SciencesPo Computational Economics Spring 2017

Florian Oswald

March 15, 2018

1 MPEC for Starters

Think of a consumer who decides how much of a certain good to consume, given prices. The consumer's behaviour is dictated by a preference parameter β, which is unobserved. We only observe data on choices, i.e. how much was consumed, at which price of the good. We will specify an economic model, and estimate the value of the preference parameter.

The utility function is given generically as

$$u(c;\beta)$$

and there is a structural relationship dictated by theory between demand for consumption and price of the consumption good: demand will be such that the marginal utility of consumption is equal to price, in other words

$$u_c(c;\beta) = p \quad \text{(FOC)} \tag{1}$$

Our aim here is to show how one can estimate the parameter vector β quite easily with MPEC. This means that we set up an estimation procedure that is a constrainted maximization problem. Our estimate of β will be a parameter vector that satisfies FOC and at the same time minimizes some criterion function.

In order to faciliate estimation, we assume that consumption is measured with error. In that case we don't observe actual consumption, but another variable `demand`, i.e. $q = c + \epsilon$, where ϵ follows some distribution function. Suppose we have N measurements of price-demand pairs $\{q_i, p_i\}_{i=1}^N$. For the sake of this example, we assume a very simple form of the utility function; bear in mind, however, that the actual strength of MPEC is that one can dispense with the need to find convenient closed form solutions, just so that the model can be solved. Our utility will be quadratic, as in

$$u(c;\beta) = c - \beta c^2$$

and marginal utility is

$$u_c = 1 - 2\beta c$$

Given our assumption about measurement error, we substitute out c

$$u_c = 1 - 2\beta(q - \epsilon)$$

The economic model prescribes that

$$p = u_c$$

or, in other words

$$p_i = 1 - 2\beta(q_i - \epsilon_i)$$

Our estimation will be based on minimizing the sum of squared errors, ϵ subject to complying with constraint FOC. We solve the following problem:

$$\min_{\{\epsilon\}_{i=1}^N, \beta} \sum_{i=1}^N \epsilon_i^2 s.t. u_c(q_i - \epsilon_i; \beta) = p_i$$

We notice right away that the choice of β does not directly influence the value of the objective function, therefore both the objective and it's gradient will be independent of β. However, every choice of β needs to go together with a corresponding choice of ϵ's which need to be chosen s.t. the constraint is satisfied. This is not a general feature of MPEC, as in many cases the parameters of interest would appear in the objective. Also, in a more elaborate model, we would probably have some data that we would want to model to come close to; In that case our objective function would be augmented by the likelihood function of that data together with our model, or we could have a moment criterion etc.

1.0.1 Task 1: write down the Lagrangian of this problem

1.0.2 Task 2: What is the length of the choice vector?

- What is the length of the choice vector x here? what are it's elements?
- What is the number of constraints?

1.0.3 Task 3: write down the constraint function and it's gradient wrt to x

- Denote the constraint function g as the collection of all constraints (there are $m = N$ constraints and $n = N + 1$ choice variables). Write down a typical element g_i, i.e. what does the i-th constraint look like?
- What does the gradient of g look like? It's a matrix. Different solvers make different assumptions here. NLopt expects an n by m matrix, where `g[j,i]` should be $\frac{\partial g_i}{\partial x_j}$

1.0.4 Task 4: Implement using plain vanilla NLopt

Take

- N=100 grid points
- $\beta = 0.1$ as true value
- setup this as an MPEC problem in NLopt!

```
In [8]: using Distributions
        srand(12345)
        normal = Normal(0,0.01)
        N = 100
        price = collect(linspace(0.05,0.95,N))
```

```
    beta0 = 0.1
    demand0 = (1.0-price) / (2*beta0)
    demand = demand0 - rand(normal,N);
```

```
In [9]: module nlopt

            using Distributions
            using NLopt
            srand(12345)
            normal = Normal(0,0.01)
            N = 100
            price = collect(linspace(0.05,0.95,N))
            beta0 = 0.1
            demand0 = (1.0-price) / (2*beta0)
            demand = demand0 - rand(normal,N)

            # objective function
            function obj(x,g)
                # x = [e_1,...,e_N,beta] i.e. (N+1,1)

                # gradient = [grad w.r.t e_1,
                #             ...,grad w.r.t e_N,
                #                 grad w.r.t beta] i.e. (N+1,1)
                if length(g)>0
                   g[:] = vcat(2.0*x[1:(end-1)],0.0)

                end
                #ăvalue of objective
                r = sum(x[1:(end-1)].^2)
                return r
            end

            #
            function constr(r::Vector,x::Vector,g::Matrix,n,q,p)
                if length(g) > 0
                    #ăg has to be n by m for nlopt
                    # \partial g_i / \partial x_j
                    g[:,:] = cat(1,diagm(2*x[end]*ones(n)),-2*(q'.-x[1:(end-1)]'))
                end
                # value of contraints
                r[:] = 1.0 - 2*x[end].*(q.-x[1:(end-1)]) .- p  # g_i
            end
            constr_clos(r::Vector,x::Vector,g::Matrix) = constr(r::Vector,x::Vector,g::Matrix,N,

            function run()
                opt = Opt(:LD_SLSQP,N+1)
                lower_bounds!(opt,[[-Inf for i in 1:N]...,0.0])
                upper_bounds!(opt,[[Inf for i in 1:N]...,1.0])
                min_objective!(opt,obj)
                equality_constraint!(opt,constr_clos,[1e-10 for i in 1:N])
                xtol_rel!(opt,1e-4)
                ftol_rel!(opt,1e-6)

                res = optimize(opt, vcat(rand(normal,N),0.9))
                println("beta0 = $beta0")
                println("beta  = $(res[2][end])")
                r = zeros(N)
                g =zeros(N+1,N)
                constr(r,res[2],g,N,demand,price);
                println("maximal error of constraint at solution = $(maxabs(r))")
                return res
            end

        end # module
```

```
Out[9]: nlopt
```

1.0.5 Task 5: Implement using JuMP

- Here is where the fun starts.
- Go back to HW-constrained for some inspiration!

```
In [10]: module jump

             using JuMP
             using Distributions
             srand(12345)
             normal = Normal(0,0.01)

             function run()

                 # create a model

                 # define constants (N, price, etc)
                 N = 100
                 price = collect(linspace(0.05,0.95,N))
                 beta0 = 0.1
                 demand0 = (1.0-price) / (2*beta0)
                 demand = demand0 - rand(normal,N)

                 #define JuMP variables


                 # define constraints and objective
```

```
                # solve
                status = solve(m)
                Dict(:obj=>getobjectivevalue(m),:beta=>getvalue(beta),:eps=>getvalue(eps))
            end
        end
```

```
WARNING: replacing module jump
```

```
Out[10]: jump
```