the concept works, it was time to start on version two. The goal for version two was to make it more usuable in an integrated fashion in our software "pipeline". For the second version we created a few variants of the actual hardware platform and started to build our own tooling and monitoring for it. Amongst the variants one dubbed "catamaran" was created which basically was a soldered version of the first version. It had screw terminals and an ATMEGA 2560 MCU instead of an Arduino Nano. The ATMEGA 2560 MCU opened possibilities of using a lot more pins, because it had 56 GPIOs in contrast to the 13 that the Arduino Nano has. It also hade 8 times as large flash memory as the Arduino Nano. The screw terminals enabled us to have more freedom in how we coupled the pins to eachother, and being able to more permanently afix cables than with a breadboard.



*Re-*



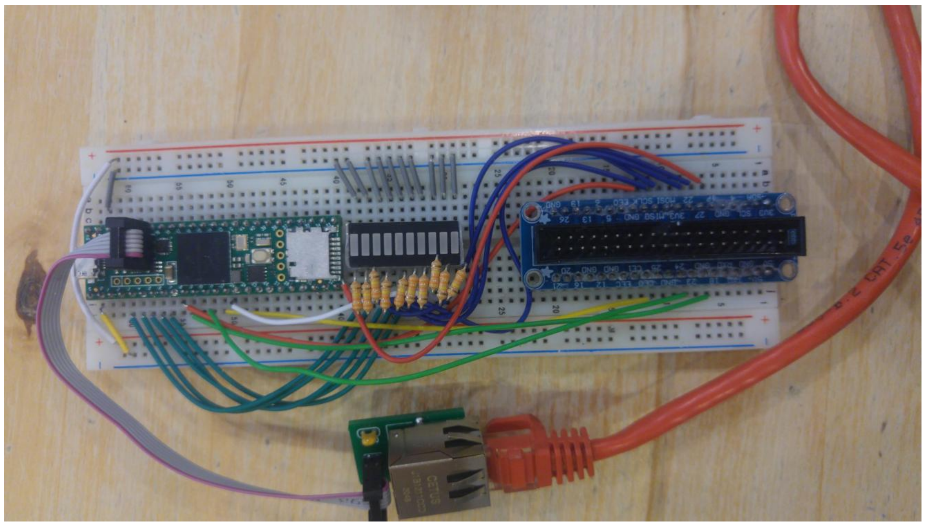
*Re-*

*fuduino v2, Breadboard version*

One of the problems with the OpenPLC platform is that the built-in monitoring is in the form of a simple web-based dashboard, that is generated from the actual PLC code that gets compiled. This dashboard has a fairly slow minimum refresh rate of one second. That means it doesn't really catch the activity that we generate on the PLC, because the runtime on the PLC is usually measured in tens or hundreds of milliseconds instead of seconds. It also became apparent that for the hardware platform to be able to fufill its "oracle" role, it had to have a way of receiving known inputs from the GP-framework and return the output back to the framework.

Thus two python scripts were developed that used the dynamic instrumentation framework called Frida where we hook the digitalRead and digitalWrite functions of the resulting binary that OpenPLC compiles. These scripts primary function initially was to help visualize whats happening in the PLC as a way of debugging the PLC code. One of the scripts has a similar function as the web-based dashboard from OpenPLC, but with a much faster, configurable, refresh timer. It also functions directly from a terminal instead of being webbased. The other scripts function is to interact with specific pins on the PLC and read/write from/to them. One of the benefits of using the Frida framework for the instrumentation is that if we decide that we want to fetch more internal state from the OpenPLC binary, we already have the harness setup and functioning to get that internal state.

In conjuction with the development of the Frida-based tools a third version of the hardware platform was made, replacing the MCU with a Teensyboard 4.1 (Teensy). The two main reasons for replacing the MCU with a Teensy is that the 4.1 version of Teensy has a 100 Mbps ethernet extension available. It also has much better hardware in the form of a stronger CPU, larger RAM and drastically larger flash memory than even the ATMEGA 2560 MCU. This opened up a lot more options of what we could have the hardware platform perform. We did interoperability tests by hosting a MODBUS TCP server on the Teensy and interacting with it through a python script that used the PyModBus library from a computer on the network. The main reason for switching to MODBUS for the PLC communication was two-fold, it would better simulate how a sensor/PLC would be interacted with in an actual system. It would also take care of solving most, if not all, of the synchronization issues we had when manually handling the inputs/outputs like bit-input/outputs.


*Refuduino v3*

**Future work for hardware platform**

There were plans for the hardware platform and tooling that didn't come to fruition due to several reasons, which will be briefly outlined in this subsection.

The switch to the teensy MCU was done for several reasons with the most noteworthy reason being its ethernet capability. The plan was to have it host both a MODBUS TCP server and some rudimentary webserver/RPC server. The MODBUS TCP server would mainly serve as the entrypoint to getting inputs/outputs to/from the PLC, whereas the webserver/RPC server was supposed to be the entrypoint to enable logging, sync timestamps and similar functions. There was also a plan to have some type of file-based "cache" for the oracle function, so that the return value wouldn't have to be computed each time. This cache would be stored on a microSD card that you could equip the teensy with. The benefits of making the hardware platform network connected is that there wouldn't need to be a 1:1 mapping between machines that is running the GP-framework and instances of the hardware platform. The hardware platform being connected to the network meant that several machines could utilize an instance.

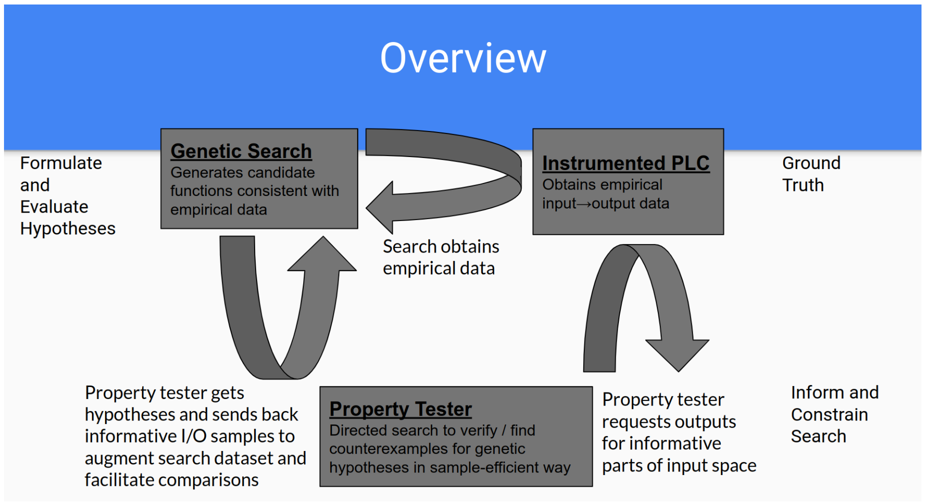
Another plan was to develop the Frida-based tools more, making the read-all-pins script into a proper Terminal UI (TUI) dashboard, that updated in place and could track the state changes of pins and visualize that with a graph or similar representation. It was also planned to have a function of streaming the data to a CSV file, that then could be analyzed with other software.

One thing that was discussed was to make a proper Human-Machine Interface (HMI) that mirrors how operators would monitor processes that were being controlled by PLCs and sensors. This would be possible to make with vFactory, a software made by Velocio the company behind the ACE PLCs. This would've been possible due to vFactory being capable of probing PLCs through MODBUS, and tying values from sensors/PLCs through MODBUS to different controls/visualisations in vFactory.

## Property Testing

Anthony Di Franco Principal Applied Scientist Special Circumstances

### Role of property testing


Overview
Formulate and Evaluate Hypotheses
Genetic Search
Generates candidate functions consistent with empirical data
Instrumented PLC
Obtains empirical input→output data
Ground Truth
Search obtains empirical data
Property tester gets hypotheses and sends back informative I/O samples to augment search dataset and facilitate comparisons
Property Tester
Directed search to verify / find counterexamples for genetic hypotheses in sample-efficient way
Property tester requests outputs for informative parts of input space
Inform and Constrain Search

Role of property testing

We developed property testing tools to provide side information useful for identifying functions under study and comparing hypotheses about them to one another and to the ground truth function. By virtue of how efficient property testing tools operate, they can also be used to bias the collection of samples from the ground truth function to inputs expected to be most informative regarding the ground truth function's behavior.

### Junta review and junta tester overview

A k-junta is a set of k inputs to a (boolean, single-output) function that together entirely determine the output of the function. That is, knowing those k inputs is sufficient to know the function's output. It is one of the simplest widely studied properties of boolean functions with nontrivial potential for informing applications such as ours here.

Most directly, determining that a set of inputs forms a junta can be used to "preprocess" a problem to restrict the search space to only the relevant inputs, which yields a speedup exponential in the number of relevant inputs vs nominal inputs. However, this is highly problem-specific, because in general there is no guarantee that nominal inputs will not also be inputs that contribute to determining the output. However, the search used to determine junta membership of inputs can also yield other useful information.

We use a state of the art adaptive sampling approach to junta search from "Distribution-Free Junta Testing" by Liu, Chen, Servedio, Sheng, and Xie in STOC 2018. This approach adaptively samples from recursively refined partitions of the input space to find single-bit perturbations of the input vectors that produce changes in the outputs.

In previous work, we considered both using testing techniques for a battery of specific properties, and using general property testing techniques to characterize and compare general functions. General property testing in terms of adaptive sampling procedures proved to be a useful technique in developing our novel adaptive sample set construction technique for function characterization, which is based on the sensitivity information yielded by the junta tester's search.

**Sensitivity info from Junta sampling and pointwise property testing** State of the art junta testing uses adaptive sampling strategies to deal effectively with combinatorially large search spaces and to take into account specific function behavior and input data distribution. We developed a new approach that adapts the junta search, using in part concepts from the general purpose function property testing methodology in Chapter 6 of Oded Goldreich, *Introduction to Property Testing*, 2021; and also draws on ideas about function sensitivity measures to augment the existing junta tester with the ability to characterize a function based on local rather than global properties, which yields more information and can be implemented in a simpler and more efficient way than the general property tester.

Since the junta proof relies on finding single-bit input perturbations at specific input point pairs that influence output, the junta search also yields information about the function's input-output sensitivity, in that input point pairs that prove junta the property also reveal where in input space the output is sensitive to input.

Therefore, we modified the junta search sampler to collect multiple input pair samples at each input bit position, so as to accumulate a sample set of known sensitive points on which other properties could be tested.

This defines an adaptive method for characterizing functions by testing *pointwise properties* i.e. local properties of the function defined in terms of the perturbation of a single bit of a point in the domain of the binary function being characterized. Single-point perturbation sensitivity is a more local version of measures such as Nisan sensitivity and Dirichlet energy, (both properties of interest in the broader work here,) which consider the entire neighborhood of single-bit perturbations at a given input point.

The notion of pointwise property also includes other properties of interest, notably 1-monotonicity. *k*-monotonicity is defined such that a function *f* is said to be *k*-monotonic if, given an ordering relation   on inputs, and the function *f*, for all sequences of inputs X(i), for all i, X(i-1)   X(i), then f(X(i))   f(X(i-1)) at most *k* times. In the case of 1-monotonicity, checking sequences of length 1, that is, perturbations of a single input at a single bit, suffice to establish the property locally in the neighborhood of the original and perturbed inputs.

In summary, the resulting approach to property testing consists of the following features:

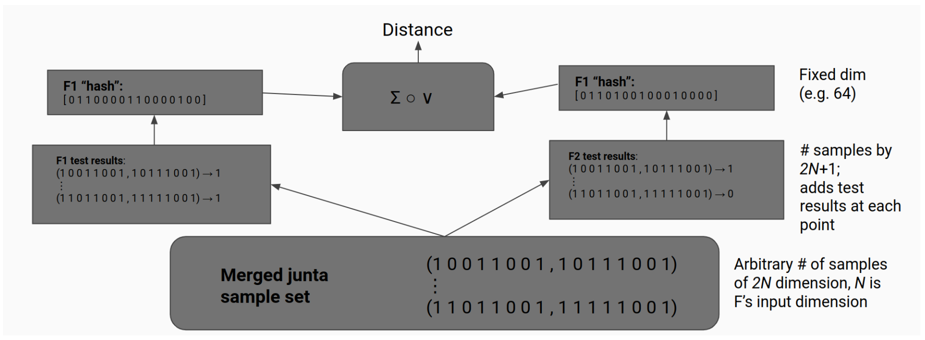
- First, it provides information about where in the input space the function's output is sensitive to certain perturbations of its input, which is useful information for identifying and comparing functions, and for reducing the space of searches associated with these queries.
- Second, the state of the art algorithm for junta testing we chose for implementation is able to adapt its sampling strategy to both the function under study and the input distribution of interest, which is useful for focusing the search effort in the junta checking itself and in the generation of samples for other queries that may benefit from a bias toward the sensitive samples identified by the junta search.
- Third, the vector of sensitive input positions is itself a generally informative property that can be used to characterize and distinguish functions without assuming much about them.

The ability to determine sensitive points and inspired the following approach to function identification by computing fingerprints based on pointwise property behavior at sensitive points.

**Pointwise test-based function fingerprinting** Based on the junta tester's operation in terms of a search for inputs that, when perturbed at a single bit position, yield a change in the output, testing of arbitrary properties at such points was implemented. The junta tester can optionally test a given predicate of two points at the sensitive input points and their perturbed versions, and maintains a log of the result of testing the predicate between the original and perturbed point at each point pair.

This log is then accumulatd into an approximately information-preserving fixed dimensional binary vector that serves as a signature of the function's pointwise property behavior, computed in a way most directly informed by Kanerva's hyperdimensional codes, and also by Panigrahy's sketch-based memory for neural networks, and Bloom filters and variants thereof more broadly. By means of these fingerprints, functions can be compared directly in terms of where in their input spaces the pointwise property under consideration does or does not hold; similarity in this sense is reflected by the inverse Hamming distance between the computed fingerprint vectors.

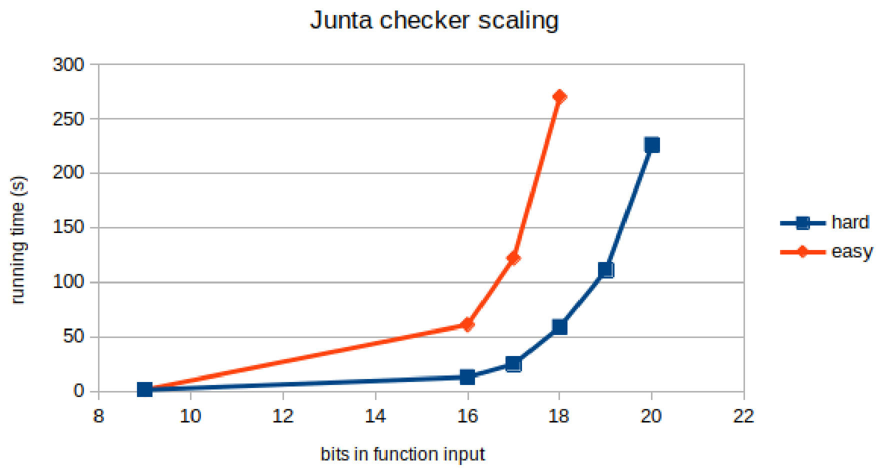
The fingerprinting process is outlined in the diagram above and in psuedocode



Fingerprinting process diagram

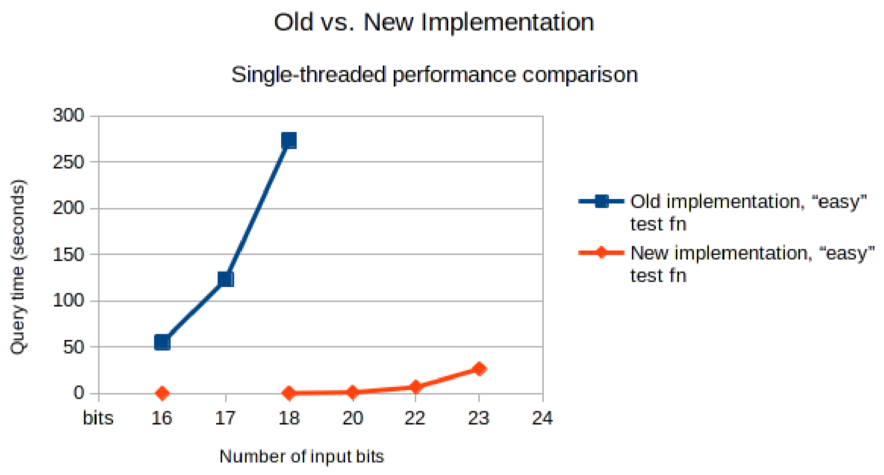
below.

1. The junta sampler yields a set of *sensitive input points*
2. Properties are tested in these local neighborhoods as described above
3. These (point, point, property) tuples are hashed.
4. The hash is used to compute a fixed size bit pattern with few set bits.
5. The patterns of each (point, point, property) tuple are combined into a composite vector of the same fixed dimension representing function behavior on the entire sample set. This vector is the *function fingerprint*.
6. Two functions are compared by measuring the distance between their fingerprint vectors on the same sample set.



Original Implementation Scaling

**Performance of property testing** The junta search is by far the costliest operation under consideration here, being essentially a combinatorial search of subspaces of the entire input space (see initial implementation performance in figure above). Beginning from the performance-naive algorithm in the original source, we obtained several orders of magnitude speedups by memoizing a key subset calculation involved in directing the search, and by parallelizing across the tests for candidate sensitive input points.



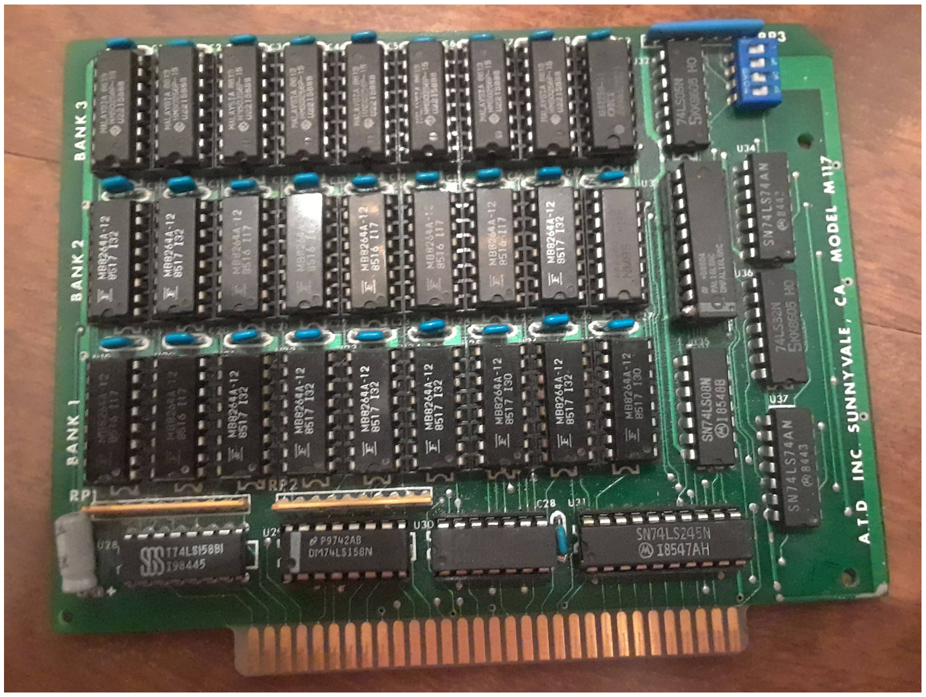
Optimized vs. Naive Junta Search Performance

The result is an implementation that can comfortably handle a 24 bit function in a few minutes on on a 16 core, 32 thread AMD 5950x machine with 32 GB RAM, becoming memory bound on larger problems. In limited testing on a machine with 256 GB RAM, 32 bit problems could be solved at comparable near-interactive rates.

Because performance depends strongly on the relative frequency of sensitive input points which in turn is highly problem-specific, number of input bits alone is only a rough guide to the difficulty of a problem and the expected cost of solving it, so performance will ultimately need to be assessed by the end user in terms of function behavior, input space size, and available processing and memory resources.

## Real Data: PAL16L8

PAL16L8 is a programmable array logic chip at the heart of the ATD Model M117, "an ancient memory expansion ISA card for IBM PC and compatibles, especially 8088-class systems" with an 11 bit input space and 7 bit output space, and 18 of the 20 total pins having a function of some sort.



PAL16L8

A programmable array logic device (PAL) consists of a programmable logic device which operates according to and which can be configured by writing rules tabulated in a programmable read-only memory (PROM). It is a simple but relatively low performance instance of reconfigurable hardware capable of implementing arbitrary digital logic functions. It bears some resemblance to a look up table element of an FPGA, and can be thought of as a standalone version thereof.

The M117's original appliction was to expand an early IBM PC system with up to 384kB of 4164-type and 41256-type DRAM. The address ranges handled by the card were configurable using 4 DIP switches. The PAL16L8 controlled most of the card's behavior and was configured using 3 of the 4 DIP switches. The device has no internal state, thus the output bits are strictly functions of the input bits.

The pins of the card are as follows:

| PAL Pin | Signal Name | Pin Direction |
|---|---|---|
| 1 | SW 1 | I |
| 2 | SW 2 | I |
| 3 | SW 3 | I |
| 4 | A16 | I |
| 5 | A17 | I |
| 6 | A18 | I |
| 7 | A19 | I |
| 8 | Delayed /SMEM{R,W} | I |
| 9 | /REFRESH | I |
| 10 | GND | PWR |
| 11 | /SMEMR | I |
| 12 | DRAM A8 All Banks | O |
| 13 | /CAS Bank 3 | O |
| 14 | /CAS Bank 2 | O |
| 15 | /CAS Bank 1 | O |
| 16 | Data Bus Direction | O |
| 17 | /RAS All Banks | O |
| 18 | /SMEMW | I |
| 19 | /WE All Banks | O |
| 20 | 5V | PWR |

The pin functions are as follows:

- SW1-3 are user-input DIP switches (SW4 is not connected to the PAL).
- A16-A19, Delayed /SMEM{R,W}, /REFRESH, /SMEMR, and /SMEMW are control signals from the computer to the ATD M117.
- DRAM A8 All Banks, /CAS Bank 3-1, /RAS All Banks, and /WE All Banks control the DRAM. /CAS Bank 3-1 uses a one-hot encoding.
- Data Bus Direction determines the data transfer direction between the

card and the computer.

See the project Github page for details.

We obtained a tabulated dump of data representing the PAL16L8's input-output behavior on the entire 11 bit input space and implemented access to tabulated data for the junta checker. Because junta checking operates on only a single output at a time, we implemented a simple iteration over all outputs to find the junta set for each output bit. Unlike in the case of the automatic junta size finding implemented previously, there is in general no opportunity to share information among these queries. Though an heuristic approach that biases the search with sensitive input points from previously queried outputs may plausibly be useful, this is not considered further here.

The junta checker finds juntas for all 7 outputs in about a minute on a 16 core, 32 thread AMD 5950x machine with 32 GB RAM.

#### Software availability

The junta checker is implemented as a standalone Julia library, suitable for interactive query use within a broader hypothesis search such as that undertaken by the genetic search module, and has provisions for loading and analyzing batch data from csv files.

See the project Github page.

#### Future Work

Recently, we became aware that work in symmetric property estimation may be applicable to vastly expanding the scope of the properties we can efficiently test. Symmetric properties are those that do not depend on the identities of the symbols in the alphabet, rather only their relative frequencies.

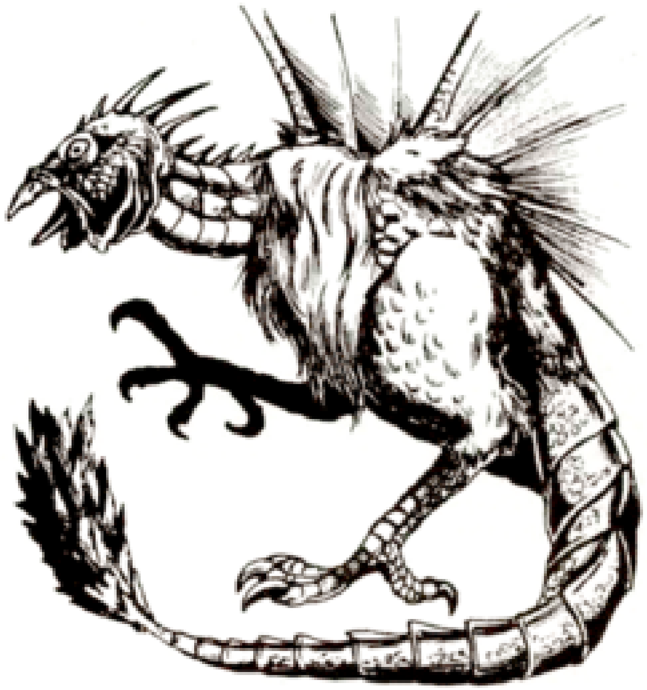
Examples of symmetric properties are many of the various entropies and energies, including the Dirichlet energy of interest in the broader work here in the genetic module.

Symmetric properties can be calculated in terms of profile sufficient statistics, which can be found with profile maximum likelihood algorithms. Recent work by Pavlichin, Jiao, and Weissman (2017) permits efficient profile maximum likelihood calculations.

We are currently investigating implementation of symmetric property estimation using their algorithm.

## The Cockatrice Genetic Programming Engine

At the core of our symbolic regression engine is a genetic programming framework called *Cockatrice*, implemented in the Julia programming language as the module `Cockatrice.jl`. Cockatrice provides a multiprocessing architecture for



evolutionary search: a collection of geographically structured "island" populations are maintained and tournament selection is scheduled and dispatched. The system is polymorphic by design, and allows for a variety of genotype and phenotype structures and fitness functions to be used.

As a means of supporting population diversity and facilitating parallelism, we have divided our populations into a set of "islands", which evolve independently from one another, save for occasional migration. Each island has its own two-dimensional, toroidal geography.

When a tournament is arranged, Cockatrice begins by chosing a random point on the island, and then randomly selects competitors from the neighbourhood of that point. The closer an individual is to the point in question, the more likely their participation in that tournament becomes. The steepness of the distribution curve around that point can be adjusted by tweaking the `locality` parameter in the Cockatrice's configuration file.

For the task at hand, we adopted the approach of *linear genetic programming*, wherein genotypes are represented as sequences of register transfer instructions, without jumps or conditionals. These can be thought of as programs in assembly language for a very simple virtual machine. The primitive operations consist of basic boolean functions, such as `NOT`, `AND`, `XOR`, `OR` and `IDENTITY` (i.e., `MOV`). Each instruction takes a source and a destination register, applies the operation to the value in each, and then writes the result to the destination register.

Each program implements a boolean function of type

$$\mathbb{B}^n \to \mathbb{B}^m$$

allowing us to model functions that return multiple bits of output.

The simplicity of the programming language used in our linear genotypes makes it a simple matter to vectoralize our program evaluations, and so, in practice, each function can be treated as a map

$$\mathbb{B}^{nk} \to \mathbb{B}^{mk}$$

where $k$ is the number of test cases being evaluated in a batch. For relatively small functions, we can let $k$ cover the entire set of possible inputs, i.e., $k = 2^n$.

It should be noted that *every concatenation of register transfer instructions* constitutes a valid linear program. The only restriction we impose is on length, for which we set an upper bound, configurable for each experiment, in order to guard against the exhaustion of computational resources.

A subset of registers are used to load the $n$ arguments of the function being modelled, and these are fixed as read-only registers. The reason for this decision is to prevent the gradual deterioration of information that occurs in random linear programs, such that the likelihood of the program calculating a constant function approaches 1 as the length approaches infinity

Cockatrice seeks to model the boolean functions described by input/output maps (obtained, in the context of this project, by tracing the behaviour of boolean programs on PLCs and other systems, like the PAL16L8 described above) by means of genetic search. When it discovers a linear program whose behaviour fits the observed data, it then "decompiles" that program into a simplified symbolic representation that can be easily interpreted by a subject area expert.

To assist this search, we have brought together a number of heuristics, including:

1. the use of *interaction matrices* to facilitate niching and forestall the premature convergence of populations that succeed only in emulating the relatively "easy" aspects of a function (see Milestone 2 report)
2. tracking information-theoretic relationships between intermediate execution states and the target function graph (see Milestone 3 report)
3. measuring and comparing the Dirichlet energy between candidate programs and the target function, a viewpoint that has shown itself to be particularly useful in seeking out otherwise recondite checksum functions (see Milestone 4 report)

Finally, and this will be the focus of the following section, we have developed a parallel modelling engine, *Climb*, which employs *reinforcement learning* to pick up the search where *Cockatrice* plateaus. These two engines share a common program representation – linear VM machine code – and so are able to operate in a "tag-team" fashion, each picking up the problem afresh when its counterpart hits a wall.

## Climb: A Policy Gradient Approach to Symbolic Regression

**Code** located in the RL directory

Inspired both by our prior work in AIMEE and *Deep Symbolic Regression: Recovering Mathematical Expressions* we investigated a reinforcement learning approach **Climb** that uses **Cockatrice's** fitness landscape as an environment for solving a symbolic regression task. With Cockatrice we are exploring the discrete space of possible program space. Linear programs are sequences of instructions which require an auto-regressive model to generate instructions. Additionally, distant components can be highly correlated with one another meaning a locality bias will cause a loss in accuracy, especially for more complex programs. With AIIMEE, we learned that policy gradient methods with RNN based actor/critic models are effective means of recovering a gradient given a problem with a discrete action space. Instead of using continuous relaxation or a natural gradient (like in Natural Evolutionary Strategies), we use proximal policy optimization (PPO) and in-situ constraints to restrict the search and discover the direction in each dimension of the policy that optimizes an extrinsic and intrinsic reward condition.

We treat the problem as a vanilla symbolic regression problem rather than the method used in **Cockatrice** which peeks at the function which is being regressed. **Climb** instead presumes no observations of the oracle function and works purely by minimizing the normalized mean squared error between observed **candidate programs** output given a limited number of inputs and the actual output of the oracle function. This requires the use of a vanilla PPO agent and a highly custom environment consisting of a linear compiler and a mechanism for characterizing novelty of an execution trace adapted from "Learning to Execute Programs with Instruction Pointer Attention Graph Neural Networks". Critically, instruction by instruction, we can predict both the reward (proximity to the ground truth output observations) and the similarity of that instruction conditoned on its context. The latter allows us coerce the RL algorithm into exploring increasingly novel sub-routines using a learned embedding constructed across candidate programs observed in the linear compiler environment. This score could additionally allow us to in future work maximize the diversity of the candidate programs in each batch reducing variance with the addition of the bias of the curiosity module.

Inspired by "Recovering Mathematical Expressions" we train the policy to maximize the performance of the top quintile of policies. The kind of policy $\pi_\theta(A_t|S_t)$ we learn is stochastic (it learns a distribution of actions given the state and parameterized by parameters $\theta$. To get the top quintile of linear programs (those that produce outputs most similar to the ground truth outputs), you take the softmax of each state's action distribution, repeated k times this would give us the top-k samples from the policy distribution. We are interested in the top-k quintile because this is a search/anomaly detection problem rather than one where you want to optimize for the average reward of samples from the policy distributon. This policy class, called a risk-aware policy, something we retrospectively should have used in AIMEE and will likely use in later work.

### Curiosity or Adaptation: Directed Exploration and EvoRL

Reinforcement learning works indirectly on a large number of parameters via sampling from a buffer, making estimations, taking action, and updating the estimations a large number of times. Genetic algorithms work directly on the parameter values via crossover and mutation operations leading to greater performance scalability (parallelism) and less sensitivity to initial conditions (weight initialization) than deep reinforcement learning models.

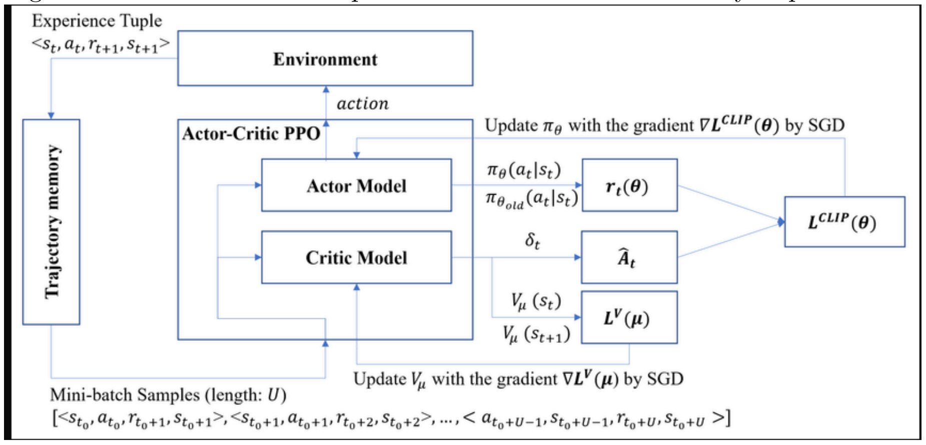
Reinforcement learning researchers often discuss the high variance of samples of the policy distribution and how to minimize this variance. While there are a number of trade-offs to introduce bias (and some that reduce variance without introducing bias), this variance tends to cause reinforcement learning algorithms to converge to a local optima. While this is generally negative, there is some argument in deep learning as a whole that a local optima is often good enough (due to symmetries in model parameters or just demands of the task ). This tendency generally negatively affects accuracy and generalizability, but for problems where the goal is to find a few, anomalously high reward samples from the policy distribution a local optima isn't good enough and getting as close as possible to a global optima - crucially on the specific task on which the algorithm is trained- is essential

Given this, there is incentive in these classes of problems to 1) maximize exploration 2) find a way to combine genetic algorithm's increased robustness to initial condition and reinforcement learning's ability to scale to highly complex problems.

There are many examples of evolutionary reinforcement learning with vast differences on the entry point of GA to RL or vice versa. For this problem, I believe that using evolutionary algorithms to perform data augmentation (given a seed of n candidate programs run a few iterations of Cockatrice to produce more examples with just diversity as a fitness function ) would allow for parallelism of some exploration as well as coercing greater variance into the sample reducing the variance of the estimate of the policy gradient. Another interesting direction would be using a nearest-neighbors cache of policies and having the fitness landscape consist of policies vs. candidate program genotypes.

## Policy

Figure 1.0: A General Representation of Proximal Policy Optimization


Experience Tuple
$<s_t, a_t, r_{t+1}, s_{t+1}>$
Environment
action
Actor-Critic PPO
Trajectory memory
Actor Model
Critic Model
Update $\pi_\theta$ with the gradient $\nabla L^{CLIP}(\theta)$ by SGD
$\pi_\theta(a_t|s_t)$
$\pi_{\theta_{old}}(a_t|s_t)$
$r_t(\theta)$
$L^{CLIP}(\theta)$
$\delta_t$
$\hat{A}_t$
$V_\mu\,(s_t)$
$V_\mu\,(s_{t+1})$
$L^V(\mu)$
Update $V_\mu$ with the gradient $\nabla L^V(\mu)$ by SGD
Mini-batch Samples (length: $U$)
$[<s_{t_0}, a_{t_0}, r_{t_0+1}, s_{t_0+1}>, <s_{t_0+1}, a_{t_0+1}, r_{t_0+2}, s_{t_0+2}>, \ldots, < a_{t_0+U-1}, s_{t_0+U-1}, r_{t_0+U}, s_{t_0+U} >]$

We use a simple RNN (in the future attention) for the actor model. This model samples instructions from the categorical distribution formed over the set of all possible instructions. Combined with an MLP for estimating the value of a given action this forms the general advantage estimation central to Proximal Policy Optimization or PPO. Another critical component of the PPO method in particular is its bounding of KL-divergence with an additional hyper-parameter which reduces some of the variance of the policy estimates by restricting how much the policy can change epoch over epoch.

Policy gradient methods are a marriage of actor critic methods and trust region optimization used to estimate the direction of the policy via sampling and temporal difference learning. Core to this method is the policy gradient theorem which states that the derivative of the total reward:

$$ {}_{\theta}J(\theta) = \sum_{s \in S} d^{\pi}(s) \sum_{a \in A} Q^{\pi}(s,a) \ {}_{\theta}\pi_{\theta}(a|s) $$

can be expressed as

$$ J(\theta) = \mathbb{E}_{\pi}[G_t \ ln\pi_{\theta}(A_t|S_t)] $$

That is, the reward $J(\theta)$ is the sum over the stationary state distribution and the action value function parameterized by the policy (with its current parameter configuration). This reward is equivalent to the expectation over the policy estimates which simplifies taking the derivative. Estimating the policy in this way is unbiased but high variance. Two crtiical components mitigate this variance 1. introducing the abstraction of a batch, along with methods that increase batch diversity 2. the introduction of a critic $Q_w(a|s)$ , namely instead of just evaluating the value at the first step it is evaluated on the second state as well. This single-step value estimate of the actor $\pi_{\theta}(a|s)$ acts as a baseline. In fact, single-step value function is the optimal baseline for reducing variance of the policy.

The fundamental policy gradient algorithm REINFORCE uses Monte Carlo methods to estimate return using samples of episodes to update policy parameters. in order to reach an estimate of $J_{\theta}$ the following process is followed until convergence criteria is met 1. intiialize the policy parameters $\theta$ at random 2. generate a single trajectory (on policy) $\pi^i_{\theta} : S_1, A_1, R_2, S_2, A_2....S_T$. Each state in **Climb** is an instruction that is appended to previous states in the trajectory until the parameter *maximum sequence length* is reached and then becomes a candidate program. 3. For $t = 1....T$ 1. estimate the return $G_t$ 2. update the policy parameters $\theta \leftarrow \theta\alpha\lambda^t G_t \ {}_{\theta}ln\pi_{\theta}(A_t|S_t)]$

We are using proximal policy optimization rather than vanilla REINFORCE. The difference lies in step 3.2. In PPO, a probability ratio is computed between old and new policies (KL divergence)

$$ r(\theta) = \frac{\pi_{\theta}(a|s)}{\pi_{\theta_{old}}(a|s)} $$

and instead of $J(\theta)$, new hyperparameters are introduced and the reward is

$$ J^{CLIP}(\theta) = \mathbb{E}[min(r - \theta)A_{\theta}(s,a), clip(r(\theta), 1 - \epsilon, 1 + \epsilon)A_{\theta}(s,a) $$

This bounds how much the policy estimation can change step over step biasing and restricting the search for an optimal policy in a greedy manner. While finding surprising policies was needed in AIMEE due to the highly anomalous, needle-in-the-haystack nature of ROP attacks, in this problem we want continuous steady progress in training.

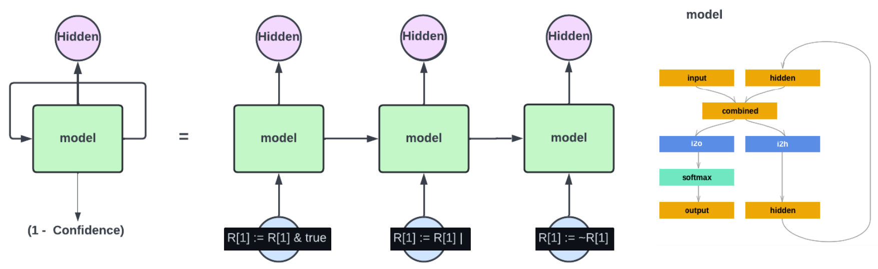
**Risk Aware Policy Gradient Methods** The conventional loss function for policy gradient methods $J_{std}\theta$ optimizes both for the expectation of the policy diistribution. However, as symbolic regression tasks (like our work in AIMEE) is primarily a search problem, we are interested in optimiizing not for the expectation but for the top-k trajectories (in our case linear programs) sampled from the policy. Therefore we use a risk-aware policy-gradient method $J_{risk}(\theta;\epsilon)$ whixh maximizes the conditonal expectation of rewards above the $(1-\epsilon)$ quintile.

**The Fitness Landscape**

Agents in a reinforcement learning problem learn by interacting with their environment. In our case the environment is a simple compiler for linear programs. The action space is the **set of all possible instructions constrained by the static, configured architecture**. Environments contain states which are often not markov independent. In our case a state is the partial linear program at $t = 0$ , as such it is best described as a graph neural network embedding of the control flow graph / execution trace of the program or a constrained language model (ultimately there are benefits depending on the task to describing it as both but current SOTA research is biased to the graph representaiton). The environment that **Climb** has to negotiate has a complex action space and the probabilites of the sampled action are conditioned on non-adjacent instructions from the sub-sequence that has already been built in the given episode. Readers who have been bombarded by buzzy paper titles in the last five years will recognize this as a problem that is best served by an attentional model vs. a simple autoregressive one - both due to the sparsity of the action space, fixed vocabulary size, and the fact that a locality bias would not serve this problem. This informed our choices of the architecture for the actor and critic network.

Figure 1.1: Instrinsic Reward based on program novelty

We use two different kinds of reward, one is a non-discounted and episode level reward computation that is typical of symbolic regression tasks. Strict reliance on extrinsic rewards constrain program performance to the limits of the imagination of the ressearcher coding it. To avoid this, we add an intrinsic reward, specifically one based off of model uncertainty, This consists of a simple RNN trained across candidate programs on execution traces. Given a candidate program parameterized by some input that could be generated randomly or from a knowledge of the program's typical behavior based on prior observations, predict the output of the program. This is a kind of self-supervision because it is trivial for us to compile the program and produce the ground-truth output. Over time, the model regularizes across examples of candidate programs and



learns patterns of instruction sequences (ideally sub-rotuines) as they appear in observations of the non-stationary distribution of candidate program episode over episode. Reward is simply $1 - confidence$, rewarding a policy for generating unfamiliar sub-sequence examples and encouraging more diverse exploration of the control flow graph as well as weighing the probability of a given candidate program being sampled as part of a batch. Eventually performance of a graph neural network should be evaluated as it retains more semantic information than an RNN and has been adapted by prior work to integrate with the sequential nature of training on the execution trace.

Another characteristic of this framework that makes it promising for both tech transfer and future DARPA programs is the fact it can be extended with richer embeddings, a kind of expressive scalability, and also extended to limited knowledge regimes where there are a low number of observation tuples, noise/error in the observations, or lack of knowledge of system architecture constraining the possible program being observed.

## Software

All of the software developed in connection with this project can be found in our public Github repository, and is licensed under Version 3 of the GNU Public License.