Library for Evolutionary Algorithms in Python (LEAP)

GECCO 2020 —– EvoSoft

Mark A. Coletti
colettima@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee

Eric O. Scott
escott8@gmu.edu
George Mason University
Fairfax, Virginia

Jeffrey K. Bassett
jbasset1@gmu.edu
George Mason University
Fairfax, Virginia

ABSTRACT

There are generally three types of scientific software users: users that solve problems using existing science software tools, researchers that explore new approaches by extending existing code, and educators that teach students scientific concepts. Python is a general-purpose programming language that is accessible to beginners, such as students, but also as a language that has a rich scientific programming ecosystem that facilitates writing research software. Additionally, as high-performance computing (HPC) resources become more readily available, software support for parallel processing becomes more relevant to scientific software.

There currently are no Python-based evolutionary computation frameworks that support all three types of scientific software users. Moreover, some support synchronous concurrent fitness evaluation that do not efficiently use HPC resources. We pose here a new Python-based EC framework that uses an established generalized unified approach to EA concepts to provide an easy to use toolkit for users wishing to use an EA to solve a problem, for researchers to implement novel approaches, and for providing a low-bar to entry to EA concepts for students. Additionally, this toolkit provides a scalable asynchronous fitness evaluation implementation friendly to HPC that has been vetted on hardware ranging from laptops to the worlds fastest supercomputer, Summit.

ACM Reference Format:
Mark A. Coletti, Eric O. Scott, and Jeffrey K. Bassett. 2020. Library for Evolutionary Algorithms in Python (LEAP): GECCO 2020 EvoSoft. In GECCO '20: The Genetic and Evolutionary Computation Conference, July 08–12, 2020, Cancun, MX. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

GECCO '20, July 08–12, 2018, Cancun, MX

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Evolutionary algorithms (EAs) are a powerful tool for approaching all manner of complex design and optimization problems that would otherwise be intractable. But they bring with them a non-negligible implementation cost: as with machine learning more broadly, it takes time and skill to build a suitable EA solution for a new application. In our experience working with partners in academia and industry, we often find that teams who want to apply EAs to their problems rapidly hit up against a complexity barrier. A basic, generational EA that runs on a single CPU is easy enough to implement by hand and from scratch; but the moment that more sophisticated features are needed—such as distributed fitness evaluation, convenient visualization, or multi-objective optimization—the complexity and error-proneness of the software grows by an order of magnitude, and development grinds to a snail's pace.

Many software libraries are available that offer remediation for these challenges, and most common programming languages sport at least one general-purpose EA framework (such as Evolving Objects for C++ [15], and ECJ for Java [27]). But—as so often happens with any software—most libraries bake quite a few assumptions and constraints into their basic design. The structure of the library itself dictates which algorithms are easy to create, and which are difficult.

We think that the EA community has not quite succeeded in creating libraries that do for metaheuristics what TensorFlow [1], Keras [5], and PyTorch [20] have done for neural networks: provide a highly flexible and easy-to-learn framework that both assists researchers in implementing and comparing novel algorithms, and also gives industry practitioners easy access to state-of-the-art solutions that scale to modern HPC infrastructure.

In this paper, we present a new effort to design an EA framework that has both the flexibility needed for rapid expert prototyping and the gentle learning curve expected by educators and much of industry. Our core approach takes its inspiration from De Jong's theoretical EA framework, which provides a unified view of how different kinds of evolutionary algorithm can be seen as variations on a basic theme of operators acting on a population [8]. This operator-centric view of EA design has led us to create the Library for Evolutionary Algorithms in Python (LEAP). LEAP's signature lies in

```
generation = 0
while generation < 50:
    offspring = pipe(parents,
                     ops.tournament,
                     ops.clone,
                     ops.mutate_bitflip,
                     ops.evaluate,
                     ops.pool(size=len(parents)))

    parents = offspring
    generation += 1
```

Listing 1 `Simple LEAP syntax example.` This shows the use of `pipe()` to implement a simple genetic algorithm (GA). It uses binary tournament parent selection and bit-flip mutation to accumulate offspring that become parents for the next generation. Notice that we did not include a crossover operator—we show a similar example with crossover in Listing 5.

leveraging Python's support for convenient functional syntax to concisely describe the operator pipeline at the heart of an algorithm. A preview of this syntax is given in Listing 1 and is further discussed in §3.2.

This concise, operator-centric style of describing EAs offers a powerful approach for meeting the needs of three main classes of users:

(1) Researchers, who want to rapidly customize and design novel metaheuristics, composing many algorithm components (ex. operators, distribution facilities) that can be swapped in and out for various experiments
(2) Engineers, who want convenient and feature-rich software that they can rapidly use to solve problems—often with the help of considerable HPC infrastructure
(3) Educators, who want a well-documented environment for training students, and strong visualizations to demonstrate different aspects of optimization algorithms' behavior

In LEAP's first release, we have focused on implementing a divergent set of EA classes, so that we can solve problems that they pose for a pipeline-based design and preserve clean configuration syntax for all kinds of algorithms. LEAP currently supports classic generational and steady-state EAs, distributed fitness evaluation, cooperative coevolution, island models, and Pitt-approach learning classifier systems. Because it has come to play an outsized role in many applications in recent years, we have given special attention to distribution support for multiple types of clusters, and in particular to asynchronous steady-state evolutionary algorithms (ASEAs), which are especially effective at taking advantage of large HPC clusters. On the education and demonstration front, we also offer a seamless approach to inserting visualization and data-collection probes into all of these algorithms. These in turn integrate naturally with Jupyter Notebook.

2 BACKGROUND

There are a number of very capable Python-based libraries for creating Evolutionary Algorithms. However, the main open issue that we faced with all of these was the inability to define the operator pipeline on the fly. The Java-based ECJ library [17, 27] comes the closest to having this type of ability, but even this had some limitations that led us to develop this library.

The original impetus for developing LEAP came while performing research that involved the tracking the performance of algorithms and their components. We wanted to be able to place special probes into the pipeline to measure the effects that operators were having on the population. This was done by measuring certain aspects of individuals both before and after a reproductive operator made modifications to them. Having the ability to quickly and easily reconfigure the locations of these probes was critical to the success of that research.

It quickly became evident that the benefits of a flexible pipeline implementation extend far beyond specialized metric probes: another advantage is that a new algorithm can essentially be defined just by changing the various operators and their order. This can reduce the amount of code required to experiment with new algorithm designs and new types of operators.

Some of the most capable Python-based libraries include DEAP [10], Inspyred [11], and jMetalPy [3]. All of these offer the ability to customize many aspects of the EA that one can create, including the representations and algorithms themselves. However, algorithm variants must all be defined in the code itself, and do not offer the ability to add probes by simply modifying a data structure.

The Pyevolve [21], pyeasyga [23], and pySTEP [16] libraries each tend to be more fixed in their algorithmic structures, and thus more difficult to adapt to implement new algorithms.

High Performance Computing (HPC) is also becoming an increasingly important environment for EAs, as sophisticated clusters of computers become increasingly ubiquitous. The "embarrassingly parallel" inherent nature of EAs make them particularly suitable for HPC platforms, but there is very little support in the way of Python-based EA toolkits to take advantage of HPC capabilities in the form of parallel fitness evaluations. DEAP supports concurrent evaluations, but in a synchronous manner that does not make the most efficient use of resources [26], though its reliance on SCOOP [13] does mean it can leverage multiple hardware resources. Similarly, Inspyred takes a map/reduce approach to evaluating individuals that is also synchronous in nature, and requires significant modifications to work with other third party tools, such as `schwimmbad` [22], to leverage concurrent fitness evaluations on available cluster hardware. jMetalPy, pyeasyga, and pysteps do not support parallel fitness evaluation.

LEAP's readability has also benefited greatly from taking advantage of Python's support for basic functional programming patterns. Some research platforms for EA experiments in other languages (ex. Scala) have taken this further, providing purely functional strategies for defining algorithms in terms of algebraic data types (i.e. semigroups, monoids) [19]. We make more lightweight use of functional programming here, using it to simplify the syntax of a fundamentally imperative framework.

3 DESIGN

There are three basic ways to use LEAP as a library, depending on how much control the user wishes to exert over the details of the evolutionary process.

The simplest approach is to use a built-in solver, which uses a default representation and operator set to optimize a user-provided function (in the tradition of MATLAB and SciPy's `fsolve()` function). Listing 2 shows a complete example of how to use LEAP in this fashion.

```
from leap.simple import ea_solve

def function(values):
    return sum([x ** 2 for x in values])

ea_solve(function,
         bounds=[(-5.12, 5.12) for _ in range(5)],
         mutation_std=0.1)
```

Listing 2 A basic "practitioners" interface for an evolutionary algorithm consists of a domain-specific function call.

The second approach is to use a richer metaheuristic function, which offers full control over the operator pipeline, representation, and other design decisions that are specific to a particular metaheuristic family. This interface still hides the details of the evolutionary loop inside a high-level function, but allows the user to apply a deeper understanding of LEAP's architecture to choose operators and other components to compose into a customized algorithm. The result is typically a block of about 20 lines of code that describes a full algorithm and its parameters. This high-level view of the algorithm doubles as a concise summary of a complete experiment (playing the role that a parameter file would play in some other frameworks).

Because LEAP takes a very general view of evolutionary algorithms, some well-known algorithm families are more naturally implemented via a special choice of operators or representation than as first-class metaheuristics. For example, Figure 1 shows two uses of LEAP's generic `multi_population_ea()` metaheuristic. In one, a special evaluation operator is used to produce a cooperative coevolutionary algorithm, while a different choice of operators in the other yields an island model. In a similar vein, we implement Pitt-approach learning classifier systems [9] as a special representation component that can be used with many kinds of metaheuristic functions.

Finally, it is also straightforward to write one's own evolutionary loop from scratch, manually stitching together pipeline operators, population initialization, and other components in unique ways. This allows power users and researchers maximum flexibility, albeit potentially at some cost in terms of conciseness. It also offers a viable fallback for cases where a simple linear pipeline of operators cannot express the desired algorithm: if an algorithm requires operators to interact along a non-trivial Directed Acyclic Graph (DAG), for example, a completely custom approach may be preferred.

3.1 Top-level components

Three classes work together to represent and evaluate solutions: `Individual`, `Problem`, and `Decoder`. As shown, `Individual` is the design's center-piece, and encapsulates a posed solution for a `Problem`. The `Problem` implements the semantics for a given problem to be solved by an EA, which `Individual` uses to compute its fitness when requested to do so. The `Problem` is also the canonical source for comparing fitness of two `Individuals` to define the meaning of "better than" and "worse than." Lastly, the `Decoder` is the mechanism whereby an internal "genome" that stores the unique state for a given Individual is decoded into a phenome—i.e. values meaningful to the `Problem` for fitness evaluation. For example, this may entail decoding a series of bits into real-valued numbers used as fitness function arguments, or decoding a list of thresholds for a rule system into an executable controller.

As with most EA frameworks, practitioners wishing to use LEAP to solve a real-world problem will need to define a `Problem` that implements the semantics associated with that problem and, depending on representation, an appropriate `Decoder`; these classes would then be "plugged in" to the existing framework of EA pipeline operators (which we will explain in §3.2). The loose coupling between the `Individual`, `Decoder`, and `Problem` classes facilitates evolutionary computation (EC)-related research into new representation and encoding strategies, as well as unusual `Problem` implementations.

LEAP comes with a fully documented stock of common real-valued and binary benchmark `Problem` implementations, such as Griewank, Rosenbock, Rastrigin, and MAX ONES. There is also support for scaling, translating, and rotating real-valued functions, which is important for benchmark testing, particularly for removing axis and origin-based biases when exercising novel real-valued optimization operators [25]. This set of common benchmark functions—as well as standard operators defined within the general, theoretical framework—are also ideal for educators to implement examples of common EA configurations to study design decision ramifications for selection, representation, and reproduction, such as GAs, evolutionary programming (EP), and evolutionary strategies (ES).

Figure 1: Two instantiations of LEAP's generic **`multi_population_ea()`** metaheuristic. In this case, we can convert a cooperative coevolutionary algorithm into an analogous island model simply by replacing a single operator in the pipeline.

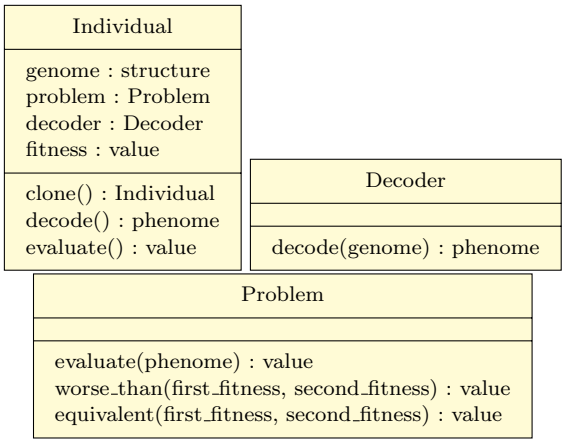

Figure 2: This Unified Modeling Language (UML) diagram shows the Individual, Decoder, and Problem classes.

Fig. 2 shows the UML diagram for the Individual, Decoder, and Problem classes. Each Individual has a genome, which contains state representing a posed solution; it is listed as a "structure" because that encompasses the general notion that the genome can be a sequence or a matrix or a tree or some other data structure. In practice, however, a genome is usually a binary string or a real-valued vector. Every Individual is connected to an associated Problem. The Individual relies on the Problem to evaluate its fitness and to compare itself with another Individual to determine the better of the two.

clone() will create a duplicate of a given Individual; the new Individual gets a deep copy of the genome and refers to the same Problem and Decoder. evaluate() calls decode() to translate the genome into phenomes, or values meaningful to the Problem, and then passes those values to the Problem where it returns a fitness. This fitness is then assigned to the Individual.

The abstract-base class, Decoder has one function intended to be overridden by sub-classes, decode(), that returns a phenome meaningful to a given Problem, which is usually a sequence of values. There are a number of supplied Decoder classes in LEAP, mostly for converting binary strings into integers or real values.

The Problem abstract-base class has three abstract methods. evaluate() takes a phenome that was decode()d from an Individual's genome, and returns a value denoting the quality, or fitness, of that individual. Problems are also used to compare the fitnesses between Individuals. worse_than() returns true if the first individual is less fit than the second, which is a determination suited for the Problem as it "understands" the respective problem domain. Similarly, equivalent() is used to determine if two given fitnesses are effectively the same.

3.2 Pipeline Workflow

```
> grep newacronym acronyms.tex |  awk '{print $1}' |
    egrep -o '{[a-z]{4}}' | sed -e 's/{//' -e 's/}//'
alcc
dice
dsge
```

Listing 3 `Example *nix command line use of pipes.` This shows how the output of one text processing command is used as the input for the next, in this case to extract acronyms of four letters in length from a LaTeX source file.

Tying `Individual`, `Problem`, and `Decoder` together is the notion of an "operator pipeline", which is similar to the *nix (i.e. the family of Unix operating systems) command-line text processing support. Listing 3 demonstrates the basic concept of a pipeline with an example *nix shell command line that extracts four-letter acronyms from a LaTeXfile. The

Figure 3: **`Typical pipeline.`** This shows the typical sequence of pipeline objects. The first is a collection of prospective parents from which a selection operator draws individuals. These individuals are then cloned, crossover applied, mutated, and then evaluated by the next set of operators. The last component pools a specified number of offspring from the preceding pipeline components.

output of the first `grep` is piped as input to the next command, `awk`, which is then, in turn passed to the next command, and so on. This concept allows for creating arbitrary and non-trivial command sequences without having to write a program or script.

An EA can be similarly thought of as a pipeline of operators that start with a "source" of data, which is typically a population of individuals representing prospective solutions to a given problem, through which selected individuals are passed through a pipeline of functions into a "sink" of newly created offspring (while perhaps less obvious, it is also typically easy to model estimation of distribution algorithms (EDAs) in this way, such as PBIL [2] or CMA-ES [12]—see discussion in [27]). Along the way, pipeline operators will be applied to individuals that pass through, and may include selecting a subset of individuals from a collection, making changes to individuals, generating a visualization, or simply reporting some salient individual characteristic. Each operator will accept and send either a collection of individuals or a single individual, depending the specific operator semantics. For example, a selection operator will typically select from a sequence and return an individual, whereas a mutation operator will both accept and return a single individual at a time.

LEAP embodies this concept by supporting a pipeline that pulls individuals from a sequence of prospective parents, applies a sequence of functions, or operators, to those individuals, and collects them in a final sequence of offspring. Some run-time type-checking is also included to ensure that the chain of operators accepts and returns compatible types at each stage. The process is depicted in Fig. 3: one pass through this pipeline will generate offspring from which survival selection can be applied to provide prospective parents for the next generation. These new parents are then supplied as a collection for the next invocation of the pipeline. This process repeats until a halting criteria is met.

`toolz` is a popular third-party package that provides iterator and functional programming support for Python applications [24]. One of these is `pipe()`, which supports semantics similar to the aforementioned *nix command line. It takes a data source as its first argument, which is supplied as input to the next argument, which is a function. The output from that function is fed as input to the next function, which is repeated in daisy-chain fashion to the last function. The output of the last function is returned.

```
>>> double = lambda i: 2 * i
>>> toolz.functoolz.pipe(3, double, str)
'6'
```

Listing 4 `A simple toolz pipe example.`

Listing 4 shows a simple example of a `toolz pipe()` that has a 3 as a data source that is passed into the given lambda function, which doubles the value; in turn, the output from that is passed to `str()`, which converts its input into a string, and then that result is returned.

```
generation = 0
while generation < 50:
    offspring = pipe(parents,
                     ops.tournament,
                     ops.clone,
                     ops.uniform_crossover, # new
                     ops.mutate_bitflip,
                     ops.evaluate,
                     ops.pool(size=len(parents)))

    parents = offspring
    generation += 1
```

Listing 5 `Simple LEAP example with crossover.` This is the code in Listing 1, but with a uniform crossover operator inserted after mutation. Note that now two parents are selected to satisfy the crossover operator, but the selection operator did not change; instead, the selection operator dynamically adjusts to the needs of the downstream operators and returns two individuals instead of just one, as was the case when there was no crossover.

Most algorithms in LEAP use `pipe()` as their fundamental design pattern. Listing 1 shows a LEAP code snippet that exemplifies using toolz `pipe()` to create offspring in a simplistic GA. Parents will initially be a sequence of randomly generated individuals from which parents are selected by the next operator via binary tournament selection. The parent is then cloned so that the subsequent operations do not damage the original parent. Then the clone is mutated, evaluated, and added to a pool of offspring. This process continues until the pool has as many offspring as there were parents. Then the offspring are returned to become the parents for the next generation. This process repeats for 50 generations.

One advantage of this approach is that it is trivial to modify this pipeline given its built-in dynamic nature. For

example, Listing 5 inserts a uniform crossover operator at Line 6. Even though this operator needs two selected parents (rather than the one needed for the previous mutation-only example), the same binary tournament selection operator on Line 4 did not have to be changed. Instead, it automatically compensates for the demands of the needs of downstream operators and implicitly returns the two necessary individuals. Similarly, if the size of the desired number of offspring was increased, which would also put more demand on selection, the selection operator would, once again, supply the additional selected parents as necessary.

3.3 Operators

```
1 @toolz.curry
2 def tournament(population: List, k: int = 2) -> Iterator:
3     while True:
4         choices = random.choices(population, k=k)
5         best = max(choices)
6
7         yield best
```

Listing 6 `Example of a pipeline operator.` This shows the implementation of the tournament selection operator used in the previous examples. It is a generator function that will yield an individual from a set of individuals randomly drawn from the population.

```
1  @toolz.curry
2  def uniform_crossover(next_individual: Iterator, p_swap:
       float = 0.5) -> Iterator:
3      def _uniform_crossover(ind1, ind2, p_swap):
4          for i in range(len(ind1.genome)):
5              if random.random() < p_swap:
6                  ind1.genome[i], ind2.genome[i] = ind2.
       genome[i], ind1.genome[i]
7          return ind1, ind2
8
9      while True:
10         parent1 = next(next_individual)
11         parent2 = next(next_individual)
12
13         child1, child2 = _uniform_crossover(parent1,
       parent2, p_swap)
14
15         yield child1
16         yield child2
```

Listing 7 `Another example of pipeline operator.` This shows the uniform crossover operator and how it couples with the preceding and following pipeline operators via calls to `next()` and `yield`.

The dynamic nature of the operators is achieved by loosely coupling a series of Python generator functions (i.e., functions that return iterators). Listing 6 shows shows an example of this approach in the implementation for the tournament selection operator used in Listing 1 and Listing 5. `tournament()` accepts a population as its first parameter, which it will sample from, and an optional second parameter for the number of individuals to be drawn. By default, this implements binary tournament selection, given that $k = 2$. It is a generator function, in that it will always `yield` the best of the selected individuals; so if a downstream operator wants more individuals, this generator function will just `yield` more to meet the demand.

The `@toolz.curry` at the top of the function is a Python function decorator that allows the operator to be used as a partial function (akin to the `functools.partial()` capability in the Python standard library). That is, it allows us to fix the keyword arguments of the function to produce an operator with a specific configuration. This has much the same effect as calling the constructor on a class. For example, if we write `ops.tournament(k=3)` within our `toolz.pipe()` invocation, an operator will be added to the pipeline that selects three individuals (instead of the default two).

Listing 7 shows another example of a pipeline operator: the uniform crossover operator that was used earlier. This further demonstrates the loose coupling between pipeline operators. The `uniform_crossover()` function uses two consecutive calls to `next()` to pull in individuals from upstream, performs the uniform crossover, and then `yield`s the two offspring for the next pipeline operator. Moreover, it first `yield`s one offspring based on demand, then the second, thus ensuring that both offspring eventually move down the pipeline. As before, the `@toolz.curry` decorator is there to allow for adjusting the probability for gene swapping by optionally setting `p_swap`.

A fundamental limitation of the pipeline design pattern is that it only gives operators access to the output of the previous operator. We get around this by permitting operators to have both direct and indirect inputs: some operators take a `context` parameter, which allows us to pass a shared key-value store into multiple operators, where they can write and read shared state information. This proves useful for many algorithmic mechanisms—for example, we use this for the coevolution and island model implementations in Figure 1. The example operators we present here also all use functions and `@toolz.curry` to implement pipeline operators. This is the preferred pattern for working with operators in LEAP. But operators can also be implemented as classes when it is useful to do so (for example, to maintain an internal state): both approaches have their strengths and limitations.

3.4 Metrics and Visualization

Visualizing algorithm behavior and saving data out to a file stream are two of the most important activities in a research workflow. In our view, the less ad-hoc code necessary to achieve these two goals, the better. LEAP accomplishes these tasks via specialized pipeline operators. LEAP provides operators that write fitness information to files, that collect genome information about individuals during evolution, and that produce real-time visualizations of best-so-far curves or the population's location on a fitness landscape. These are inserted directly into the pipeline like any other operator, as shown in Listing 8, and the visuals are based on Matplotlib,

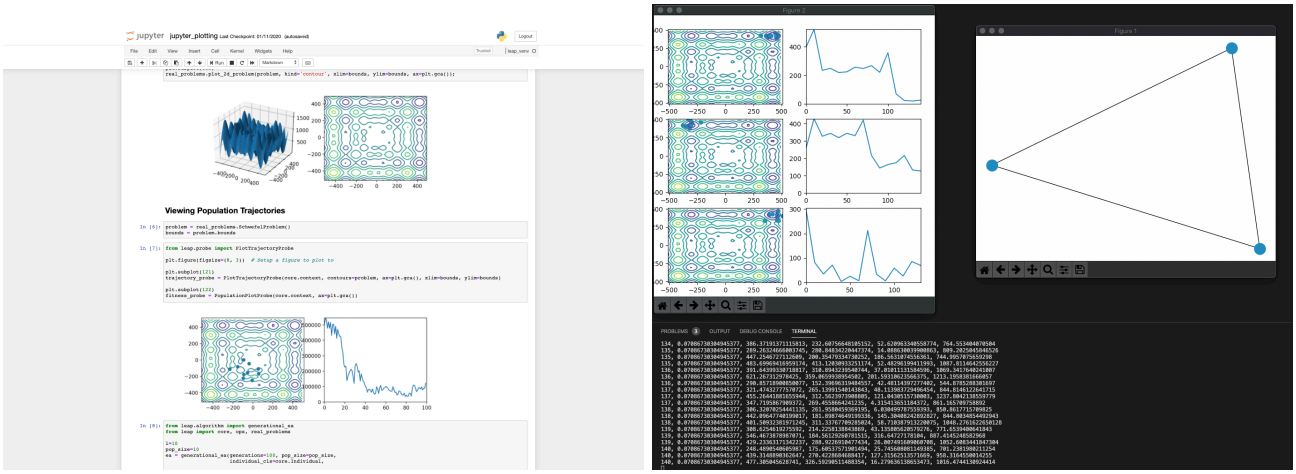

Figure 4: LEAP's visualization operators can be used interactively in conjunction with Jupyter Notebook (left), or as a standalone application (right). The example on the right demonstrates an island model running in real time, displaying plots for the population phenotypes and best-of-generation fitness for each island, and a network diagram of the topology of allowed migration routes.

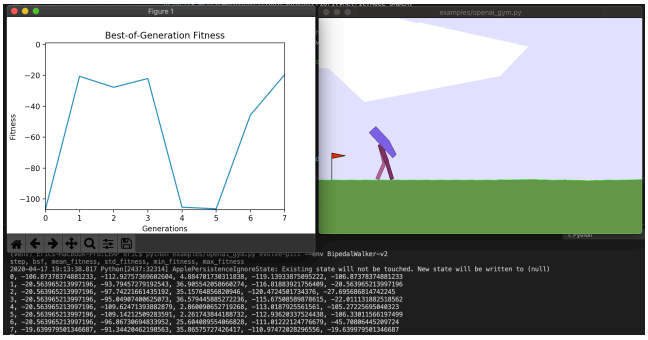

Figure 5: An experiment that uses a Pitt-Approach learning classifier system to learn a robot control problem in the OpenAI Gym simulation set.

```
generation = 0
while generation < 50:
    offspring = pipe(parents,
                     ops.tournament,
                     ops.clone,
                     ops.mutate_bitflip,
                     ops.evaluate,
                     ops.pool(size=len(parents)),
                     probe.FitnessStatsCSVProbe(core.
     context, stream=sys.stdout),
                     probe.PopulationPlotProbe(core.
     context))

    parents = offspring
    generation += 1
```

Listing 8 Simple LEAP example with `probe` operators. This pipeline captures fitness information to `stdout`, and plots an animated best-of-generation fitness curve in a pop-up window.

which allows them to integrate seamlessly into a Jupyter Notebook environment (see Figure 4).

Fitness functions called by the `evaluate` operator can also produce visualizations. This is often useful when fitness evaluation involves executing a complex simulator to evaluate the behavior of a controller. For example, Figure 5 shows LEAP executing a Pitt-approach learning classifier system to learn a controller for a simulated robot, where fitness evaluation involves launching an instance of the OpenAI Gym simulation environment [4].

3.5 Support for HPC

EAs are "embarrassingly parallelizable" in that fitness evaluations can be done concurrently, which makes them particularly suitable for HPC environments. EAs are also scalable in this context in that the number of such parallel evaluations can ramp up or down depending on available computational capability—so EAs can leverage parallel evaluations on everything from multicore desktop processors to all the nodes on the world's current largest supercomputer, Summit [14].

There are two broad approaches to implementing concurrent EAs. The first is to use a map/reduce approach such that with each generation newly created offspring are scattered to available resources to be concurrently evaluated. Once all the offspring are evaluated, they are incorporated into the subsequent set of parents as with any traditional by-generation approach. However, the problem with this synchronous approach is that no further progress can be made until the last offspring is evaluated, which means that the computational resources that finish evaluating offspring earlier will sit idle until the next generation. Alternatively, an asynchronous approach continuously updates a single pool

Algorithm 1 `ASEA design`. This shows details on how we asynchronously update a pool of individuals of posed solutions.

```
P_0 ← init_pop()                          ▷ Initial population
b ← size(P_o)                             ▷ Initial number of births
P_p ← ∅                                   ▷ Initialize an empty pool
async_eval(P_0)                           ▷ Fan out population to workers
while I_e ← evaluated() do                ▷ Next evaluated individual
    if is_full(P_p) then
        if I_e > min(P_p) then            ▷ Replace only if better
            remove(P_p, min(P_p))         ▷ than weakest in pool
            insert(P_p, I_e)
        end if
    else                                  ▷ Pool not full yet, so just insert
        insert(P_p, I_e)
    end if
    if b < birth_budget then
        I_p ← select(P_p)                 ▷ Select parent
        I_o ← reproduce(I_p)              ▷ Create offspring
        async_submit(I_o)                 ▷ Send to worker for evaluation
        b ← b + 1                         ▷ Increment births
    end if
end while
return P_p                                ▷ Return best individuals
```

of individuals: as soon as an offspring is done being evaluated, it is incorporated into the pool, and immediately a new offspring is created and begins evaluation on that resource, thus ensuring that resource minimizes time being idle [26]. This ASEA is shown in more detail in Algorithm 1.

LEAP supports both synchronous and asynchronous distributed fitness evaluation methods, and uses the Python distributed process package Dask to implement both approaches [7]. Dask easily allows LEAP to fan out fitness evaluations on a local machine, to nodes on a cluster, or to supercomputer nodes while the EA is unaware of the environment in which it is running. The same code can scale to all those platforms without change.

```
final_pop =
    steady_state(dask_client,
                 births=100,
                 init_pop_size=10,
                 bag_size=10,
                 initializer=
                    create_binary_sequence(4),
                 inserter=greedy_insert_into_bag,
                 decoder=core.IdentityDecoder(),
                 problem=MaxOnes(),
                 offspring_pipeline=
                    [ops.random_selection,
                     ops.clone,
                     ops.mutate_bitflip,
                     ops.pool(size=1)])
```

Listing 9 An ASEA metaheuristic function in LEAP, applied here to solve the MAX ONES problem.

Listing 9 shows a LEAP code snippet that demonstrates an ASEA to solve the MAX ONES problem. We give it a Dask client object that could be running a scheduler with workers for all the cores of a desktop, or workers running on a supercomputer, or any configuration in between. **`births`** indicates a budget for 100 births, including those of the initial population. The initial randomly generated population size is indicated by **`init_pop_size`**, and will be ten. Similarly, the size of the pool, or bag, is set by **`bag_size=10`**, which means there will be at most ten individuals that will be constantly updated from newly updated offspring. **`initializer`** is a function used to generate the genomes for all the individuals, and in this case we will generate random four-bit sequences. Given that this is the MAX ONES problem where fitness is just the number of ones, as dictated by **`problem`**, that means we set the **`decoder`** to the identity decoder. This means we do not need to decode those bits into phenomes since we will be directly counting the bits in the genome. **`inserter`** sets the policy for how newly evaluated individuals are inserted into the pool; in this case the new offspring is only inserted into the pool if it is better than the current least fit individual. And, finally, the **`offspring_pipeline`** specifies how we want to create offspring. In this case we randomly select a parent from the ten individuals in the pool, clone it, perform bit-flip mutation, and then send the newly created offspring via the Dask scheduler to an available computing resource for evaluation.

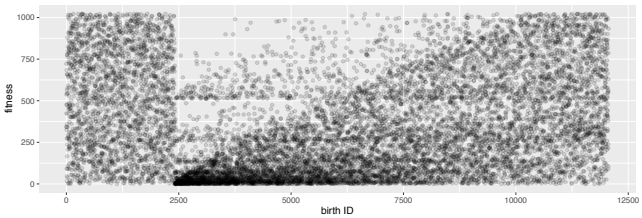

Figure 6: **Results of running LEAP on Summit.** This shows the results of a single scaling test run of LEAP on 402 nodes Oak Ridge National Laboratory (ORNL)'s Summit supercomputer with six dask workers per node, for 2,412 Dask workers. The test problem, SLEEPER, is a maximization problem where the fitness is the number of seconds to sleep.

LEAP has been successfully used as a deep-learning training data diagnostic tool on ORNL's Summit supercomputer [6] using an ASEA. Fig. 6 shows the results of a single run on 402 nodes of six dask workers per node, for a total of 2,412 parallel evaluations at a time. The test problem was SLEEPER, where the 10-bit genomes represent an integer value for how long to sleep during evaluation—this is a maximization problem, so the longer an evaluation takes, the fitter the corresponding individual. On the left can be seen the block of 2,500 initial randomly generated individuals, and after that are shown the evaluations of each newly created individual from that initial population. At first the new individuals are dominated by the individuals with the shortest evaluation time, but eventually the fitter individuals that take longer begin to emerge. These results demonstrate that

LEAP can scale from low-end machines, such as laptops, to the largest of supercomputers, such as Summit [18].

4 FUTURE WORK

This initial release of LEAP demonstrates that our basic EA design, visualization, and distribution mechanisms can support a wide variety of EA families, including coevolution and ASEAs. One key focus for future work will involve fleshing out the algorithms, operators and representations so that a large set of capabilities will be available. This includes canonical EA algorithms such as Genetic Programming (GP), Evolution Strategies (ES), Covariance Matrix Adaptation ES (CMA-ES), and rule-based classifier systems. Other stochastic search algorithms will also be developed, such as Particle Swarm Optimization, Gradient Descent, and Tabu Search.

Work will also continue on making sure all the necessary educational materials will be available for all types of users, from novice to expert. These include online tutorials, class documentation along with numerous practical examples of various algorithms and approaches.

Finally a long term goal of ours is to make sure LEAP is compatible with at least one of the machine learning ecosystems, such as Scipy or Scikit-learn. Additionally we want to make sure it is easily compatible with other machine learning libraries such as PyTorch, TensorFlow, and NLTK.

ACKNOWLEDGMENTS

We would like to thank Thomas Papatheodore of the ORNL Leadership Computing Facility (OLCF) for his helpful assistance in debugging Summit problems. And, lastly, we would like to thank the OLCF for their generous grant of 20,000 Summit node-hours via Director's Discretion grant CSC363 that made the Summit results possible.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: large-scale machine learning on heterogeneous systems, 2015. URL: https://www.tensorflow.org/. Software available from tensorflow.org.

[2] Shumeet Baluja. Population-based incremental learning. a method for integrating genetic search based function optimization and competitive learning. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Dept Of Computer Science, 1994.

[3] Antonio Bentez-Hidalgo, Antonio J. Nebro, Jos Garca-Nieto, Izaskun Oregi, and Javier [Del Ser]. jMetalPy: a Python framework for multi-objective optimization with metaheuristics. Swarm and Evolutionary Computation, 51:100598, 2019. URL: http://www.sciencedirect.com/science/article/pii/S2210650219301397.

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. arXiv preprint arXiv:1606.01540, 2016.

[5] François Chollet et al. Keras. https://keras.io, 2015.

[6] M. Coletti, A. Fafard, and D. Page. Troubleshooting deep-learner training data problems using an evolutionary algorithm on summit. IBM Journal of Research and Development:1–1, in press.

[7] Dask Development Team. Dask: Library for dynamic task scheduling. 2016. URL: https://dask.org.

[8] Kenneth De Jong. Evolutionary Computation: A Unified Approach. The MIT Press, 2006.

[9] Kenneth De Jong. Learning with genetic algorithms: an overview. Machine learning, 3(2-3):121–138, 1988.

[10] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: evolutionary algorithms made easy. Journal of Machine Learning Research, 13:2171–2175, 2012.

[11] A. Garrett. Inspyred: bio-inspired algorithms in python. 2017. URL: http://aarongarrett.github.io/inspyred/.

[12] Nikolaus Hansen, Sibylle D Müller, and Petros Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). Evolutionary computation, 11(1):1–18, 2003.

[13] Yannick Hold-Geoffroy, Olivier Gagnon, and Marc Parizeau. Once you scoop, no need to fork. In Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment, page 60. ACM, 2014.

[14] June 2019 | TOP500 supercomputer sites, 2019. URL: https://www.top500.org/lists/2019/06/.

[15] Maarten Keijzer, Juan J Merelo, Gustavo Romero, and Marc Schoenauer. Evolving objects: a general purpose evolutionary computation library. In International Conference on Artificial Evolution (Evolution Artificielle), pages 231–242. Springer, 2001.

[16] Mehdi Khoury et al. pySTEP or Python strongly typed genetic programming. https://sourceforge.net/projects/pystep/.

[17] Sean Luke. ECJ evolutionary computation library, 1998. Available for free at http://cs.gmu.edu/~eclab/projects/ecj/.

[18] ORNL Leadership Computing Facility. Summit: america's newest and smartest supercomputer, 2019. URL: https://www.olcf.ornl.gov/for-users/system-user-guides/summit/summit-user-guide/#system-overview.

[19] Gary Pamparà and Andries P Engelbrecht. Evolutionary and swarm-intelligence algorithms through monadic composition. In Proceedings of the Genetic and Evolutionary Computation Conference Companion, pages 1382–1390, 2019.

[20] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: an imperative style, high-performance deep learning library. In Advances in Neural Information Processing Systems 32, pages 8024–8035. 2019. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[21] Christian S Perone. Pyevolve: a python open-source framework for genetic algorithms. Acm Sigevolution, 4(1):12–20, 2009.

[22] Adrian M. Price-Whelan and Daniel Foreman-Mackey. Schwimmbad: a uniform interface to parallel processing pools in python. The Journal of Open Source Software, 2(17), 2017. URL: https://doi.org/10.21105/joss.00357.

[23] Ayodeji Remi-Omosowon and Yasser Gonzalez. Pyeasyga: A simple and easy-to-use implementation of a Genetic Algorithm library in Python. https://github.com/remiomosowon/pyeasyga, 2014.

[24] Matthew Rocklin and John Jacobsen. Toolz. 2020. URL: https://toolz.readthedocs.io/en/latest/index.html.

[25] Ralf Salomon. Re-evaluating genetic algorithm performance under coordinate rotation of benchmark functions. A survey of some theoretical and practical aspects of genetic algorithms. BioSystems, 39(3):263–278, 1996.

[26] Eric O Scott and Kenneth A De Jong. Understanding simple asynchronous evolutionary algorithms. In Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII, pages 85–98, 2015.

[27] Eric O Scott and Sean Luke. Ecj at 20: toward a general metaheuristics toolkit. In Proceedings of the Genetic and Evolutionary Computation Conference Companion, pages 1391–1398, 2019.