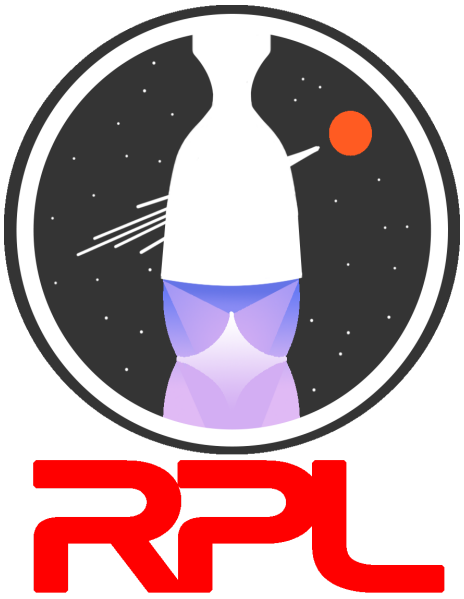
DEAP Tutorials & Walkthroughs

Rocket Propulsion Laboratory, UCSD

Augmented Design Team



DISTRIBUTED
EVOLUTIONARY
ALGORITHMS IN
PYTHON

DEAP Overview

DEAP (Distributed Evolutionary Algorithms in Python) is a novel evolutionary computation framework for rapid prototyping and testing of ideas. It seeks to make algorithms explicit and data structures transparent. It works in perfect harmony with parallelisation mechanism such as multiprocessing and `SCOOP`. The following documentation presents the key concepts and many features to build your own evolutions.

In short, DEAP is broken into two major modules, `creator` and `toolbox`. The `creator` is a meta-factory allowing to create classes that will fulfill the needs of your evolutionary algorithms. In effect, new classes can be built from any imaginable type, from `list to set, dict, PrimitiveTree` and more, providing the possibility to implement genetic algorithms, genetic programming, evolution strategies, particle swarm optimizers, and many more.

The `tools` module contains the operators for evolutionary algorithms. They are used to modify, select and move the individuals in their environment. The set of operators it contains are readily usable in the `Toolbox`. In addition to the basic operators this module also contains utility tools to enhance the basic algorithms with `Statistics, HallOfFame,` and `History.` In addition to `creator` and `toolbox`, The `base` module provides basic structures to build evolutionary algorithms. It contains the `Toolbox`, useful to store evolutionary operators, and a virtual `Fitness` class used as base class, for the fitness member of any individual.

Why DEAP?

CASE 1 - BASIC COMPRESSOR THERMODYNAMICS PROOF OF CONCEPT

A simple, straightforward, proof of concept is the compressor pressure ratio (CPR) equation.iThis equation is a good choice because it contains two non constant variables and an ideal solution is easily calculated by hand given upper and lower bounds. The equation is:

$$CPR = \left(\frac{Exit\ Temp}{Entry\ Temp}\right)^{\frac{\gamma}{\gamma-1}}$$

Where:
`Exit temp` = the temperature of the air leaving the compressor
`Entry temp` = the temperature of the air entering the compressor
γ = specific heat ratio

In this case, given a lower temperature bound of 250° C, an upper temperature bound of 600° C, and a specific heat ratio (gamma) of 1.4. The ideal solution is a CPR of 21.416. As you will see at the end of this tutorial, the evolutionary algorithm finds this exact solution.

Code

Due to the fact that DEAP is not a part of the standard Python library, we have to start with a number of imports to allow us to use the framework.

- `random` is used for random number generation
- `matplotlib.pyplot` is used for visualization of data /results

- `base`, `creator`, and `tools` are the modules/classes contained in DEAP which allow us to actually use the DEAP framework

Once the imports are taken care of, we will then define two global constants, NON_CONST_VARS which simply represents the number of non constant variables in the CPR equation (the entry temp and exit temp) and our specific heat ratio (gamma).

```
import random
import matplotlib.pyplot as plt
from deap import base
from deap import creator
from deap import tools

NON_CONST_VARS = 2
GAMMA = 1.4
```

Creator

The first line creates a maximizing fitness function by replacing, in the base type `Fitness`, the pure virtual **weights** attribute by (1.0,) that means to maximize a single objective fitness. If we were to want to minimize this function, "`FitnessMax`" would be changed to "`FitnessMin`" and weights would be set to a weight of `(-1.0,).` If we were to want to maximize the first parameter and minimize the second parameter, "`FitnessMax`" would be changed to "`FitnessMulti`" and weights would be set to a weight of `(1.0,-1.0,).`

```
#Define the fitnessMax function as a maximization function
#i.e. Maximize the compressor ratio
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

The next step is to actually create the class which will represent the component to be analyzed. In this case, we choose to derive from the `list` class which will allow us easy indexing of component equation variables and allows the class to be iterable. By using a `list` to represent our compressor we can use particular indices to store our non constant variables in the CPR equation like so:

|___Entry Temp___|___Exit Temp___|

`list` index: 0 1

```
#Create the Compressor class which derives from list and assign the fitnessMax to this class
creator.create("Compressor", list, fitness=creator.FitnessMax)
```

Now that we have created our class and linked it to the `FitnessMax` attribute, it is time to define how we will evaluate the compressor. As stated before, CPR will be the metric used for this example. A higher CPR means a higher air pressure exiting the compressor relative to ambient pressure. This will equate to more thrust through various subsequent factors in the engine. Note the comma after `cpr` in the `return`, this comma is necessary because DEAP requires tuples.

```
#Calculate CPR, the larger the better
def evalCompressor(Compressor):
    cpr = (Compressor[1] / Compressor[0]) ** (GAMMA / (GAMMA-1))
    return cpr,
```

Toolbox

To begin registering functions to the toolbox, we first need to actually create the toolbox. Although mentioned earlier, it will be explained a bit more here. A toolbox is the container which contains the evolutionary operators. At first the toolbox contains a `clone()` method that duplicates any element it is passed as an argument, this method defaults to the `copy.deepcopy()` function. And a `map()` method that applies the function given as first argument to every items of the iterables given as next arguments, this method defaults to the `map()` function. You may populate the toolbox with any other function by using the `register()` method.

One part of the toolbox that cannot be stated enough is the registration of tools in the toolbox only associates an *alias* to an already existing function and freezes part of its arguments. This allows us to call the alias even if the majority of the (or every) arguments have already been given. For example, the `attr_bool()` generator is made from the `randint()` that takes two arguments *a* and *b*, with `a <= n <= b`, where `n` is the returned integer. Here, we fix `a = 450` and `b = 500`.

```
#Register the toolbox as toolbox, standard line, should be included in every DEAP program
toolbox = base.Toolbox()
```

Once the toolbox has been registered, we can now register all of the functions to give our program functionality. Beginning with the attribute generator, this function is what will be called when populating the data fields of the class/object we are using as individuals in our evolutionary algorithms.

```
#Attribute generator
toolbox.register("attr_rand", random.randint, 450, 500)
```

Next, the structure initializers are what will be used to actually create/populate both the individual (compressor in our case) and the population. In regards to the first function, whenever “individual” is called, a new individual will be created. The arguments for this function are described as follows:

`tools.initRepeat` - This is the function that will be called again and again until the proper number of
`creator.Compressor` - This is the class which will be created with each call
`toolbox.attr_rand` - The alias for the attribute generator function for our class
`NON_CONST_VARS` - The number of times to generate attributes, in this case twice. Once for entry temp and once for exit temp

```
#Structure initializers
toolbox.register("individual", tools.initRepeat, creator.Compressor,
    toolbox.attr_bool, NON_CONST_VARS)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

Evaluating fitness of our individuals (compressors) is massively important so we need to register the previously defined evaluation function in the toolbox so we can use it later.

```
#Register the fitness function in toolbox
toolbox.register("evaluate", evalCompressor)
```

Next we will register the mating function which is simply how to create a new Compressor child given two Compressor “parents”. We will use the prepackaged `cxTwoPoint` mating function which comes with the DEAP framework. This function will be registered in the toolbox under the alias **“mate”.**

```
#Register mating function in toolbox
toolbox.register("mate", tools.cxTwoPoint)
```

Another extremely important part of evolutionary algorithms is the mutation of individuals which currently exist within the population. In this example, we use the `tools.mutUniformInt` function which simply chooses a random integer between 250 and 600, inclusive, with probability of 5%. This function will be registred in the toolbox under the alias **"mutate"**.

```
#Register mutate function in toolbox
toolbox.register("mutate", tools.mutUniformInt, low=250, up=600, indpb=0.05)
```

Finally, the last function to register is the selection/tournament function. This function, renamed under the alias `"select"`, grabs the `tournsize` best individuals in the current population and uses them as the basis for creating/populating the new generation.

```
#Register tournament function in toolbox
toolbox.register("select", tools.selTournament, tournsize=3)
```

Main Method

Now comes the actual implementation of the work done so far. A vast majority of the code that follows will be simple to follow with a background in Python. Again, remember that functions in the main method may just be alias' and preceded by the `toolbox` DEAP keyword.

```
#Main method
def main():
    # initialize internal state of the random number generator with seed 64
    random.seed(64)

    # create an initial population of 500 individuals (where
    # each individual is a list of integers)
    pop = toolbox.population(n=500)

    # CXPB  is the probability with which two individuals
    #       are crossed
    #
    # MUTPB is the probability for mutating an individual
    #
    # NGEN  is the number of generations for which the
    #       evolution runs
    CXPB, MUTPB, NGEN = 0.5, 0.2, 40

    print("Start of evolution")

    # Evaluate the entire population
    fitnesses = list(map(toolbox.evaluate, pop))
    for ind, fit in zip(pop, fitnesses):
```

```
    ind.fitness.values = fit

print("  Evaluated %i Compressors" % len(pop))

# Begin the evolution
for g in range(NGEN):
    print("-- Generation %i --" % g)

    # Select the next generation individuals
    offspring = toolbox.select(pop, len(pop))
    # Clone the selected individuals
    offspring = list(map(toolbox.clone, offspring))

    # Apply crossover and mutation on the offspring
    for child1, child2 in zip(offspring[::2], offspring[1::2]):

        # cross two individuals with probability CXPB
        if random.random() < CXPB:
            toolbox.mate(child1, child2)

            # fitness values of the children
            # must be recalculated later
            del child1.fitness.values
            del child2.fitness.values

    for mutant in offspring:

        # mutate an individual with probability MUTPB
        if random.random() < MUTPB:
            toolbox.mutate(mutant)
            del mutant.fitness.values

    # Evaluate the individuals with an invalid fitness
    invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
    fitnesses = map(toolbox.evaluate, invalid_ind)
    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit

    print("  Evaluated %i Compressors" % len(invalid_ind))

    # The population is entirely replaced by the offspring
    pop[:] = offspring

    # Gather all the fitnesses in one list and print the stats
    fits = [ind.fitness.values[0] for ind in pop]

    length = len(pop)
    mean = sum(fits) / length
```

```
        sum2 = sum(x*x for x in fits)
        std = abs(sum2 / length - mean**2)**0.5

        print("  Min %s" % min(fits))
        print("  Max %s" % max(fits))
        print("  Avg %s" % mean)
        print("  Std %s" % std)
        print("  Best compressor temps %s" % tools.selBest(pop, 1)[0])

    print("-- End of (successful) evolution --")

    best_ind = tools.selBest(pop, 1)[0]
    print("Best individual is %s, with a CPR of %s" % (best_ind, best_ind.fitness.values))

if __name__ == "__main__":
    main()
```

Results

We can see that in the first generation, the best compressor entry and exit temperatures in this population yield a CPR of only 3.67.

```
Start of evolution
  Evaluated 500 Compressors
-- Generation 0 --
  Evaluated 302 Compressors
  Min 0.110180231752112
  Max 3.673622247811698
  Avg 1.1440093339927697
  Std 0.2050248750776529
  Best compressor temps [342, 496]
```

At the end of execution, in generation 39, we see that the algorithm has found the most ideal compressor entry and exit temperatures with 250° C and 600° C, respectively.

```
-- Generation 39 --
  Evaluated 307 Compressors
  Min 1.3734011972400235
  Max 21.416048711188537
  Avg 21.164685881967703
  Std 2.04227369729934
  Best compressor temps [250, 600]
-- End of (successful) evolution --
Best individual is [250, 600], with a CPR of (21.416048711188537,)
```

While this is a simple equation paired with a relatively simple algorithm strategy, this example is a powerful proof of concept and serves as a baseline understanding for future use in more complex evolutions.