**ECON 6130**

# *Problem Set 5*

Gabe Sekeres

November 19, 2024

**Remark.** As in earlier problem sets, I completed this in both Matlab and Julia. Results are presented from the Julia code for everything except for part (e), which relies on a specific random seed. I attained the same results in both languages. The entirety of the Matlab code ran in 1.977 seconds, and the entirety of the Julia code ran in 0.465 seconds. I used the Matlab symbolic toolbox and the Julia packages `LinearAlgebra`, `Symbolics`, `Plots`, `Random`, `Statistics`, and `StatsBase`. All code was written in the Cursor IDE, which natively implements LLM suggestions – specifically, in my case, from GPT-4o and Claude 3.5 Sonnet (2024.10.22). While legally the code is my own (per Cursor TOS 6.2), I am uncomfortable claiming full ownership of it. After the school year, I will reformat and publicize a Github repo with the entirety of my code, and link it through my website, with an open source MIT license. Code is included at the end, each file separated by a pagebreak and the two languages in their own sections

## Contents



## 1 Solutions

1. **Solution to linearized model:**

   (a) From the notes, we know that the steady state is characterized by the following equations, making the standard CRRA and Cobb-Douglas assumptions as well as assuming technology does not move ($A = 1$):

$$\begin{aligned}
\frac{K}{N} &= \left(\frac{\beta^{-1} - 1 + \delta_k}{\alpha}\right)^{\frac{1}{\alpha - 1}} \\
V &= \left(\frac{\varepsilon\chi}{\phi_n}\frac{1-\alpha}{1-\beta(1-\delta_n)}\left(\frac{K}{N}\right)^{\alpha}\right)^{\frac{1}{1-\varepsilon}} \\
N &= \frac{\chi V^{\varepsilon}}{\delta_n} \\
K &= \frac{K}{N} \cdot N \\
Y &= K^{\alpha} N^{1-\alpha} \\
I &= \delta K \\
C &= Y - I - \phi_n V
\end{aligned}$$

   After running the respective functions (`pset5_parameters.m` and `pset5_model_ss.m` in Matlab, or `crra_cd_ss(parameters)`, defined in `pset5_functions.jl` in Julia), I get the

steady state values:[1]

$$
\begin{aligned}
A &= 1.0000 \\
K &= 348.5424 \\
N &= 19.6668 \\
Y &= 46.5897 \\
C &= 28.6534 \\
I &= 10.4563 \\
N &= 19.6668 \\
V &= 14.9600
\end{aligned}
$$

(b) We have that the model is characterized by the following equations, derived from the Euler equations for capital and labor, as well as the constraints and the laws of motion.

$$
\begin{aligned}
0 &= 1 - \beta \left[ \left( \frac{C_{t+1}}{C_t} \right)^{-\sigma} \left( A_{t+1} \alpha \left( \frac{K_{t+1}}{K_t} \right)^{\alpha - 1} + 1 - \delta_k \right) \right] \\
0 &= \frac{\phi_n}{\varepsilon \chi V_t^{\varepsilon - 1}} - A_t (1 - \alpha) \left( \frac{K_t}{N_t} \right)^{\alpha} + \beta \left[ \left( \frac{C_{t+1}}{C_t} \right)^{-\sigma} \frac{\phi_n}{\varepsilon \chi V_t^{\varepsilon - 1}} (1 - \delta_n) \right] \\
0 &= Y_t - A_t K_t^{\alpha} N_t^{1 - \alpha} \\
0 &= Y_t - C_t - I_t - \phi_n V_t \\
0 &= K_{t+1} - (1 - \delta_k) K_t - I_t \\
0 &= N_t - (1 - \delta_n) N_{t-1} - \chi V_t^{\varepsilon} \\
0 &= \log(A_{t+1}) - \rho \log(A_t) \\
0 &= (N_{t-1})' - N_t
\end{aligned}
$$

Where the final equation exists only for coding purposes. I wrote the file `pset5_model.m` in Matlab and the function `crra_cd_model()` in the file `pset5_functions.jl` in Julia. When testing the steady state numerically in both, I got that all values were within $10^{-14}$ of 0 for both.

(c) I used the given code in Matlab (renamed `pset5_gx_hx.m`), and wrote my own function in Julia (`gx_hx()`, in the file `pset5_functions.jl`). I got the solution matrices:

$$
h_x = \begin{bmatrix} 0.9500 & 0 & 0 \\ 0.0812 & 0.9643 & 0.0707 \\ 0.0220 & 0.0128 & 0.8962 \end{bmatrix}
$$

$$
g_x = \begin{bmatrix} 1.0154 & 0.3090 & 0.6273 \\ 0.4332 & 0.4378 & 0.2000 \\ 2.7081 & -0.1904 & 2.3569 \\ 0.0220 & 0.0128 & 0.8962 \\ 0.8795 & 0.5136 & -0.1535 \end{bmatrix}
$$

$$
\eta = \begin{bmatrix} 0.01 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}
$$

(d) Recall from class, that for a state variable $X$ and a ~~control~~ jump variable $Y$, in response to a

[1] Julia returned further precision (to the $10^{-15}$); for readability I present the values to the $10^{-4}$.

technology shock $\varepsilon_t^1$, we have that

$$\mathbb{E}[X_{t+j} \mid \varepsilon_t^i = 1] - \mathbb{E}[X_{t+j} \mid \varepsilon_t^i = 0] = h^j \eta \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

and

$$\mathbb{E}[Y_{t+j} \mid \varepsilon_t^i = 1] - \mathbb{E}[Y_{t+j} \mid \varepsilon_t^i = 0] = g h^j \eta \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

where $g$, $h$, and $\eta$ are the matrices solved for above. Solving for the effect of a one standard deviation shock to technology over 20 periods, we get:

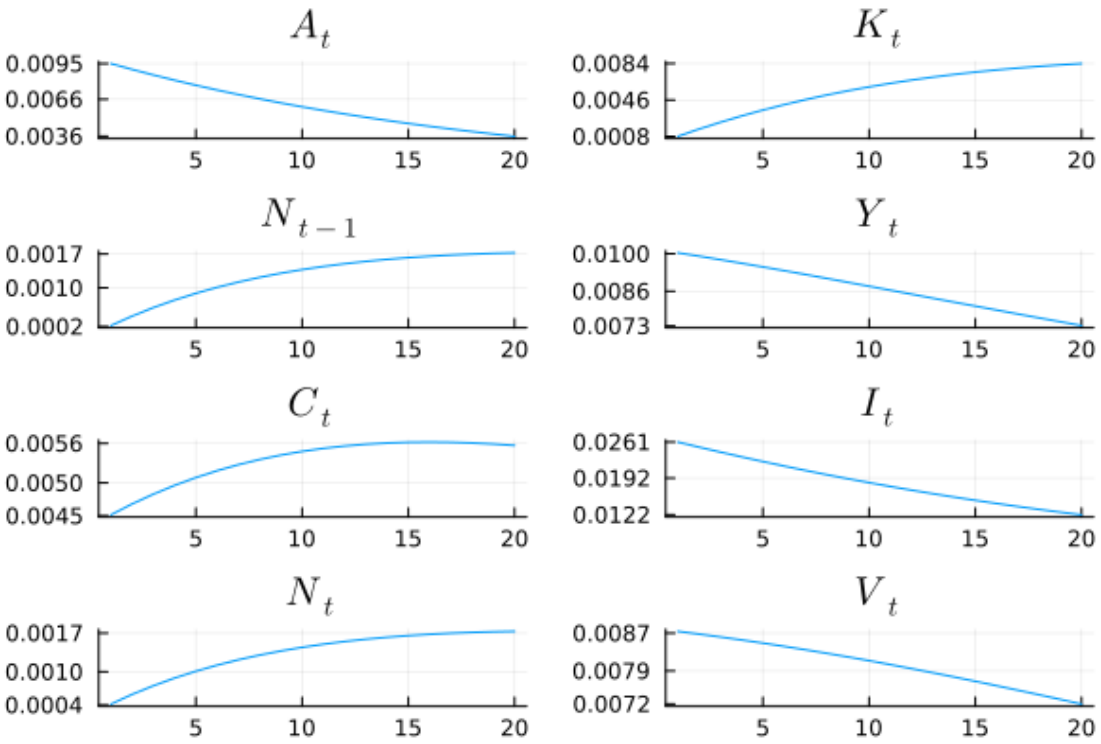


Figure 1: The Effect of a Technology Shock on Endogenous Variables

(e) Similarly to part (d), we have that the effect of a random shock $\varepsilon_t$ on state variables $\hat{x}$ and control variables $\hat{y}$ is

$$\hat{x}_{t+1} = h\hat{x}_t + \eta\varepsilon_{t+1} \quad ; \quad \hat{y}_t = g\hat{x}_t$$

By simulating 5,000 periods of shocks, we get that the first five realizations of log productivity are:

$$\begin{bmatrix} 0.0054 & 0.0234 & -0.0003 & 0.0083 & 0.0111 \end{bmatrix}$$

This also gets us the completed table of moments:

| Moment | Model Value | Data Value |
|---|---|---|
| Std $\log(Y)$ | 0.0522 | 0.0149 |
| Std $\log(C)$ | 0.0391 | 0.0082 |
| Std $\log(I)$ | 0.0975 | 0.0318 |
| Std $\log(N)$ | 0.0119 | 0.0131 |
| Autocorr $\log(Y)$ | 0.98 | 0.85 |
| Autocorr $\log(C)$ | 0.99 | 0.85 |
| Autocorr $\log(I)$ | 0.96 | 0.84 |
| Autocorr $\log(N)$ | $>0.99$ | 0.93 |

Table 1: Moments from Simulated Linear Model

(f) We have that in the model, the ratio $\text{std}\log N/\text{std}\log Y$ is roughly 0.228, while the ratio in the data is about 0.879. Disciplining the model such that the model ratio matches the data ratio could involve one of two strategies – increasing the standard deviation of $N$, or decreasing the standard deviation of $Y$. By inspection, it looks like the standard deviation of $N$ in the model is a lot closer to the data than the standard deviation of $Y$. I will pursue the strategy of working to decrease the standard deviation of $Y$, while leaving $N$ unchanged.

First, a quick programming note. The model values in Table 1 come from Matlab, as the random seed is impossible[2] to fully recreate in Julia. Using an arbitrary seed in Julia, I attained the values:

$$\text{std}\log Y = 0.0512 \quad ; \quad \text{std}\log N = 0.0117 \quad ; \quad \frac{\text{std}\log N}{\text{std}\log Y} \approx 0.229$$

I will be using Julia going forward, as I'm significantly more comfortable in it than Matlab. These values are qualitatively identical, so I feel comfortable doing that. Also, I didn't include this section of the code in the timing benchmarks above. This is, to be frank, ridiculously computationally intensive. It takes 318.186 seconds to run on its own, as compared to the 0.465 seconds for the rest of the problem set. It's currently at approximately $O(n^2)$ complexity, where $n$ is the number of discrete values of the variable we are optimizing over. That can be improved, but not by a massive amount.

Intuitively, we are trying to make output less variable. It seems natural to work with a restriction on consumption – if we change people's incentives, they will act in a way that promotes more or less variable consumption. To that end, we will work with changing the inverse IES, $\sigma$. In general terms, we will be solving the problem

$$\min_{\sigma} \left| \frac{\text{std}\log N}{\text{std}\log Y} - 0.879 \right|$$

I initially discretized $\sigma$ over the grid $[1.5, 10]$ (taking 10,000 values), and found that the optimal $\sigma$ for this problem was at the corner, 10. I increased the grid to $[1.5, 50]$, again taking 10,000 values, and again found a corner. Since each run was intensive and there were diminishing returns, I ended up discretizing the grid $[1.5, 100]$ over 10,000 values of $\sigma$. The problem was optimized when $\sigma = 100$, which led to standard deviations:

$$\text{std}\log Y = 0.114 \quad ; \quad \text{std}\log N = 0.0397 \quad ; \quad \frac{\text{std}\log N}{\text{std}\log Y} \approx 0.347$$

I can say that my mission to better match the data's ratio was a success! Unfortunately, I did so by massively decreasing the stability of both output and employment. Intuitively, it appears

[2] Ok, not technically impossible. But recreating it would involve saving the first 5,000 elements of `randn` and then converting them to a readable data structure and importing them into Julia. This is way too much work, so I didn't do it.

that when $\sigma$ increases (*i.e.*when the representative household becomes more risk averse), stability decreases in general. I have no idea why this would happen.

I first was interested in how the difference between the model ratio and the data ratio was affected by $\sigma$. As expected, the difference is decreasing as $\sigma$ increases, and the relationship looks monotonic. Figure 2 shows the calibration difference as $\sigma$ increases – note that the difference is negative, so a positive slope means less of a difference.

I then found the standard deviations and autocorrelations of the other relevant variables and added them to the table of moments from part (e). That is Table 2. As you can see, all variables became less stable from the original model. Interestingly, investment was the most unstable by a large margin, while consumption barely changed.

Finally, I was interested in what effect $\sigma$ had on whether there were issues solving the model. In an earlier test, I varied $\phi_n$, and this caused some massive headaches for me. Interestingly, varying $\sigma$ did nothing – everything remained feasibly. I intentionally avoided $\sigma = 1$, as I wasn't sure how Julia would handle CRRA utility approaching log utility, but otherwise it had no effects. There is no figure for this part, as it is just a horizontal line.

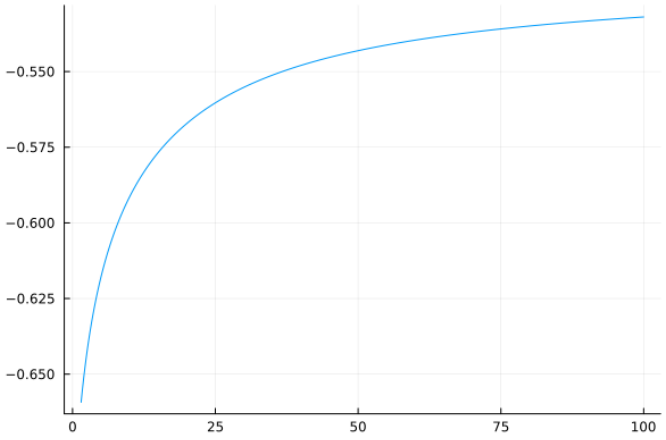


Figure 2: Calibration Difference, by Risk Aversion Parameter

| Moment | Model Value | Data Value | Optimal Model Value |
|---|---|---|---|
| Std $\log(Y)$ | 0.0522 | 0.0149 | 0.114 |
| Std $\log(C)$ | 0.0391 | 0.0082 | 0.0432 |
| Std $\log(I)$ | 0.0975 | 0.0318 | 0.2777 |
| Std $\log(N)$ | 0.0119 | 0.0131 | 0.0397 |
| Autocorr $\log(Y)$ | 0.98 | 0.85 | $>$0.99 |
| Autocorr $\log(C)$ | 0.99 | 0.85 | $>$0.99 |
| Autocorr $\log(I)$ | 0.96 | 0.84 | $>$0.99 |
| Autocorr $\log(N)$ | $>$0.99 | 0.93 | $>$0.99 |

Table 2: Moments from Simulated Linear Model, with Optimal $\sigma$ added

## 2 Matlab Code

What follows is Matlab code, split into different files, that solves parts (a) through (e) of this problem set. All of the code is run from `pset5_main.m`, which calls the other files. They are:

- `pset5_main.m`:

```
clear

tic

%Load Parameters
param = pset5_parameters;

disp('ss param:')
disp(pset5_model_ss(param))

%Compute the first-order coefficiencients of the model
[fyn, fxn, fypn, fxpn, fn] = pset5_model(param);

[gx,hx]= pset5_gx_hx(fyn,fxn,fypn,fxpn);

disp('gx:')
disp(gx)
disp('hx:')
disp(hx)

%Initialize shock
eta = zeros(3,3);
eta(1,1) = param.siga;

disp('eta:')
disp(eta)

%Eigenvalues of hx
disp('Computing eigenvalues of hx')
disp(eig(hx))

%Number of periods for shock
T = 20;

%Impulse response functions
IRF_x = zeros(3,T);
IRF_y = zeros(5,T);

for i = 1:T
    IRF_x(:,i) = hx^i * eta * [1 0 0]';
    IRF_y(:,i) = gx * hx^i * eta * [1 0 0]';
end

%Figure of impulse response functions
subplot(4,2,1)
plot(IRF_x(1,:))
```

```
title('Technology shock on A_{t}')
subplot(4,2,2)
plot(IRF_x(2,:))
title('Technology shock on K_{t}')
subplot(4,2,3)
plot(IRF_x(3,:))
title('Technology shock on N_{t-1}')
subplot(4,2,4)
plot(IRF_y(1,:))
title('Technology shock on Y_{t}')
subplot(4,2,5)
plot(IRF_y(2,:))
title('Technology shock on C_{t}')
subplot(4,2,6)
plot(IRF_y(3,:))
title('Technology shock on I_{t}')
subplot(4,2,7)
plot(IRF_y(4,:))
title('Technology shock on N_{t}')
subplot(4,2,8)
plot(IRF_y(5,:))
title('Technology shock on V_{t}')

saveas(gcf, '/Users/gabesekeres/Dropbox/Notes/Cornell_Notes/Fall_2024/
   Macro/Matlab/pset5_tech_shock.png')


%Simulate with random shocks
rng(0);
L = 5000;
epsilon = randn(1,L);
epsilon = [0 epsilon];

simX = zeros(3,L+1);
simY = zeros(5,L+1);
for i = 1:L
    simX(:,i+1) = hx * simX(:,i) + eta * [epsilon(i+1) 0 0]';
    simY(:,i+1) = gx * simX(:,i+1);
end

simYt = simY(1,:);
simC = simY(2,:);
simI = simY(3,:);
simN = simY(4,:);
simV = simY(5,:);

%First five realizations of productivity
disp("First five realizations of productivity:")
disp(simX(1,2:6))

%Standard Deviations
```

```
disp("Standard Deviations:")
disp("SD Y: " + std(simYt))
disp("SD C: " + std(simC))
disp("SD I: " + std(simI))
disp("SD N: " + std(simN))


%Autocorrelations
[acf_Y, lags] = autocorr(simYt, 'NumLags', 1);
[acf_C, ~] = autocorr(simC, 'NumLags', 1);
[acf_I, ~] = autocorr(simI, 'NumLags', 1);
[acf_N, ~] = autocorr(simN, 'NumLags', 1);

% Display results (take second value as first is always 1)
disp("Autocorrelations:")
disp("Y: " + num2str(acf_Y(2)))
disp("C: " + num2str(acf_C(2)))
disp("I: " + num2str(acf_I(2)))
disp("N: " + num2str(acf_N(2)))


toc
```

- pset5_parameters.m

```
function param = pset5_parameters()
    param.bet = 0.99;
    param.sig = 2.00;
    param.alpha = 0.30;
    param.deltak = 0.03;
    param.deltan = 0.10;
    param.phin = 0.50;
    param.chi = 1.00;
    param.eps = 0.25;
    param.rho = 0.95;
    param.siga = 0.01;
end
```

- pset5_model_ss.m

```
function [ss, param] = pset5_model_ss(param)
%Parameters from param object
bet  = param.bet;
alpha = param.alpha;
deltak = param.deltak;
deltan = param.deltan;
phin = param.phin;
chi = param.chi;
eps = param.eps;

%Use closed form expressions for the ss values.
kn = ((1/bet - 1 + deltak) / alpha)^(1 / (alpha - 1));
v = (((eps*chi) / phin) * (1 - alpha) / (1 - bet * (1 - deltan)) * kn ^
   alpha)^(1 / (1 - eps));
n = chi * v ^ eps / deltan;
k = kn * n;
y = k^alpha * n^(1-alpha);
i = deltak * k;
c = y - i - phin * v;


%Put the ss values in a vector
xx = [1 k n];
yy = [y c i n v];
ss = [xx yy];
end
```

- pset5_model.m

```
function [fyn, fxn, fypn, fxpn, fn, log_var] = pset5_model(param)

%Steady State
[ss, param] = pset5_model_ss(param);

%Declare parameters
bet = param.bet;
sig = param.sig;
alpha = param.alpha;
deltak = param.deltak;
deltan = param.deltan;
phin = param.phin;
chi = param.chi;
eps = param.eps;
rho = param.rho;

%Declare symbols
syms A A_p K K_p N_m N_m_p
syms Yt Yt_p C C_p I I_p N N_p V V_p

%Declare X and Y vectors
X = [A K N_m];
XP = [A_p K_p N_m_p];

Y = [Yt C I N V];
YP = [Yt_p C_p I_p N_p V_p];

%Model Equations
f(1) = 1 - bet * (C_p / C)^(-sig) * (A_p * alpha * (K_p / N_p)^(alpha - 1)
    + 1 - deltak);
f(2) = phin / (eps * chi * V^(eps - 1)) - A * (1 - alpha) * (K / N)^alpha
   - bet * (C_p / C)^(-sig) * (phin / (eps * chi * V_p^(eps - 1))) * (1 -
    deltan);
f(3) = Yt - A * K^alpha * N^(1 - alpha);
f(4) = Yt - C - I - phin * V;
f(5) = K_p - (1 - deltak) * K - I;
f(6) = N - (1 - deltan) * N_m - chi * V^eps;
f(7) = log(A_p) - rho * log(A);
f(8) = N_m_p - N;

%Chech computation of steady state numerically
fnum = double(subs(f, [X Y XP YP], [ss ss]));
disp('Checking steady state:');
disp(fnum);

%Log-linearize the model
log_var = [X Y XP YP];
f = subs(f, log_var, exp(log_var));
ss = log(ss);
```

```
%Differentiate
fx = jacobian(f, X);
fy = jacobian(f, Y);
fxp = jacobian(f, XP);
fyp = jacobian(f, YP);

%Compute numerical values
fyn = double(subs(fy, [X Y XP YP], [ss ss]));
fxn = double(subs(fx, [X Y XP YP], [ss ss]));
fypn = double(subs(fyp, [X Y XP YP], [ss ss]));
fxpn = double(subs(fxp, [X Y XP YP], [ss ss]));
fn = double(subs(f, [X Y XP YP], [ss ss]));

end
```

- pset5_gx_hx.m

```
function [gx,hx,exitflag]=pset5_gx_hx(fy,fx,fyp,fxp,stake)
    if nargin<5
        stake=1;
    end
    exitflag = 1;

    %Create system matrices A,B
    A = [-fxp -fyp];
    B = [fx fy];
    NK = size(fx,2);

    %Complex Schur Decomposition
    [s,t,q,z] = qz(A,B);

    %Pick non-explosive (stable) eigenvalues
    slt = (abs(diag(t))<stake*abs(diag(s)));
    nk=sum(slt);

    %Reorder the system with stable eigs in upper-left
    [s,t,~,z] = ordqz(s,t,q,z,slt);

    %Split up the results appropriately
    z21 = z(nk+1:end,1:nk);
    z11 = z(1:nk,1:nk);

    s11 = s(1:nk,1:nk);
    t11 = t(1:nk,1:nk);

    %Catch cases with no/multiple solutions
    if nk>NK
        warning('The Equilibrium is Locally Indeterminate')
        exitflag=2;
    elseif nk<NK
        warning('No Local Equilibrium Exists')
        exitflag = 0;
    elseif rank(z11)<nk
        warning('Invertibility condition violated')
        exitflag = 3;
    end

    %Compute the Solution
    z11i = z11\eye(nk);
    gx = real(z21*z11i);
    hx = real(z11*(s11\t11)*z11i);
end
```

# 3 Julia Code

The Julia code is split only into two files. One defines the functions I reference throughout. The other is the main shell file that calls the functions.

- pset5_main.jl

```
using LinearAlgebra, Symbolics, Plots, Random, Statistics, StatsBase

# Load functions
include("pset5_functions.jl")

@time begin
# Set the parameters
param = [0.99, 2.00, 0.30, 0.03, 0.10, 0.50, 1.00, 0.25, 0.95, 0.01]

# Compute the model equations
fyn, fxn, fypn, fxpn, fn = crra_cd_model(param)

# Compute the gx and hx matrices
gx, hx, exitflag = gx_hx(fyn, fxn, fypn, fxpn)

# Initialize shock
eta = zeros(3,3)
eta[1,1] = param[10]

# Timing of shock
T = 20

# Impulse response functions
IRF_x = zeros(3,T)
IRF_y = zeros(5,T)

for i = 1:T
    IRF_x[:,i] = hx^i * eta * [1 0 0]'
    IRF_y[:,i] = gx * hx^i * eta * [1 0 0]'
end

# Plot the impulse response functions
function three_ticks(y_values)
    min_val = round(minimum(y_values), digits=4)
    max_val = round(maximum(y_values), digits=4)
    mid_val = round((min_val + max_val) / 2, digits=4)
    return [min_val, mid_val, max_val]
end
plot(layout=(4,2), legend=false)
plot!(subplot=1, IRF_x[1,:], title="\$A_{t}\$", yticks=three_ticks(IRF_x
    [1,:]))
plot!(subplot=2, IRF_x[2,:], title="\$K_{t}\$", yticks=three_ticks(IRF_x
    [2,:]))
plot!(subplot=3, IRF_x[3,:], title="\$N_{t-1}\$", yticks=three_ticks(IRF_x
    [3,:]))
plot!(subplot=4, IRF_y[1,:], title="\$Y_{t}\$", yticks=three_ticks(IRF_y
```

```
    [1,:]))
plot!(subplot=5, IRF_y[2,:], title="\$C_{t}\$", yticks=three_ticks(IRF_y
    [2,:]))
plot!(subplot=6, IRF_y[3,:], title="\$I_{t}\$", yticks=three_ticks(IRF_y
    [3,:]))
plot!(subplot=7, IRF_y[4,:], title="\$N_{t}\$", yticks=three_ticks(IRF_y
    [4,:]))
plot!(subplot=8, IRF_y[5,:], title="\$V_{t}\$", yticks=three_ticks(IRF_y
    [5,:]))


# Save the plot
savefig("/Users/gabesekeres/Dropbox/Notes/Cornell_Notes/Fall_2024/Macro/
    Julia/pset5_tech_shock.png")

# Simulate with random shocks
Random.seed!(0)
L = 5000
epsilon = vcat(0.0, randn(L))

simX = zeros(3, L+1)
simY = zeros(5, L+1)
for i in 1:L
    simX[:, i+1] = hx * simX[:, i] + eta * [epsilon[i+1], 0, 0]
    simY[:, i+1] = gx * simX[:, i+1]
end

simYt = simY[1, :]
simC = simY[2, :]
simI = simY[3, :]
simN = simY[4, :]
simV = simY[5, :]

# First five realizations of productivity
println("First five realizations of productivity:")
println(simX[1, 2:6])

# Standard Deviations
println("Standard Deviations:")
println("SD Y: ", std(simYt))
println("SD C: ", std(simC))
println("SD I: ", std(simI))
println("SD N: ", std(simN))

# Autocorrelations
acf_Y = autocor(simYt)
acf_C = autocor(simC)
acf_I = autocor(simI)
acf_N = autocor(simN)

# Display results (take second value as first is always 1)
```

```
println("Autocorrelations:")
println("Y: ", acf_Y[2])
println("C: ", acf_C[2])
println("I: ", acf_I[2])
println("N: ", acf_N[2])


end

## Optimize the value of sigma to match std ratio to data
@time begin
# Number of sigmas
num = 10000
# Initialize sigmas
sigmas = collect(range(start = 1.5, stop = 100, length = num))

returns = zeros(4,num)
function iterate_sigmas(sigma)
    # Create parameter vector with new sigma
    new_params = [0.99, sigma, 0.30, 0.03, 0.10, 0.50, 1.00, 0.25, 0.95,
       0.01]

    # Get model matrices with new parameters
    fyn, fxn, fypn, fxpn, fn = crra_cd_model(new_params, verbose=false)
    gx, hx, exitflag = gx_hx(fyn, fxn, fypn, fxpn, stake=1.0, verbose=
       false)

    if exitflag != 1
        return [1.0, 1.0, 1.0, Float64(exitflag)]
    end

    # Create new simulation matrices for each sigma
    simX_local = zeros(3, L+1)
    simY_local = zeros(5, L+1)

    # Create shock matrix for this iteration
    eta_local = zeros(3,3)
    eta_local[1,1] = new_params[10]  # Use the shock parameter from
       new_params

    # Simulate with these matrices
    for i in 1:L
        simX_local[:, i+1] = hx * simX_local[:, i] + eta_local * [epsilon[
           i+1], 0, 0]
        simY_local[:, i+1] = gx * simX_local[:, i+1]
    end

    stdY = std(simY_local[1,:])
    stdN = std(simY_local[4,:])
    ratio_diff = stdN / stdY - 0.8791946  # Target ratio from data
```

```
    return [stdY, stdN, ratio_diff, Float64(exitflag)]
end

returns = zeros(4,num)
# Iterate over sigmas
for i in 1:num
    returns[:, i] = iterate_sigmas(sigmas[i])
end

valid_indices = findall(x -> x == 1, returns[4,:])  # Find where exitflag
    == 1

# Filter sigmas and returns
valid_sigmas = sigmas[valid_indices]
valid_returns = returns[:, valid_indices]

# Plot only the valid results
plot(valid_sigmas, valid_returns[3,:],
     legend=false,
    yformatter=x->round(x, digits=3))
savefig("/Users/gabesekeres/Dropbox/Notes/Cornell_Notes/Fall_2024/Macro/
    Julia/pset5_sigma_ratio.png")


# Find optimal sigma among valid results only
min_index = argmin(abs.(valid_returns[3, :]))
optimal_sigma = valid_sigmas[min_index]

println("Optimal sigma: ", optimal_sigma)
println("Ratio: ", valid_returns[3, min_index] + 0.8791946)
println("Std Y: ", valid_returns[1, min_index])
println("Std N: ", valid_returns[2, min_index])

# Create a vector to indicate validity of each sigma
validity = zeros(Int, num)
validity[valid_indices] .= 1

# Plot the validity of each sigma
plot(sigmas, validity,
     legend=false,
     yformatter=x->round(x, digits=0),
     xlabel="Sigma", ylabel="Validity (1=valid, 0=invalid)")
savefig("/Users/gabesekeres/Dropbox/Notes/Cornell_Notes/Fall_2024/Macro/
    Julia/pset5_sigma_validity.png")

# Get full optimal std and autocorrelations
opt_params = [0.99, optimal_sigma, 0.30, 0.03, 0.10, 0.50, 1.00, 0.25,
    0.95, 0.01]

opt_fyn, opt_fxn, opt_fypn, opt_fxpn, opt_fn = crra_cd_model(opt_params,
    verbose=false)
```

```
opt_gx, opt_hx, opt_exitflag = gx_hx(opt_fyn, opt_fxn, opt_fypn, opt_fxpn,
    stake=1.0, verbose=false)

simX_opt = zeros(3, L+1)
simY_opt = zeros(5, L+1)

eta_opt = zeros(3,3)
eta_opt[1,1] = opt_params[10]
for i in 1:L
    simX_opt[:, i+1] = opt_hx * simX_opt[:, i] + eta_opt * [epsilon[i+1],
       0, 0]
    simY_opt[:, i+1] = opt_gx * simX_opt[:, i+1]
end

simYt_opt = simY_opt[1, :]
simC_opt = simY_opt[2, :]
simI_opt = simY_opt[3, :]
simN_opt = simY_opt[4, :]

# Standard Deviations
println("Standard Deviations:")
println("SD Y: ", std(simYt_opt))
println("SD C: ", std(simC_opt))
println("SD I: ", std(simI_opt))
println("SD N: ", std(simN_opt))

# Autocorrelations
acf_Y_opt = autocor(simYt_opt)
acf_C_opt = autocor(simC_opt)
acf_I_opt = autocor(simI_opt)
acf_N_opt = autocor(simN_opt)

# Display results (take second value as first is always 1)
println("Autocorrelations:")
println("Y: ", acf_Y_opt[2])
println("C: ", acf_C_opt[2])
println("I: ", acf_I_opt[2])
println("N: ", acf_N_opt[2])


end
```

- pset5_functions.jl

```
"""
    crra_cd_ss(param)

Compute the steady state of the CRRA-CD model.
param = [beta, sig, alpha, deltak, deltan, phin, chi, eps, rho, siga]
"""
function crra_cd_ss(param::Vector{Float64})
    kn = ((1/param[1] - 1 + param[4]) / param[3])^(1 / (param[3] - 1))
    v = (((param[8]*param[7]) / param[6]) * (1 - param[3]) / (1 - param[1]
        * (1 - param[5])) * kn ^ param[3])^(1 / (1 - param[8]))
    n = param[7] * v ^ param[8] / param[5]
    k = kn * n
    y = k^param[3] * n^(1-param[3])
    i = param[4] * k
    c = y - i - param[6] * v
    return [1 k n y c i n v]
end

"""
    crra_cd_model(param)

Compute the first-order coefficients of the CRRA-CD model.
param = [beta, sig, alpha, deltak, deltan, phin, chi, eps, rho, siga]
"""
function crra_cd_model(param::Vector{Float64}; verbose::Bool=true)
    # Steady State
    ss = crra_cd_ss(param)

    # Convert ss to vector
    ss = vec(ss)

    # Declare parameters
    bet, sig, alpha, deltak, deltan, phin, chi, eps, rho = param[1:9]

    # Declare symbolic variables
    @variables A A_p K K_p N_m N_m_p
    @variables Yt Yt_p C C_p I I_p N N_p V V_p

    # Declare X and Y vectors
    X = [A, K, N_m]
    XP = [A_p, K_p, N_m_p]
    Y = [Yt, C, I, N, V]
    YP = [Yt_p, C_p, I_p, N_p, V_p]

    # Model Equations
    f = [
        1 - bet * (C_p / C)^(-sig) * (A_p * alpha * (K_p / N_p)^(alpha -
           1) + 1 - deltak),
        phin / (eps * chi * V^(eps - 1)) - A * (1 - alpha) * (K / N)^alpha
            -
```

```
            bet * (C_p / C)^(-sig) * (phin / (eps * chi * V_p^(eps - 1)))
                * (1 - deltan),
        Yt - A * K^alpha * N^(1 - alpha),
        Yt - C - I - phin * V,
        K_p - (1 - deltak) * K - I,
        N - (1 - deltan) * N_m - chi * V^eps,
        log(A_p) - rho * log(A),
        N_m_p - N
    ]

    # Check steady state
    all_vars = vcat(X, Y, XP, YP)
    ss_double = vcat(ss, ss)  # Changed from vcat(ss; ss)
    ss_vals = Dict(zip(all_vars, ss_double))
    fnum = [substitute(fi, ss_vals) for fi in f]
    if verbose
        println("Checking steady state:")
        println(fnum)
    end

    # Log-linearize the model
    log_vars = Dict(v => exp(v) for v in all_vars)
    f = [substitute(fi, log_vars) for fi in f]
    ss = log.(ss)
    ss_double = vcat(ss, ss)  # Changed from vcat(ss; ss)
    ss_vals = Dict(zip(all_vars, ss_double))

    # Compute Jacobians
    fx = Symbolics.jacobian(f, X)
    fy = Symbolics.jacobian(f, Y)
    fxp = Symbolics.jacobian(f, XP)
    fyp = Symbolics.jacobian(f, YP)

    # Evaluate at steady state
    fxn = Float64.(Symbolics.value.(substitute.(fx, Ref(ss_vals))))
    fyn = Float64.(Symbolics.value.(substitute.(fy, Ref(ss_vals))))
    fxpn = Float64.(Symbolics.value.(substitute.(fxp, Ref(ss_vals))))
    fypn = Float64.(Symbolics.value.(substitute.(fyp, Ref(ss_vals))))
    fn = Float64.(Symbolics.value.(substitute.(f, Ref(ss_vals))))

    return fyn, fxn, fypn, fxpn, fn
end
"""
    gx_hx(fyn, fxn, fypn, fxpn, stake)

Compute the gx and hx matrices of the CRRA-CD model.
"""
function gx_hx(fyn::Matrix{Float64}, fxn::Matrix{Float64},
    fypn::Matrix{Float64}, fxpn::Matrix{Float64};
    stake::Real=1.0, verbose::Bool=true)
# Initialize exitflag
```

```
exitflag = 1

# Create system matrices A, B
A = [-fxpn -fypn]
B = [fxn fyn]
NK = size(fxn, 2)

# Complex Schur Decomposition
F = schur(A, B)

# Pick non-explosive (stable) eigenvalues
slt = abs.(diag(F.T)) .< stake .* abs.(diag(F.S))
nk = sum(slt)

# Reorder the system with stable eigenvalues in upper-left
F = ordschur(F, slt)

# Get the Z matrix from the decomposition
Z = F.Z

# Split up the results appropriately
z21 = Z[nk+1:end, 1:nk]
z11 = Z[1:nk, 1:nk]
s11 = F.S[1:nk, 1:nk]
t11 = F.T[1:nk, 1:nk]

# Catch cases with no / multiple solutions
if nk > NK
    verbose && @warn "The Equilibrium is Locally Indeterminate"
    exitflag = 2
    return zeros(size(z21)), zeros(size(z11)), exitflag
elseif nk < NK
    verbose && @warn "No Local Equilibrium Exists"
    exitflag = 0
    return zeros(size(z21)), zeros(size(z11)), exitflag
elseif rank(z11) < nk
    verbose && @warn "Invertibility condition violated"
    exitflag = 3
    return zeros(size(z21)), zeros(size(z11)), exitflag
end

# Compute the Solution
    try
        z11i = z11 \ I(nk)
        gx = real(z21 * z11i)
        hx = real(z11 * (s11 \ t11) * z11i)
        return gx, hx, exitflag
    catch e
        if isa(e, SingularException)
            verbose && @warn "Singular matrix encountered in computation"
            exitflag = 3
```

```
            return zeros(size(z21)), zeros(size(z11)), exitflag
        else
            rethrow(e)
        end
    end
end
```