Untitled

June 9, 2020

0.1 What's this?

In this package you'll find some experiments in optimization sprout from the realisation of a project for the course Computational Mathematics for Machine Learning and Data Analysis, held at the University of Pisa. For additional details, scroll through the Julia code. As a by side, please be mindful of the fact that this is the first time we get in touch with both Julia and mathematical programming.

0.1.1 Setup

Copy the repo to your favourite folder, then add the path to the `LOAD_PATH` variable in your Julia REPL. This can be accomplished with

[ ]:
```
push!(LOAD_PATH, "/path/to/the/repo")
```

You can make it permanent by adding the command to `~/.julia/config/startup.jl`

0.1.2 Usage

Load the package with

[ ]:
```
using Optimization
```

Example: Subgradient with Automatic Parameter Tuning In this example you'll choose a subgradient step update and nest it inside the parameter tuner, together with the parameter searcher, in this case the standard Nelder Mead. Hence you'll generate a random quadratic Min Cost Flow boxed separable problem (`QMCFBProblem`), run the algorithm and finally plot some data from the execution. We'll guide you through the process line by line.

Disclaimer: this is an experimental algorithm for parameter tuning, in its early infancy, plenty of defects.

[ ]:
```
subgradient = Subgradient.HarmonicErgodicPrimalStep(k=4, a=0.01, b=0.1);
```

Here you create an instance of a subgradient method, the `HarmonicErgodicPrimalStep`.

You can find such method, and many other, in the file `subgradient.jl`.

This algorithm is specific to dual problems; in fact its peculiarity is that it keeps a convex combination of the primal values corresponding with the dual ones, through a choice of the subgradients,

and such combination is guaranteed to converge to the primal optimal solution. More details in the docstring above the `struct` definition.

[ ]:
```
algorithm = QMCFBPAlgorithmD1SG(
   localization=subgradient,      # subgradient method of choice
   verbosity=1,                   # verbosity level (WIP)
   max_iter=1000,                 # maximum number of iterations
    =1e-6,                        # \varepsilon find the optimal solution with tolerance
    =1e-12);                      # \epsilon geometric extension of vertices
```

Here you create an instance of an algorithm taylored to solve `QMCFBProblem`s with the use of subgradient methods. You can find such method in the file `algorithm/QMCFBP_D1_SG.jl`.

The specifies how vertices are expanded, so that they are reached if nearer than .

Other members of the `struct` not shown here are: * : the starting point in the dual space; if unspecified is set to zeros * `stopped` : a boolean indicating if the algorithm is fresh new or not (so there should be no initialization). This is used especially when you want to *prosecute from the result of an algorithm execution*, which can be attained with `function set!(algorithm::QMCFBPAlgorithmD1SG, result::OptimizationResult{QMCFBProblem})` * `memorabilia` : when setting up a solver, it is possible to specify a set of `memoranda`, which contains the names of the variables to be tracked during the execution of the algorithm. *In `memorabilia` there is a list of the trackable variables in the algorithm.*

[ ]:
```
searcher = NelderMead();
```

This is the object we are using for the parameter search.

[ ]:
```
halgorithm = WithParameterSearch{QMCFBProblem, typeof(algorithm), NelderMead}(
   algorithm=algorithm,
   searcher=searcher,
   objective="L ",                        # objective to be used in comparisons between point of the simplex
   cmp=(a, b)->a>b,                       # comparison to be used to which value of the objective is better
   searcher_iter=10,                      # see below
   algorithm_iter=10000,                  # see below
   algorithm_iter_per_search=200,         # see below
    =0.1,                                 # see below
   param_ranges=Dict(:a=>[1e-8, 1.0], :b=>[1e-8, 1.0]));   # ranges of the parameters to be used as a start
```

This is our *hyper algorithm*, which is just a wrapper for the `algorithm` to be used by the `searcher` together with the dynamics that we want to use for the search.

You can find it in the file `hyper.jl`.

At the moment, a naive strategy is implemented, consisting in a percentage of algorithm iterations

to be explored by the `searcher`, followed by an `1-` remaining iterations with the parameters chosen by the preceding search.

All this is repeated until the total `algorithm_iter` are accounted for.

As an example, in this case we have `algorithm_iter_per_search = 200`, so the search phase costs 200 iterations; they contitute `=0.1` of the total iterations per cycle, so there are 1800 iterations following the search with the parameters set by the search. Note that, from the computational cost point of view, the search will actually cost `algorithm_iter_per_search*searcher_iter = 2000` iterations.

Finally, the cycle is repeated 5 times to reach the total number of iterations, `algorithm_iter=10000`.

You see that `param_ranges`, which is a `Dict{Symbol, Array}`, is indicating the range to start with in the parameter search. If you desire to keep fixed some parameter, you can do it with `fixed_params = Dict( dictionary of parameters to value to be held fixed )`, as an example we could specify `fixed_params = Dict(:a=>0.5)` in case we wanted the `a` parameter to be 0.5

[ ]:
```
test = get_test(
    halgorithm,     # algorithm to be set in the solver
    m=100,          # number of arcs (rows of incidence matrix E)
    n=200,          # number of edges (columns of incidence matrix E)
    singular=30,    # dim ker Q, where Q is the diagonal semidefinite positive␣
↪hessian of the obj
    active=0);      # the generator will try to position the minimum so that␣
↪there will be active active constraints
```

This is generating a random problem. Multiple dispatch will drive you to the code in `mincostflow.jl`, thanks to the type of `halgorithm <: OptimizationAlgorithm{QMCFBProblem}`.

If you do not provide a problem, as in this case, the problem is generated by the function `generate_quadratic_min_cost_flow_boxed_problem`. If you already have a problem `my_problem` to solve, you can specify it with

```
=my_problem
```

Scroll through the code to see how to reduce the problem, in case you want to consider separately each connected component.

[ ]:
```
test.solver.options.memoranda = Set(["norm L ", "L ","i ", "params_best",␣
↪"result_best"]);
```

Remember of `memoranda`?

`get_test` is returning an `OptimizationInstance{QMCFBProblem}`. You can look at the structure in the file `Optimization.jl`.

Each `OptimizationInstance` contains a **problem** (whose type is parametrizing the instance), a **solver** and a **result**.

Each solver contains an **algorithm** and some **options**, which are the ones we are specifying here, setting the variables to be tracked during the execution of the algorithm. Note that we are setting the variables to be tracked globally, so both in the `halgorithm` and the `algorithm`.

[ ]:
```
run!(test);
```

Well, guess it...

[ ]:
```
Ls = [x.memoria["L "] for x in test.result.memoria["result_best"]];
norm Ls = [x.memoria["norm L "] for x in test.result.memoria["result_best"]];
is = [x.memoria["i "] for x in test.result.memoria["result_best"]];
```

Here we are extracting some data which we'd love to represent. If the resulting graph will surprise you, recall how the `halgorithm` is repeating a search step 10 times...

What about the `is`? Since we are dealing with subgradient algorithms, and not descent ones, we are not driving toward a better objective value at each step, hence the `i\prime` are recording at which step a new best value is found. Note the `\prime` also to the right of `L` and `norm L`: the meaning is the same, they are the values corresponding to a best new point.

[ ]:
```
plot(is, Ls);
plot(vcat(Ls...));
```

The first line is plotting a superposition of the objective value along the iterations, where the iterator is resetted at each step of the `halgorithm`.

The second line is showing the same but without the superposition, so in chronological order.

[ ]:
```
plot(Dict{Symbol, Float64}.(test.result.memoria["params_best"]));
```

This is plotting a graph of the parameters (`a` and `b`) through the `halgorithm` iterations.

[ ]:
```
L_hyper = test.result.result["L "]
params = test.result.result["params_best"]
```

Here we are extracting part of the result, the best value for the lagrangian dual and the best parameter values at the last cycle of the `halgorithm` - the ones to be used in case you want to prosecute with a pure `algorithm`.

[ ]:
```
  = test.problem;
Q, q, l, u, E, b = ( .Q,  .q,  .l,  .u,  .E,  .b);
```

Here you are extracting the problem.

Example: Subgradient with Polyak step and Elliptic rank-1 linear operator We'll show you a little code snippets to test the Polyak step size with linear elliptic operator (in file `subgradient.jl`). Such method is enlarging the angle span by the last 2 directions of search, bringing the ellipsis generated by them to a circle. The method is applied to a prototypical example, where Polyak step size are easier because the minimum is known.

The purpose of this example is to give you finer control on the usage of the methods.

[ ]:
```
import Plots
function test(;n=2, max_iter=100, mapping=x->x)
    t = max.(1.0, 0.5 .+ rand(n))
    f = x -> t' * abs.(x)
     f = x -> t .* sign.(x)
    subgradient = Subgradient.PolyakEllipStepSize(f_opt=0.0)    # optimum is␣
↪known
    x = rand(n).-0.5
    init!(subgradient, f,  f, x)
    fs = Float64[]
    for i in 1:max_iter
        x ,  , sg = step!(subgradient, f,  f, x)
        fs = [fs; f(x )]
        x = x
    end
    Plots.plot([i for i in 1:length(fs)], mapping(fs))
end
```

As you see, there are 3 main steps in the usage of subgradient methods:

- `subgradient = ...` : object construction
- `init!(subgradient, f,  f, x)` : initialization
- `step!(subgradient, f,  f, x)` : a single step of the subgradient method

If you give it a try, you'll see how fast the method is converging (well, it has been devised to overcome exactly such kind of problems…)

Example: Quadratic programming in a box with projected cojugate gradient: long search + local search You'll see how to set up a convex quadratic problem for solution with projected conjugate gradient. This is used inside the conjugate gradient algorithm for `QMCFBProblem`s.

[ ]:
```
  = MinQuadratic.MQBProblem(
            Q,
            q,
            l,
            u)
instance = OptimizationInstance{MQBProblem}()
algorithm = MQBPAlgorithmPG1(
    localization=QuadraticBoxPCGDescent(),
    verbosity=-1,
    max_iter=3000,
     =1e-8,          # required maximum error
      =1e-12)        #
set!(instance,
    problem= ,
    algorithm=algorithm,
    options=MQBPSolverOptions(),      # set here the options you need, or␣
↪afterward...
```

```
    solver=OptimizationSolver{MQBProblem}())
run!(instance)
```

To see more of how it works, take a look at the submodule MinQuadratic, in the file `minquadratic.jl` for the general settings of the problem and at `MQBP_P_PG.jl` for the specific projected gradient algorithm. As usual, if you want to set some variable to track, you should specify it in the `memoranda` member of `instanve.solver.options`.

Example: QMCFBProblem solution with conjugate gradient and exact search This method is still not working as we hoped for singular `Q`, further specific analysis should be carried on to find out what's going wrong.

By now you should be able to foresee the lines of code needed to setup a `QMCFBProblem` and then an `QMCFBPAlgorithmD1D` object to solve it.

[ ]:
```
algorithm = QMCFBPAlgorithmD1D(
    descent=ConjugateGradientDescent(),
    verbosity=1,
    max_iter=1000,
     =1e-6,       # max error required
      =1e-12,     # epsilon up to which to approximate being inside, outside
    cure_singularity=true)   # if true, approach iteratively the singular Q
test = get_test(
    algorithm,
    m=1000,
    n=2000,
    singular=2);
run!(test);
```

Example: something of numerical analysis If you like numerical analysis, we collected also some of the algorithm that we have been writing in numerical analysis in the file `numerical.jl`. There was the plan to drive toward pseudospectra, singular values and pseudosv, but in the end, no time to do it.

As an example, if you want to give a try to fast GMRES, as described in the exercise of the book of Trefethen-Baum on Numerical Analysis, just give a call to

[ ]:
```
GMRES_naive(A, b, k,  ,  )
```

Please be advised that code there is really a stub, maybe better than nothing.

[ ]:
```

```