“Robust Multiobjective Optimization using Regression Models and Linear Subproblems” Toolbox

Fillipe Goulart

This is a small toolbox that implements the Differential Evolution for Multiobjective Optimization (DEMO) but adapted to handle uncertainties in the variables. The details of the method are given in [1].

In the `algorithms` folder there are the `demo_opt.m` and `demo_robust.m` files, together with some auxiliary ones. The first file implements the original DEMO that can (or rather tries to) solve any multi-objective problem, while the second is the adapted version to handle the case when the variables have uncertainties. By typing `help demo_opt` or `help demo_robust` the user can see the syntax of the function, together with examples of how to use them.

The test functions are in the obvious folder `functions`, and there are the usual DTLZ functions and the one proposed by Deb in [2] but adapted in [1]. Again, the user can see how to use each file with the `help` function. Just keep in mind that [1] was tested with Deb’s test problem. Even if `demo_robust.m` should run fine with any of the DTLZ (or any other function created by the user), the validity of the results should be subject to another analysis.

First time user:

To just start using the code, try the following in the Octave workspace:

```
includepaths %to add all required paths to the workspace
% First test: solve the DTLZ2 function with 2 objectives with the
regular DEMO
f = @(x) dtlz2(x, 2) %write the test function
f =

@(x) dtlz2 (x, 2)

xrange = dtlz_range ("dtlz2", 2) %get the search limits
xrange =

   0   1
   0   1
   0   1
   0   1
   0   1
   0   1
   0   1
   0   1
```

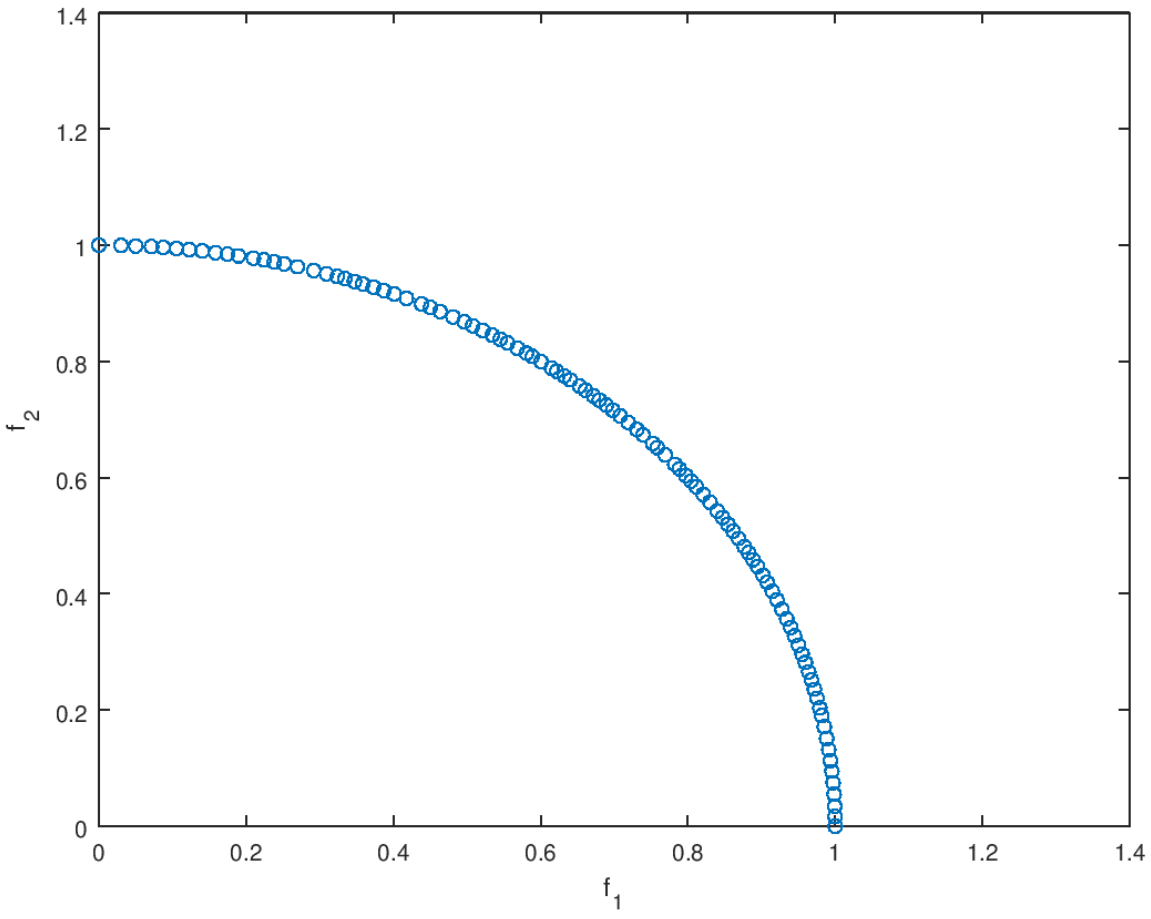
```
    0    1
    0    1
    0    1
```

```
Popt = demo_opt(f, xrange); %wait until 300 iterations are complete
```

`Popt` is a struct containing the final population with fields `.x` (variables) and `.f` (objective values). To check if the results are o.k., you can plot them:

```
plot(Popt.f(1,:), Popt.f(2,:), "o"); xlabel("f_1"); ylabel("f_2");
```

which gives the following figure. Users that know the DLTZ function will notice that the points are almost in the quarter circle of the first hemisphere, which is a good visual indicator of convergence to the Pareto-optimal front. Of course, the skeptical ones can check this by computing the distances to the optimal solutions:




```
d = dtlz_distance ("dtlz2", Popt.x);
min(d), mean(d), max(d),
ans =    3.7714e-11
ans =    1.7123e-10
ans =    4.1795e-10
```

which is an even better indicator of good results. Check the helper to get more information on how to change the default parameters and more examples.

Now, since the focus is with robustness, let us try to solve the Deb's function in the `Deb` folder with similar settings of the first test in Section 3.2 of [1].

```
% Set the objective function
f = @(x) deb_robust (x, 1, 1); %alpha =  beta = 1
n = 10; %dimension
xrange = deb_range(n); %n = 10 variables
% Set parameters of the algorithm
options.dx = [0.01; 0.02*ones(n-1,1)]; %vector of uncertainties
options.eta = 1; % sensitivity tolerance
options.display = true; %to see the population evolving
Popt = demo_robust(f, xrange, options);
```

This time a graph was shown while the population evolved. Compare these results with the ones from Figures 4 and 5 of [1]. Also try different sensitivity tolerances, uncertainties etc.

Writing your own functions

Any function that receives a matrix `x` of size $n \times \mu$ with μ points and n dimensions each and returns another matrix `y` of size $m \times \mu$ should work fine. If there are optional parameters in the function, e.g., if the m file containing the test function is of the form

```
function y = myfunction(x, par1, par2, …)
```

the user can use as input for `demo_opt` or `demo_robust` a function handle

```
f = @(x) myfunction(x, par1, par2, …);
```

where `par1`, `par2` etc. should be set here. Different function handles can be created every time one of the parameters needs to be changed. See the example in `demo_robust` since the `deb_robust.m` function has this form.

Important observation about Matlab

The code is meant to be run in Octave, version 4.0 or newer. However, the code was not optimized for Octave only, so most functions should run fine in recent versions of Matlab as well, with an obvious exception of the `ols` and `glpk` functions used in the linear regression of `compute_sensitivity.m`. The interested user who wishes to run Matlab instead should adapt this portion. Unfortunately, I did not have access to a Matlab copy, let alone the optimization toolbox, so I could not correct for possible errors. With that said, a few friends mentioned that the linear solver of Matlab is *way slower* than the equivalent `glpk` of Octave, so they still preferred the latter software. It

should not be an issue for most people, considering Octave is free and can run most of the code written for Matlab.

Contact information

Have any questions? Send me an e-mail: fillipe.gsm@gmail.com.

References:

[1] Goulart, Fillipe, et al. "Robust multiobjective optimization using regression models and linear subproblems." Proceedings of the Genetic and Evolutionary Computation Conference. ACM, 2017.
[2] Kalyanmoy Deb and Himanshu Gupta. 2006. Introducing robustness in multi-objective optimization. Evolutionary Computation 14, 4 (Dec. 2006), 463–494. DOI:http://dx.doi.org/10.1162/evco.2006.14.4.463