```julia
using SparseArrays, Statistics, ForwardDiff, MAT, Optim, LinearAlgebra
```

```julia
cd("/Users/isaacgeng/OneDrive - The Chinese University of Hong Kong/ECON5480-IO")
```

Data preparation

1. Read the input matrice
2. calculate the miscelleous matrices for later use

```julia
begin
    # load the files into matrixes, use meta-programming to reduce later
    ps2raw = matopen("data/ps2.mat")
    ivraw = matopen("data/iv.mat")
    x2 = read(ps2raw,  "x2")
    id = read(ps2raw, "id")
    s_jt = read(ps2raw, "s_jt")
    x1 = read(ps2raw, "x1")
    v = read(ps2raw,"v")
    demogr = read(ps2raw, "demogr")
    id_demo = read(ps2raw, "id_demo")
    iv = read(ivraw, "iv")
    close(ps2raw)
    close(ivraw)
end
```

```
(20, 94, 24, 20)
```

```julia
ns, nmkt, nbrn, n_inst = 20, 94, 24, 20
```

```
cdid = 24×1 Array{LinearAlgebra.Adjoint{Float64,Array{Float64,1}},2}:
        [1.0 2.0 … 93.0 94.0]
        [1.0 2.0 … 93.0 94.0]
        [1.0 2.0 … 93.0 94.0]
        [1.0 2.0 … 93.0 94.0]
        [1.0 2.0 … 93.0 94.0]
        [1.0 2.0 … 93.0 94.0]
        [1.0 2.0 … 93.0 94.0]
        ⋮
        [1.0 2.0 … 93.0 94.0]
        [1.0 2.0 … 93.0 94.0]
        [1.0 2.0 … 93.0 94.0]
        [1.0 2.0 … 93.0 94.0]
        [1.0 2.0 … 93.0 94.0]
        [1.0 2.0 … 93.0 94.0]
```

```
cdid = kron([1:nmkt]', ones(nbrn, 1))
```

```
cdindex = 24:24:2256
```

```
cdindex = 24:24:2256
```

```
IV = 2256×44 SparseMatrixCSC{Float64,Int64} with 47376 stored entries
    :
      [1   ,  1]  =  -0.215973
      [2   ,  1]  =  -0.245239
      [3   ,  1]  =  -0.176459
      [4   ,  1]  =  -0.121401
      [5   ,  1]  =  -0.132611
      [6   ,  1]  =  -0.1535
      ⋮
      [2112, 44]  =  1.0
      [2136, 44]  =  1.0
      [2160, 44]  =  1.0
      [2184, 44]  =  1.0
      [2208, 44]  =  1.0
      [2232, 44]  =  1.0
      [2256, 44]  =  1.0
```

```
IV = [iv[:,2:n_inst+1] x1[:, 2:nbrn+1]]
```

Starting values: 0 means corresponding coef not maxed over.

```
θ_2w = 4×5 Array{Float64,2}:
  0.3772   3.0888   0.0     1.1859   0.0
  1.848   16.598   -0.659   0.0     11.6245
 -0.0035  -0.1925   0.0     0.0296   0.0
  0.081    1.4684   0.0    -1.5143   0.0
```

```
θ_2w = [0.3772    3.0888        0    1.1859         0;
             1.8480   16.5980    -.6590         0   11.6245;
            -0.0035   -0.1925         0    0.0296         0;
             0.0810    1.4684         0   -1.5143         0]
```

```
ind =
 [CartesianIndex(1, 1),  CartesianIndex(2, 1),  CartesianIndex(3, 1),  CartesianIndex
```

```
ind = findall(!iszero,θ_2w)
```

```
θ_2_init =
 [0.3772,  1.848,  -0.0035,  0.081,  3.0888,  16.598,  -0.1925,  1.4684,  -0.659,  1.18
```

```
θ_2_init = filter(!iszero,θ_2w)
```

```
θ_i =  [1,  2,  3,  4,  1,  2,  3,  4,  2,  1,  3,  4,  2]
```

```
θ_i = getindex.(ind,1)
```

```
θ_j =  [1,  1,  1,  1,  2,  2,  2,  2,  3,  4,  4,  4,  5]
```

```
θ_j = getindex.(ind,2)
```

create a weight matrix

- This may not need optimized, IV'*IV is not sparse?

```
44×44 SparseMatrixCSC{Float64,Int64} with 1384 stored entries:
  [1 ,  1]  =  319.472
  [2 ,  1]  =  -3.42701
  [3 ,  1]  =  641.219
  [4 ,  1]  =  24.5327
  [5 ,  1]  =  -20.7317
  [6 ,  1]  =  -2.50864
  ⋮
  [15, 44]  =  10.1538
  [16, 44]  =  10.2782
  [17, 44]  =  10.2372
  [18, 44]  =  10.0945
  [19, 44]  =  10.0649
  [20, 44]  =  10.0597
  [44, 44]  =  94.0
```

```
IV'*IV
```

```
invA = 44×44 Array{Float64,2}:
          0.00344519    0.00210364   -5.98984e-5   …   0.000420291   0.000245385
          0.00210364    0.233132      5.79456e-5      -0.00471337    0.00636716
         -5.98984e-5    5.79456e-5    2.9956e-5        1.56934e-5   -1.78572e-5
         -0.000677831   0.000462207   6.07925e-6      -0.000818284  -0.000721377
          0.00321384    0.0226931    -0.000383199      0.0316399     0.0719253
         -0.228229     -0.977446      0.0337041    …   0.338339      2.34971
          1.69742e-5   -0.00120553    1.67574e-6       1.45939e-5   -0.00370381
          ⋮                                         ⋱
         -5.8233e-5    -0.00390934    0.000349586      0.017126      0.0188
          0.000427227  -0.00276034   -4.85427e-5       0.0223205     0.0390922
          0.00032356   -0.00755811    7.20235e-5   …   0.0234568     0.0253501
          0.00054432   -0.00113857   -6.77893e-5       0.0210975     0.0386224
          0.000420291  -0.00471337    1.56934e-5       0.0292198     0.0226411
          0.000245385   0.00636716   -1.78572e-5       0.0226411     0.076844
```

```
invA = inv(Matrix(IV'*IV))
```

Logit results and save the mean utility as initial values for the search below

```
temp = 2256×1 Array{Float64,2}:
         0.012417211928625965
         0.02022659873144509
         0.033221109690458016
         0.038991071031856776
         0.05692521231589029
         0.0835271039078989
         0.10854186992864956
         ⋮
        44.65292262728556
        44.677624485133734
        44.6855381988558
        44.687767381537256
        44.699230050703
        44.725438371923566
```

```julia
# compute the outside good market share by market
temp = cumsum(s_jt, dims=1)
```

```
sum1 = 94×1 Array{Float64,2}:
         0.44477547187965527
         0.8595949123499734
         1.5281796851321725
         1.95250640130562
         2.5471270403141375
         2.967975619889587
         3.444067536541951
         ⋮
        42.23550898632855
        42.701755210767544
        43.317401458555324
        43.75837014082409
        44.36791515168355
        44.725438371923566
```

```julia
sum1 = temp[cdindex,:] # total market share in each market besides outside option
```

```
93×1 Array{Float64,2}:
 0.41481944047031816
 0.6685847727821991
 0.4243267161734474
 0.5946206390085176
 0.4208485795754493
 0.47609191665236406
 0.4621368512560129
 ⋮
 0.39430060899942987
 0.4662462244389971
 0.6156462477877795
 0.4409686822687675
 0.6095450108594562
 0.3575232202400187
```

```julia
sum1[2:size(sum1,1),:] = diff(sum1,dims=1)
```

```
outshr = 2256×1 Array{Float64,2}:
          0.5552245281203447
          0.5552245281203447
          0.5552245281203447
          0.5552245281203447
          0.5552245281203447
          0.5552245281203447
          0.5552245281203447
          ⋮
          0.6424767797599813
          0.6424767797599813
          0.6424767797599813
          0.6424767797599813
          0.6424767797599813
          0.6424767797599813
```

```
outshr = 1.0 .- repeat(sum1,inner=(24,1))
```

```
y = 2256×1 Array{Float64,2}:
     -3.800289018199724
     -4.264046140702415
     -3.7548455527342353
     -4.5667072065411745
     -3.4326663566100533
     -3.0383902613670397
     -3.0999062957692622
      ⋮
     -4.225279599872224
     -3.258452218334516
     -4.3967335069639795
     -5.663695667934123
     -4.026235080001827
     -3.1992537114848822
```

```
y = log.(s_jt) - log.(outshr)
```

```
mid = 25×2256 Array{Float64,2}:
        0.0703482     0.117966      0.131403     …   0.0947895     0.121742
        1.0          -3.37204e-15   5.46223e-15     -8.89001e-16   8.62712e-15
       -1.06942e-14   1.0          -6.70117e-15     -9.12441e-16  -1.43567e-14
       -5.09751e-14  -7.77451e-14   1.0              3.19841e-15  -4.81218e-14
       -2.84986e-14  -4.70755e-14  -2.27042e-14     -2.22055e-15  -3.6642e-14
       -1.51129e-13  -2.50554e-13  -2.24045e-13  …  -2.47472e-14  -2.90746e-13
       -6.7847e-14   -1.03961e-13  -5.15169e-14     -1.59608e-15  -7.88695e-14
        ⋮                                        ⋱                 ⋮
        3.42298e-15   2.96302e-15   1.36878e-14     -7.29119e-16   1.34115e-14
       -1.31374e-13  -1.94889e-13  -1.18046e-13  …  -4.66692e-15  -1.80158e-13
       -1.02314e-13  -1.70265e-13  -1.41652e-13     -1.24146e-14  -1.8776e-13
       -3.58453e-14  -6.61556e-14  -4.35236e-14     -5.55035e-15  -6.24508e-14
       -3.54246e-14  -6.58468e-14  -5.05876e-14      1.0          -6.93136e-14
       -1.24215e-13  -2.04567e-13  -1.82605e-13     -2.17346e-14   1.0
```

```
mid = x1'*IV*invA*IV'
```

```
t = 25×1 Array{Float64,2}:
     -30.0977549512755
      -1.7746816664495049
       0.5913114852760736
       0.026462795365476414
      -1.0110885552391162
       2.2344781361184483
       0.038057457340183405
       ⋮
      -0.007368574096889394
       1.0330609455366153
       0.9053325462480496
      -1.302887694765856
      -0.39966497788707905
       0.39737532411088705
```

```
t = (mid*x1) \ (mid*y)
# instead of using inv(mid*x1)*(mid*y), use this to utilize qr factorization.
```

```
mvalold = 2256×1 Array{Float64,2}:
           0.019363470627957263
           0.05812236754340788
           0.01909747890087745
           0.007196555381675834
           0.08844663949544648
           0.016792613173624037
           0.013919925913178111
           ⋮
           0.021893561509520074
           0.06317577766104211
           0.006163237773629371
           0.004397126099454333
           0.03288668004952659
           0.03200643560754751
```

```
mvalold = exp.(x1*t) # fitted log shares
```

```
oldt2 =  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

```
oldt2 = zeros(size(θ_2_init))
```

```
vfull =
2256×80 Array{Float64,2}:
 0.434101  -0.726649  -0.623061  -0.041317  …  2.02119   0.190172  -0.0397307
 0.434101  -0.726649  -0.623061  -0.041317     2.02119   0.190172  -0.0397307
 0.434101  -0.726649  -0.623061  -0.041317     2.02119   0.190172  -0.0397307
 0.434101  -0.726649  -0.623061  -0.041317     2.02119   0.190172  -0.0397307
 0.434101  -0.726649  -0.623061  -0.041317     2.02119   0.190172  -0.0397307
 0.434101  -0.726649  -0.623061  -0.041317  …  2.02119   0.190172  -0.0397307
 0.434101  -0.726649  -0.623061  -0.041317     2.02119   0.190172  -0.0397307
 ⋮                                          ⋱
 1.84276   -0.650623  -0.384532   0.574788  …  0.748736  1.76507   -2.10289
 1.84276   -0.650623  -0.384532   0.574788     0.748736  1.76507   -2.10289
 1.84276   -0.650623  -0.384532   0.574788     0.748736  1.76507   -2.10289
 1.84276   -0.650623  -0.384532   0.574788     0.748736  1.76507   -2.10289
 1.84276   -0.650623  -0.384532   0.574788     0.748736  1.76507   -2.10289
 1.84276   -0.650623  -0.384532   0.574788  …  0.748736  1.76507   -2.10289
```

```
vfull = repeat(v,inner=(24,1))
```

```
dfull =
2256×80 Array{Float64,2}:
  0.495123   0.378762  0.105015  -1.48548   …  -0.230851  -0.230851  -0.230851
  0.495123   0.378762  0.105015  -1.48548      -0.230851  -0.230851  -0.230851
  0.495123   0.378762  0.105015  -1.48548      -0.230851  -0.230851  -0.230851
  0.495123   0.378762  0.105015  -1.48548      -0.230851  -0.230851  -0.230851
  0.495123   0.378762  0.105015  -1.48548      -0.230851  -0.230851  -0.230851
  0.495123   0.378762  0.105015  -1.48548   …  -0.230851  -0.230851  -0.230851
  0.495123   0.378762  0.105015  -1.48548      -0.230851  -0.230851  -0.230851
  ⋮                                         ⋱
 -0.0316119  0.486454  0.86236    0.368064  …  -0.230851   0.769149   0.769149
 -0.0316119  0.486454  0.86236    0.368064     -0.230851   0.769149   0.769149
 -0.0316119  0.486454  0.86236    0.368064     -0.230851   0.769149   0.769149
 -0.0316119  0.486454  0.86236    0.368064     -0.230851   0.769149   0.769149
 -0.0316119  0.486454  0.86236    0.368064     -0.230851   0.769149   0.769149
 -0.0316119  0.486454  0.86236    0.368064  …  -0.230851   0.769149   0.769149
```

```
dfull = repeat(demogr,inner=(24,1))
```

individual choice probabilities

- original ind_sh matlab code

```
function f = ind_sh(expmval,expmu)
# This function computes the "individual" probabilities of choosing each brand
global ns cdindex cdid
eg = expmu.*kron(ones(1,ns),expmval);
temp = cumsum(eg);
sum1 = temp(cdindex,:);
sum1(2:size(sum1,1),:) = diff(sum1);

denom1 = 1./(1+sum1);
denom = denom1(cdid,:);
f = eg.*denom;
```

- We first show the math of market share, note that the integrad is individual choice probabilities.

$$s_j = \int \frac{e^{\delta_j + x_j \sigma \nu}}{1 + \sum_{i=1}^{J} e^{\delta_i + x_i \sigma \nu}} dF_\nu(\nu)$$

- where

$$\delta_j = \beta[1] * p[j] + x[:, j]' * \beta[2 : end] + \xi[j]$$

- here, we choose to replicate the nevo's way, that is, we donot write out the δ_j in ind_sh function.

```
ind_sh (generic function with 1 method)
```

```
function ind_sh(expmval, expmu)
    eg = expmu.*kron(ones(1,ns), expmval)
    # common techinique in nevo: begin
    temp = cumsum(eg,dims=1);
    sum1 = temp[cdindex,:]
    sum1[2:size(sum1, 1),:] = diff(sum1, dims=1)
    # end
    denom1 = 1 ./ (1 .+ sum1)
    # denom. = denom1[cdid,:]
    denom = repeat(denom1, inner=(24,1))
    f = eg.*denom
end
```

- the ind_sh function's second parameter `expmu` is the result of mufunc, we change the name to μ _func instead.

```
μ_func (generic function with 1 method)
```

```
function μ_func(x2,θ_2w)
    n,k = size(x2)
    j = size(θ_2w,2)-1
    μ = Matrix{Real}(undef, n,ns); # 2256, 20
    for i = 1:ns
        v_i = vfull[:, i:ns:k*ns] # try v_i = vfull[:,3:20:80] if unclear
        d_i = dfull[:, i:ns:j*ns]
        μ[:,i] = x2.*v_i*θ_2w[:,1] + x2.*(d_i*θ_2w[:,2:j+1]')*ones(k,1);
    end
    f = μ;
end
```

- Note that here I use a different sentence to create a empty container where the type is implicitly specified as being subtypes of `Real`. This is needed since later on we use `Autodiff.jl` and `Autodiff.jl` uses dual to calculate its direvatives and Float64 type is not capable of obtaining its dual.

```
md"""
- Note that here I use a different sentence to create a empty container where the type is implicitly specified as being subtypes of `Real`. This is needed since later on we use `Autodiff.jl` and `Autodiff.jl` uses dual to calculate its direvatives and Float64 type is not capable of obtaining its dual.
"""
```

debug: ind_sh

- we test `ind_sh` work or not. And we found those julia syntax difference to matlab:

1. exp() in matlab need to be exp.() in case the matrix is not square
2. the square brackets near `x2.*v_i*θ_2w[:,1]` to remove
3. the square brackets near `d_i*θ_2w[:,2:j+1]'` should be round brackets.

- debug console, for record only.

```
exp.(μ_func(x2,θ_2w))
size(x2.*v_i*θ_2w[:,1])
v_i = vfull[:,3:20:80]
```

```
2256×20 Array{Real,2}:
  2.11579    2.82664   -0.000206899  -5.71468   …  1.78783     6.57991   -0.202249
  0.43141    0.970107  -0.450538     -1.3473       0.905542    2.41477   -0.633585
  1.75501    3.22422   -0.159715     -4.97789      1.69439     6.20854   -0.446988
  0.770159  -1.05501   -0.232977     -4.34228      1.33732     4.62603    0.616178
 -0.179497  -2.09012   -0.487795     -1.88285      0.841282    2.28532    0.370767
 -0.327173  -2.5857    -0.500889     -1.41681   …  0.723944    1.7022     0.396711
  1.82085    3.50555   -0.158666     -5.20074      1.75396     6.50805   -0.470124
  ⋮                                             ⋱
  0.50448    0.342741   0.781991      0.598769  …  0.201967   -3.64451    0.0974525
  0.371518   1.89337    1.61524       1.36262      1.40074    -8.85789   -2.34911
  0.339751   1.44563    0.589734      0.958038     0.251378   -5.99706   -1.55861
  0.362223   1.96344    1.79555       1.42958      1.56878    -9.47501   -2.32325
  0.399383   1.55126    0.759131      1.04011      0.565678   -6.05262   -2.29172
  0.355032   2.3624     2.75871       1.79918   …  2.56354   -12.4458    -2.65546
```

```
μ_func(x2,θ_2w)
```

```
(2256×1 Array{Float64,2}:,  2256×20 Array{Float64,2}:
  0.0193635                 8.29613   16.8887     0.999793  0.00329721  …   5.976
  0.0581224                 1.53943    2.63823    0.637285  0.25994         2.473
  0.0190975                 5.78353   25.134      0.852386  0.0068886       5.443
  0.00719656                2.16011    0.348191   0.792172  0.0130068       3.808
  0.0884466                 0.83569    0.123673   0.613979  0.152156        2.319
  0.0167926                 0.720959   0.0753435  0.605992  0.242487    …   2.062
  0.0139199                 6.17711   33.2997     0.853281  0.00551247      5.777
  ⋮                         ⋮                                           ⋱
  0.0218936                 1.65612    1.4088     2.18582   1.81988     …   1.223
  0.0631758                 1.44993    6.64173    5.02907   3.9064          4.058
  0.00616324                1.4046     4.24451    1.80351   2.60658         1.285
  0.00439713                1.43652    7.12376    6.02281   4.17696         4.800
  0.0328867                 1.4909     4.7174     2.13642   2.82954         1.760
  0.0320064                 1.42623   10.6164    15.7794    6.04472     …  12.981
```

```
a, b = mvalold, exp.(μ_func(x2,θ_2w))
```

```
2256×20 Array{Float64,2}:
 0.0672621   0.0798907    0.0135933   …  0.0392761   0.162527     0.00913435
 0.037464    0.0374605    0.0260081      0.0487885   0.00757506   0.0178119
 0.0462467   0.117261     0.0114299      0.0352812   0.110569     0.00705313
 0.00650896  0.000612152  0.0040029      0.0093029   0.00856067   0.00769587
 0.0309484   0.00267222   0.0381299      0.0696223   0.0101276    0.0740003
 0.00506921  0.000309088  0.00714523  …  0.0117551   0.00107324   0.0144191
 0.0360026   0.113239     0.00833989     0.0272944   0.108735     0.00502337
 ⋮                                    ⋱
 0.0174324   0.00695026   0.00482998  …  0.00387822  0.000571189  0.0195765
 0.04404     0.094551     0.0320666      0.0371095   8.97167e-6   0.00489149
 0.00416207  0.00589483   0.00112187     0.00114705  1.52958e-5   0.00105198
 0.00303689  0.00705851   0.0026729      0.00305553  3.36882e-7   0.000349374
 0.0235732   0.0349589    0.00709122     0.00838096  7.72063e-5   0.0026967
 0.021947    0.0765681    0.0509733   …  0.0601411   1.25702e-7   0.00182423
```

```
ind_sh(a, b)
```

market share function calculation

original matlab code

```
function f = mktsh(mval, expmu)
% This function computes the market share for each product

% Written by Aviv Nevo, May 1998.

global ns
f = sum((ind_sh(mval,expmu))')/ns;
f = f';
```

```
mktsh (generic function with 1 method)
```

```julia
function mktsh(mval, expmu)
    f = sum(ind_sh(mval, expmu)', dims=1)./ns;
    f = f'
end
```

```
2256×1 LinearAlgebra.Adjoint{Float64,Array{Float64,2}}:
 0.0401675767914942
 0.027018775321159278
 0.0357693354806783
 0.005440613902253944
 0.039801265872969685
 0.007613829789776675
 0.03287555438179839
 ⋮
 0.014087928954088821
 0.036391826769690265
 0.004155650447011357
 0.0026598265451732842
 0.017412693616885393
 0.027020995418857323
```

```julia
mktsh(a, b)
```

Mean utility level

```
meanval (generic function with 3 methods)
```

```
function meanval(θ_2, oldt2=oldt2, mvalold=mvalold)
    if maximum(abs.(θ_2 - oldt2)) < 0.01
        tol = 1e-9
        flag = 0
    else
        tol = 1e-6
        flag = 1
    end

    θ_2w = Matrix(sparse(θ_i,θ_j, θ_2))
    expmu = exp.(μ_func(x2, θ_2w))
    norm = 1
    avgnorm = 1

    i = 0

    # contraction mapping to compute mean utility
    while (norm > tol*10^(flag*floor(i/50))) & (avgnorm > 1e-
3*tol*10^(flag*floor(i/50)))
        mval = mvalold .* s_jt ./ mktsh(mvalold, expmu);
        t = abs.(mval - mvalold)
        norm = maximum(t)
        avgnorm = mean(t)
        mvalold = mval
        i = i + 1
    end

    println("# of iterations. for delta convergence: "*string(i))

    if flag == 1 & maximum(isnan.(mval)) < 1
        mval = mvalold .* s_jt ./ mktsh(mvalold, expmu)
        mvalold = mval
        oldt2 = θ_2
    end

    log.(mval)
end
```

```
16.598
```

```
maximum(abs.(θ_2_init - oldt2))
```

```
2256×1 Array{Float64,2}:
 -6.028177833920886
 -4.377498429092451
 -5.850799205135971
 -5.4474547778298
 -3.435192462214368
 -2.9785765691419868
 -5.470251465124257
  ⋮
 -4.610510922176886
 -3.697397185609129
 -4.858198456796975
 -6.169864908290707
 -4.283885281988619
 -4.127636032805655
```

```
meanval(θ_2_init)
```

Jacobian

I use autodiff to save the approx calculation process.

```
md"""
## Jacobian
I use autodiff to save the approx calculation process.
"""
```

```
jacob = #1 (generic function with 1 method)
```

```
jacob = θ_2 -> ForwardDiff.jacobian(x -> meanval(x), θ_2)
```

```
jacob! (generic function with 1 method)
```

```
function jacob!(g,θ_2)
    g = Matrix{Real}(undef, nbrn*nmkt,size(θ_2)[1])
    g = jacob(θ_2)
    return g
end
```

```
g =
2256×13 Array{Real,2}:
 #undef  #undef  #undef  #undef  #undef  …  #undef  #undef  #undef  #undef  #undef
 #undef  #undef  #undef  #undef  #undef     #undef  #undef  #undef  #undef  #undef
 #undef  #undef  #undef  #undef  #undef     #undef  #undef  #undef  #undef  #undef
 #undef  #undef  #undef  #undef  #undef     #undef  #undef  #undef  #undef  #undef
 #undef  #undef  #undef  #undef  #undef     #undef  #undef  #undef  #undef  #undef
 #undef  #undef  #undef  #undef  #undef  …  #undef  #undef  #undef  #undef  #undef
 #undef  #undef  #undef  #undef  #undef     #undef  #undef  #undef  #undef  #undef
   ⋮                                     ⋱                   ⋮
 #undef  #undef  #undef  #undef  #undef  …  #undef  #undef  #undef  #undef  #undef
 #undef  #undef  #undef  #undef  #undef     #undef  #undef  #undef  #undef  #undef
 #undef  #undef  #undef  #undef  #undef     #undef  #undef  #undef  #undef  #undef
 #undef  #undef  #undef  #undef  #undef     #undef  #undef  #undef  #undef  #undef
 #undef  #undef  #undef  #undef  #undef     #undef  #undef  #undef  #undef  #undef
 #undef  #undef  #undef  #undef  #undef  …  #undef  #undef  #undef  #undef  #undef
```

```
g = Matrix{Real}(undef, 2256,13)
```

```
2256×13 Array{Float64,2}:
 -0.113572    0.0564604  -1.07871    0.066295   …   1.20769   0.288679   -0.0130281
 -0.178001    0.0251814  -0.453407  -0.105183       6.27111   0.364888   -0.0118995
 -0.0873896   0.0201807  -0.651902   0.060925       1.46125   0.404353   -0.0116321
 -0.142381    0.03142    -0.729319  -0.0639244      1.35705   0.213018    0.00236581
 -0.215294    0.0348447  -3.22648   -0.0205948     -0.34304   0.106863    0.00407475
 -0.236009    0.0344811  -4.22441   -0.0113813  …  -0.828587  0.0896996   0.00400528
 -0.0842909   0.0112049  -0.884968   0.0696001      1.18087   0.413128   -0.0115617
  ⋮                                             ⋱   ⋮
 -0.590207   -0.0509698   6.25714   -0.346959   …   7.61154   0.378679   -0.036364
 -0.131617   -0.0471219   1.75481    0.0587896     -0.492494  0.125976    0.00060209
 -0.421002   -0.137627    4.70053    0.0331532     -0.539062  0.0618994  -0.0269819
 -0.130671   -0.0488683   1.42295    0.0669783     -0.106503  0.128859    0.00146716
 -0.163183   -0.0471014   3.86034    0.0293275     -2.36057   0.101856   -0.0020668
 -0.0602672  -0.0326707  -0.212633   0.0930127  …   2.06783   0.154437    0.00363363
```

```julia
jacob(θ_2_init)
```


```
2256×13 Array{Float64,2}:
 -0.113572    0.0564604  -1.07871    0.066295   …   1.20769   0.288679   -0.0130281
 -0.178001    0.0251814  -0.453407  -0.105183       6.27111   0.364888   -0.0118995
 -0.0873896   0.0201807  -0.651902   0.060925       1.46125   0.404353   -0.0116321
 -0.142381    0.03142    -0.729319  -0.0639244      1.35705   0.213018    0.00236581
 -0.215294    0.0348447  -3.22648   -0.0205948     -0.34304   0.106863    0.00407475
 -0.236009    0.0344811  -4.22441   -0.0113813  …  -0.828587  0.0896996   0.00400528
 -0.0842909   0.0112049  -0.884968   0.0696001      1.18087   0.413128   -0.0115617
  ⋮                                             ⋱   ⋮
 -0.590207   -0.0509698   6.25714   -0.346959   …   7.61154   0.378679   -0.036364
 -0.131617   -0.0471219   1.75481    0.0587896     -0.492494  0.125976    0.00060209
 -0.421002   -0.137627    4.70053    0.0331532     -0.539062  0.0618994  -0.0269819
 -0.130671   -0.0488683   1.42295    0.0669783     -0.106503  0.128859    0.00146716
 -0.163183   -0.0471014   3.86034    0.0293275     -2.36057   0.101856   -0.0020668
 -0.0602672  -0.0326707  -0.212633   0.0930127  …   2.06783   0.154437    0.00363363
```


```julia
jacob!(g, θ_2_init)
```

- I compared the results of hand written approximated jacobian (matlab result) and found that there is a considerable difference between the audodiff result and the jacobian approximation result.

```julia
begin
    jcb_approx_raw = matopen("data/jcb.mat")
    jcb_approx = read(jcb_approx_raw,"jcb")
    close(jcb_approx_raw)
end
```

```
2256×13 Array{Float64,2}:
 -0.135854   0.0495283  -0.85803    …  -0.0539627  0.197726   -0.00623629
 -0.218614   0.0283561  -1.19337        4.06796    0.207756   -0.00568217
 -0.132355   0.0281426  -0.598074       0.20359    0.263632   -0.00539735
 -0.142942   0.0378521  -0.595299       0.385622   0.210046    0.00567736
 -0.227667   0.0427094  -4.09623       -1.13683    0.100752    0.00686473
 -0.251945   0.0393491  -5.24152    …  -1.63661    0.0897562   0.00641203
 -0.127702   0.021502   -0.714417      -0.155577   0.275678   -0.00612964
  ⋮                                 ⋱   ⋮
 -0.549464  -0.0568355   5.43001    …   8.56362    0.420358   -0.0362712
 -0.209086  -0.0554244   1.35485        0.223064   0.189339   -0.00609143
 -0.434294  -0.145005    4.0423         0.407729   0.0928363  -0.0331071
 -0.212416  -0.0580662   1.06265        0.564873   0.194313   -0.00566825
 -0.216335  -0.0508981   3.23395       -1.41078    0.151301   -0.00701013
 -0.174555  -0.0428443  -0.390959   …   2.64175    0.242072   -0.00418428
```

```
jcb_approx
```

GMM objective function: gmmobjg

```
gmmobjg (generic function with 2 methods)
```

```
function gmmobjg(θ_2,nargout=0)
    δ = meanval(θ_2)
    # deals with cases were the min algorithm drifts into region where the objective is not defined
    if maximum(isnan.(δ)) == 1
        f = 1e+10
    else
        temp1 = x1' * IV
        temp2 = δ' * IV
        θ_1= (temp1 * invA * temp1') \ (temp1 * invA * temp2')
        gmmresid = δ - x1*θ_1
        temp1 = gmmresid' * IV;
        f1 = temp1 * invA * temp1';
        f = f1
        # donot know what is for
        if nargout > 1
            temp = jacob(mvalold,θ_2)'
            df = 2 * temp * IV * invA * IV' * gmmresid
        end
    end
    # if nargout > 1
    #   f, df, gmmresid
    # else
    #   f, gmmresid
    # end
    real(f[1,1])
end
```

```
gmmresid (generic function with 2 methods)
```

```julia
function gmmresid(θ_2, nargout=0)
    δ = meanval(θ_2)
    # deals with cases were the min algorithm drifts into region where the
objective is not defined
    if maximum(isnan.(δ)) == 1
        f = 1e+10
    else
        temp1 = x1' * IV
        temp2 = δ' * IV
        θ_1= (temp1 * invA * temp1') \ (temp1 * invA * temp2')
        gmmresid = δ - x1*θ_1
        temp1 = gmmresid' * IV;
        f1 = temp1 * invA * temp1';
        f = f1
        # donot know what is for
        if nargout > 1
            temp = jacob(mvalold,θ_2)'
            df = 2 * temp * IV * invA * IV' * gmmresid
        end
    end
    gmmresid
end
```

θ_1 (generic function with 2 methods)

```julia
function θ_1(θ_2, nargout=0)
    δ = meanval(θ_2)
    # deals with cases were the min algorithm drifts into region where the
objective is not defined
    if maximum(isnan.(δ)) == 1
        f = 1e+10
    else
        temp1 = x1' * IV
        temp2 = δ' * IV
        θ_1= (temp1 * invA * temp1') \ (temp1 * invA * temp2')
        gmmresid = δ - x1*θ_1
        temp1 = gmmresid' * IV;
        f1 = temp1 * invA * temp1';
        f = f1
        # donot know what is for
        if nargout > 1
            temp = jacob(mvalold,θ_2)'
            df = 2 * temp * IV * invA * IV' * gmmresid
        end
    end
    θ_1
end
```

14.900788598425498

```julia
gmmobjg(θ_2_init)
```

```
2256×1 Array{Float64,2}:
 -0.2169875391113809
 -1.4126699582690376
 -0.16712588928427508
  0.49705280346193437
 -0.7746928031843456
  1.3354680714179188
  0.7497642559447719
  ⋮
 -0.730826367879919
 -0.3929790901818202
  0.5293764728179831
 -0.03440879284107101
 -0.6210771449038215
  0.6913301928578068
```

```
gmmresid(θ_2_init)
```

```
[-0.0324637,  0.0120607,  0.249196,  -0.0320783,  0.0930545,  -0.0227259,  1.21883,
```

```
ForwardDiff.gradient(gmmobjg,θ_2_init)
```

```
jcb! (generic function with 1 method)
```

```
function jcb!(g,θ_2)
    g = Matrix{Real}(undef, 1,size(θ_2)[1])
    g = ForwardDiff.gradient(gmmobjg,θ_2)
    g
end
```

Variance-covariance matrix of gmmobjg func wrt θ_2

```
md"""
## Variance-covariance matrix of gmmobjg func wrt $\theta_2$
"""
```

```
var_cov (generic function with 2 methods)
```

```
function var_cov(θ_2,   gmmresid =gmmresid(θ_2))
    N = size(x1,1)
    Z = size(IV,2)
    temp = jacob(θ_2)
    a = [x1 temp]' * IV
    IVres = IV.*(gmmresid*ones(1, Z))
    b = IVres' * IVres
    f = inv(a*invA*a')*a*invA*b*invA*a'*inv(a*invA*a')
end
```

```
38×38 Array{Float64,2}:
  59.9255    -3.13425     -4.10071    …   0.118315    1.25808    -4.5728
  -3.13425    0.36214      0.254012      -0.00245855 -0.153116   -0.169943
  -4.10071    0.254012     0.368527      -0.0146237  -0.347283    1.01468
  -3.37068    0.338634     0.287947      -0.00582736 -0.245155    0.306161
  -2.15901    0.179783     0.16151       -0.00283312 -0.037137   -0.129512
  -2.5139     0.176865     0.19501    …  -0.00359811 -0.0665287  -0.0574784
  -2.66035    0.188699     0.219973      -0.00541826 -0.173015    0.304722
   ⋮                                  ⋱   ⋮
  -1.69301    0.222024     0.26744       -0.0143178  -0.680385    1.85668
 -66.2865     3.05049      4.68002       -0.175856   -3.03129    13.2952
  -0.383333   0.027936     0.268789      -0.0266834  -1.07089     3.9574
   0.118315  -0.00245855  -0.0146237  …   0.00132035  0.0257068  -0.0976947
   1.25808   -0.153116    -0.347283       0.0257068   1.21691    -3.70788
  -4.5728    -0.169943     1.01468       -0.0976947  -3.70788    27.1241
```

```
var_cov(θ_2_init)
```

Final routine solving for optimal θ_2 to gmmobjg

- I found that using derivatives on mealval function or on gmmobjg itself has no difference as below shows.

```
[0.3772, 1.848, -0.0035, 0.081, 3.0888, 16.598, -0.1925, 1.4684, -0.659, 1.18
```

```
Optim.minimizer(optimize(gmmobjg, jacob!,θ_2_init, BFGS()))
```

```
θ2 =
 [0.3772, 1.848, -0.0035, 0.081, 3.0888, 16.598, -0.1925, 1.4684, -0.659, 1.18
```

```
θ2 = Optim.minimizer(optimize(gmmobjg, jcb!,θ_2_init, BFGS()))
```

```
ttime = 0.260131333
```

```
ttime = @elapsed res = optimize(gmmobjg, jcb!,θ_2_init, BFGS())
```

```
fval = 14.900788598425498
```

```
fval = Optim.minimum(res)
```

```
38×38 Array{Float64,2}:
  1.99546     -0.161053    -0.13906     …  -0.00331636   -0.297945     0.733544
 -0.161053     0.251598     0.0309435       0.00378899   -0.0679572   -0.0948127
 -0.13906      0.0309435    0.160682       -0.00728675   -0.0942188    0.0512943
 -0.137674     0.14325      0.0386999       0.00224776   -0.0868595   -0.0298376
 -0.171606     0.0472731    0.0203403      -0.00165798   -0.0240024   -0.0519346
 -0.223019     0.0211718    0.0280513   …   0.00192219   -0.0321983   -0.0506335
 -0.160874     0.032874     0.0230462       0.00353204   -0.0753077   -0.00957576
  ⋮                                     ⋱   ⋮
  0.146927     0.133146     0.0690405      -0.00237438   -0.250005     0.196913
 -0.246743    -0.012411     0.0168194      -0.000865385  -0.0425038    0.199821
  0.243567    -0.00411279   0.0978389      -0.018309     -0.404085     0.571678
 -0.00331636   0.00378899  -0.00728675  …   0.0076147     0.00436815  -0.00973686
 -0.297945    -0.0679572   -0.0942188       0.00436815    0.597148    -0.470613
  0.733544    -0.0948127    0.0512943      -0.00973686   -0.470613     4.86568
```

```julia
begin
    vcov = var_cov(θ2)
    se = real(sqrt(var_cov(θ2)))
end
```

```
4×5 Array{Float64,2}:
  0.0129682    -0.426681     0.0         0.243567    0.0
  0.0342588     7.3717      -0.246743    0.0         0.733544
 -0.00195039    0.00126177   0.0        -0.00331636  0.0
  0.000320648   0.146927     0.0        -0.297945    0.0
```

```julia
begin
    θ2w = Matrix(sparse(θ_i,θ_j,θ2))
    t_new = size(se,1) - size(θ2,1)
    se2w = Matrix(sparse(θ_i, θ_j, se[t_new+1:size(se,1)]))
end
```

```
θ1 = 25×1 Array{Float64,2}:
      -32.433704933071326
       -3.473111184143974
        0.7384029600756631
       -1.3897536055318538
       -1.716966094523778
        2.3609938332345752
        0.1309691054430907
        ⋮
        0.23053716825465842
        0.7850069586586796
        1.0721552602429705
       -1.6914738842582127
       -0.4137833140631842
       -0.6818047377989064
```

```julia
θ1 = θ_1(θ2)
```

```
[0.25789,  1.99546,  0.25789,  0.012875,  0.202054]
```

```julia
begin
    Ω = inv(vcov[2:25,2:25])
    xmd = [x2[1:24,1] x2[1:24,3:4]];
    ymd = θ1[2:25];
    β = (xmd' * Ω * xmd) \ (xmd' * Ω * ymd)
    resmd = ymd - xmd * β
    semd = sqrt.(diag(inv(xmd' * Ω * xmd)))
    mcoef = [β[1]; θ1[1]; β[2:3]];
    semcoef = [semd[1]; se[1]; semd]
end
```

Rsq = 0.27877049152814415

```julia
Rsq = 1 - ((resmd .- mean(resmd))'*(resmd .- mean(resmd)))/ ((ymd .-
mean(ymd))'*(ymd .-mean(ymd)))
```

Rsq_G = 0.0944935447857097

```julia
Rsq_G = 1-(resmd'*Ω*resmd)/((ymd .- mean(ymd))'*Ω*(ymd .- mean(ymd)))
```

Chisq = 3.795133673249084e6

```julia
Chisq = size(id,1)*resmd'*Ω*resmd
```

vert = ["constant ", "price ", "sugar ", "mushy "]

```julia
vert=["constant  ";
      "price     ";
      "sugar     ";
      "mushy     "]
```

["\n\t[\"constant \", \"price   \", \"sugar   \", \"mushy   \"][i,end]\n\t[-1.8

```julia
map(1:size(θ2w,1)) do i
    content = "
    $vert[i,end]
    $mcoef[i]
    $θ2w[i,end]
    $semcoef[i]
    $se2w[i,end]
    "
    md"$content"
end
```

"GMM Objective: 14.900788598425498"

```julia
@show "GMM Objective: $fval"
```

"MR R-squared: 0.27877049152814415"

```julia
@show "MR R-squared: $Rsq"
```

"MR Weighted R-squared: 0.0944935447857097"

```
@show "MR Weighted R-squared: $Rsq_G"
```

"run time: 0.260131333 seconds"

```
@show "run time: $ttime seconds"
```