Co-optimizer Manual

Introduction

The co-optimizer comes with a variety of optimization options that are aimed at solving multi-objective optimization problems. There are pre-defined options the user can choose from (see below) and there is the possibility to query self-defined objective functions. In the following, we describe step-by-step how and what files to change in order to run the given options. At the end, we outline how the user can define their own objective functions.

File hierarchy

Below we describe the relationships between the individual files.

fuelsdb\_interface.py

calls

**co\_optimizer\_par.py**

GPmerit.py

blend\_functions.py

merit\_functions.py

cooptimizer\_input.py

nsga2\_k.py

calls

calls

calls

The co-optimizer uses python 2.7 and requires several libraries to be available, such as numpy, scipy, multiprocessing, deap, pandas, and sklearn. co\_optimizer\_par.py is the main file from which to run the code. Currently, there are several multi-objective optimization options that can be run. They require the user to make option selections and changes in two files, namely in cooptimizer\_input.py and nsga2\_k.py. Below we describe the options.

Co-optimizer options

1. Deterministic optimization: return one Pareto front

* cost\_vs\_merit\_Pareto:
* in cooptimizer\_input.py l.39: *task\_list['cost\_vs\_merit\_Pareto'] = True*
* several combinations of objective functions are possible:
  1. maximize the Miles merit function MMF, minimize the associated costs
* in nsga2\_k.py:
  + - use option in l.569, *toolbox.register("evaluate", eval\_mo, propvec=propvec, Kinp=KK)*
    - in l. 543: *NDIM = len(propvec['COST'])*
    - in l 547: *creator.create("FitnessMin", base.Fitness, weights=(1.0, -1.0))*
    - l. 578 & 579 – must be active
  1. maximize the Miles merit function, maximize the net mean effective pressure (NMEP);
     + in nsga2\_k.py:
     + use option in l.567, *toolbox.register("evaluate", eval\_MMF\_gp, propvec=propvec, Kinp=KK, GP = GP, scal = scal)*
     + in l. 543: *NDIM = len(propvec['COST'])+3*
     + in l 547: *creator.create("FitnessMin", base.Fitness, weights=(1.0, 1.0))*
     + in l. 538: use *GP, scal = run\_GP()*
     + l. 578 & 579 – must be active
  2. maximize the Miles merit function, maximize NMEP by solving optimization subproblem on NMEP-related variables;
     + in nsga2\_k.py:
     + use option in l.566, *toolbox.register("evaluate", eval\_MMF\_gp\_opt, propvec=propvec, Kinp=KK, GP = GP, scal = scal)*
     + in l. 543: *NDIM = len(propvec['COST'])*
     + in l 547: *creator.create("FitnessMin", base.Fitness, weights=(1.0, 1.0))*
     + in l. 538: use *GP, scal = run\_GP()*
     + l. 578 & 579 – must be active
  3. maximize the expected NMEP, minimize the variance of NMEP;
     + in nsga2\_k.py:
     + use option in l.568, *toolbox.register("evaluate", eval\_gp, GP = GP, scal = scal)*
     + in l. 543: *NDIM= 6*
     + in l 547: *creator.create("FitnessMin", base.Fitness, weights=(1.0, -1.0))*
     + in l. 538: use *GP, scal = run\_GP()*
     + outcomment l. 578 & 579

1. Optimization under uncertainty in MMF coefficients: return one Pareto front

* mean\_vs\_var\_Pareto:
* in cooptimizer\_input.py l.41: *task\_list['mean\_vs\_var\_Pareto'] = True*
* maximize the mean of MMF and minimize the MMF variance (assuming uncertainty in coefficients of the MMF, as defined in lines 75-108 in cooptimizer\_input.py)
* in nsga2\_k.py:
* use option in l. 570: *toolbox.*register*("evaluate", eval\_mo2, propvec=propvec, Kinp=KK)*
* in l. 543: *NDIM= = len(propvec['COST'])*
* in l 547: *creator.create("FitnessMin", base.Fitness, weights=(1.0, -1.0))*
* l. 578 & 579 – must be active
* l. 338 and following: choose distribution from which to sample coefficients

1. Optimization under uncertainty in one cost coefficient: returns several Pareto fronts

* cost\_vs\_merit\_Pareto\_UP\_single:
* in cooptimizer\_input.py: *task\_list['cost\_vs\_merit\_Pareto\_UP\_single'] = True*
* maximizes the MMF, and minimizes the cost, where the cost is a random variable for a single fuel component (all other fuel components are kept with deterministic costs). We draw *nsamples* (see cooptimizer\_input.py l. 58) values for the *i*-th cost component and thus do *nsamples* optimizations, yielding *nsamples* Pareto fronts
  + - * in nsga2\_k.py:
      * use option in l.569, *toolbox.register("evaluate", eval\_mo, propvec=propvec, Kinp=KK)*
      * in l. 543: *NDIM= = len(propvec['COST'])*
      * in l 547: *creator.create("FitnessMin", base.Fitness, weights=(1.0, -1.0))*
      * l. 578 & 579 – must be active

1. Optimization under uncertainty in all cost coefficients: returns several Pareto fronts
   * cost\_vs\_merit\_Pareto\_UP:
   * in cooptimizer\_input.py: *task\_list['cost\_vs\_merit\_Pareto\_UP'] = True*
   * maximizes the MMF, and minimizes the cost, where all component costs are randomly drawn from a distribution. We draw *nsamples* (see cooptimizer\_input.py l. 58) values for each cost component and thus do *nsamples* optimizations, yielding *nsamples* Pareto fronts

* in nsga2\_k.py:
* use option in l.569, *toolbox.register("evaluate", eval\_mo, propvec=propvec, Kinp=KK)*
* in l. 543: *NDIM= = len(propvec['COST'])*
* in l 547: *creator.create("FitnessMin", base.Fitness, weights=(1.0, -1.0))*
* l. 578 & 579 – must be active

1. Single objective optimization for cheap-to-compute and expensive-to-compute functions: returns one solution only

* K\_vs\_merit\_sweep:
* in cooptimizer\_input.py: *task\_list['K\_vs\_merit\_sweep'] = True*
* maximizes the MMF for different values of K
* select a range of K values in cooptimizer\_input.py l. 165 (does as many single objective optimizations as there are K values)
* select optimization option:
  + if cheap-to-compute objective function, use in co\_optimizer\_par.py, l. 515: *merit = maximize\_merit(KK, propvec, propDB);* and l. 516: *M.append(merit);* and l. 527 for plot: *plt.scatter(cooptimizer\_input.KVEC, M, marker = 'o', c= 'r',s=40,label='python optimizer')*
  + if expensive-to-compute objective function, use in co\_optimizer\_par.py, l. 518: *data = surrogate\_optimization(KK, propvec, propDB),* and l. 519: *F.append(-data.Fbest);* andl. 528 for plot: *plt.scatter(cooptimizer\_input.KVEC, F, marker = '^', c = 'b', s=40, label='surrogate optimizer')*

Merit functions

1. Miles merit function (MMF): this merit function is defined in the report [insert name] and measures the efficiency one can obtain from a given fuel blend. In this implementation, we assume that we have 22 fuel components, whose properties and costs are assumed to be given (synthetic data for experiments). The fuel components are mixed with a linear blending model to obtain values for RON, S, HOV, etc., which enter the MMF. The costs are derived in the same linear fashion. See the files merit\_functions.py (revised\_mf) for computing MMF and blend\_functions.py (blend\_fancy\_vec) for computation of RON, S, HOV, COST.
2. Net mean effective pressure (NMEP): this merit function is based on data from Ratcliff [insert paper]. We used their data to train a Gaussian process (GP) model (GPmerit.py). We use the Gaussian process model as “truth model” objective function in the optimization. The GP gives for an unsampled parameter vector an estimate of the mean response and its variance. Thus, we can maximize the expected (mean) NMEP while minimizing the associated predicted variance as done in 1d above.
3. Optimization under uncertainty: this is generally done by assuming a distribution on either the coefficients in the MMF (2 above), or assuming a distribution on the costs of the fuel components (3 and 4 above). For the fuel component uncertainty, we run several multi-objective optimizations from a realization of the random cost variables and we obtain several Pareto fronts. We can look at the mean and median Pareto fronts to obtain an idea of the spread and sensitivity of the tradeoff solutions. When considering uncertainty in the coefficients of the MMF, for a given parameter vector, we draw a large number *N* of random values form the distribution, compute *N* MMF values, and then maximize the mean of these *N* values while minimizing their variance (2 above).
4. Singe objective optimization (see 5 above): there is an option for computationally cheap and computationally expensive single objective optimization. The current implementation allows the user to select a range of K values that go into the Miles function (MMF). Given K, the MMF is maximized with a local optimizer and a single best solution is returned. For expensive-to-evaluate objective functions, there is an option that uses surrogate model approximations during the optimization in order to minimize the number of expensive evaluations that is required. In the current implementation, we use the MMF as placeholder for an expensive evaluation.

User defined objective functions

In order to solve your own multi-objective problems, the user has to implement their functions such that it is a simple function call that returns *m* scalars, *m* being the number of objective functions. In the following is information on the necessary modifications. It is probably easiest to use one of the given objectives and modify them:

* In nsga2\_k.py: use for example the objective *eval\_mo* starting on l. 162. The important part is that the return values have to contain the objective function values (in the example *mmf\_p\_p, c\_p\_p*). If you have more than 2 objectives, then correspondingly more values have to be returned. The user is responsible for implementing the computation of the return values.
* In nsga2\_k.py: in nsga2\_pareto\_K starting l. 535:
  + l. 543: *NDIM* (the number of optimization parameters) must be adjusted
  + l. 547: *creator.create("FitnessMin", base.Fitness, weights=(1.0, -1.0)):* the weights represent whether we maximize (1.0) or minimize (-1.0). These weights must be adjusted depending on the sequence of objective function values that are returned. For example, if the order of returned objectives is *f1, f2, f3, f4*, and aw want to maximize *f1* and *f2*, and minimize *f3* and *f4,* the weight vector would be *weights=(1.0, 1.0, -1.0, -1.0).*
  + l. 556: *BOUND\_LOW, BOUND\_UP = 0,1:*  the lower and upper bounds for the optimization must be adjusted. However, we recommend doing the optimization over the scaled hyper unit cube and rescaling the parameters to their true ranges for evaluating the user-defined objective functions.
  + l. 578, 579: *toolbox.decorate("mate", scale\_2()), toolbox.decorate("mutate", scale\_2()):* this is the place where additional optimization constraints are defined.