

solve-transition

April 8, 2019

```
In [1]: using Pkg
        pkg"activate ."
        pkg"instantiate"
        pkg"precompile"

Updating registry at `C:\Users\Chiyoungh Ahn\.julia\registries\General`
Updating git-repo `https://github.com/JuliaRegistries/General.git`
Precompiling project...
```

```
In [ ]: using PerlaTonettiWaugh, Plots, BenchmarkTools, CSV
```

```
In [3]: gr(fmt = :png)
```

```
Out[3]: Plots.GRBackend()
```

1 Steady states

1.1 Solving steady state solutions

Set up parameters and find the corresponding stationary solution:

Below are the results from the **updated** matlab calibration with the corresponding letters. Here I added the μ parameter in the calibration routine to fit the firm dynamic moments. Fit improved a lot and it generated a negative drift term which is what we need to keep S from going negative.

```
In [4]: # NEW CALIBRATION
```

```
parameters = parameter_defaults()
```

```
parameters = merge(parameters, ( = 0.0215, d = 3.0426 , = 5.0018 , = 0.0732, = 1/
```

```
Out[4]: ( = 0.0215, = 3.1725, N = 10, = 5.0018, = 1.0, = 0.0732, = 1.0, = 0.0, Theta = 1
```

```
In [5]: parameters.
```

```
Out[5]: 0.02
```

```
In [6]: changed_parameters = false # set this to true if you drastically change the parameters
```

Out[6]: false

Side note on previous version. Essentially, under this calibration, the parameters are very similar. Theta, chi, kappa are quite close to what we had in the prior version.

```
In [7]: # Define common objects.
        #parameters = parameter_defaults()

        settings = settings_defaults()
        settings = merge(settings, (transition_penalty_coefficient = 1.0, ))

        settings = merge(settings, (z_ex = unique([range(0., 0.1, length = 150)' range(0.1, 1.
        settings = merge(settings, (z = settings.z_ex[2:end-1], ))

        P = length(settings.z)

        d_0 = parameters.d # Here is the 10 percent tariff increase
        d_T = 1 + (parameters.d-1)*0.90

        d_autarky = 1 + (parameters.d-1)*2.5

        params_0 = merge(parameters, (d = d_0, )) # parameters to be used at t = 0
        params_T = merge(parameters, (d = d_T, )) # parameters to be used at t = T
        params_autarky = merge(parameters, (d = d_autarky, )) # parameters to be used in autarky

        # initial value for numerical solver on (g, z_hat, Omega)
        initial_x = [0.02; 2; .57]

        # solve for stationary solution at t = 0
        stationary_sol_0 = stationary_algebraic(params_0, initial_x) # solution at t = 0
        stationary_sol = stationary_algebraic(params_T, initial_x) # solution at t = T
        stationary_autarky = stationary_algebraic(params_autarky, initial_x) # solution at t =

        _0 = stationary_sol_0.;
        _T = stationary_sol.;
```

In [8]: d_0

Out[8]: 3.0426

```
In [9]: print(stationary_sol.U_bar, '\n')
        print(stationary_sol_0.U_bar, '\n')
        print(stationary_sol_0.S, '\n')
        print(stationary_autarky.S, '\n')
```

```
19.536172711003402
16.106813814683527
0.0989517579364377
```

0.07742571652180796

```
In [10]: #T = solved.t[end]
        lambda_ss = 100*(consumption_equivalent(stationary_sol.U_bar, stationary_sol_0.U_bar,

        print("SS to SS welfare gain: ", lambda_ss, "\n")
        print("SS to SS welfare gain: ", stationary_sol._ii, "\n")
```

SS to SS welfare gain: 7.6517416240797775

SS to SS welfare gain: 0.8574023485038085

```
In [11]: print("SS to SS welfare gain: ", stationary_sol.g, "\n")
```

SS to SS welfare gain: 0.008570416914745884

1.1.1 This is the autarky calculation

```
In [12]: #T = solved.t[end]
        lambda_ss_autarky = 100*(consumption_equivalent(stationary_autarky.U_bar, stationary_

        print("SS to SS welfare gain: ", lambda_ss_autarky, "\n")
        print("Autarky Home Share, should be close to 1: ", stationary_autarky._ii, "\n")
        print("Autarky TFP growth rate ", stationary_autarky.g, "\n")
```

SS to SS welfare gain: -15.522308452599521

Autarky Home Share, should be close to 1: 0.9965183434231

Autarky TFP growth rate 0.00243807896292438

1.2 Welfare in Steady States

1.2.1 Steady state at T

```
In [13]: stationary_sol.U_bar
```

```
Out[13]: 19.536172711003402
```

1.2.2 Steady state at 0

```
In [14]: stationary_sol_0.U_bar
```

```
Out[14]: 16.106813814683527
```

1.2.3 Outstanding Issue #1: Sensitivity of growth to trade.

This is a big difference relative to previous version. As noted above, with parameter values that are quite similar to what we had before, the growth rate is changing a lot with only a very small change in trade flows. Why?

```
In [15]: @show stationary_sol.g, stationary_sol_0.g;

(stationary_sol.g, stationary_sol_0.g) = (0.008570416914745884, 0.0067417379286226755)
```

2 Transition dynamics

Setup for optimizer:

```
In [16]: settings = merge(settings, (params_T = params_T, stationary_sol_T = stationary_numeri
```

Use the solution found with calibrated parameters above for E

```
In [ ]: settings = merge(settings, (transition_x0 = [ -0.8614905788913748, -0.4492651749115799,
           weights = [10.0; ones(7)])); # remove tstops when solving the model for the fi

           settings = merge(settings, (transition_lb = settings.transition_x0 .- 1e-4,
           transition_ub = settings.transition_x0 .+ 1e-4));
```

Find the corresponding solution

```
In [18]: @time result = solve_full_model(merge(settings, (transition_iterations = (changed_par
           impose_E_monotonicity_constraints = true, write_data = false,
           run_global = true)

           solved = result.solution;
           E_nodes = result.E_nodes;
           solved = solved.results;
```

216.478492 seconds (1.73 G allocations: 129.947 GiB, 8.19% gc time)

```
In [19]: E_nodes
```

```
Out[19]: 8-element Array{Float64,1}:
 -0.8614905788913748
 -0.44926517491157925
 -0.283795655883632
 -0.0651727069993808
 -0.050745349310809824
 -0.01213734877329569
 -0.005088513022818065
 -0.004779208323042127
```

2.1 Welfare Gains

```
In [20]: print("Utility in initial SS: ", stationary_sol_0.U_bar, "\n")
        print("Utility in new SS: ", stationary_sol.U_bar, "\n")
```

```
Utility in initial SS: 16.106813814683527
```

```
Utility in new SS: 19.536172711003402
```

```
In [21]: print("Utility immediately after change ", solved.U[1], "\n")
```

```
Utility immediately after change 20.125091306477238
```

Summary so far... In the old paper, what we did was take $U_{0,ss}$ at some date t , then compare it to U_{ss} at the same date t . This is like an instantaneous jump to the new ss . This is what the first cell is looking at and note that this is like a 17 percent increase in utility. Higher than what we had in the paper, but in of the same order of magnitude.

The next cell reprots the utility just after the change. Utility here includes the future path of consumption and change in growth rate, so it "bakes in" the transition path. Here it goes up by much more than utility in the new SS . This is what I was expecting given the dynamics of consumption.

```
In [22]: solved.r[100] + 0.05
```

```
stationary_sol_0.r
```

```
Out [22]: 0.04824173792862267
```

Just a reminder about how the function `counsumption_equivalent(U_{new} , U_{old} , parameters)` works, it takes U_{new} and then U_{old} in that order, then evaluates the **gross** increase in consumption. 100 times this value **minus one** gives the permanent, percent increase in consumption required to make the agent indifferent between the two paths.

```
In [23]: #T = solved.t[end]
        lambda_ss = 100*(consumption_equivalent(stationary_sol.U_bar, stationary_sol_0.U_bar,
        print("SS to SS welfare gain: ", lambda_ss, "\n")
```

```
SS to SS welfare gain: 7.6517416240797775
```

```
In [24]: lambda_tpath = 100*(consumption_equivalent(solved.U[1], stationary_sol_0.U_bar, param
        print("Inclusive of the Transition Path: ", lambda_tpath, "\n")
```

```
Inclusive of the Transition Path: 9.023466937515078
```

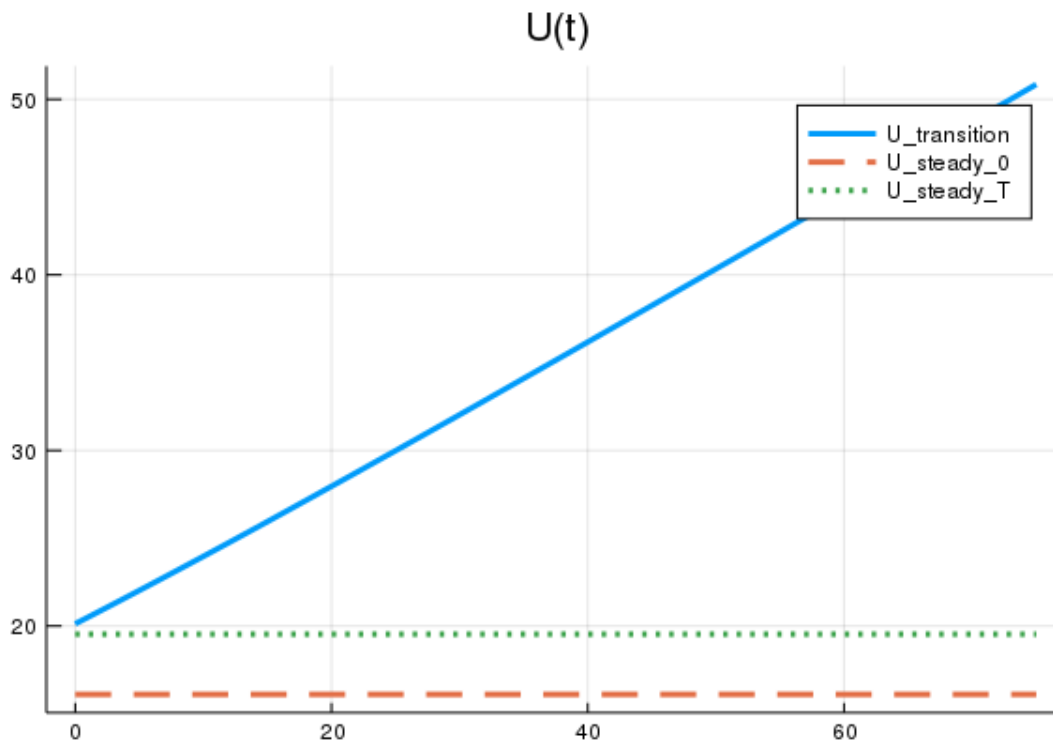
2.1.1 Relative to initial notebook computations

- SS to SS, the issue here is that we should compare **at date 0** utility in the first ss vs. the second ss. This is what we must have been doing in the previous version of the paper. The previous calculation in the old notebook had `consumption_equivalent(solved.U[end], stationary_sol_0.U_bar, parameters)` which took **date T** utility and compared them. The problem is that this now depends on date T. So if we picked T to be arbitrarily large, then utility will be arbitrarily different.
- Following the same logic, the transition path should compare **date 0** utility with the initial value from the transition path. So what we want to do is to compare everything at 0. In the previous calculation, we had `consumption_equivalent(solved.U[1], stationary_sol_0.U_bar, parameters)` were comparing the initial utility relative to ss utility on the old path at date T. So the initial blue point below verses the last orange dashed point. **See the figure below**

```
In [25]: # generate the plot!
U_steady_0(t) = stationary_sol_0.U_bar
U_steady_T(t) = stationary_sol.U_bar

plot(solved.t,
     [solved.U, U_steady_0, U_steady_T],
     label = ["U_transition", "U_steady_0", "U_steady_T"] ,
     title = "U(t)", linestyle = :auto, lw = 3)
```

Out[25]:



2.1.2 Outstanding Issue #2: Welfare Gains still depend on T in transition path.

The stuff above I think is correct, the one issue is why does the welfare gains, inclusive of the transition path seem to depend on T??? So change T above from 40 to 75 or 100, then the welfare gains fall alot? Why?

2.2 Plotting

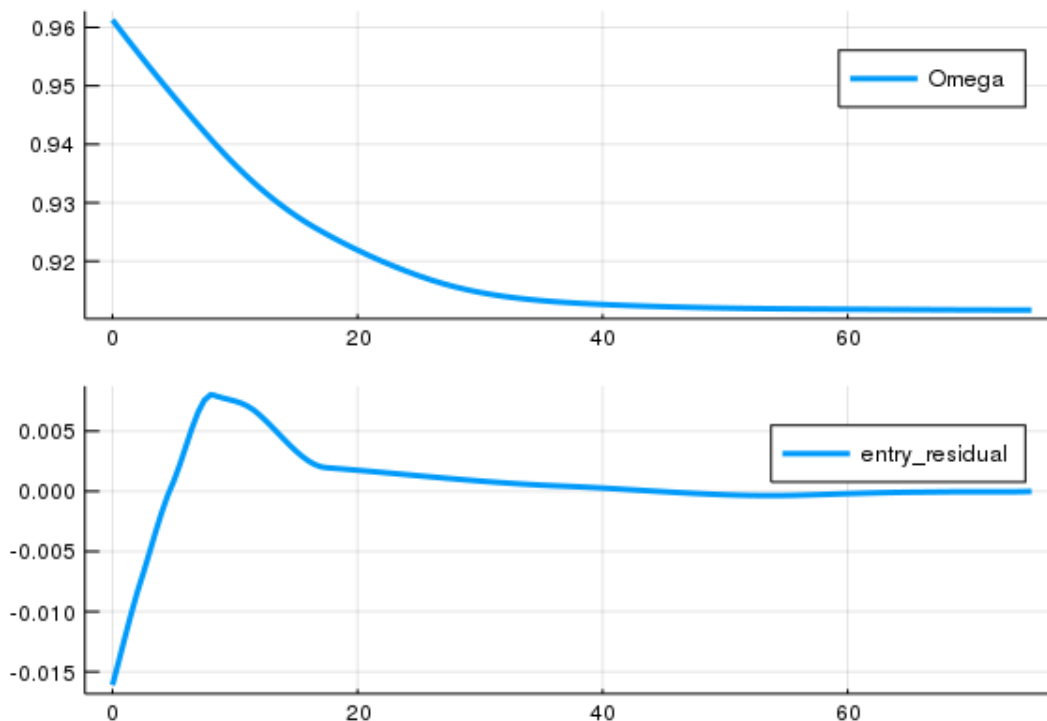
```
In [26]: solved.U[end]
```

```
Out [26]: 50.848692012905175
```

2.3 Plots for and residuals

```
In [27]: #solved = solved.results;
plot_ = plot(solved.t, solved., label = "Omega", lw = 3)
plot_residual = plot(solved.t, solved.entry_residual, label = "entry_residual", lw = 3)
plot(plot_, plot_residual, layout = (2,1))
```

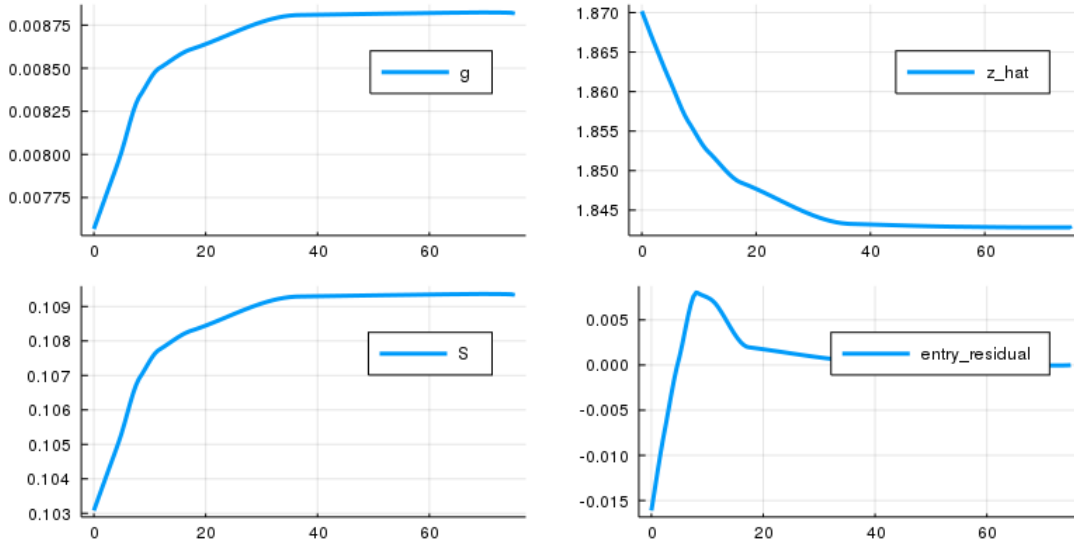
```
Out [27]:
```



2.4 Primary Plots

```
In [28]: plot1 = plot(solved.t, solved.g, label = "g", lw = 3)
plot2 = plot(solved.t, solved.z_hat, label = "z_hat", lw = 3)
plot3 = plot(solved.t, solved.S, label = "S", lw = 3)
plot4 = plot(solved.t, solved.entry_residual, label = "entry_residual", lw = 3)
plot(plot1, plot2, plot3, plot4, layout=(2,2), size = (800, 400))
```

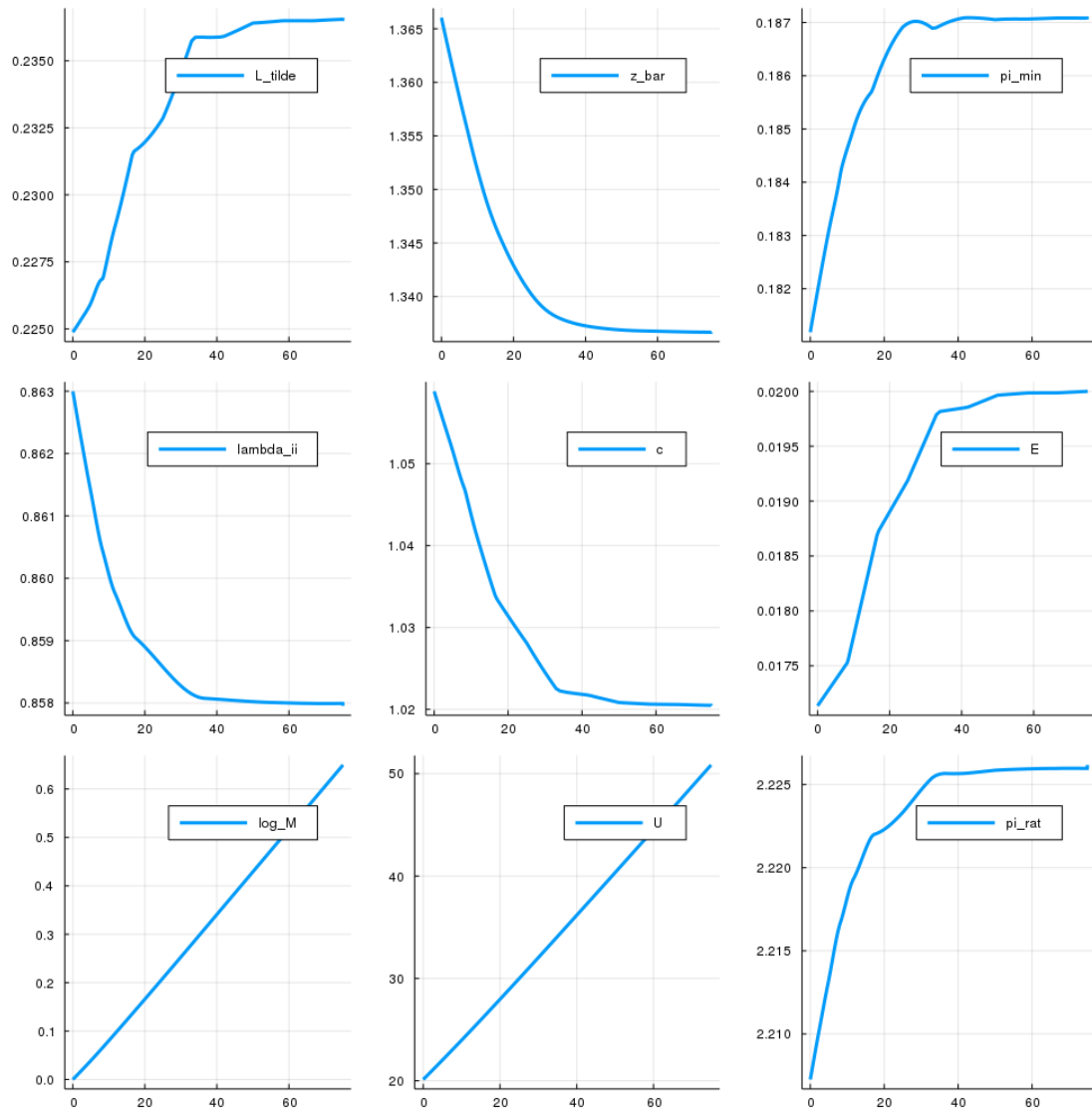
Out [28] :



2.5 Static Equations

```
In [49]: plot1 = plot(solved.t, solved.L_tilde, label = "L_tilde", lw = 3)
plot2 = plot(solved.t, solved.z_bar, label = "z_bar", lw = 3)
plot3 = plot(solved.t, solved._min, label = "pi_min", lw = 3)
plot4 = plot(solved.t, solved._ii, label = "lambda_ii", lw = 3)
plot5 = plot(solved.t, solved.c, label = "c", lw = 3)
plot6 = plot(solved.t, solved.E, label = "E", lw = 3)
plot7 = plot(solved.t, solved.log_M, label = "log_M", lw = 3)
plot8 = plot(solved.t, solved.U, label = "U", lw = 3)
plot9 = plot(solved.t, solved._rat, label = "pi_rat", lw = 3)
plot(plot1, plot2, plot3, plot4, plot5, plot6, plot7, plot8, plot9, layout=(3,3), size = (1000, 1000))
```

Out [49] :



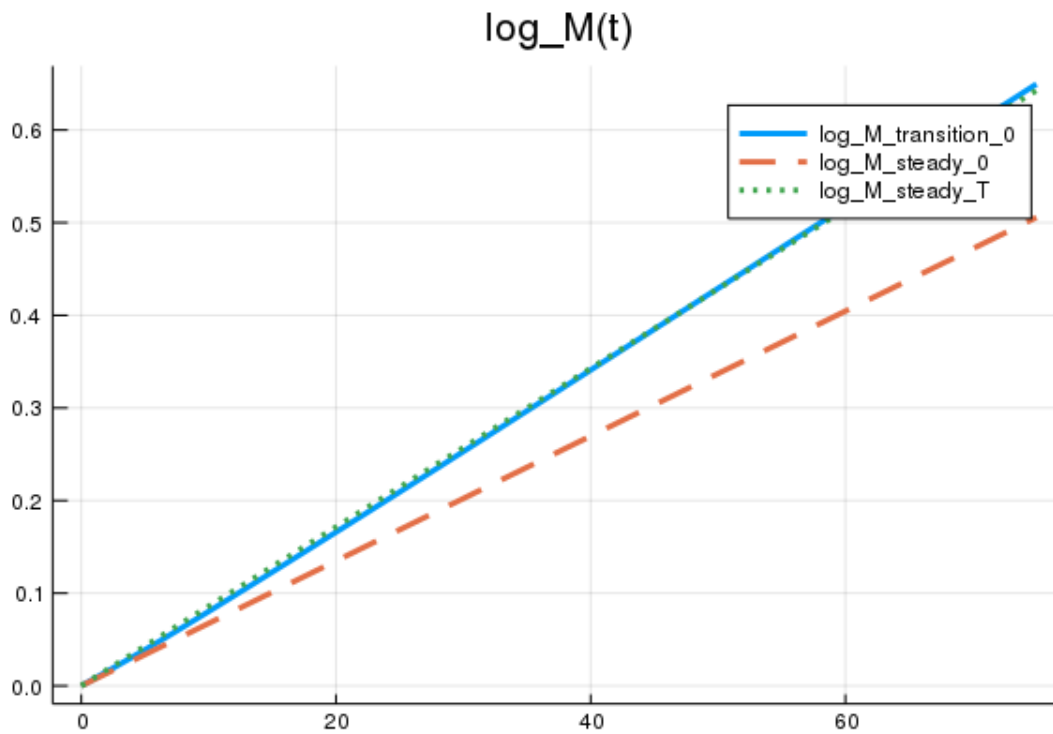
3 Welfare analysis

3.0.1 log_M(t)

```
In [30]: # define log_M with steady state g
log_M_steady_0(t) = stationary_sol_0.g * t
log_M_steady_T(t) = stationary_sol.g * t

# generate the plot!
plot(solved.t,
     [solved.log_M, log_M_steady_0, log_M_steady_T],
     label = ["log_M_transition_0", "log_M_steady_0", "log_M_steady_T"] ,
     title = "log_M(t)", linestyle = :auto, lw = 3)
```

Out [30] :

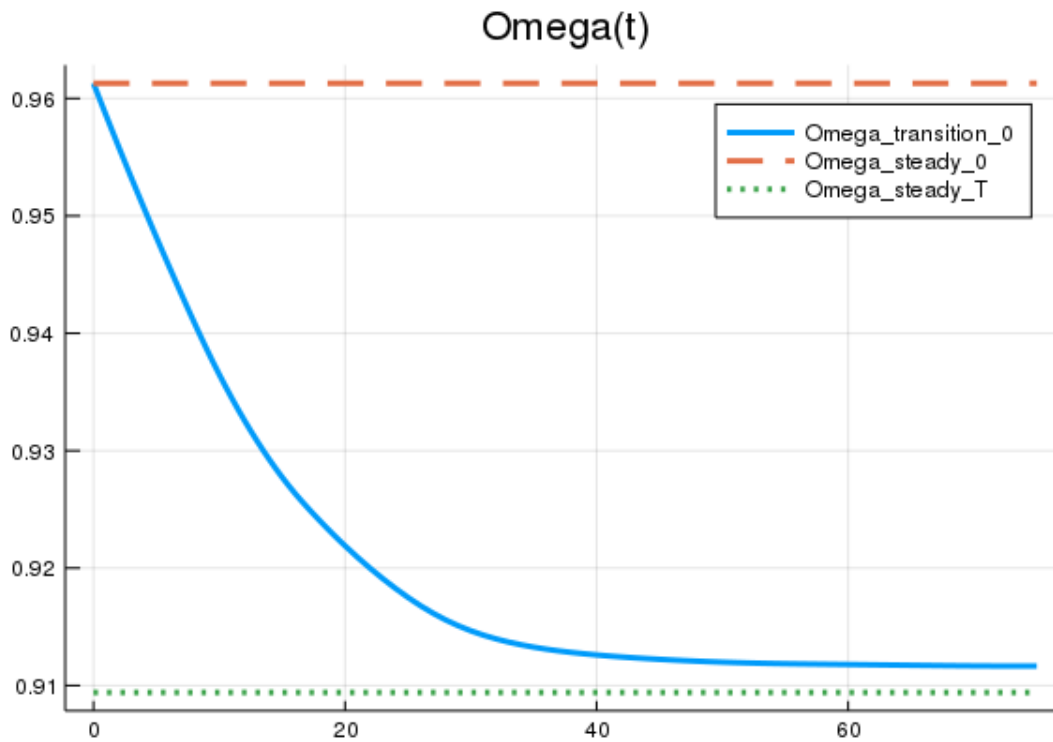


3.0.2 (t)

```
In [31]: # define function (constant)
         _steady_0(t) = stationary_sol_0.
         _steady_T(t) = stationary_sol.

         # generate the plot!
         plot(solved.t,
              [solved., _steady_0, _steady_T],
              label = ["Omega_transition_0", "Omega_steady_0", "Omega_steady_T"] ,
              title = "Omega(t)", linestyle = :auto, lw = 3)
```

Out [31] :

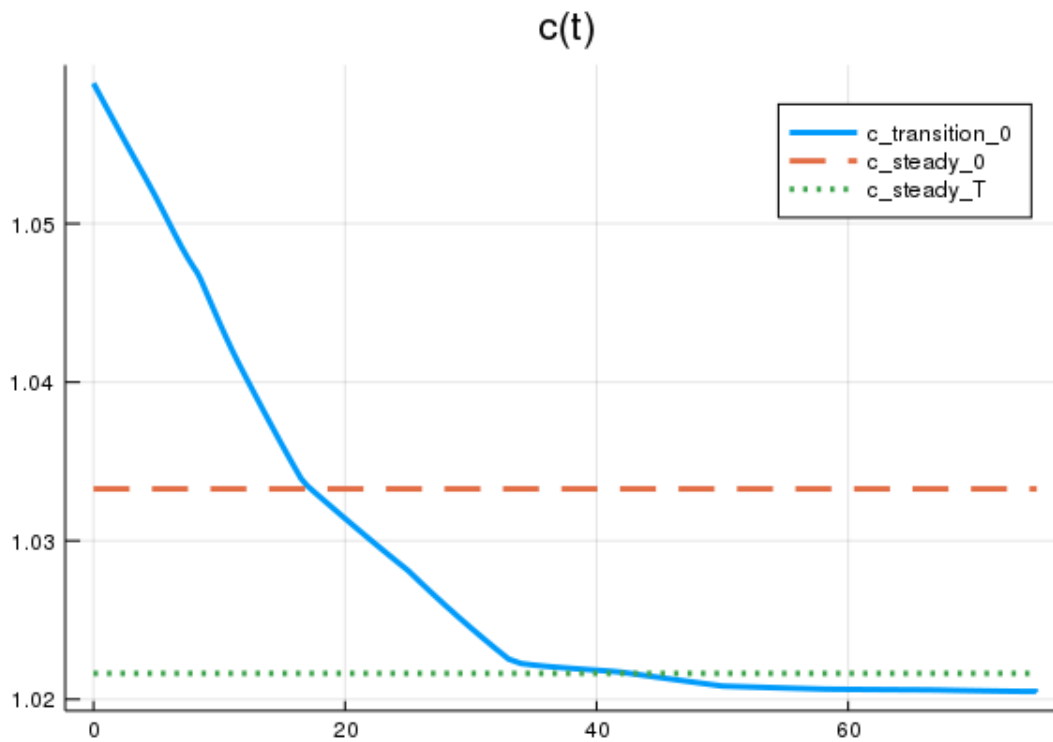


3.0.3 $c(t)$

```
In [32]: # define c function (constant)
c_steady_0(t) = stationary_sol_0.c
c_steady_T(t) = stationary_sol.c

# generate the plot!
plot(solved.t,
     [solved.c, c_steady_0, c_steady_T],
     label = ["c_transition_0", "c_steady_0", "c_steady_T"] ,
     title = "c(t)", linestyle = :auto, lw = 3)
```

Out[32]:



3.0.4 $U(t)$

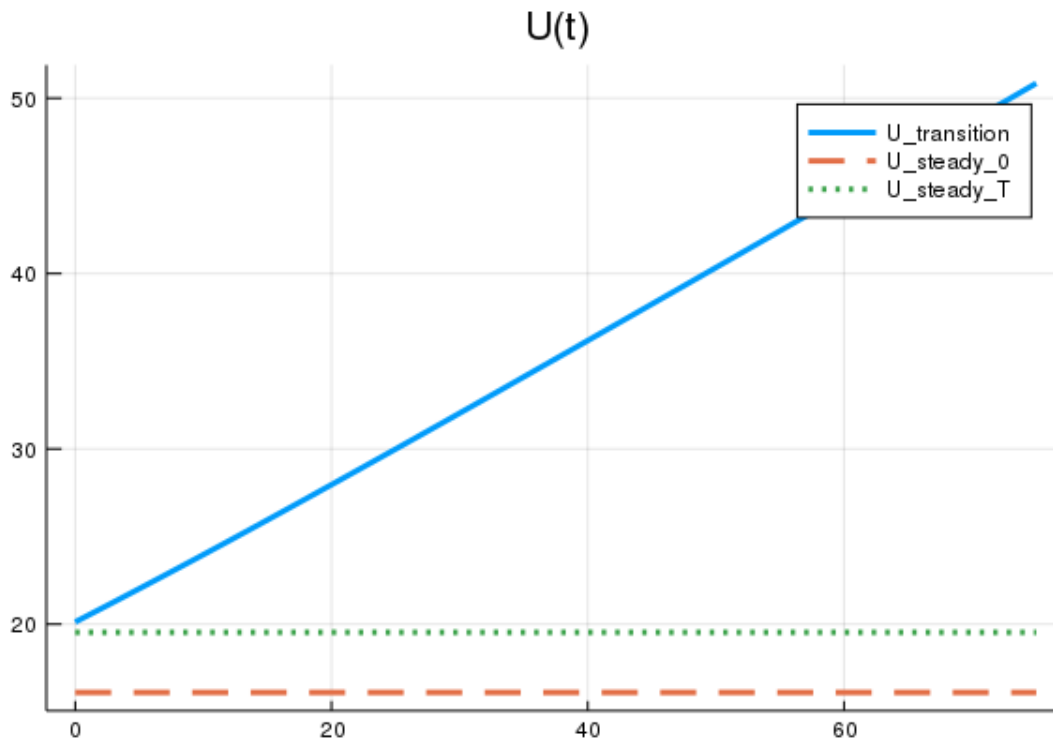
In [33]: *# generate the plot!*

```
U_steady_0(t) = stationary_sol_0.U_bar
```

```
U_steady_T(t) = stationary_sol.U_bar
```

```
plot(solved.t,
     [solved.U, U_steady_0, U_steady_T],
     label = ["U_transition", "U_steady_0", "U_steady_T"] ,
     title = "U(t)", linestyle = :auto, lw = 3)
```

Out[33]:



3.1 Consumption equivalent for search threshold ($M(0)$)

See computational appendix for details.

3.1.1 $M(0)$ by two steady states (autarky and steady state at T)

```
In [34]: T = solved.t[end]
          consumption_equivalent(solved.U[end], stationary_sol_0.U_bar, parameters)
```

```
Out [34]: 2.110553807928471
```

3.1.2 $M(0)$ by autarky and transition from $t=0$

```
In [35]: T = solved.t[end]
          consumption_equivalent(solved.U[1], stationary_sol_0.U_bar, parameters)
```

```
Out [35]: 1.0902346693751508
```

But if we include the transition path, this falls a lot. Like down to 13 percent gain.

```
In [36]: # solved
          # Run the above to see the whole dataframe
```

```
In [37]: using DataFrames
```

```
df_stationary = DataFrame(t = -1.00, g =stationary_sol_0.g, _ii = stationary_sol_0._ii,  
    _rat = stationary_sol_0._rat, L_tilde_a = stationary_sol_0.L_tilde_a, L_tilde_x =  
    L_tilde_E= stationary_sol_0.L_tilde_E, r = stationary_sol_0.r);
```

```
In [38]: CSV.write("stationary_results.csv", df_stationary)
```

```
Out[38]: "stationary_results.csv"
```

```
In [39]: df_transition = DataFrame(t = solved.t, g =solved.g, _ii = solved._ii, c = solved.c, l  
    _rat = solved._rat, L_tilde_a = solved.L_tilde_a, L_tilde_x = solved.L_tilde_x,  
    L_tilde_E= solved.L_tilde_E, r = solved.r .+ parameters.);
```

```
In [40]: CSV.write("transition_results.csv", df_transition)
```

```
Out[40]: "transition_results.csv"
```

```
In [41]: stationary_sol_0._rat
```

```
Out[41]: 1.8110075391436142
```

```
In [42]: df_welfare = DataFrame(steady_state = lambda_ss, transition_path = lambda_tpath, growt  
    CSV.write("welfare_results.csv", df_welfare)
```

```
Out[42]: "welfare_results.csv"
```

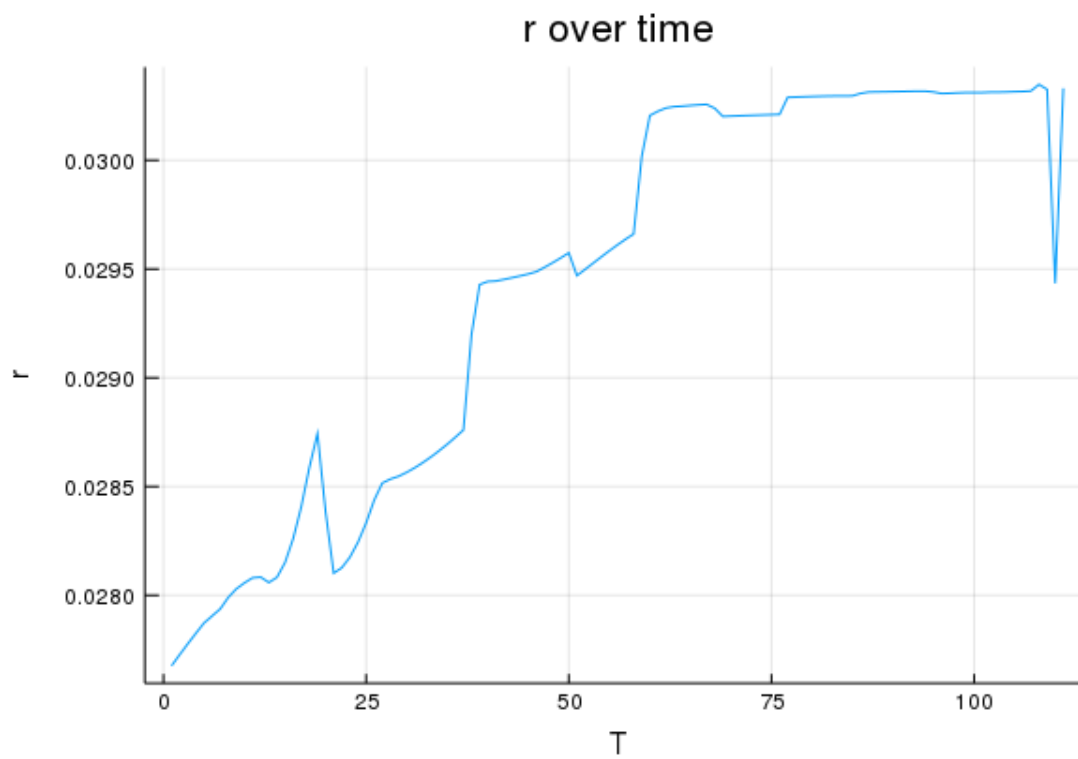
```
In [43]: df_autarky = DataFrame(steady_state = lambda_ss_autarky, growth_rate = stationary_aut  
    CSV.write("autarky_welfare_results.csv", df_autarky)
```

```
Out[43]: "autarky_welfare_results.csv"
```

3.1.3 R and W

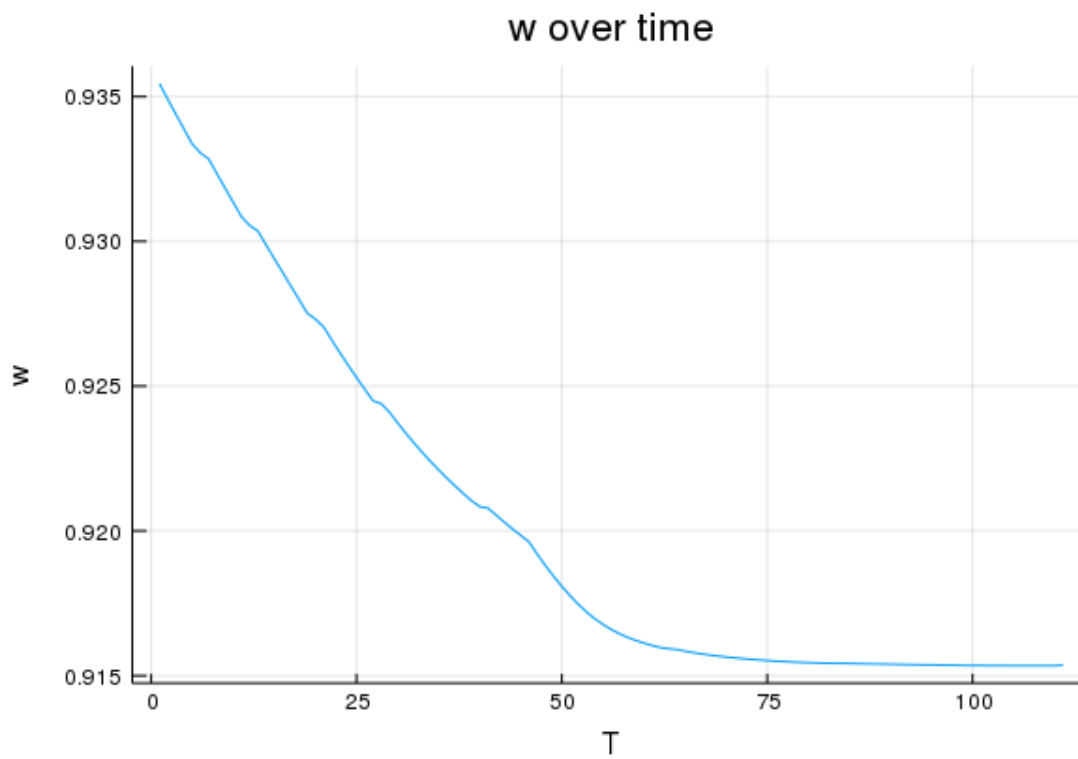
```
In [44]: plot(solved.r, legend = false, title = "r over time", xlabel = "T", ylabel = "r")
```

```
Out[44]:
```



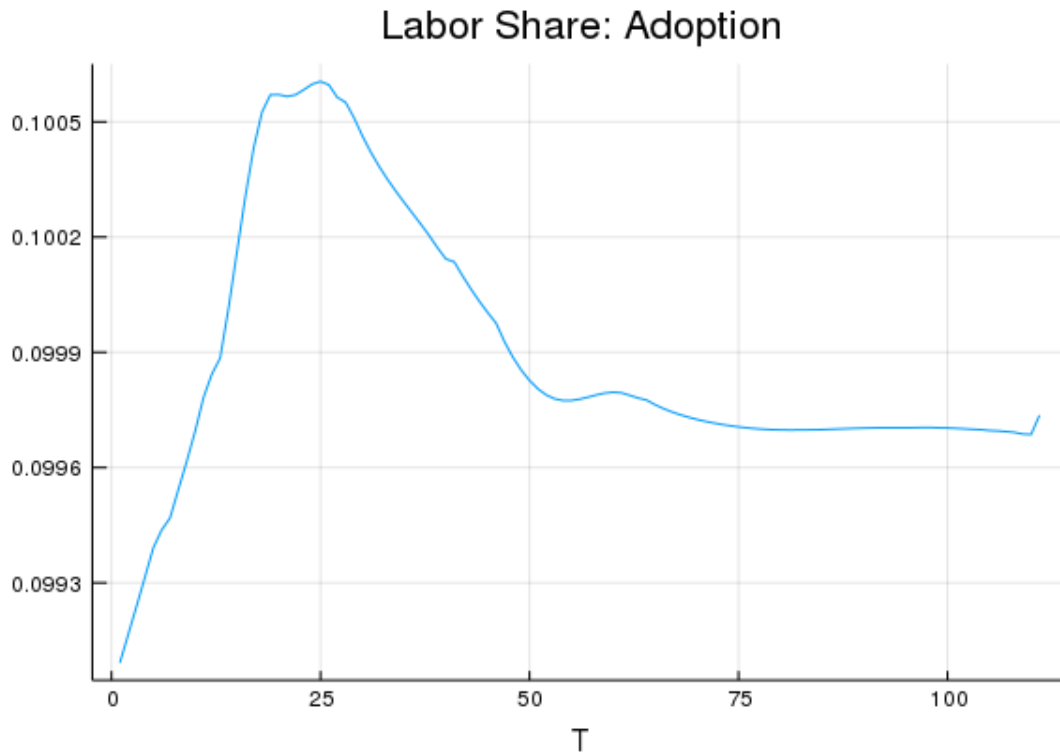
In [45]: `plot(solved.w, legend = false, title = "w over time", xlabel = "T", ylabel = "w")`

Out[45]:



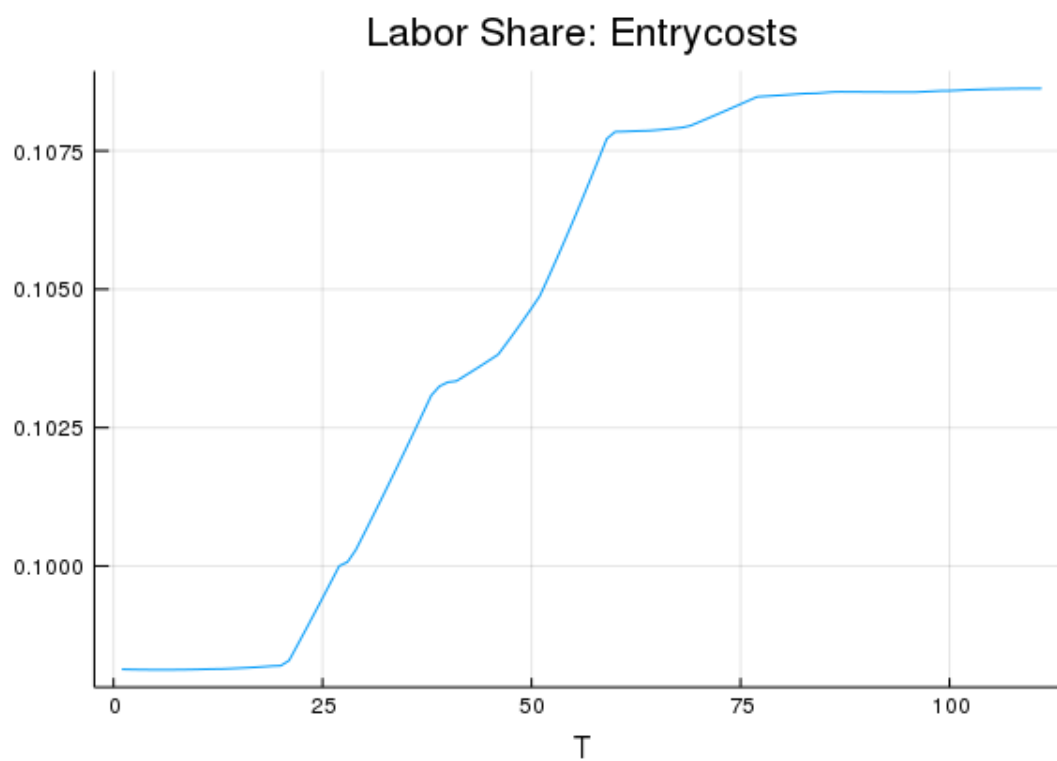
```
In [46]: plot(solved.L_tilde_a, legend = false, xlabel = "T", title = "Labor Share: Adoption")
```

Out[46]:



```
In [47]: plot(solved.L_tilde_E, legend = false, xlabel = "T", title = "Labor Share: Entrycosts")
```

Out[47]:



In [48]: `plot(solved.L_tilde_x, legend = false, xlabel = "T", title = "Labor Share: Export")`
 Out[48]:

