Numerical Methods in Economics: PS 1

Runnan (Ron) Yang

1 Wealth Distribution

Dependencies

`python`, `numpy`, `matplotlib`

Function Description

`wealthDistribution.py` has a function `wealthDistribution(alpha, gamma, pi0, pi1, N)`, where the inputs are `alpha` = α, `gamma` = γ, `pi0` = π_0, `pi1` = π_1, `N` = N. This function approximates the ergodic distribution by computing successive iterations of the Markov process as described in the text.

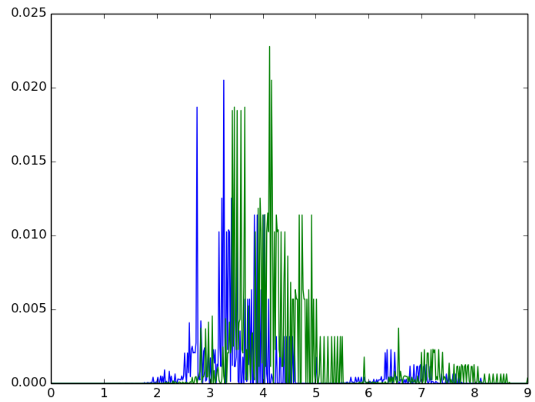
Beginning with $(w, A) = (0, 0)$, we compute the successive movements of (w, A) with the function `wealthMovement(grid, prob, alpha, gamma, pi0, pi1)` where `grid` is an `Nx2 numpy` array of N points on the range $(0, Amax)$ for $w = 0$ and $w = 1$, and `prob` is an `Nx2 numpy` array with `prob[a, w]` being the probability of being in asset level a with wage w.

Observations

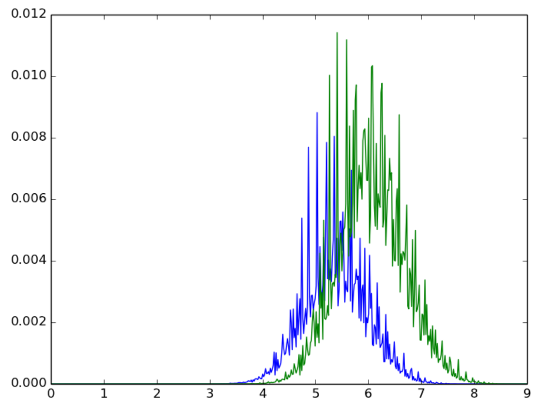
We generated random values between 0 and 1 for alpha, gamma, p0, p1, and several values between 100 and 10000 for N. We also tested using 10/100/1000 iterations; the 100 and 1000 iteration distributions tend to be similar. Images of the distributions generated are attached, in the format `alpha_gamma_p0_p1_N_iterations_plot.png`.

As described in the text, as we increase the number of iterations, the approximation rapidly improves.

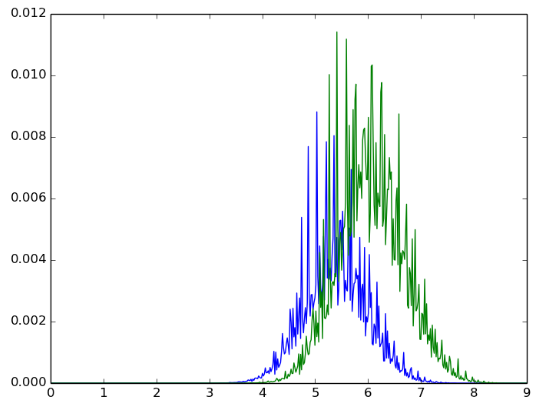
$\alpha = 0.1, \gamma = 0.1, p_0 = 0.1, p_1 = 0.5, N = 500$, 10 iterations



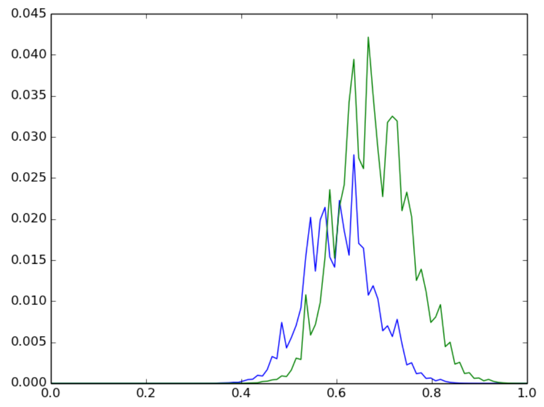
$\alpha = 0.1, \gamma = 0.1, p_0 = 0.1, p_1 = 0.5, N = 500$, 100 iterations



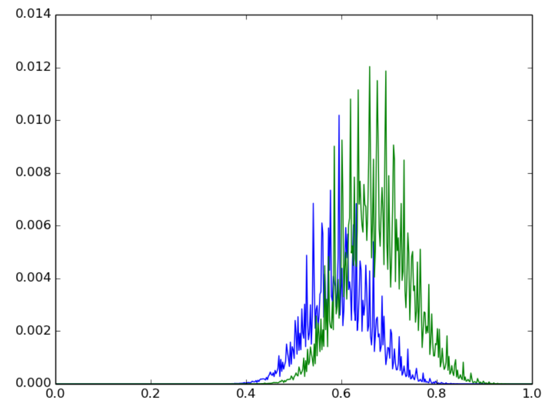
$\alpha = 0.1, \gamma = 0.1, p_0 = 0.1, p_1 = 0.5, N = 500$, 1000 iterations



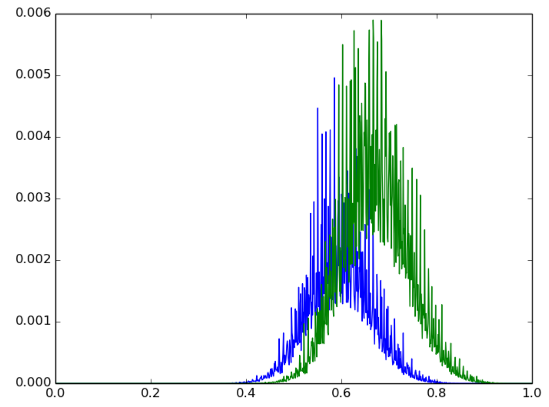
We also observe the distribution changing as N increases.
$\alpha = 0.9, \gamma = 0.1, p_0 = 0.1, p_1 = 0.5, N = 100$, 1000 iterations



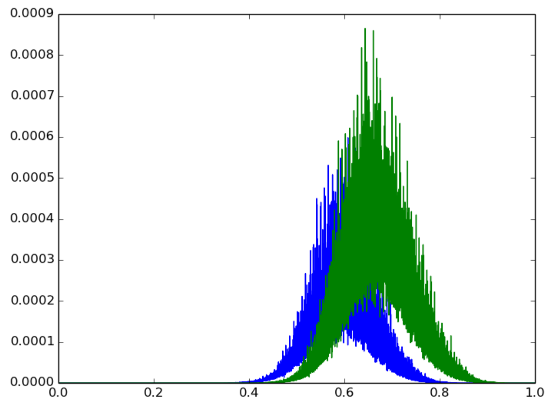
$\alpha = 0.9, \gamma = 0.1, p_0 = 0.1, p_1 = 0.5, N = 500$, 1000 iterations



$\alpha = 0.9, \gamma = 0.1, p_0 = 0.1, p_1 = 0.5, N = 1000$, 1000 iterations



$\alpha = 0.9, \gamma = 0.1, p_0 = 0.1, p_1 = 0.5, N = 10000$, 1000 iterations



2 DAgMirN selection criterion

Dependencies

`python, numpy`

Best Response Function (Gauss-Jacobi Step)

Under the class of games described, we have $\frac{\partial c_i}{\partial q_i} = k_i q_i$ (linear marginal cost, zero at zero output), $q_d(p) = d_0 - d_1 p$ (linear demand), $p(q) = \frac{d_0 - q}{d_1}$ (linear inverse demand). Then at each iteration, $\frac{\partial \Pi_i}{\partial q_i} = 0$, i.e. $\frac{\partial \Pi_i}{\partial q_i} = p + q\frac{\partial p}{\partial q_i} - k_i q_i = \frac{d_0 - \sum q}{d_1} - \frac{q_i}{d_1} - k_i q_i \implies d_0 - \sum q - q_i - k_1 q_i d_1 = 0$. Each firm's best response function to other firms producing $Q = \sum_{j \neq i} q_j$ is $q_i(Q) = \frac{d_0 - \sum_{j \neq i} q_j}{2 + k_i d_1}$. Our Gauss-Jacobi step involves solving this equation for each firm i.

Function Description

`dagmirnCriterion.py` has a function `cournotEqmSelection(cost, demand, iters = 100, tol = 1e − 10, algo = ”gj”)` where `cost` is an N-length numpy vector, `demand` is a 2-length numpy vector, `iters` specifies the maximum number of iterations, and `algo` specifies ”gj” for Gauss-Jacobi or ”gs” for Gauss-Seidel. This function can be called by an external script by importing `dagmirnCriterion` as a module and calling the function `dagmirnCriterion.cournotEqmSelection(cost, demand)`.

Observations

We report the results of some random specifications, where the costs are uniformly drawn from $[0, 10]$, d_0 is drawn from $[0, 100]$, and d_1 is drawn from $[0, 10]$:

```
2 players
Marginal Cost
[ 4.90213296  3.20632889]
Demand Parameters
[ 22.03606003   4.51805961]
Initial Guess
[ -1.00000000e+10  -1.00000000e+10]

Beginning Iterations
=================================
CONVERGED: Converged after 8 iterations
=================================

Quantity
[ 0.85934454  1.2844972 ]



3 players
Marginal Cost
[ 0.29073973  6.4574052   5.21175093]
Demand Parameters
[ 84.56483784   8.32974705]
Initial Guess
[ -1.00000000e+10  -1.00000000e+10  -1.00000000e+10]

Beginning Iterations
=================================
CONVERGED: Converged after 11 iterations
```

```
=================================

Quantity
[ 18.5396825    1.15788552   1.4283991 ]



4 players
Marginal Cost
[ 3.9065339   7.28876551  2.34488755  2.4313192 ]
Demand Parameters
[ 3.30000703  8.403995  ]
Initial Guess
[ -1.00000000e+10  -1.00000000e+10  -1.00000000e+10  -1.00000000e+10]

Beginning Iterations
=================================
CONVERGED: Converged after 11 iterations
=================================

Quantity
[ 0.08552305  0.04647495  0.13972894  0.13499344]



5 players
Marginal Cost
[ 1.1487386   5.36965384  4.06731457  5.88749088  8.58715914]
Demand Parameters
[ 69.2162079    4.56477534]
Initial Guess
[ -1.00000000e+10  -1.00000000e+10  -1.00000000e+10  -1.00000000e+10
  -1.00000000e+10]

Beginning Iterations
=================================
CONVERGED: Converged after 15 iterations
=================================

Quantity
[ 8.45451204  2.06919274  2.69787914  1.89372493  1.31317795]



6 players
Marginal Cost
[ 2.26657717  6.77082062  6.30663913  0.5049631   1.59630169  5.42129355]
Demand Parameters
[ 3.38779728  3.34861707]
Initial Guess
[ -1.00000000e+10  -1.00000000e+10  -1.00000000e+10  -1.00000000e+10
  -1.00000000e+10  -1.00000000e+10]

Beginning Iterations
```

```
=================================
CONVERGED: Converged after 34 iterations
=================================

Quantity
[ 0.22091247  0.08015989  0.08579307  0.70519008  0.29905362  0.09907236]
```

Note Since we use Gauss-Jacobi iteration rather than Gauss-Seidel iteration, we run into one problem where the function cycle between two guesses. The initial guess is each firm produces zero output, so if the sum of the firms' best responses to all other firms producing 0 is great than d_0, then in the next iteration each firm will choose to produce 0 (since firms cannot produce negative values). This returns us to the initial guess and causes the algorithm to throttle back and forth . With certain specifications where N is large or $k_i d_1$ is small, this becomes an issue (and may make a case for using Gauss-Seidel iteration or some other method).

3 Pareto efficient allocations

Dependencies

`R`, `IPOPT`, `IPTOPR`

For solving this nonlinear problem, we used the IPOPT solver with the R interface IPOPTR, testing on a Debian linux VM and acropolis.

Function Description

In `ipoptSPP.R`, we define a function `ipopt_solve(a, nu, lambda, e)`, where `a[i * nGoods + j]` $= a_{ij}$, `nu[i * nGoods + j]` $= \nu_{ij}$, `lambda[i]` $= \lambda_i$ (agent i's social weight), and `e[i * nGoods + j]` $= e_{i,j}$ (where `nGoods` is the number of goods and `nAgents` is the number of agents).

Observations

We noticed that convergence is slow when the social weights differed across agents. Meanwhile, changing nu, a, or the endowments did not significantly increase convergence time. We tested with random parameters, and were able to compute solutions for $m = n$ up through $m = n = 25$.

The following are results from a sample of runs for different parameters (I've omitted larger examples with very large matrices).

A case with random endowments (uniformly drawn from (0,1)):

```
[1] "nAgents:"
[1] 3
[1] "nGoods:"
[1] 3
[1] "a:"
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
[3,]    1    1    1
[1] "nu:"
     [,1] [,2] [,3]
[1,]   -2   -2   -2
[2,]   -2   -2   -2
[3,]   -2   -2   -2
[1] "e:"
          [,1]       [,2]      [,3]
[1,] 0.3929545 0.02189036 0.8674487
[2,] 0.4985871 0.18939153 0.9105017
[3,] 0.9029751 0.38216926 0.4662514
[1] "lambda:"
[1] 1 1 1


Ipopt solver status: 0 ( SUCCESS: Algorithm terminated successfully at a
locally optimal point, satisfying the convergence tolerances (can be specified
by options). )

Number of Iterations....: 198
```

```
Optimal value of objective function:  -4.63216978021757
Optimal value of controls: 0.5981723 0.1978171 0.7480673 0.5981722 0.1978171 0.7480673 0.5981723 0.19
0.7480673
```

A case with random a, uniformly drawn from (0,10).

```
[1] "nAgents:"
[1] 5
[1] "nGoods:"
[1] 5
[1] "a:"
         [,1]     [,2]     [,3]     [,4]     [,5]
[1,] 4.152463 7.889862 2.002005 7.746430 3.985748
[2,] 5.274914 5.863594 7.069927 4.390570 9.128934
[3,] 8.413369 9.656015 3.297319 4.335415 5.790411
[4,] 8.925589 5.936143 5.524899 6.692621 6.588603
[5,] 2.954638 1.227553 6.852863 3.686212 3.356048
[1] "nu:"
     [,1] [,2] [,3] [,4] [,5]
[1,]   -2   -2   -2   -2   -2
[2,]   -2   -2   -2   -2   -2
[3,]   -2   -2   -2   -2   -2
[4,]   -2   -2   -2   -2   -2
[5,]   -2   -2   -2   -2   -2
[1] "e:"
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1    1    1
[2,]    1    1    1    1    1
[3,]    1    1    1    1    1
[4,]    1    1    1    1    1
[5,]    1    1    1    1    1
[1] "lambda:"
[1] 1 1 1 1 1

Ipopt solver status: 0 ( SUCCESS: Algorithm terminated successfully at a
locally optimal point, satisfying the convergence tolerances (can be specified
by options). )

Number of Iterations....: 61
Optimal value of objective function:  -25.0000002499907
Optimal value of controls: 0.8532233 1.181978 0.6515566 1.213509 0.8443042 0.961651 1.018959 1.22441
0.913593 1.277774 1.214492 1.307596 0.8361805 0.9078365 1.017651 1.250916
1.025243 1.082385 1.127951 1.085527 0.7197173 0.4662242 1.205468 0.8371099
0.7747439
```

A case with a drawn from (0,1), nu drawn from (-2,0), and e drawn from (1,3)

```
[1] "nAgents:"
[1] 5
[1] "nGoods:"
[1] 5
[1] "a:"
          [,1]      [,2]      [,3]      [,4]      [,5]
```

```
[1,] 3.929545 0.2189036 8.674487 4.9858712 1.893915
[2,] 9.105017 9.0297508 3.821693 4.6625141 8.115859
[3,] 3.492380 6.7209786 8.670412 0.2722837 4.873563
[4,] 7.659178 0.3105772 3.707635 9.7326738 5.463511
[5,] 4.034273 7.4675498 1.314126 6.9956421 4.232383
[1] "nu:"
          [,1]      [,2]       [,3]       [,4]        [,5]
[1,] -1.696143 -1.401518 -0.7370320 -0.5011700 -0.66759407
[2,] -1.779224 -1.842813 -1.1789093 -1.7306918 -0.29637024
[3,] -1.345405 -1.528708 -1.8360544 -1.2786964 -0.35211142
[4,] -1.157474 -1.497777 -1.1733568 -1.7332032 -1.31758259
[5,] -1.522537 -0.873092 -0.5213084 -0.7201539 -0.06042532
[1] "e:"
         [,1]     [,2]     [,3]     [,4]     [,5]
[1,] 1.869775 1.570333 2.215124 2.560133 2.997978
[2,] 2.579706 1.997839 1.725603 2.875269 2.625977
[3,] 2.331369 1.603074 1.570924 1.929646 1.635851
[4,] 1.100651 2.823873 2.039500 1.907595 2.198084
[5,] 2.145223 1.830017 2.466477 2.282098 2.626176
[1] "lambda:"
[1] 1 1 1 1 1

Ipopt solver status: 0 ( SUCCESS: Algorithm terminated successfully at a
locally optimal point, satisfying the convergence tolerances (can be specified
by options). )

Number of Iterations....: 125
Optimal value of objective function:  -53.5082990769477
Optimal value of controls: 1.47734 0.2122414 5.09791 3.774704 0.2915649 2.326363 2.315473 1.381289
1.413274 8.446167 1.498276 2.268154 1.922434 0.1732197 1.415587 3.153269
0.2961566 1.348136 2.159803 1.196741 1.571476 4.733112 0.2678605 4.03374
0.734007
```