MOpt.jl Documentation

Release 0.1

Forian Oswald and Thibaut Lamadon

October 01, 2014

Contents

This is a Julia library to run moment optimization in parallel.

Contents:

CHAPTER 1

Getting Started

Let's see how to get started very quickly. You start by ....

1.1 Setting the problem

This package implements several MCMC algorithms to optimize a non-differentiable objective function. The main application are **likelihood-free estimators**, which requires evaluating the objective at many regions. In general, this implements *Simulated Method of Moments*. The library is targeted to run MCMC on an SGE cluster, where each node is a chain.

For an R implementation which is the basis of this package, see https://github.com/tlamadon/mopt

1.2 Example Useage

```
using Mopt

# get a parameter vector
p = ["a" => 3.1 , "b" => 4.9]
# define params to use with bounds
pb= [ "a" => [0,1] , "b" => [0,1] ]

# get some moments
# first entry is moment estimate, second is standard deviation
moms = [
  "alpha" => [ 0.8 , 0.02 ],
  "beta"  => [ 0.8 , 0.02 ],
  "gamma" => [ 0.8 , 0.02 ]
]

# a subset of moments to match
submoms = ["alpha", "beta"]

# call objective
x = Mopt.Testobj(p,moms,submoms)

# Define an Moment Optimization Problem
mprob = Mopt.MProb(p,pb,Mopt.Testobj,moms;moments_subset=submoms)

# show
```

```
mprob

# step 2: choose an algorithm
# ----------------------
algo = Mopt.MAlgoRandom(mprob,opts=["mode"=>"serial","maxiter"=>100])

# step 3: run estimation
# ----------------------
runMopt(algo)
```

CHAPTER 2

Setting up a moment problems

This section describes how to create the *MProb* object that will store the description of the problem. In general this is composed of the set of parameters we will iterate over together with their bounds, as well as the set of moments to match, with their value and their precision.

We start by creating an empty MProb object and we progressively add content to it.

```
using Mopt
mprob = MProb()
```

We then describe the step required to scuccessfuly set up the object.

2.1 Step 1: add parameters

There are two types of parameters, some are sampled by the algorythm and some are not. Here are ways to add arguments to the description:

```
addParam!(mprob, "c", 0.1)
addParam!(mprob, "d", 0.2)
addSampledParam!(mprob, "a", 0.1, 0, 1)
addSampledParam!(mprob, "b", 0.1, 0, 1)
```

But parameters can also be added all at once using for instance a dictionary

```
ps  = { "c" => 0.1 , "d" => 0.2}
addParam!(mprob,ps)
pss = { "a" => [0.1, 0, 1] , "b" => [0.1, 0, 1]}
addSampledParam!(mprob,pss)
```

2.2 Step 2: add moments

The second step is to add moments to the description. These are moments that the objective function will use as matching criteria. This is more for analysis purposes, as ex-post it can be very informative to look at what parameter is affected by what particular moments, or to check the shape of the moment conditions at maximum.

In a spirit similar to the parameters, moments can be added as follows:

```
addMoment!(mprob, "m1", 0.1, 0.001)
addMoment!(mprob, "m2", 0.1, 0.001)
```

But as well using a DataFrame which is usually how moments are loaded, from a csv file or other source. In this case the used needs to provide the names of the columns that must be used.

```
dd = DataFrame(name= ["m1", "m2"], value=[0.1,0.1], sd=[0.01,0.01])
addMoment!(mprob, dd, [:name,:value,:sd])
```

2.3 Done, next is selecting an algorithm

This is it, your problem is now created, you can move to the second step which is to select an algorithm.

2.4 Putting all at once using Lazy.jl

```
mprob = @> begin
  Mprob()
  addParam!("c", 0.1)
  addParam!("d", 0.2)
  addSampledParam!("a", 0.1, 0, 1)
  addSampledParam!("b", 0.1, 0, 1)
  addMoment!("m1", 0.1, 0.001)
  addMoment!("m2", 0.1, 0.001)
end
```

CHAPTER 3

Writing an objective function

We show here how to write your own objective function. The signature is:

> obj(x::Dict,mom::DataFrame,whichmom::Array{ASCIIString,1},vargs...)

where the first agrument is a dictionary of values for the parameters. The second is a dataFrame that contains information about the different moments to match. This will include the value of the moments together with their standard error.

The objective function should return another Dictionary that includes:

- *value*: the value of the objective function
- *params*: the list of parameters with their value, this should just be a deep copy of x
- *time*: the amount of time spent in the objective function
- *moments*: a DataFrame with the evaluated moments

here is a full example:

```
function Testobj(x::Dict,mom::DataFrame,whichmom::Array{ASCIIString,1},vargs...)

  t0 = time()

  if length(vargs) > 0
      if get(vargs[1],"printlevel",0) > 0
          info("in Test objective function")
      end
  end

  mm = deepcopy(mom)
  nm0 = names(mm)
  DataFrames.insert_single_column!(mm,zeros(nrow(mm)),ncol(mm)+1)
  names!(mm,[nm0,:model_value])

  for ir in eachrow(mm)
      ir[:model_value] = ir[:data_value] + 2.2
  end

  # output all moments
  mout = transpose(mm[[:moment,:model_value]],1)

  # subset mm to required moments to compute function value
  mm = mm[findin(mm[:moment],whichmom),:]

  # compute distance
  v = sum((mm[:data_value] - mm[:model_value]).^2)

  # status
  status = 1

  # time out
  t0 = time() - t0

  # return a dict
  ret = ["value" => v, "params" => deepcopy(x), "time" => t0, "status" => status, "moments" => mout]
  return ret

end
```

CHAPTER 4

Indices and tables

- *genindex*
- *modindex*
- *search*