Comparison Fortran and Julia Methods

October 3, 2021

1 Fortran vs Julia

I this notebook I intend to compare the Fortran and Julia solvers for the PS3 model.

First we will compare both solvers in regards to solving the Value Function.

[4]:
```
using Distributed, SharedArrays, BenchmarkTools
cd("/home/mitchv34/Work/2nd Year/ECON 899 (Computational Methods)/1st Quarter/
 ↪Problem Sets/Shared Repo/Shared Repo")
```

[8]:
```
include("../JuliaCode/conesa_kueger.jl");
```

[17]:
```
# This funciton will compile and run the Fortran code
function V_fun_Fortran()
    path = "/home/mitchv34/Work/2nd Year/ECON 899 (Computational Methods)/1st␣
 ↪Quarter/Problem Sets/Shared Repo/Shared Repo/PS3/FortranCode/"
    run(`gfortran -fopenmp -O2 -o $(path)V_fun_Fortran $(path)conesa_kueger.
 ↪f90`)
    run(`$(path)V_Fortran`)
end
```

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

[20]:
```
# We can benchmark the Fortran code
@benchmark V_fun_Fortran()
```

```
BenchmarkTools.Trial: 4 samples with 1 evaluation.
 Range (min … max):  1.422 s …   1.775 s   GC (min … max): 0.00% … 0.00%
 Time  (median):     1.695 s               GC (median):    0.00%
 Time  (mean ± ):   1.647 s ± 157.521 ms  GC (mean ± ):  0.00% ± 0.00%



  1.42 s        Histogram: frequency by time        1.77 s <

 Memory estimate: 7.62 KiB, allocs estimate: 166.
```

[21]:
```
# We can also benchmark the Julia code
@benchmark begin
    prim, res = Initialize()
    V_ret(prim, res)
    V_workers(prim, res)
end
```

```
BenchmarkTools.Trial: 3 samples with 1 evaluation.
 Range (min … max):  1.825 s …   1.911 s   GC (min … max): 0.70% … 0.86%
 Time  (median):     1.861 s               GC (median):    0.69%
 Time  (mean ± σ):   1.865 s ± 43.327 ms   GC (mean ± σ):  0.75% ± 0.10%



  1.82 s         Histogram: frequency by time        1.91 s <

 Memory estimate: 312.40 MiB, allocs estimate: 44575.
```

Being fair just runing the fortran code will not help a lot, next we will use a function that reads in the data from the Fortran code in a format that is easier to work with.

[28]:
```
function V_fun_Fortran2(prim::Primitives, res::Results)
    @unpack r, w, b =res
    # PS3/FortranCode/conesa_kueger.f90
    # Compile Fortran code
    path = "/home/mitchv34/Work/2nd Year/ECON 899 (Computational Methods)/1st␣
↪Quarter/Problem Sets/Shared Repo/Shared Repo/PS3/FortranCode/"
    run(`gfortran -fopenmp -O2 -o $(path)V_Fortran $(path)conesa_kueger.f90`)
    # run(`./T_op $q $n_iter`)
    run(`$(path)V_Fortran`)

    results_raw =  readdlm("$(path)results.csv");

    val_fun = zeros(prim.nA, prim.nZ, prim.N_final)    # Initialize the value␣
↪function
    pol_fun = zeros(prim.nA, prim.nZ, prim.N_final)    # Initialize the policy␣
↪function
    pol_fun_ind = zeros(prim.nA, prim.nZ, prim.N_final + 1)    # Initialize the␣
↪policy function index
    consumption = zeros(prim.nA, prim.nZ, prim.N_final)    # Initialize the␣
↪consumption function
    l_fun = zeros(prim.nA, prim.nZ, prim.N_final)    # Initialize the labor␣
↪policy function
    for j in 1:prim.N_final
        range_a = (j-1) * 2*prim.nA + 1 : j * 2*prim.nA |> collect
        val_fun[:,:,j] = hcat(results_raw[range_a[1:prim.nA],end],␣
↪results_raw[range_a[prim.nA+1:end],end])
```

```
        pol_fun_ind[:,:,j] = hcat(results_raw[range_a[1:prim.nA],end-1],␣
↪results_raw[range_a[prim.nA+1:end],end-1])
        pol_fun[:,:,j] = hcat(results_raw[range_a[1:prim.nA],end-2],␣
↪results_raw[range_a[prim.nA+1:end],end-2])
        consumption[:,:,j]   = hcat(results_raw[range_a[1:prim.nA],end-3],␣
↪results_raw[range_a[prim.nA+1:end],end-3])
        l_fun[:,:,j] = hcat(results_raw[range_a[1:prim.nA],end-4],␣
↪results_raw[range_a[prim.nA+1:end],end-4])
    end
    A_grid_fortran = results_raw[1:prim.nA,end-5]
    res.val_fun = val_fun
    res.pol_fun = pol_fun
    res.pol_fun_ind = pol_fun_ind
    res.l_fun = l_fun
    return A_grid_fortran, consumption

end # run_Fortran()
```

```
V_fun_Fortran2 (generic function with 1 method)
```

[34]:
```
@benchmark begin
    prim, res = Initialize()
    V_fun_Fortran2(prim, res)
end
```

```
BenchmarkTools.Trial: 3 samples with 1 evaluation.
 Range (min … max):  1.882 s …   1.955 s   GC (min … max): 6.95% … 1.13%
 Time  (median):     1.917 s               GC (median):    1.15%
 Time  (mean ± ):   1.918 s ± 36.259 ms   GC (mean ± ):  2.77% ± 3.61%



  1.88 s         Histogram: frequency by time        1.95 s <

 Memory estimate: 142.82 MiB, allocs estimate: 3169629.
```

But Fortran code is actually doing more than just solving the Value Function. It also solves for the stationary distribution and the aggregate levels of capital and labor.

Les's compare the results with a Julia code that does the same.

[36]:
```
@benchmark begin
    prim, res = Initialize()

    V_ret(prim, res);
    V_workers(prim, res);

    SteadyStateDist(prim, res);
    # calculate aggregate capital and labor
    K = sum(res.F[:, :, :] .* prim.a_grid)
    L = sum(res.F[:, :, :] .* res.l_fun) # Labor supply grid
end
```

```
BenchmarkTools.Trial: 2 samples with 1 evaluation.
 Range (min … max):  2.799 s …  2.866 s   GC (min … max): 2.75% … 2.89%
 Time  (median):     2.832 s              GC (median):    2.82%
 Time  (mean ± ):   2.832 s ± 46.813 ms   GC (mean ± ):  2.82% ± 0.09%



  2.8 s          Histogram: frequency by time        2.87 s <

 Memory estimate: 1.33 GiB, allocs estimate: 1925921.
```

This means that we can use Fortran to solve the model usign Fortran just with the aggreates which will reduce the computational load of reading in the data.

[42]:
```
function solve_model_Fortran(prim, res;  ::Float64=0.7, tol::Float64=1e-2, err::
↪Float64=100.0)
    prim, res= Initialize()
    @unpack w_mkt, r_mkt, b_mkt, J_R, a_grid = prim
    path = "/home/mitchv34/Work/2nd Year/ECON 899 (Computational Methods)/1st␣
↪Quarter/Problem Sets/Shared Repo/Shared Repo/PS3/FortranCode/"

    n = 0 # loop counter
    while err > tol

        # calculate prices and payments at current K, L, and F
        res.r = r_mkt(res.K, res.L)
        res.w = w_mkt(res.K, res.L)
        res.b = b_mkt(res.L, res.w, sum(res. [J_R:end]))

        run(`gfortran -fopenmp -O2 -o $(path)V_Fortran $(path)conesa_kueger.
↪f90`)
        # run(`./T_op $q $n_iter`)
        run(`$(path)V_Fortran $0 $(res.r) $(res.w) $(res.b)`)

        K, L = readdlm("$(path)agg_results.csv");

        # calculate error
        err = maximum(abs.([res.K, res.L] - [K, L]))

        if (err > tol*10)
            # Leave   at the default
        elseif (err > tol*5) & (  <= 0.85)
```

```
              = 0.85
        elseif (err > tol*1.01) & (  <= 0.90)
              = 0.90
        elseif   <= 0.95
              = 0.95
        end

        # update guess
        res.K = (1- )*K +  *res.K
        res.L = (1- )*L +  *res.L

        n+=1

        println("$n iterations; err = $err, K = ", round(res.K, digits = 4), ",␣
 ↪L = ",
        round(res.L, digits = 4), ",   = $ ")

    end # while err > tol
    # return prim, res
end
```

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

[47]:
```
@time begin
    prim, res = Initialize()
    solve_model_Fortran(prim, res);
end
```

```
1 iterations; err = 1.853654283196522, K = 4.5561, L = 0.7164,   = 0.7
2 iterations; err = 0.628222627598225, K = 4.3676, L = 0.5981,   = 0.7
3 iterations; err = 0.7490854426911806, K = 4.1429, L = 0.5175,   = 0.7
4 iterations; err = 0.6394546435929285, K = 3.9511, L = 0.4625,   = 0.7
5 iterations; err = 0.5252353649988502, K = 3.7935, L = 0.4248,   = 0.7
6 iterations; err = 0.39666691660967324, K = 3.6745, L = 0.3991,   = 0.7
7 iterations; err = 0.29359646814189544, K = 3.5864, L = 0.3815,   = 0.7
8 iterations; err = 0.21407905080393608, K = 3.5222, L = 0.3695,   = 0.7
9 iterations; err = 0.15419735072324903, K = 3.4759, L = 0.3612,   = 0.7
10 iterations; err = 0.11656975646133416, K = 3.441, L = 0.3556,   = 0.7
11 iterations; err = 0.07834938237442968, K = 3.4292, L = 0.3537,   = 0.85
12 iterations; err = 0.06733148735338235, K = 3.4191, L = 0.352,   = 0.85
13 iterations; err = 0.05741369700356236, K = 3.4105, L = 0.3506,   = 0.85
14 iterations; err = 0.04806313110832816, K = 3.4057, L = 0.3499,   = 0.9
15 iterations; err = 0.04283516602035853, K = 3.4014, L = 0.3492,   = 0.9
16 iterations; err = 0.038778460525281666, K = 3.3975, L = 0.3486,   = 0.9
17 iterations; err = 0.03493051684980175, K = 3.394, L = 0.348,   = 0.9
18 iterations; err = 0.031047594362970443, K = 3.3909, L = 0.3475,   = 0.9
19 iterations; err = 0.02785262599422067, K = 3.3881, L = 0.3471,   = 0.9
```

```
20 iterations; err = 0.025149609438808618, K = 3.3856, L = 0.3467,   = 0.9
21 iterations; err = 0.022751006757098846, K = 3.3834, L = 0.3463,   = 0.9
22 iterations; err = 0.020527959303886956, K = 3.3813, L = 0.346,   = 0.9
23 iterations; err = 0.01845025031883818, K = 3.3795, L = 0.3457,   = 0.9
24 iterations; err = 0.01589216569091123, K = 3.3779, L = 0.3454,   = 0.9
25 iterations; err = 0.014489272438640288, K = 3.3764, L = 0.3452,   = 0.9
26 iterations; err = 0.013008477086864367, K = 3.3751, L = 0.345,   = 0.9
27 iterations; err = 0.011832110795989781, K = 3.3739, L = 0.3448,   = 0.9
28 iterations; err = 0.010596875039666998, K = 3.3729, L = 0.3447,   = 0.9
29 iterations; err = 0.00958704319094128, K = 3.3724, L = 0.3446,   = 0.95
 46.302382 seconds (14.30 k allocations: 6.037 MiB, 0.03% gc time)
```

Compared with the same code using Julia functions:

```
[45]: function MarketClearing_Julia(prim, res,  ::Float64=0.7, tol::Float64=1e-2, err:
       ↪:Float64=100.0)

          # unpack relevant variables and functions
          @unpack w_mkt, r_mkt, b_mkt, J_R, a_grid = prim

          n = 0 # loop counter

          # iteratively solve the model until excess savings converge to zero
          while err > tol

              # calculate prices and payments at current K, L, and F
              res.r = r_mkt(res.K, res.L)
              res.w = w_mkt(res.K, res.L)
              res.b = b_mkt(res.L, res.w, sum(res. [J_R:end]))

              # solve model with current model and payments

              V_ret(prim, res);
              V_workers(prim, res);

              SteadyStateDist(prim, res);

              # calculate aggregate capital and labor
              K = sum(res.F[:, :, :] .* a_grid)
              L = sum(res.F[:, :, :] .* res.l_fun) # Labor supply grid

              # calculate error
              err = maximum(abs.([res.K, res.L] - [K, L]))

              if (err > tol*10)
                  # Leave   at the default
              elseif (err > tol*5) & (  <= 0.85)
                     = 0.85
```

```
        elseif (err > tol*1.01) & (  <= 0.90)
              = 0.90
        elseif   <= 0.95
              = 0.95
        end

        # update guess
        res.K = (1- )*K +  *res.K
        res.L = (1- )*L +  *res.L

        n+=1

        println("$n iterations; err = $err, K = ", round(res.K, digits = 4), ",␣
↪L = ",
        round(res.L, digits = 4), ",   = $ ")

    end # while err > tol
    # return prim, res
end # MarketClearing
```

```
MarketClearing_Julia (generic function with 4 methods)
```

[48]:
```
@time begin
    prim, res = Initialize()
    solve_model_Fortran(prim, res);
end
```

```
1 iterations; err = 1.853654283196522, K = 4.5561, L = 0.7164,   = 0.7
2 iterations; err = 0.628222627598225, K = 4.3676, L = 0.5981,   = 0.7
3 iterations; err = 0.7490854426911806, K = 4.1429, L = 0.5175,   = 0.7
4 iterations; err = 0.6394546435929285, K = 3.9511, L = 0.4625,   = 0.7
5 iterations; err = 0.5252353649988502, K = 3.7935, L = 0.4248,   = 0.7
6 iterations; err = 0.39666691660967324, K = 3.6745, L = 0.3991,   = 0.7
7 iterations; err = 0.29359646814189544, K = 3.5864, L = 0.3815,   = 0.7
8 iterations; err = 0.21407905080393608, K = 3.5222, L = 0.3695,   = 0.7
9 iterations; err = 0.15419735072324903, K = 3.4759, L = 0.3612,   = 0.7
10 iterations; err = 0.11656975646133416, K = 3.441, L = 0.3556,   = 0.7
11 iterations; err = 0.07834938237442968, K = 3.4292, L = 0.3537,   = 0.85
12 iterations; err = 0.06733148735338235, K = 3.4191, L = 0.352,   = 0.85
13 iterations; err = 0.05741369700356236, K = 3.4105, L = 0.3506,   = 0.85
14 iterations; err = 0.04806313110832816, K = 3.4057, L = 0.3499,   = 0.9
15 iterations; err = 0.04283516602035853, K = 3.4014, L = 0.3492,   = 0.9
16 iterations; err = 0.038778460525281666, K = 3.3975, L = 0.3486,   = 0.9
17 iterations; err = 0.03493051684980175, K = 3.394, L = 0.348,   = 0.9
18 iterations; err = 0.031047594362970443, K = 3.3909, L = 0.3475,   = 0.9
19 iterations; err = 0.02785262599422067, K = 3.3881, L = 0.3471,   = 0.9
20 iterations; err = 0.025149609438808618, K = 3.3856, L = 0.3467,   = 0.9
```

```
21 iterations; err = 0.022751006757098846, K = 3.3834, L = 0.3463,   = 0.9
22 iterations; err = 0.020527959303886956, K = 3.3813, L = 0.346,   = 0.9
23 iterations; err = 0.01845025031883818, K = 3.3795, L = 0.3457,   = 0.9
24 iterations; err = 0.01589216569091123, K = 3.3779, L = 0.3454,   = 0.9
25 iterations; err = 0.014489272438640288, K = 3.3764, L = 0.3452,   = 0.9
26 iterations; err = 0.013008477086864367, K = 3.3751, L = 0.345,   = 0.9
27 iterations; err = 0.011832110795989781, K = 3.3739, L = 0.3448,   = 0.9
28 iterations; err = 0.010596875039666998, K = 3.3729, L = 0.3447,   = 0.9
29 iterations; err = 0.00958704319094128, K = 3.3724, L = 0.3446,   = 0.95
 46.064710 seconds (14.26 k allocations: 6.034 MiB)
```

We can see that both converge in the same number of iterations to the same solution at virtually the same speed.

Finally we will use Fortran to solve the model completely.

[52]:
```julia
function Fortran_sove_the_whole_thing()
    path = "/home/mitchv34/Work/2nd Year/ECON 899 (Computational Methods)/1st␣
↪Quarter/Problem Sets/Shared Repo/Shared Repo/PS3/FortranCode/"
    run(`gfortran -fopenmp -O2 -o $(path)V_Fortran $(path)conesa_kueger.f90`)
    # run(`./T_op $q $n_iter`)
    run(`$(path)V_Fortran $1 $4 $0.9`)

    results_raw =  readdlm("$(path)results.csv");

    val_fun = zeros(prim.nA, prim.nZ, prim.N_final)     # Initialize the value␣
↪function
    pol_fun = zeros(prim.nA, prim.nZ, prim.N_final)     # Initialize the policy␣
↪function
    pol_fun_ind = zeros(prim.nA, prim.nZ, prim.N_final + 1)     # Initialize the␣
↪policy function index
    consumption = zeros(prim.nA, prim.nZ, prim.N_final)     # Initialize the␣
↪consumption function
    l_fun = zeros(prim.nA, prim.nZ, prim.N_final)     # Initialize the labor␣
↪policy function
    for j in 1:prim.N_final
        range_a = (j-1) * 2*prim.nA + 1 : j * 2*prim.nA |> collect
        val_fun[:,:,j] = hcat(results_raw[range_a[1:prim.nA],end],␣
↪results_raw[range_a[prim.nA+1:end],end])
        pol_fun_ind[:,:,j] = hcat(results_raw[range_a[1:prim.nA],end-1],␣
↪results_raw[range_a[prim.nA+1:end],end-1])
        pol_fun[:,:,j] = hcat(results_raw[range_a[1:prim.nA],end-2],␣
↪results_raw[range_a[prim.nA+1:end],end-2])
        consumption[:,:,j]    = hcat(results_raw[range_a[1:prim.nA],end-3],␣
↪results_raw[range_a[prim.nA+1:end],end-3])
        l_fun[:,:,j] = hcat(results_raw[range_a[1:prim.nA],end-4],␣
↪results_raw[range_a[prim.nA+1:end],end-4])
    end
    A_grid_fortran = results_raw[1:prim.nA,end-5]
    res.val_fun = val_fun
    res.pol_fun = pol_fun
    res.pol_fun_ind = pol_fun_ind
    res.l_fun = l_fun

    return nothing
end
```

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

[54]:
```
@time begin
    Fortran_sove_the_whole_thing();
end
```

```
 Iteration:           0 ERR=   1.9040204465183033      K=   4.2856030215823608
L=  0.80821288785858592      LAMBDA=  0.85000002384185791
 Iteration:           1 ERR=   1.0099861783401369      K=   4.4371009242534347
L=  0.73361572785500595      LAMBDA=  0.85000002384185791
 Iteration:           2 ERR=  0.42798928423478255      K=   4.5012993066845919
L=  0.67234078960674060      LAMBDA=  0.85000002384185791
 Iteration:           3 ERR=  0.33920737965243147      K=   4.5304648847239344
L=  0.62145969074621010      LAMBDA=  0.85000002384185791
 Iteration:           4 ERR=  0.28289431902224044      K=   4.5308357670169617
L=  0.57902554963760022      LAMBDA=  0.85000002384185791
 Iteration:           5 ERR=  0.23569706986203343      K=   4.5174741018390527
L=  0.54367099477775127      LAMBDA=  0.85000002384185791
 Iteration:           6 ERR=  0.19692534421772662      K=   4.4948240412565621
L=  0.51413219784015840      LAMBDA=  0.85000002384185791
 Iteration:           7 ERR=  0.18453722564133113      K=   4.4671434618100729
L=  0.48943985373748833      LAMBDA=  0.85000002384185791
 Iteration:           8 ERR=  0.19222130238274460      K=   4.4383102710355740
L=  0.46877368147745607      LAMBDA=  0.85000002384185791
 Iteration:           9 ERR=  0.20022151062164184      K=   4.4082770492159806
L=  0.45144976616503224      LAMBDA=  0.85000002384185791
 Iteration:          10 ERR=  0.18864153332136269      K=   4.3799808237153410
L=  0.43695496521219507      LAMBDA=  0.85000002384185791
 Iteration:          11 ERR=  0.18507281545369825      K=   4.3522199058097657
L=  0.42479761836536156      LAMBDA=  0.85000002384185791
 Iteration:          12 ERR=  0.15743349619959623      K=   4.3286048851333332
L=  0.41461844342526361      LAMBDA=  0.85000002384185791
 Iteration:          13 ERR=  0.14849255370031145      K=   4.3063310056186248
L=  0.40607666746699228      LAMBDA=  0.85000002384185791
 Iteration:          14 ERR=  0.12492295332510128      K=   4.2875925655982554
L=  0.39893407149173404      LAMBDA=  0.85000002384185791
 Iteration:          15 ERR=  0.11476719629930709      K=   4.2703774888896220
L=  0.39294224267973626      LAMBDA=  0.85000002384185791
 Iteration:          16 ERR=  0.10370603783454690      K=   4.2548215856869849
```

```
L=  0.38791750986416057      LAMBDA=  0.85000002384185791
 Iteration:           17 ERR=   8.4965291911740515E-002 K=   4.2420767939259543
L=  0.38372762476537414      LAMBDA=  0.85000002384185791
 Iteration:           18 ERR=   7.7219590550297035E-002 K=   4.2304938571844684
L=  0.38020776577883691      LAMBDA=  0.85000002384185791
 Iteration:           19 ERR=   5.7853087478218157E-002 K=   4.2218158954420613
L=  0.37727664818335488      LAMBDA=  0.85000002384185791
 Iteration:           20 ERR=   5.1686047433292970E-002 K=   4.2140629895593591
L=  0.37481734600466499      LAMBDA=  0.85000002384185791
 Iteration:           21 ERR=   4.2919769740304758E-002 K=   4.2097710115620419
L=  0.37345499680719380      LAMBDA=  0.89999997615814209
 Iteration:           22 ERR=   3.8275399284274592E-002 K=   4.2059434707210581
L=  0.37224060968198430      LAMBDA=  0.89999997615814209
 Iteration:           23 ERR=   3.4616022613914055E-002 K=   4.2024818676343560
L=  0.37115775932448880      LAMBDA=  0.89999997615814209
 Iteration:           24 ERR=   2.8031696877328649E-002 K=   4.1996786972782951
L=  0.37020631277651050      LAMBDA=  0.89999997615814209
 Iteration:           25 ERR=   2.5190848753730677E-002 K=   4.1971596118023253
L=  0.36935816934804000      LAMBDA=  0.89999997615814209
 Iteration:           26 ERR=   2.2183401568691252E-002 K=   4.1949412711165630
L=  0.36860219259034466      LAMBDA=  0.89999997615814209
 Iteration:           27 ERR=   1.9412719170563086E-002 K=   4.1929999987366715
L=  0.36792770016824189      LAMBDA=  0.89999997615814209
 Iteration:           28 ERR=   1.7795482906533877E-002 K=   4.1912204500217403
L=  0.36732539295246452      LAMBDA=  0.89999997615814209
 Iteration:           29 ERR=   1.6398700802177757E-002 K=   4.1895805795505474
L=  0.36678758420362323      LAMBDA=  0.89999997615814209
 Iteration:           30 ERR=   5.2032612827401792E-003 K=   4.1893204164243825
L=  0.36655609161406699      LAMBDA=  0.94999998807907104
 Total elapsed time =    7.99200010      seconds
  9.812714 seconds (3.18 M allocations: 141.008 MiB, 0.09% gc time)

1000×2×66 Array{Float64, 3}:
[:, :, 1] =
 1.07313  0.125839
 1.07263  0.0935282
 1.07213  0.0930274
 1.07163  0.0925267
 1.07113  0.0920259
 1.07063  0.0597153
 1.03832  0.0592145
 1.03782  0.0587138
 1.03732  0.058213
 1.03681  0.0577122

 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      0.0
 0.0      0.0
 0.0      0.0

[:, :, 2] =
 1.15486  0.134159
 1.15436  0.101848
 1.15386  0.101347
 1.15336  0.100847
 1.15286  0.100346
 1.15236  0.0680352
 1.15186  0.0675344
 1.11955  0.0670337
 1.11904  0.0665329
 1.11854  0.0660322

 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      0.0
 0.0      0.0
 0.0      0.0

[:, :, 3] =
 1.23659  0.142479
 1.23609  0.110168
 1.23559  0.109667
 1.23509  0.109167
 1.23459  0.108666
 1.23409  0.0763551
 1.23359  0.0758544
 1.23308  0.0753536
 1.23258  0.0748528
 1.23208  0.0743521

 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      0.0
 0.0       0.0
 0.0       0.0

…

[:, :, 64] =
 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  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
 0.0  0.0
 0.0  0.0
 0.0  0.0
 0.0  0.0
 0.0  0.0
 0.0  0.0

[:, :, 65] =
 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  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
 0.0  0.0
 0.0  0.0
 0.0  0.0
 0.0  0.0
 0.0  0.0
 0.0  0.0
```

```
[:, :, 66] =
 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  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
 0.0  0.0
 0.0  0.0
 0.0  0.0
 0.0  0.0
 0.0  0.0
 0.0  0.0
```

Clearly we only get significant speedup from using Fortran to run the whole model.