Econ 413R: Computational Economics

Spring Term 2013

Dynamic Stochastic General Equilibrium Modeling Homework Key

Week 2

Homework 1

For the Brock and Mirman model, find the value of A in the policy function. Show that your value is correct.

For this case find an algebraic solution for the policy function, $k_{t+1} = \Phi(k_t, z_t)$. Couple of good sources for hints are **?**, excercise 2.2, p. 12 and **?**, exercise 1.1, p. 47.

Answer:

We take a good educated guess and assume that $A = \alpha\beta$

Household problem is:

$$\begin{aligned}
V(k_t, \theta_t) &= \max_{k_{t+1}} \ln[w_t + r_t k_t - k_{t+1}] + \beta V(k_{t+1}, \theta_{t+1}) \\
&= \max_{k_{t+1}} \ln[(1-\alpha)e^{z_t}k_t^\alpha + \alpha e^{z_t}k_t^{\alpha-1}k_t - k_{t+1}] + \beta V(k_{t+1}, \theta_{t+1}) \\
&= \max_{k_{t+1}} \ln[e^{z_t}k_t^\alpha - k_{t+1}] + \beta V(k_{t+1}, \theta_{t+1})
\end{aligned}$$

Which gives the following Euler equation:

$$[e^{z_t}k_t^\alpha - k_{t+1}]^{-1} = \beta E_t\{[e^{z_{t+1}}k_{t+1}^\alpha - k_{t+2}]^{-1}\alpha k_{t+1}^{\alpha-1}e^{z_{t+1}}\}$$

We try a guess and verify method. We guess that the policy function is $k_{t+1} = \beta\alpha k_t^\alpha e^{z_t}$.

If this is the true policy function then the Euler equation must hold exactly. Substituting the policy function into the Euler equation we have:

$$\begin{aligned}
[e^{z_t}k_t^\alpha - \beta\alpha k_t^\alpha e^{z_t}]^{-1} &= \beta E_t\{[e^{z_{t+1}}k_{t+1}^\alpha - \beta\alpha k_{t+1}^\alpha e^{z_{t+1}}]^{-1}\alpha k_{t+1}^{\alpha-1}e^{z_{t+1}}\} \\
\frac{1}{e^{z_t}k_t^\alpha(1-\beta\alpha)} &= \beta E_t\left\{\frac{\alpha k_{t+1}^{\alpha-1}e^{z_{t+1}}}{e^{z_{t+1}}k_{t+1}^\alpha(1-\beta\alpha)}\right\} \\
\frac{1}{e^{z_t}k_t^\alpha} &= \beta E_t\left\{\frac{\alpha}{k_{t+1}}\right\} \\
\frac{1}{e^{z_t}k_t^\alpha} &= \beta E_t\left\{\frac{\alpha}{\beta\alpha k_t^\alpha e^{z_t}}\right\} \qquad \square
\end{aligned}$$

Homework 2

For the model in section 3 of the notes consider the following functional forms:

$$u(c_t, \ell_t) = \ln c_t + a \ln (1 - \ell_t)$$
$$F(K_t, L_t, z_t) = e^{z_t} K_t^{\alpha} L_t^{1-\alpha}$$

Write out all the characterizing equations for the model using these functional forms.

Can you use the same tricks as in homework 1 to solve for the policy function in this case? Why or why not?

Answer:

Household's maximize utility

$$\beta E_t \left\{ \frac{c_t}{c_{t+1}} [(r_{t+1} - \delta)(1 - \tau) + 1] \right\} = 1$$
$$\frac{w_t(1-\tau)(1-\ell_t)}{a c_t} = 1$$
$$(1 - \tau)\left[w_t \ell_t + (r_t - \delta) k_t\right] + k_t + T_t - k_{t+1} = c_t$$

Firm's maximize profits

$$W_t = (1 - \alpha) e^{z_t} K_t^{\alpha} L_t^{-\alpha}$$
$$R_t = \alpha e^{z_t} K_t^{\alpha - 1} L_t^{1-\alpha}$$

Adding up

$$k_t = K_t \quad \ell_t = L_t$$
$$w_t = W_t \quad r_t = R_t$$

$$T_t = \tau\left[w_t \ell_t + (r_t - \delta) k_t\right]$$
$$z_t = (1 - \rho_z)\bar{z} + \rho_z z_{t-1} + \epsilon_t^z; \quad \epsilon_t^z \sim \text{i.i.d.}(0, \sigma_z^2)$$

Can't solve for the policy function as easily since we have an additive term inside the log utility function when depreciation is not 100%.

Homework 3

For the model in section 3 of the notes consider the following functional forms:

$$\begin{aligned} u(c_t, \ell_t) &= \frac{c_t^{1-\gamma} - 1}{1-\gamma} + a \ln(1 - \ell_t) \\ F(K_t, L_t, z_t) &= e^{z_t} K_t^{\alpha} L_t^{1-\alpha} \end{aligned}$$

Write out all the characterizing equations for the model using these functional forms.

Answer:

The equations are:

$$\begin{aligned} 1 &= \beta E_t \left\{ \left(\frac{c_t}{c_{t+1}} \right)^{\gamma} [(r_{t+1} - \delta)(1-\tau) + 1] \right\} \\ 1 &= \frac{w_t (1-\tau)(1-\ell_t)}{a c_t^{\gamma}} \\ c_t &= (1-\tau)\left[w_t \ell_t + (r_t - \delta) k_t \right] + k_t + T_t - k_{t+1} \\ W_t &= (1-\alpha) e^{z_t} K_t^{\alpha} L_t^{-\alpha} \\ R_t &= \alpha e^{z_t} K_t^{\alpha-1} L_t^{1-\alpha} \\ k_t &= K_t \\ \ell_t &= L_t \\ w_t &= W_t \\ r_t &= R_t \\ T_t &= \tau \left[w_t \ell_t + (r_t - \delta) k_t \right] \\ z_t &= (1-\rho_z)\bar{z} + \rho_z z_{t-1} + \epsilon_t^z; \quad \epsilon_t^z \sim \text{i.i.d.}(0, \sigma_z^2) \end{aligned}$$

Homework 4

For the model in section 3 of the notes consider the following functional forms:

$$u(c_t, \ell_t) = \frac{c_t^{1-\gamma} - 1}{1-\gamma} + a\frac{(1-\ell_t)^{1-\xi} - 1}{1-\xi}$$
$$F(K_t, L_t, z_t) = e^{z_t}\left[\alpha K_t^{\eta} + (1-\alpha)L_t^{\eta}\right]^{\frac{1}{\eta}}$$

Write out all the characterizing equations for the model using these functional forms.

Answer:

The equations are:

$$1 = \beta E_t\left\{\left(\frac{c_t}{c_{t+1}}\right)^{\gamma}\left[(r_{t+1}-\delta)(1-\tau)+1\right]\right\}$$
$$1 = \frac{w_t(1-\tau)(1-\ell_t)^{\xi}}{ac_t^{\gamma}}$$
$$c_t = (1-\tau)\left[w_t\ell_t + (r_t-\delta)k_t\right] + k_t + T_t - k_{t+1}$$
$$W_t = (1-\alpha)e^{z_t}\left[K_t^{\eta} + L_t^{\eta}\right]^{\frac{1}{\eta}-1}L_t^{\eta-1}$$
$$R_t = \alpha e^{z_t}\left[K_t^{\eta} + L_t^{\eta}\right]^{\frac{1}{\eta}-1}K_t^{\eta-1}$$
$$k_t = K_t$$
$$\ell_t = L_t$$
$$w_t = W_t$$
$$r_t = R_t$$
$$T_t = \tau\left[w_t\ell_t + (r_t-\delta)k_t\right]$$
$$z_t = (1-\rho_z)\bar{z} + \rho_z z_{t-1} + \epsilon_t^z; \quad \epsilon_t^z \sim \text{i.i.d.}(0, \sigma_z^2)$$

Homework & Lab 5a

For the model in section 3 of the notes consider the following functional forms:

$$u(c_t) = \frac{c_t^{1-\gamma} - 1}{1-\gamma}$$
$$F(K_t, L_t, z_t) = K_t^{\alpha}(L_t e^{z_t})^{1-\alpha}$$

Write out all the characterizing equations for the model using these functional forms. Assume $\ell_t = 1$.

Write out the steady state versions of these equations. Solve algebraically for the steady state value of k as a function of the steady state value of z and the parameters of the model. Numerically solve for the steady state values of all variables using the following parameter values: $\gamma = 2.5$, $\beta = .98$, $\alpha = .40$, $a = .5$, $\delta = .10$, $\bar{z} = 0$ and $\tau = .05$. Also solve for the steady state values of output and investment. Compare these values with the ones implied by the algebraic solution.

Answer:

Equations are

$$1 = \beta E_t \left\{ \left(\frac{c_t}{c_{t+1}} \right)^{\gamma} [(1 + r_{t+1} - \delta)(1-\tau) + 1] \right\}$$
$$c_t = (1-\tau)\left[w_t + (r_t - \delta)k_t\right] + k_t + T_t - k_{t+1}$$
$$W_t = (1-\alpha)e^{(1-\alpha)z_t} K_t^{\alpha} L_t^{-\alpha}$$
$$R_t = \alpha e^{1-\alpha)z_t} K_t^{\alpha-1} L_t^{1-\alpha}$$
$$k_t = K_t$$
$$1 = L_t$$
$$w_t = W_t$$
$$r_t = R_t$$
$$T_t = \tau\left[w_t \ell_t + (r_t - \delta)k_t\right]$$
$$z_t = (1-\rho_z)\bar{z} + \rho_z z_{t-1} + \epsilon_t^z; \quad \epsilon_t^z \sim \text{i.i.d.}(0, \sigma_z^2)$$

Reducing the dimensionality by substituting out variables in capital letters (except

T_t):

$$
\begin{aligned}
1 &= \beta E_t \left\{ \left(\frac{c_t}{c_{t+1}} \right)^{\gamma} [(r_{t+1} - \delta)(1-\tau) + 1] \right\} \\
c_t &= (1-\tau)\left[w_t + (r_t - \delta)k_t\right] + k_t + T_t - k_{t+1} \\
w_t &= (1-\alpha)e^{(1-\alpha)z_t} k_t^{\alpha} \\
r_t &= \alpha e^{1-\alpha)z_t} k_t^{\alpha-1} \\
T_t &= \tau\left[w_t \ell_t + (r_t - \delta)k_t\right]
\end{aligned}
$$

The steady state versions are:

$$
\begin{aligned}
1 &= \beta[(\bar{r} - \delta)(1-\tau) + 1] \\
\bar{c} &= (1-\tau)\left[\bar{w} + (\bar{r} - \delta)\bar{k}\right] + \bar{T} \\
\bar{w} &= (1-\alpha)e^{(1-\alpha)\bar{z}} \bar{k}^{\alpha} \\
\bar{r} &= \alpha e^{(1-\alpha)\bar{z}} \bar{k}^{\alpha-1} \\
\bar{T} &= \tau\left[\bar{w} + (\bar{r} - \delta)\bar{k}\right]
\end{aligned}
$$

These five equations define a steady state in $\bar{r}, \bar{w}, \bar{c}, \bar{k}$ and $\bar{T}$.

Taking the first equation and subsituting the definition of $\bar{r}$:

$$
1 = \beta[(\alpha e^{(1-\alpha)\bar{z}} \bar{k}^{\alpha-1})(1-\tau) + 1]
$$

Solving for $\bar{k}$:

$$
\bar{k} = \left\{ \left[\left(\frac{1}{\beta} - 1 \right) \frac{1}{1-\tau} + \delta \right] \frac{1}{\alpha e^{(1-\alpha)\bar{z}}} \right\}^{\frac{1}{\alpha-1}}
$$

The numerical and analytical solutions are both $\bar{k} = 7.2875$.

In addition we have:

$$
\begin{aligned}
&\bar{y} = 2.2133 \quad \bar{r} = 0.1215 \quad \bar{w} = 1.3280 \\
&\bar{T} = 0.0742 \quad \bar{i} = 0.7287 \quad \bar{c} = 1.4845
\end{aligned}
$$

For this problem (and problem 5b) assume we have run the code below:

```
from __future__ import division
import pandas as pd
import numpy as np
import scipy as sp
from scipy.optimize import fsolve
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

#------------------------------ Define Parameters ----------------------#
gamma = 2.5
beta = 0.98
alpha = 0.40
delta = 0.10
zbar = 0.0
tau = 0.05
xi = 1.5 # used in solving problem 6.
a = 0.5 # used in solving problem 6.
param_vec = np.array([delta, tau, zbar, gamma, beta, alpha, xi, a])
```

The solution for this problem was computed using the following code:

```
#------------------------ #5a: Solving for Steady State ----------------#
def state_defs5a(kbar):
    """
    This function solves for each of the steady state variables as a
    function of kbar, which is steady state captial.

    Parameters
    ----------
    kbar: number, float
        The steady state value of the capital stock

    Returns
    -------
    arr: array, dtype = float, shape = (6 x 1)
        The value of each of the variables as a function of steady
        state capital. The values are ordered as follows:
            y: output
            i: investment
            c: consumption
            r: interest rate
            w: wage
            T: government transfers
    """
    y = kbar ** alpha * np.exp(zbar) ** (1 - alpha)
    i = kbar - (1 - delta) * kbar
    r = alpha * np.exp((1 - alpha) * zbar) * kbar ** (alpha - 1)
    w = (1 - alpha) * np.exp((1 - alpha) * zbar) * kbar ** (alpha)
    T = tau * (w + (r - delta) * kbar)
    c = (1 - tau) * (w + (r - delta) * kbar) + T

    return np.array([y, i, c, r, w, T])


def opt_func5a(guess):
    """
    This function is what allows us to solve for the steady state
    capital stock. We will be minimizing the error in the euler
    equation and passing that to fsolve.

    Parameters
    ----------
    guess: number, float
       The guess for the steady state captial stock.

    Returns
    -------
    eul_err: number, float
       The error in the Euler equation using the specified value of
       kbar.
    """

    kbar = guess
    r = alpha * np.exp((1 - alpha) * zbar) * kbar ** (alpha - 1)

    eul_err = beta * ((r - delta) * (1 - tau) + 1) - 1
    return eul_err

initial_guess5 = 8.0

kbar5a = fsolve(opt_func5a, initial_guess5)
ss5a_1 = state_defs5a(kbar5a)
ybar5, ibar5, cbar5, rbar5, wbar5, Tbar5 = ss5a_1
ss5a = np.append(kbar5a, ss5a_1) # append kbar to front of
ss5a_series = pd.Series(ss5a, index=['kbar', 'ybar', 'ibar', 'cbar',
```

Homework & Lab 5b

For the model in section 3 of the notes consider the following functional forms:

$$\begin{aligned} u(c_t, \ell_t) &= \frac{c_t^{1-\gamma} - 1}{1-\gamma} + a\frac{(1-\ell_t)^{1-\xi} - 1}{1-\xi} \\ F(K_t, L_t, z_t) &= K_t^\alpha (L_t e^{z_t})^{1-\alpha} \end{aligned}$$

Write out all the characterizing equations for the model using these functional forms. Write out the steady state versions of these equations. Numerically solve for the steady state values of all variables using the following parameter values: $\gamma = 2.5$, $\xi = 1.5$, $\beta = .98$, $\alpha = .40$, $a = .5$, $\delta = .10$, $\bar{z} = 0$, $a = 2$ and $\tau = .05$. Also solve for the steady state values of output and investment.

Answer:

Equations are

$$\begin{aligned} 1 &= \beta E_t \left\{ \left(\frac{c_t}{c_{t+1}} \right)^\gamma [(1 + r_{t+1} - \delta)(1-\tau) + 1] \right\} \\ 1 &= \frac{w_t(1-\tau)(1-\ell_t)^\xi}{a c_t^\gamma} \\ c_t &= (1-\tau)\left[w_t \ell_t + (r_t - \delta) k_t \right] + k_t + T_t - k_{t+1} \\ W_t &= (1-\alpha) e^{(1-\alpha) z_t} K_t^\alpha L_t^{-\alpha} \\ R_t &= \alpha e^{1-\alpha) z_t} K_t^{\alpha-1} L_t^{1-\alpha} \\ k_t &= K_t \\ \ell_t &= L_t \\ w_t &= W_t \\ r_t &= R_t \\ T_t &= \tau \left[w_t \ell_t + (r_t - \delta) k_t \right] \\ z_t &= (1-\rho_z)\bar{z} + \rho_z z_{t-1} + \epsilon_t^z; \quad \epsilon_t^z \sim \text{i.i.d.}(0, \sigma_z^2) \end{aligned}$$

Reducing the dimensionality by substituting out variables in capital letters (except

T_t):

$$
\begin{aligned}
1 &= \beta E_t\left\{\left(\frac{c_t}{c_{t+1}}\right)^{\gamma}(r_{t+1}-\delta)(1-\tau)+1\right\}\\
1 &= \frac{w_t(1-\tau)(1-\ell_t)^{\xi}}{ac_t^{\gamma}}\\
c_t &= (1-\tau)\left[w_t\ell_t+(r_t-\delta)k_t\right]+k_t+T_t-k_{t+1}\\
w_t &= (1-\alpha)e^{(1-\alpha)z_t}k_t^{\alpha}\ell_t^{-\alpha}\\
r_t &= \alpha e^{1-\alpha)z_t}k_t^{\alpha-1}\ell_t^{1-\alpha}\\
T_t &= \tau\left[w_t\ell_t+(r_t-\delta)k_t\right]
\end{aligned}
$$

The steady state versions are:

$$
\begin{aligned}
1 &= \beta[(\bar{r}-\delta)(1-\tau)+1]\\
1 &= \frac{\bar{w}(1-\tau)(1-\bar{\ell})^{\xi}}{a\bar{c}^{\gamma}}\\
\bar{c} &= (1-\tau)\left[\bar{w}\bar{\ell}+(\bar{r}-\delta)\bar{k}\right]+\bar{T}\\
\bar{w} &= (1-\alpha)e^{(1-\alpha)\bar{z}}\bar{k}^{\alpha}\bar{\ell}^{-\alpha}\\
\bar{r} &= \alpha e^{(1-\alpha)\bar{z}}\bar{k}^{\alpha-1}\bar{\ell}^{1-\alpha}\\
\bar{T} &= \tau\left[\bar{w}\bar{\ell}+(\bar{r}-\delta)\bar{k}\right]
\end{aligned}
$$

These six equations define a steady state in $\bar{r}, \bar{w}, \bar{c}, \bar{k}, \bar{\ell}$ and $\bar{T}$.

Numerical solutions yield:

$$
\begin{array}{llll}
\bar{k} = 4.2252 & \bar{\ell} = 0.5798 & \bar{y} = 1.2832 & \bar{r} = 0.1215\\
\bar{w} = 1.3280 & \bar{T} = 0.0430 & \bar{i} = 0.4225 & \bar{c} = 0.8607
\end{array}
$$

```python
#------------------------ #5b: Solving for Steady State ----------------#
def state_defs5b(kbar, lbar, parameters=param_vec):
    """
    This function solves for each of the steady state variables as a
    function of kbar and lbar, which are steady state captial
    and labor, respectively.

    Parameters
    ----------
    kbar: number, float
```

```
            The steady state value of the capital stock

        lbar: number, float
            The steady state value of labor.

        Returns
        -------
        arr: array, dtype=float, shape=(6 x 1)
            The of each of the above definitions as a function of the
            state. The values are ordered as follows:
                y: output
                i: investment
                c: consumption
                r: interest rate
                w: wage
                T: government transfers
        """
        delta, tau, zbar, gamma, beta, alpha, xi, a = parameters
        y = (kbar ** alpha) * (np.exp(zbar) * lbar ** (1 - alpha))
        i = kbar - (1 - delta) * kbar
        r = (alpha) * y / kbar
        w = (1 - alpha) * y / lbar
        T = tau * (w * lbar + (r - delta) * kbar)
        c = (1 - tau) * (w * lbar + (r-delta) * kbar) + T

        return np.array([y, i, c, r, w, T])


    def opt_func5b(guess, params=param_vec):
        """
        This function uses the characterizing equations and Euler
        equations to numerically solve for the steady state. It will be
        passed to the routine in scipy.optimize.fsolve.

        Parameters
        ----------
        guess: array, dtype = float, shape = (2 x 1)
            This is a two element array containing an inital guess for
            the steady state value of capital and labor as the first and
            second elements, respectively.

        Returns
        -------
        Eul_err: array, dtype = float, shape = (2 x 1)
            This is the error in the Euler equations that the current
            value of the parameters yeilds.

        Notes
        -----
        We make the assumption that the error in the Euler equations
        should go to zero in the stead state. That is why this function
        returns Eul_err.
        """
        delta, tau, zbar, gamma, beta, alpha, xi, a = params
        kbar, lbar = guess
        y, i, c, r, w, T = state_defs5b(kbar, lbar, params)

        eul_err1 = beta * ((r - delta) * (1 - tau) + 1) - 1
        eul_err2 = (w * (1 - tau) * (1 - lbar) ** xi) / (a * (c ** gamma)) - 1
    return np.array([eul_err1, eul_err2])

initial_guess = np.array([4.5, 0.4])

solution = fsolve(opt_func5b, initial_guess, full_output=True)
Xbar6 = kbar6, lbar6 = solution[0]
errors = solution[1]['fvec']
ss5b_1 = state_defs5b(kbar6, lbar6)
ybar6, ibar6, cbar6, rbar6, wbar6, Tbar6 = ss5b_1
```

Homework 6

For the steady state in section 3.7 of the notes use total differentiation to solve for the full set of comparative statics and sign them where possible. Find $\frac{\partial y}{\partial x}$ for $y \in \{\bar{k}, \bar{w}, \bar{r}, \bar{\ell}\}$ and $x \in \{\delta, \tau, \bar{z}\}$.

Assume $f_K > 0$, $f_{KK} < 0$, $f_L > 0$, $f_{LL} < 0$, $u_c > 0$, $u_{cc} < 0$, $u_\ell < 0$ and $u_{\ell\ell} > 0$.

Answer:

Recall the steady state equations from section 3.7:

$$\bar{c} = (1-\tau)\left[\bar{w}\bar{\ell} + (\bar{r}-\delta)\bar{k}\right] + \bar{T}$$

$$u_c(\bar{c}, \bar{\ell}) = \beta\{u_c(\bar{c}, \bar{\ell})[(\bar{r}-\delta)(1-\tau)+1]\}$$

$$- u_\ell(\bar{c}, \bar{\ell}) = u_c(\bar{c}, \bar{\ell})\bar{w}(1-\tau) = 0$$

$$(1-\tau)(\bar{r}+\delta) = f_K(\bar{k}, \bar{\ell}, \bar{z})$$

$$(1-\tau)\bar{w} = f_L(\bar{k}, \bar{\ell}, \bar{z})$$

$$\tau\left[\bar{w}\bar{\ell} + (\bar{r}-\delta)\bar{k}\right] = \bar{T}$$

Totally differentiating (NEEDS CORRECTING):

$$\begin{aligned} 0 = &- d\bar{c} - \left[\bar{w}\bar{\ell} + (\bar{r}-\delta)\bar{k}\right] d\tau + (1-\tau)\bar{\ell}\, d\bar{w} + (1-\tau)\bar{w}\, d\bar{\ell} \\ &+ (1-\tau)(\bar{r}-\delta)\, d\bar{k} + (1-\tau)\bar{k}\, d\bar{r} - (1-\tau)\bar{k}\, d\delta + d\bar{T} \end{aligned}$$

$$\begin{aligned} 0 = &- \bar{u}_{cc}d\bar{c} - \bar{u}_{c\ell}d\bar{\ell} + \beta[(\bar{r}-\delta)(1-\tau)+1]\bar{u}_{cc}d\bar{c} + \beta[(\bar{r}-\delta)(1-\tau)+1]\bar{u}_{c\ell})d\bar{\ell} \\ &+ \beta\bar{u}_c(1-\tau)d\bar{r} - \beta\bar{u}_c(1-\tau)d\delta - \beta\bar{u}_c(\bar{r}-\delta)d\tau \end{aligned}$$

$$0 = \bar{u}_{\ell\ell}d\bar{\ell} + \bar{u}_{\ell c}d\bar{c} + \bar{w}(1-\tau)\bar{u}_{c\ell}d\bar{\ell} + \bar{w}(1-\tau)\bar{u}_{cc}d\bar{c} + \bar{u}_c(1-\tau)d\bar{w} - \bar{u}_c\bar{w}d\tau$$

$$0 = -\,d\bar{r} - d\delta + \bar{f}_{KK}d\bar{k} + \bar{f}_{KL}d\bar{\ell} + \bar{f}_{Kz}d\bar{z}$$

$$0 = -\,d\bar{w} + \bar{f}_{LK}d\bar{k} + \bar{f}_{LL}d\bar{\ell} + \bar{f}_{Lz}d\bar{z}$$

$$\begin{aligned}0 = &-[\bar{w}\bar{\ell} + (\bar{r}-\delta)\bar{k}]d\tau - \tau\bar{w}d\bar{\ell} - \tau\bar{\ell}d\bar{w} - \tau(\bar{r}-\delta)d\bar{k}\\ &- \tau\bar{k}d\bar{r} + \tau\bar{k}d\delta + d\bar{T}\end{aligned}$$

In matrix format:

$$\begin{bmatrix} -(1-\tau)(\bar{r}-\delta) & -(1-\tau)\bar{\ell} & -(1-\tau)\bar{k} & -(1-\tau)\bar{w} & 1 & -1 \\ 0 & 0 & -\beta\bar{u}_c(1-\tau) & \bar{u}_{c\ell}\beta(\bar{r}-\delta)(1-\tau) & \bar{u}_{cc}\beta(\bar{r}-\delta)(1-\tau) & 0 \\ 0 & -\bar{u}_c(1-\tau) & 0 & -\bar{u}_{\ell\ell} - \bar{w}(1-\tau)\bar{u}_{c\ell} & -\bar{u}_{\ell c} & 0 \\ -\bar{f}_{KK} & 0 & 1 & -\bar{f}_{KL} & 0 & 0 \\ -\bar{f}_{LK} & 1 & 0 & -\bar{f}_{LL} & 0 & 0 \\ \tau(\bar{r}-\delta) & \tau\bar{\ell} & \tau\bar{k} & \tau\bar{w} & 0 & 1 \end{bmatrix} \begin{bmatrix} d\bar{k} \\ d\bar{w} \\ d\bar{r} \\ d\bar{\ell} \\ d\bar{c} \\ d\bar{T} \end{bmatrix}$$

$$= \begin{bmatrix} -(1-\tau)\bar{k} & \bar{w}\bar{\ell} + (\bar{r}-\delta)\bar{k} & 0 \\ -\beta\bar{u}_c(1-\tau) & -\beta\bar{u}_c(\bar{r}-\delta) & 0 \\ 0 & -\bar{u}_c\bar{w} & 0 \\ -1 & 0 & \bar{f}_{Kz} \\ 0 & 0 & \bar{f}_{Lz} \\ \tau\bar{k} & -\bar{w}\bar{\ell} - (\bar{r}-\delta)\bar{k} & 0 \end{bmatrix} \begin{bmatrix} d\delta \\ d\tau \\ d\bar{z} \end{bmatrix}$$

Rewriting in matrix notation:

$$A\,dy = B\,dx$$
$$dy = A^{-1}B\,dx$$

Taking the inverse of A is no fun. Substitution is a just-as-not-fun alternative. Good time to learn MAPLE.

Lab 7a

For the Brock and Mirman model in section 4 set up a discrete grid for K with 100 values ranging from .0001 to $5\bar{K}$. Also set up a discrete grid for z with 100 values ranging from -5σ to $+5\sigma$. Set up a value function array, V that stores the value for all 10,000 possible permuations of K and z. Also set up a policy function array, H, that stores the optimal index value of K' for all all 10,000 possible permuations of K and z.

To begin assume that all elements of V are zero and that all elements of H point to the lowest possible value for K (.0001).

Loop over all possible values of K and z and for each combination find 1) the optimal value of K' from the 100 possible values. Store this value in an updated policy function array, H_{new}. Also find 2) the value implied by this choice given the current value function. Store this in an updated value function array, V_{new}.

Once this is completed for all K and z check to see if V is approzimately equal to V_{new}. If so, output the value function and policy function arrays. If not, replace V with V_{new} and H with H_{new} and repeat the search above.

When finished plot the three-dimensional surface plot for the policy function $K' = H(K, z)$. Compare this with the closed form solution from the notes.

Answer:

The plot of the policy functions can be seen in Figure 1.

The code used to generate the solution appears on the next page.

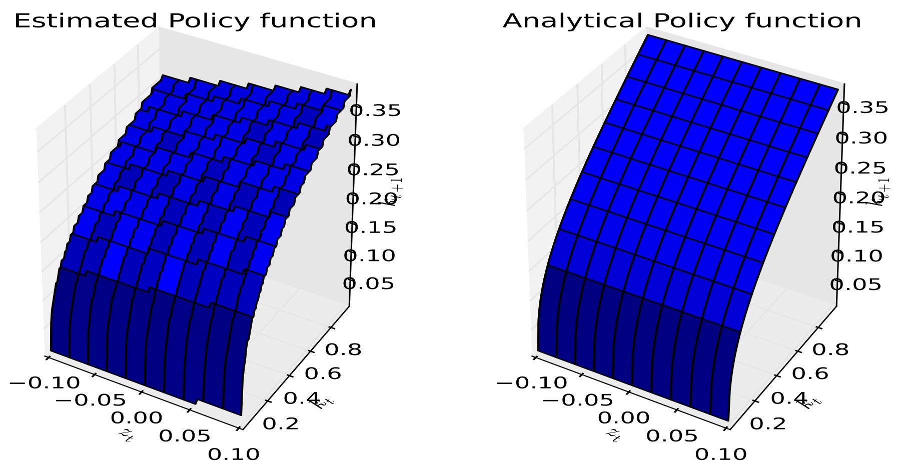

Figure 1: Plot of policy function obtained via dynamic programming and analytic expression.

```
#--------------------------- #7a: Dynamic Programming --------------------#
alpha = 0.35
beta = 0.98
rho = 0.95
sigma = 0.02

A = alpha * beta

kbar = A ** (1. / (1 - alpha))

ksize = 100
zsize = 100

k_grid = np.linspace(0.0001, 5 * kbar, ksize)
z_grid = np.linspace(-5 * sigma, 5 * sigma, zsize)
z_motion = sigma * np.array([-np.sqrt(3. / 2), 0, np.sqrt(3. / 2)])

epsilon = z_motion[np.random.randint(0, 3, zsize)]

values = pd.DataFrame(np.zeros((zsize, ksize)), index=z_grid,
                      columns=k_grid)
values.columns.name = 'k'
values.index.name = 'z'

vf = np.ones((zsize, ksize))
vfp = np.zeros((zsize, ksize))
pol = np.zeros((zsize, ksize), dtype=np.int8)

# Set up iteration parameters
max_its = 500
its = 0
tol = 1e-6
```

```
dist = 1.

while dist > tol and its < max_its:
    for zi in range(zsize):
        # NOTE: I put the z loop outside because we do some stuff with just
        #       z that we don't need to repeat for every k.

        z = z_grid[zi]

        # Get "random" shocks following law of motion for z
        znew = z * rho + z_motion

        # Get the index of z_grid closest to each element of znew
        z_inds = []
        for i in znew:
            z_inds.append(np.argmin(np.abs(z_grid - i)))

        for ki in range(ksize):
            k = k_grid[ki]

            # Utility using this k, z all possible k_{t+1}
            u = np.log(np.exp(z) * k**alpha - k_grid)

            # Calculate expected value
            e_v = 1 / 3. * np.asarray([vf[i, :] for i in
                z_inds]).sum(axis=0)

            # evaluate bellman function
            bell = u + beta * e_v

            # Get optimal policy and value
            v_star = np.nanmax(bell)
            p_star = np.nanargmax(bell)

            # Fill in the vprime and policy arrays
            vfp[zi, ki] = v_star
            pol[zi, ki] = p_star

    # Check for convergence before updating value function
    dist = np.abs(vf - vfp).max()

    # Update value function and make sure we create new array
    vf = np.array(vfp, dtype=float)
    its += 1
    if its % 3 == 0:
        print('On iteration %i the max difference is %.3e' % (its, dist))

pol_func = np.zeros((zsize, ksize))
for zi in xrange(zsize):
    for ki in xrange(ksize):
        pol_func[zi, ki] = k_grid[pol[zi, ki]]

real_pol = alpha * beta * np.exp(z_grid) * k_grid ** alpha

k_mesh, z_mesh = np.meshgrid(k_grid, z_grid)

fig = plt.figure()

ax1 = fig.add_subplot(121, projection='3d')
```

```
ax2 = fig.add_subplot(122, projection='3d')

# plot numerical estimate

ax1.plot_surface(z_mesh, k_mesh, pol_func)
ax1.set_xlabel(r'$z_t$')
ax1.set_ylabel(r'$k_t$')
ax1.set_zlabel(r'$k_{t+1}$')
ax1.set_title('Estimated Policy function')

# plot analytical solution
ax2.plot_surface(z_mesh, k_mesh, real_pol)
ax2.set_xlabel(r'$z_t$')
ax2.set_ylabel(r'$k_t$')
ax2.set_zlabel(r'$k_{t+1}$')
ax2.set_title('Analytical Policy function')
plt.savefig('dsge7a.eps', format='eps', dpi=1000)
plt.show()
```

Lab 7b

Repeat the above exercise using $k \equiv \ln K$ in place of K as the endogenous state variable.

NOTE: This solution is not totally correct. It it close, but not completely accurate.

The plot of the policy functions can be seen in Figure 2.

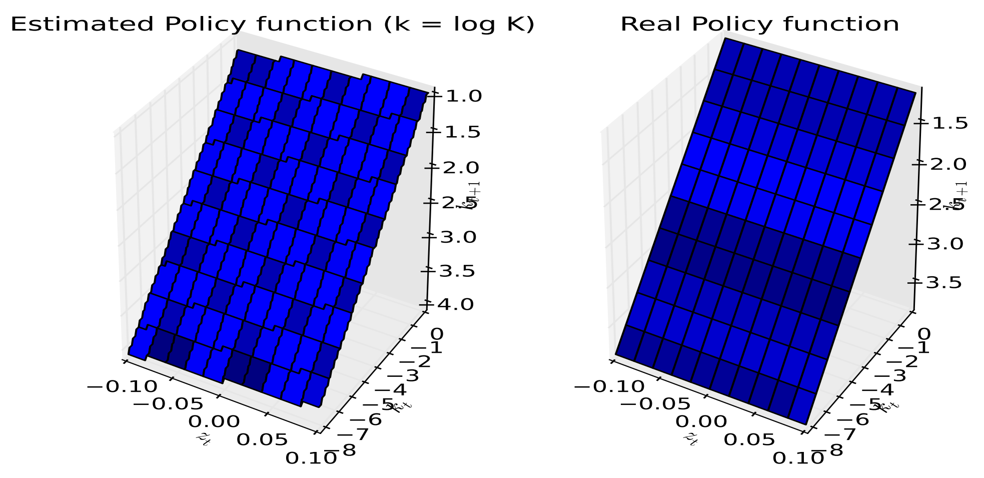

Figure 2: Plot of policy function obtained via dynamic programming when substituting $k = \log K$. Also, a plot of the and analytic expression is given.

The code used to generate the solution appears on the next page.

```
#--------------------------- #7b: Dynamic Programming --------------------#
alpha = 0.35
beta = 0.98
rho = 0.95
sigma = 0.02

A = alpha * beta

kbar = A ** (1. / (1 - alpha))

ksize = 100
zsize = 100

lk_grid = np.linspace(0.0001, 5 * np.log(kbar), ksize)
z_grid = np.linspace(-5 * sigma, 5 * sigma, zsize)
z_motion = sigma * np.array([-np.sqrt(3. / 2), 0, np.sqrt(3. / 2)])

epsilon = z_motion[np.random.randint(0, 3, zsize)]

values = pd.DataFrame(np.zeros((zsize, ksize)), index=z_grid,
                    columns=lk_grid)
values.columns.name = 'k'
values.index.name = 'z'

lvf = np.ones((zsize, ksize))
lvfp = np.zeros((zsize, ksize))
lpol = np.zeros((zsize, ksize), dtype=np.int8)

# Set up iteration parameters
max_its = 500
its = 0
tol = 1e-6
dist = 1.

while dist > tol and its < max_its:
    for zi in range(zsize):
        # NOTE: I put the z loop outside because we do some stuff with just
        #       z that we don't need to repeat for every k.

        z = z_grid[zi]

        # Get "random" shocks following law of motion for z
        znew = z * rho + z_motion

        # Get the index of z_grid closest to each element of znew
        z_inds = []
        for i in znew:
            z_inds.append(np.argmin(np.abs(z_grid - i)))

        for ki in range(ksize):
            k = np.exp(lk_grid[ki])

            # Utility using this k, z all possible k_{t+1}
            u = np.log(np.exp(z) * k**alpha - np.exp(lk_grid))

            # Calculate expected value
            e_v = 1 / 3. * np.asarray([lvf[i, :] for i in
                z_inds]).sum(axis=0)
            # evaluate bellman function
            bell = u + beta * e_v

            # Get optimal policy and value
            v_star = np.nanmax(bell)
            p_star = np.nanargmax(bell)

            # Fill in the vprime and policy arrays
            lvfp[zi, ki] = v_star
            lpol[zi, ki] = p_star

    # Check for convergence before updating value function
    dist = np.abs(lvf - lvfp).max()

    # Update value function and make sure we create new array
    lvf = np.array(lvfp, dtype=float)
    its += 1
    if its % 3 == 0:
        print('On iteration %i the max difference is %.3e' % (its, dist))

lpol_func = np.zeros((zsize, ksize))
for zi in xrange(zsize):
    for ki in xrange(ksize):
        lpol_func[zi, ki] = lk_grid[lpol[zi, ki]]

real_pol = np.log(alpha * beta * np.exp(z_grid) * np.exp(lk_grid) ** alpha)

lk_mesh, z_mesh = np.meshgrid(lk_grid, z_grid)

fig = plt.figure()

ax1 = fig.add_subplot(121, projection='3d')
ax2 = fig.add_subplot(122, projection='3d')

# plot numerical estimate
ax1.plot_surface(z_mesh, lk_mesh, lpol_func)
ax1.set_xlabel(r'$z_t$')
ax1.set_ylabel(r'$k_t$')
ax1.set_zlabel(r'$k_{t+1}$')
ax1.set_title('Estimated Policy function (k = log K)')

# plot analytical solution
ax2.plot_surface(z_mesh, lk_mesh, real_pol)
ax2.set_xlabel(r'$z_t$')
ax2.set_ylabel(r'$k_t$')
ax2.set_zlabel(r'$k_{t+1}$')
ax2.set_title('Real Policy function')
plt.savefig('dsge7b.eps', format='eps', dpi=1000)
plt.show()
```

Lab 8a

For the Brock and Mirman model in section 4 use Uhlig's notation to analytically find the values of the following matrices: F, G, H, L, M & N as functions of the parameters. Given these find the values of $P\&Q$, also as functions of the parameters. Imposing our calbrated parameter values, plot the three-dimensional surface plot for the policy function $K' = H(K, z)$. Compare this with the closed form solution fromthe notes and the solution you found using the grid search method.

Answer:

Let us first solve for the linearized approximation, not the log-linearized one. Recall the Euler equation from the Brock and Mirman model.

$$\frac{1}{e^{z_t}K_t^{\alpha} - K_{t+1}} = \beta E_t\left\{\frac{e^{z_{t+1}}{K_{t+1}}^{\alpha-1}}{e^{z_{t+1}}{K_{t+1}}^{\alpha} - K_{t+2}}\right\}$$

We rewrite this equation and put it in the form of (**??**).

$$E_t\left\{\beta[e^{z_{t+1}}{K_{t+1}}^{\alpha-1}(e^{z_t}K_t^{\alpha} - K_{t+1}) - e^{z_{t+1}}{K_{t+1}}^{\alpha} + K_{t+2}]\right\} = 0 \tag{1}$$

By differentiating (**??**) with respect to K_{t+2}, K_{t+1}, K_t, z_{t+1} and z_t we can recover the Uhlig matrices:

$$\begin{aligned}
F =& \beta \\
G =& \beta e^{z_{t+1}}K_{t+1}^{\alpha-1}\left[K_{t+1}^{-1}(\alpha-1)(e^{z_t}K_t^{\alpha} - K_{t+1}) - 1 - \alpha\right] \\
H =& \beta e^{z_{t+1}}K_{t+1}^{\alpha-1}e^{z_t}K_t^{\alpha-1}\alpha \\
L =& \beta\left[e^{z_{t+1}}K_{t+1}^{\alpha-1}(e^{z_t}K_t^{\alpha} - K_{t+1}) - (e^{z_{t+1}}K_{t+1}^{\alpha} - K_{t+2})\right] \\
M =& \beta e^{z_{t+1}}K_{t+1}^{\alpha-1}e^{z_t}K_t^{\alpha-1}
\end{aligned}$$

Evaluating these at their steady state values gives:

$$\begin{aligned}
F =& \beta \\
G =& \beta \bar{K}^{\alpha-1}\left[\bar{K}^{-1}(\alpha-1)(\bar{K}^{\alpha}-\bar{K})-1-\alpha\right] \\
H =& \beta \bar{K}^{2(\alpha-1)}\alpha \\
L =& \beta\left[\bar{K}^{\alpha-1}(\bar{K}^{\alpha}-\bar{K})-(\bar{K}^{\alpha}-\bar{K})\right] \\
M =& \beta \bar{K}^{2(\alpha-1)}
\end{aligned}$$

To evaluate (6) recall $\bar{K} = A^{\frac{1}{1-\alpha}}$. We can then use equations for P and Q given in the notes to derive the scalar values P and Q.

$$P = \frac{-G \pm \sqrt{G^2 - 4FH}}{2F} \tag{2}$$

$$\tag{3}$$

$$Q = -\frac{LN + M}{FN + FP + G} \tag{4}$$

Plugging things in we get the following two options (note quadratic) for P (sorry for the small font size- these solutions are ugly):

$$P_+ = \alpha\bar{K}^{\alpha-1} - \frac{1}{2}\alpha\bar{K}^{2\alpha-2} + \frac{\sqrt{\bar{K}^{2\alpha+2}\left(4\alpha^2\bar{K}^6 - 4\alpha^2\bar{K}^{\alpha+5} + \alpha^2\bar{K}^{2\alpha+4} - 4\alpha\bar{K}^6 + 4\alpha\bar{K}^{\alpha+5} - 2\alpha\bar{K}^{2\alpha+4} + \bar{K}^{2\alpha+4}\right)}}{2\bar{K}^5} + \frac{1}{2}\bar{K}^{2\alpha-2}$$

$$P_- = \frac{-1}{8\bar{K}^5}\left[-8\alpha\bar{K}^5\bar{K}^{\alpha-1} + 4\alpha\bar{K}^5\bar{K}^{2\alpha-2} - 4\bar{K}^5\bar{K}^{2\alpha-2} + 4\sqrt{\bar{K}^{2\alpha+2}\left(4\alpha^2\bar{K}^6 - 4\alpha^2\bar{K}^{\alpha+5} + \alpha^2\bar{K}^{2\alpha+4} - 4\alpha\bar{K}^6 + 4\alpha\bar{K}^{\alpha+5} - 2\alpha\bar{K}^{2\alpha+4} + \bar{K}^{2\alpha+4}\right)}\right]$$

Plugging things in we get the following for Q. Note that I define

$$AAA = \sqrt{\bar{K}^{2\alpha+2}\left(4\alpha^2\bar{K}^6 - 4\alpha^2\bar{K}^{\alpha+5} + \alpha^2\bar{K}^{2\alpha+4} - 4\alpha\bar{K}^6 + 4\alpha\bar{K}^{\alpha+5} - 2\alpha\bar{K}^{2\alpha+4} + \bar{K}^{2\alpha+4}\right)}$$

$$Q_+ = \frac{-\beta\bar{K}^{2\alpha-2} - 0.02\beta\left(\bar{K} - \bar{K}^{\alpha} + \bar{K}^{\alpha-1}\left(-\bar{K}+\bar{K}^{\alpha}\right)\right)}{\beta\bar{K}^{\alpha-0}\left(-\alpha - 1 + \frac{(\alpha-1)(-\bar{K}+\bar{K}^{\alpha})}{\bar{K}}\right) + \beta\left(\alpha\bar{K}^{\alpha-0} - \frac{1}{2}\alpha\bar{K}^{2\alpha-0} + \frac{AAA}{2\bar{K}^0} + \frac{1}{2}\bar{K}^{2\alpha-0}\right) + 0.02\beta}$$

$$Q_- = \frac{-\beta \bar{K}^{2\alpha-2} - 0.02\beta \left(\bar{K} - \bar{K}^{\alpha} + \bar{K}^{\alpha-1}\left(-\bar{K} + \bar{K}^{\alpha}\right)\right)}{-\frac{\beta\left(-8\alpha\bar{K}^5\bar{K}^{\alpha-1} + 4\alpha\bar{K}^5\bar{K}^{2\alpha-2} - 4\bar{K}^5\bar{K}^{2\alpha-2} + 4AAA\right)}{8\bar{K}^5} + \beta\bar{K}^{\alpha-1}\left(-\alpha - 1 + \frac{(\alpha-1)(-\bar{K}+\bar{K}^{\alpha})}{\bar{K}}\right) + 0.02\beta}$$

In this case, since we have linearized (not log-linearized) the policy function is:

$$K_{t+1} = \bar{K} + P(K_t - \bar{K}) + Qz_t \tag{5}$$

The plot of the policy functions can be seen in Figure 3.

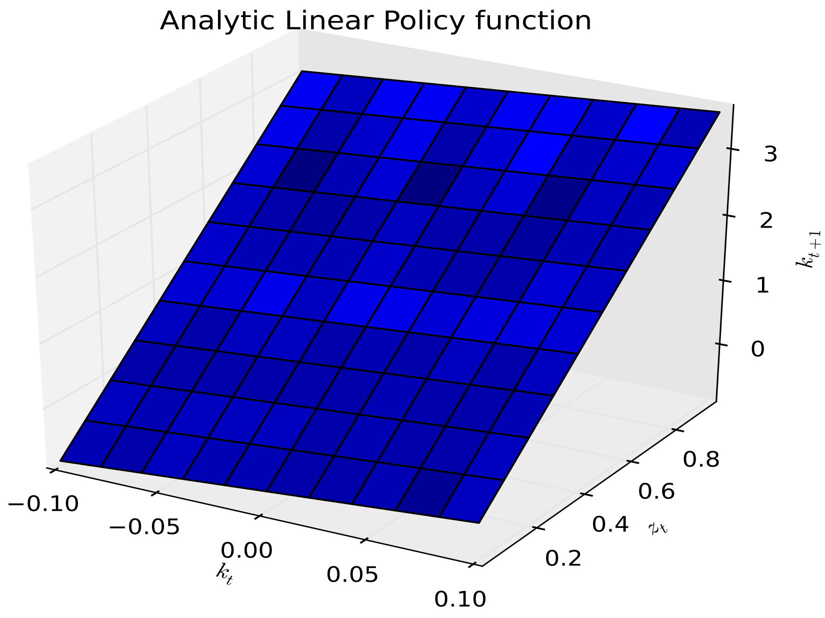

Figure 3: Plot of analytical solution for the policy function via linearization..

The code used to generate the solution appears on the next page.

```python
#---------------------- #8a: Linearizing Brock and Mirman ----------------#
alpha = 0.35
beta = 0.98
rho = 0.95
sigma = 0.02

A = alpha * beta

kbar = A ** (1. / (1 - alpha))

ksize = 100
zsize = 100

k_grid = np.linspace(0.0001, 5 * kbar, ksize)
z_grid = np.linspace(-5 * sigma, 5 * sigma, zsize)

F = beta
G = beta * kbar**(alpha - 1.) * \
    ((alpha - 1) * (kbar**alpha - kbar) / kbar - 1 - alpha)
H = beta * kbar ** (2 * (alpha - 1)) * alpha
L = beta * (kbar**(alpha - 1) * (kbar**alpha - kbar) - (kbar**alpha -
    kbar))
M = beta * kbar ** (2 * (alpha - 1))
N = sigma

P = -G - np.sqrt(G**2 - 4 * F * H) / (2*F)
Q = - (L*N + M) / (F*N + F*P + G)

linear_policy = lambda kt, zt: kbar + P *(kt - kbar) + Q * zt

k_mesh, z_mesh = np.meshgrid(k_grid, z_grid)

# Plot the analytic linearized solution
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(z_mesh, k_mesh, linear_policy(k_mesh, z_mesh))
ax.set_xlabel(r'$k_t$')
ax.set_ylabel(r'$z_t$')
ax.set_zlabel(r'$k_{t+1}$')
ax.set_title('Analytic Linear Policy function')
plt.savefig('dsge8a.eps', format='eps', dpi=1000)
plt.show()
```

Lab 8b

Repeat the above exercise using $k \equiv \ln K$ in place of K as the endogenous state variable.

Answer:

In this case we are simply log-linearizing the model. We can follow the corresponding section from the notes to do this. I have reproduced that content here.

To log-linearize we must replace K_t with $\bar{K}e^{\tilde{K}_t}$. We do this for (1) and obtain:

$$E_t\left\{\beta[\bar{K}^{\alpha-1}e^{z_{t+1}+(\alpha-1)\tilde{K}_{t+1}}(\bar{K}^{\alpha}e^{z_t+\alpha\tilde{K}_t}-\bar{K}e^{\tilde{K}_{t+1}})-\bar{K}^{\alpha}e^{z_{t+1}+\alpha\tilde{K}_{t+1}}+\bar{K}e^{\tilde{K}_{t+2}}]\right\}=0$$

We rewrtite this as:

$$E_t\left\{\begin{matrix}\bar{K}^{-1}e^{z_{t+1}+(\alpha-1)\tilde{K}_{t+1}+z_t+\alpha\tilde{K}_t}\\ -\bar{K}^{\alpha}e^{z_{t+1}+(\alpha-1)\tilde{K}_{t+1}+\tilde{K}_{t+1}}\\ -\bar{K}^{\alpha}e^{z_{t+1}+\alpha\tilde{K}_{t+1}}\\ +\bar{K}e^{\tilde{K}_{t+2}}\end{matrix}\right\}=0$$

We rewrtite this further as:

$$E_t\left\{\begin{matrix}\bar{K}^{-1}(1+z_{t+1}+(\alpha-1)\tilde{K}_{t+1}+z_t+\alpha\tilde{K}_t)\\ -\bar{K}^{\alpha}(1+z_{t+1}+(\alpha-1)\tilde{K}_{t+1}+\tilde{K}_{t+1})\\ -\bar{K}^{\alpha}(1+z_{t+1}+\alpha\tilde{K}_{t+1})\\ +\bar{K}(1+\tilde{K}_{t+2})\end{matrix}\right\}=0$$

Note that the steady state version of (1) implies that $\bar{K}^{-1}-\bar{K}^{\alpha}-\bar{K}^{\alpha}+\bar{K}=0$. This allows us to again rewrite as:

$$E_t\left\{\begin{matrix}\bar{K}^{-1}(z_{t+1}+(\alpha-1)\tilde{K}_{t+1}+z_t+\alpha\tilde{K}_t)\\ -\bar{K}^{\alpha}(z_{t+1}+(\alpha-1)\tilde{K}_{t+1}+\tilde{K}_{t+1})\\ -\bar{K}^{\alpha}(z_{t+1}+\alpha\tilde{K}_{t+1})\\ +\bar{K}(\tilde{K}_{t+2})\end{matrix}\right\}=0$$

Regrouping terms:

$$E_t \left\{ \begin{matrix} (\bar{K})\tilde{K}_{t+2} \\ (\bar{K}^{-1}(\alpha-1) - 2\bar{K}^{\alpha}\alpha)\tilde{K}_{t+1} \\ (\bar{K}^{-1}\alpha)\tilde{K}_t \\ (\bar{K}^{-1} - 2\bar{K}^{\alpha})z_{t+1} \\ (\bar{K}^{-1})z_t \end{matrix} \right\} = 0$$

This gives:

$$\begin{aligned} F =& \bar{K} \\ G =& \bar{K}^{-1}(\alpha-1) - 2\bar{K}^{\alpha}\alpha \\ H =& \bar{K}^{-1}\alpha \\ L =& \bar{K}^{-1} - 2\bar{K}^{\alpha} \\ M =& \bar{K}^{-1} \end{aligned}$$

Because we log-linearized, the policy function is

$$\tilde{K}_{t+1} = P\tilde{K}_t + Qz_t$$

Also note that because the analytical expressions for P and Q were so ugly (almost incomprehensible) for the previous part I have omitted them here.

The plot of the policy functions can be seen in Figure 4.

The code used to generate the solution appears on the next page.

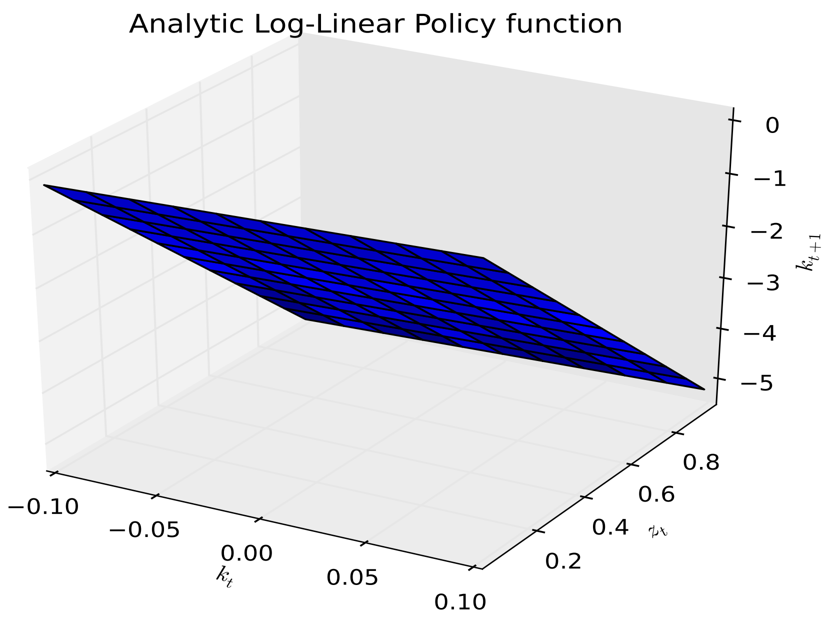

Figure 4: Plot of analytical solution for the policy function via log linearization..

```
#--------------------- #8b: Log-Linearizing Brock and Mirman -------------#
alpha = 0.35
beta = 0.98
rho = 0.95
sigma = 0.02

A = alpha * beta

kbar = A ** (1. / (1 - alpha))


ksize = 100
zsize = 100

k_grid = np.linspace(0.0001, 5 * kbar, ksize)
z_grid = np.linspace(-5 * sigma, 5 * sigma, zsize)

F = kbar
G = (alpha - 1) / kbar - 2 * (kbar**alpha) * alpha
H = alpha / kbar
L = 1. / kbar - 2 * kbar**alpha
M = 1. / kbar
N = sigma

P = -G - np.sqrt(G**2 - 4 * F * H) / (2*F)
Q = - (L*N + M) / (F*N + F*P + G)

log_linear_policy = lambda kt, zt: P * kt + Q * zt
```

```
k_mesh, z_mesh = np.meshgrid(k_grid, z_grid)

# Plot the analytic linearized solution
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(z_mesh, k_mesh, log_linear_policy(k_mesh, z_mesh))
ax.set_xlabel(r'$k_t$')
ax.set_ylabel(r'$z_t$')
ax.set_zlabel(r'$k_{t+1}$')
ax.set_title('Analytic Log-Linear Policy function')
plt.savefig('dsge8b.eps', format='eps', dpi=1000)
plt.show()
```

Lab 9

For the baseline model in section 3 numerically, find $\frac{\partial y}{\partial x}$ for $y \in \{\bar{k}, \bar{w}, \bar{r}, \bar{\ell}\}$ and $x \in \{\delta, \tau, \bar{z}\}$ assuming $u(c_t, \ell_t) = \frac{c_t^{1-\gamma}-1}{1-\gamma} + a\frac{(1-\ell_t)^{1-\xi}-1}{1-\xi}$ and $F(K_t, L_t, z_t) = K_t^{\alpha}(L_t e^{z_t})^{1-\alpha}$. Use the following parameter values: $\gamma = 2.5$, $\xi = 1.5$, $\beta = .98$, $\alpha = .40$, $a = .5$, $\delta = .10$, $\bar{z} = 0$, and $\tau = .05$.

Answer: The derivatives can be looked up in Table 1

	δ	τ	z
k	-48.34984	-2.323225	4.73048
w	-7.287498	-0.1647916	0.8853018
r	1	0.02261292	-0.1214823
l	1.319775	-0.1389237	-0.3171964

Table 1: Table of numerical derivatives

The code used to generate this table is found below:

```python
gamma = 2.5
xi = 1.5
beta = 0.98
alpha = 0.40
a = 0.5
delta = 0.10
zbar = 0.0
tau = 0.05
param_vec = np.array([delta, tau, zbar, gamma, beta, alpha, xi, a])


def state_defs9(kbar, lbar, parameters=param_vec):
    """
    This function solves for each of the steady state variables as a
    function of kbar and lbar, which are steady state captial
    and labor, respectively.

    Parameters
    ----------
    kbar: number, float
        The steady state value of the capital stock

    lbar: number, float
        The steady state value of labor.

    Returns
    -------
    arr: array, dtype=float, shape=(6 x 1)
        The of each of the above definitions as a function of the
        state. The values are ordered as follows:
            y: output
            i: investment
            c: consumption
            r: interest rate
            w: wage
            T: government transfers
    """
    delta, tau, zbar, gamma, beta, alpha, xi, a = parameters
    y = (kbar ** alpha) * (np.exp(zbar) * lbar ** (1 - alpha))
    i = kbar - (1 - delta) * kbar
    r = (alpha) * y / kbar
    w = (1 - alpha) * y / lbar
    T = tau * (w * lbar + (r - delta) * kbar)
    c = (1 - tau) * (w * lbar + (r-delta) * kbar) + T

    return np.array([y, i, c, r, w, T])


def opt_func9(guess, params=param_vec):
    """
    This function uses the characterizing equations and Euler
    equations to numerically solve for the steady state. It will be
    passed to the routine in scipy.optimize.fsolve.

    Parameters
    ----------
    guess: array, dtype = float, shape = (2 x 1)
        This is a two element array containing an inital guess for
        the steady state value of capital and labor as the first and
        second elements, respectively.

    Returns
    -------
    Eul_err: array, dtype = float, shape = (2 x 1)
        This is the error in the Euler equations that the current
        value of the parameters yeilds.

    Notes
    -----
    We make the assumption that the error in the Euler equations
    should go to zero in the stead state. That is why this function
    returns Eul_err.
    """
    delta, tau, zbar, gamma, beta, alpha, xi, a = params
    kbar, lbar = guess
    y, i, c, r, w, T = state_defs9(kbar, lbar, params)

    eul_err1 = beta * ((r - delta) * (1 - tau) + 1) - 1
    eul_err2 = (w * (1 - tau) * (1 - lbar) ** xi) / (a * (c ** gamma)) - 1

    return np.array([eul_err1, eul_err2])

ivars = ['delta', 'tau', 'z']
dvars = ['k', 'w', 'r', 'l']

derivs = pd.DataFrame(index=dvars, columns=ivars)
derivs.index.name = 'Dependent Var'
derivs.columns.name = 'Independent Var'
```

```
# Step size in numerical derivative
h = 1e-6

initial_guess = np.array([4.5, 0.4])

for i in xrange(3):
    # I will be computing numerical derivatives using centered difference
    # approximation. That expression is ((fx + h) - f(x - h)) / 2h.
    # Let's go get that stuff!

    # calculate f(x + h). I can just change element of param_vec because
    # delta, tau, and zbar are the first 3 to appear and that's what we
       need
    param_vec[i] += h
    ss_plus = fsolve(opt_func9, initial_guess, args=(param_vec),
                     full_output=True)[0]
    k_plus, l_plus = ss_plus
    y_plus, i_plus, c_plus, r_plus, w_plus, T_plus = state_defs9(k_plus,
                                                                   l_plus)

    # Now calculate f(x - h)
    param_vec[i] -= 2 * h
    ss_sub = fsolve(opt_func9, initial_guess, args=(param_vec),
                    full_output=True)[0]
    k_sub, l_sub = ss_sub
    y_sub, i_sub, c_sub, r_sub, w_sub, T_sub = state_defs9(k_sub,
                                                            l_sub)

    # Calculate derivatives
    k_der = (k_plus - k_sub) / (2 * h)
    r_der = (r_plus - r_sub) / (2 * h)
    w_der = (w_plus - w_sub) / (2 * h)
    l_der = (l_plus - l_sub) / (2 * h)

    # Stick them in the DataFrame
    derivs.ix['k', ivars[i]] = k_der
    derivs.ix['r', ivars[i]] = r_der
    derivs.ix['w', ivars[i]] = w_der
    derivs.ix['l', ivars[i]] = l_der

    # reset params
    delta = 0.10
    zbar = 0.0
    tau = 0.05
    param_vec = np.array([delta, tau, zbar, gamma, beta, alpha, xi, a])

print(derivs)
print(derivs.to_latex())
```

Homework 10

Do the necessary tedious matrix algebra necessary to transform equation (5.5) into equation (5.8).

Answer: Recall (5.5) is:

$$E_t\left\{F\tilde{X}_{t+1}+G\tilde{X}_t+H\tilde{X}_{t-1}+L\tilde{Z}_{t+1}+M\tilde{Z}_t\right\}=0 \tag{6}$$

The law of motion is:

$$\tilde{Z}_t=N\tilde{Z}_{t-1}+\varepsilon_t \tag{7}$$

The policy function is:

$$\tilde{X}_t=P\tilde{X}_{t-1}+Q\tilde{Z}_t \tag{8}$$

Substitute the $t+1$ version of (8) into (6):

$$\begin{aligned}
E_t\left\{F(P\tilde{X}_t+Q\tilde{Z}_{t+1})+G\tilde{X}_t+H\tilde{X}_{t-1}+L\tilde{Z}_{t+1}+M\tilde{Z}_t\right\} &=0\\
E_t\left\{FP\tilde{X}_t+FQ\tilde{Z}_{t+1}+G\tilde{X}_t+H\tilde{X}_{t-1}+L\tilde{Z}_{t+1}+M\tilde{Z}_t\right\} &=0\\
E_t\left\{(FP+G)\tilde{X}_t+H\tilde{X}_{t-1}++(FQ+L)\tilde{Z}_{t+1}+M\tilde{Z}_t\right\} &=0
\end{aligned}$$

Substitute the $t+1$ version of (7) into this:

$$\begin{aligned}
E_t\left\{(FP+G)\tilde{X}_t+H\tilde{X}_{t-1}+(FQ+L)(N\tilde{Z}_t+\varepsilon_{t+1})+M\tilde{Z}_t\right\} &=0\\
E_t\left\{(FP+G)\tilde{X}_t+H\tilde{X}_{t-1}+(FQ+L)N\tilde{Z}_t+(FQ+L)\varepsilon_{t+1}+M\tilde{Z}_t\right\} &=0\\
E_t\left\{(FP+G)\tilde{X}_t+H\tilde{X}_{t-1}+[(FQ+L)N+M]\tilde{Z}_t+(FQ+L)\varepsilon_{t+1}\right\} &=0
\end{aligned}$$

Substitute the t version of (8) into this:

$$\begin{aligned}
E_t\left\{(FP+G)(P\tilde{X}_{t-1}+Q\tilde{Z}_t)+H\tilde{X}_{t-1}+[(FQ+L)N+M]\tilde{Z}_t+(FQ+L)\varepsilon_{t+1}\right\} &=0\\
E_t\left\{[(FP+G)P+H]\tilde{X}_{t-1}+[(FQ+L)N+(FP+G)Q+M]\tilde{Z}_t+(FQ+L)\varepsilon_{t+1}\right\} &=0
\end{aligned}$$

Distributing the expectations operator:

$$[(FP+G)P+H]\tilde{X}_{t-1}+[(FQ+L)N+(FP+G)Q+M]\tilde{Z}_t \quad =0$$

Lab 11

Assume $n_X = n_Z = 1$. Show how to solve for the elements of P and Q. Write a program in Python that will do this given F, G, H, L, M and N as inputs.

Now assume n_X and n_Z take on arbitrary values greater than one. Show how to solve for the elements of P and Q. Write a program in Python that will do this given F, G, H, L, M and N as inputs.

The MATLAB programs for **?** are available online and may be of some help structuring the Python code. The discussion in the paper itself may also be a big help.

Answer: In his paper Uhlig shows that the solution to $\Psi P^2 - \Gamma P - \Theta = 0$ is found by constructing two matrices:

$$\Xi = \begin{bmatrix} \Gamma & \Theta \\ I_m & 0_{m\times m} \end{bmatrix} \quad \Delta = \begin{bmatrix} \Psi & 0_{m\times m} \\ 0_{m\times m} & I_m \end{bmatrix}$$

The solution involves solving the generalized eignevalue problem defined by:

$$\lambda \Delta s = \Xi s$$

Denote the m generalized eigenvalues as $\{\lambda_i\}_{i=1}^m$ and the generalized eigenvectors as $\{s_i^T\}_{i=1}^m = [\lambda x_i^T, x_i^T]$ for some $x_i \in \Re^m$. The solution for P is given by:

The solution involves solving the generalized eignevalue problem defined by:

$$P = \Omega \Lambda \Omega^T; \quad \Omega = [x_1, x_2, \ldots, x_m], \Lambda = diag(\lambda_1, \lambda_2, \ldots, \lambda_m)$$

Q is found by using:

$$Q = V^{-1}[-vec(LN + M)]; \quad V = N^T \otimes F + I_k \otimes (FP + G)$$

A python program implementing these concepts appears below (this is the uhlig.py the students were given).

```
"""
```

```
Python module for using the method outlined by Uhlig (1997) to solve a
log-linearized RBC model for policy functions.

Original adaptation of MATLAB code done by Spencer Lyon in May 2012

Additional work has been done by Chase Coleman
"""
import scipy as sp
import numpy as np
from numpy import hstack, vstack, zeros, dot, eye, kron
from scipy import linalg as la
from numpy import linalg as npla


def _nullSpaceBasis(A):
    """
    This funciton will find the basis of the null space of the matrix A.

    Parameters
    ----------
    A : array_like, dtype=float
        The matrix you want the basis for

    Returns
    -------
    vecs : array_like, dtype=float
        A numpy matrix containing the vectors as row vectors.

    Notes
    -----
    If A is an empty matrix, an empty matrix is returned.

    """
    if A:
        U, s, Vh = la.svd(A)
        vecs = np.array([])
        toAppend = A.shape[1] - s.size
        s = np.append(s, zeros((1, toAppend)))
        for i in range(0, s.size):
            if s[i] == 0:
                vecs = Vh[-toAppend:, :]
        if vecs.size == 0:
            vecs = zeros((1, A.shape[1]))
        return np.mat(vecs)
    else:
        return zeros((0, 0))


def solvePQRS(AA=None, BB=None, CC=None, DD=None, FF=None, GG=None,
    HH=None,
              JJ=None, KK=None, LL=None, MM=None, NN=None):
    """
    This function mimics the behavior of Harald Uhlig's solve.m and
    calc_qrs.m files in Uhlig's toolkit.

    In order to use this function, the user must have log-linearized the
    model they are dealing with to be in the following form (assume that
    y corresponds to the model's "jump variables", z represents the
    exogenous state variables and x is for endogenous state variables.
    nx, ny, nz correspond to the number of variables in each category.)
    The inputs to this function are the matrices found in the following
    equations.

    .. math::

        Ax_t + Bx_t-1 + Cy_t + Dz_t = 0

        E\{Fx_{t+1} + Gx_t + Hx_{t-1} + Jy_{t+1} +
           Ky_t + Lz_{t+1} Mz_t \} = 0

    The purpose of this function is to find the recursive equilibrium
    law of motion defined by the following equations.

    .. math::

        X_t = PX_{t-1} + Qz_t

        Y_t = RY_{t-1} + Sz_t

    Following outline given in Uhhlig (1997), we solve for :math:`P` and
    :math:`Q` using the following set of equations:

    .. math::

        FP^2 + Gg + H =0

        FQN + (FP+G)Q + (LN + M)=0

    Once :math:`P` and :math:`Q` are known, one ca solve for :math:`R`
    and :math:`S` using the following equations:

    .. math::

        R = -C^{-1}(AP + B)

        S = - C^{-1}(AQ + D)

    Parameters
    ----------
    AA : array_like, dtype=float, shape=(ny, nx)
        The matrix represented above by :math:`A`. It is the matrix of
        derivatives of the Y equations with repsect to :math:`X_t`
    BB : array_like, dtype=float, shape=(ny, nx)
        The matrix represented above by :math:`B`. It is the matrix of
        derivatives of the Y equations with repsect to
        :math:`X_{t-1}`.
    CC : array_like, dtype=float, shape=(ny, ny)
        The matrix represented above by :math:`C`. It is the matrix of
        derivatives of the Y equations with repsect to :math:`Y_t`
    DD : array_like, dtype=float, shape=(ny, nz)
        The matrix represented above by :math:`C`. It is the matrix of
        derivatives of the Y equations with repsect to :math:`Z_t`
    FF : array_like, dtype=float, shape=(nx, nx)
        The matrix represetned above by :math:`F`. It the matrix of
        derivatives of the model's characterizing equations with
        respect to :math:`X_{t+1}`
    GG : array_like, dtype=float, shape=(nx, nx)
        The matrix represetned above by :math:`G`. It the matrix of
```

```
        derivatives of the model’s characterizing equations with
        respect to :math:‘X_t‘
    HH : array_like, dtype=float, shape=(nx, nx)
        The matrix represetned above by :math:‘H‘. It the matrix of
        derivatives of the model’s characterizing equations with
        respect to :math:‘X_{t-1}‘
    JJ : array_like, dtype=float, shape=(nx, ny)
        The matrix represetned above by :math:‘J‘. It the matrix of
        derivatives of the model’s characterizing equations with
        respect to :math:‘Y_{t+1}‘
    KK : array_like, dtype=float, shape=(nx, ny)
        The matrix represetned above by :math:‘K‘. It the matrix of
        derivatives of the model’s characterizing equations with
        respect to :math:‘Y_t‘
    LL : array_like, dtype=float, shape=(nx, nz)
        The matrix represetned above by :math:‘L‘. It the matrix of
        derivatives of the model’s characterizing equations with
        respect to :math:‘Z_{t+1}‘
    MM : array_like, dtype=float, shape=(nx, nz)
        The matrix represetned above by :math:‘M‘. It the matrix of
        derivatives of the model’s characterizing equations with
        respect to :math:‘Z_t‘
    NN : array_like, dtype=float, shape=(nz, nz)
        The autocorrelation matrix for the exogenous state
        vector z.

    Returns
    -------
    P : array_like, dtype=float, shape=(nx, nx)
        The matrix :math:‘P‘ in the law of motion for endogenous state
        variables described above.
    Q : array_like, dtype=float, shape=(nx, nz)
        The matrix :math:‘P‘ in the law of motion for endogenous state
        variables described above.
    R : array_like, dtype=float, shape=(ny, nx)
        The matrix :math:‘P‘ in the law of motion for endogenous state
        variables described above.
    S : array_like, dtype=float, shape=(ny, nz)
        The matrix :math:‘P‘ in the law of motion for endogenous state
        variables described above.

    References
    ----------
    .. [1] Uhlig, H. (1999): "A toolkit for analyzing nonlinear dynamic
       stochastic models easily," in Computational Methods for the Study
       of Dynamic Economies, ed. by R. Marimon, pp. 30-61. Oxford
       University Press.

    """
    #The original coding we did used the np.matrix form for our matrices
        so we
    #make sure to set our inputs to numpy matrices.
    AA = np.matrix(AA)
    BB = np.matrix(BB)
    CC = np.matrix(CC)
    DD = np.matrix(DD)
    FF = np.matrix(FF)
    GG = np.matrix(GG)
    HH = np.matrix(HH)
    JJ = np.matrix(JJ)
    KK = np.matrix(KK)
    LL = np.matrix(LL)
    MM = np.matrix(MM)
    NN = np.matrix(NN)
    #Tolerance level to use
    TOL = .000001

    # Here we use matrices to get pertinent dimensions.
    nx = FF.shape[1]
    l_equ = CC.shape[0]
    ny = CC.shape[1]
    nz = LL.shape[1]

    k_exog = min(NN.shape)

    # The following if and else blocks form the
    # Psi, Gamma, Theta Xi, Delta mats
    if l_equ == 0:
        if CC.any():
            # This blcok makes sure you don't throw an error with an empty
                CC.
            CC_plus = la.pinv(CC)
            CC_0 = _nullSpaceBasis(CC.T)
        else:
            CC_plus = np.mat([])
            CC_0 = np.mat([])
        Psi_mat = FF
        Gamma_mat = -GG
        Theta_mat = -HH
        Xi_mat = np.mat(vstack((hstack((Gamma_mat, Theta_mat)),
                        hstack((eye(nx), zeros((nx, nx)))))))
        Delta_mat = np.mat(vstack((hstack((Psi_mat, zeros((nx, nx)))),
                           hstack((zeros((nx, nx)), eye(nx))))))

    else:
        CC_plus = la.pinv(CC)
        CC_0 = _nullSpaceBasis(CC.T)
        Psi_mat = vstack((zeros((l_equ - ny, nx)), FF \
                         - dot(dot(JJ, CC_plus), AA)))
        if CC_0.size == 0:
            # This block makes sure you don't throw an error with an empty
                CC.
            Gamma_mat = vstack((dot(CC_0, AA), dot(dot(JJ, CC_plus), BB) \
                        - GG + dot(dot(KK, CC_plus), AA)))
            Theta_mat = vstack((dot(CC_0, AA), dot(dot(KK, CC_plus), BB) \
                                   - HH))
        else:
            Gamma_mat = dot(dot(JJ, CC_plus), BB) - GG +\
                             dot(dot(KK, CC_plus), AA)
            Theta_mat = dot(dot(KK, CC_plus), BB) - HH
        Xi_mat = vstack((hstack((Gamma_mat, Theta_mat)), \
                         hstack((eye(nx), zeros((nx, nx))))))
        Delta_mat = vstack((hstack((Psi_mat, np.mat(zeros((nx, nx))))),\
                            hstack((zeros((nx, nx)), eye(nx)))))

    # Now we need the generalized eigenvalues/vectors for Xi with respect
        to
    # Delta. That is eVals and eVecs below.
    eVals, eVecs = la.eig(Xi_mat, Delta_mat)
    if npla.matrix_rank(eVecs) < nx:
        print('Error: Xi is not diagonalizable, stopping')

    # From here to line 158 we Diagonalize Xi, form Lambda/Omega and find
        P.
    else:
        Xi_sortabs = np.sort(abs(eVals))
        Xi_sortindex = np.argsort(abs(eVals))
        Xi_sortedVec = np.array([eVecs[:, i] for i in Xi_sortindex]).T
        Xi_sortval = Xi_sortabs
        Xi_select = np.arange(0, nx)
        if np.imag(Xi_sortedVec[nx - 1]).any():
            if (abs(Xi_sortval[nx - 1] - sp.conj(Xi_sortval[nx])) < TOL):
                drop_index = 1
                cond_1 = (abs(np.imag(Xi_sortval[drop_index])) > TOL)
                cond_2 = drop_index < nx
                while cond_1 and cond_2:
                    drop_index += 1
                if drop_index >= nx:
                    print('There is an error. Too many complex eigenvalues.'
                          +' Quitting')
                else:
                    print('droping the lowest real eigenvalue. Beware of' +
                          ' sunspots')
                    Xi_select = np.array([np.arange(0, drop_index - 1),\
                                          np.arange(drop_index + 1, nx)])

        # Here Uhlig computes stuff if user chose "Manual roots" I skip it.
        if max(abs(Xi_sortval[Xi_select])) > 1 + TOL:
            print('It looks like we have unstable roots. This might not
                work')
        if abs(max(abs(Xi_sortval[Xi_select])) - 1) < TOL:
            print('Check the model to make sure you have a unique steady' +
                  ' state we are having problems with convergence.')
        Lambda_mat = np.diag(Xi_sortval[Xi_select])
        Omega_mat = Xi_sortedVec[nx:2 * nx, Xi_select]
        #Omega_mat = sp.reshape(Omega_mat,\
        #        (math.sqrt(Omega_mat.size),math.sqrt(Omega_mat.size)))
        if npla.matrix_rank(Omega_mat) < nx:
            print("Omega matrix is not invertible, Can't solve for P")
        else:
            PP = dot(dot(Omega_mat, Lambda_mat), la.inv(Omega_mat))
            PP_imag = np.imag(PP)
            PP = np.real(PP)
            if (sum(sum(abs(PP_imag))) / sum(sum(abs(PP))) > .000001).any():
                print("A lot of P is complex. We will continue with the" +
                      " real part and hope we don't lose too much
                         information")

    # The code from here to the end was from he Uhlig file cacl_qrs.m.
    # I think for python it fits better here than in a separate file.

    # The if and else below make RR and VV depending on our model's setup.
    if l_equ == 0:
        RR = zeros((0, nx))
        VV = hstack((kron(NN.T, FF) + kron(eye(k_exog), \
            (dot(FF, PP) + GG)), kron(NN.T, JJ) + kron(eye(k_exog), KK)))
```

```python

    else:
        RR = - dot(CC_plus, (dot(AA, PP) + BB))
        VV = sp.vstack((hstack((kron(eye(k_exog), AA), \
                        kron(eye(k_exog), CC))), hstack((kron(NN.T, FF) +\
                        kron(eye(k_exog), dot(FF, PP) + dot(JJ, RR) + GG),\
                        kron(NN.T, JJ) + kron(eye(k_exog), KK)))))

    # Now we use LL, NN, RR, VV to get the QQ, RR, SS matrices.
    if (npla.matrix_rank(VV) < k_exog * (nx + ny)):
        print("Sorry but V is not invertible. Can't solve for Q and S")
    else:
        LL = sp.mat(LL)
        NN = sp.mat(NN)
        LLNN_plus_MM = dot(LL, NN) + MM

        if DD.any():
            impvec = vstack([DD.T, np.reshape(LLNN_plus_MM,
                                              (nx * k_exog, 1), 'F')])
        else:
            impvec = np.reshape(LLNN_plus_MM, (nx * k_exog, 1), 'F')

        QQSS_vec = np.matrix(la.solve(-VV, impvec))

        if (max(abs(QQSS_vec)) == sp.inf).any():
            print("We have issues with Q and S. Entries are undefined." +
                  " Probably because V is no inverible.")

        QQ = np.reshape(np.matrix(QQSS_vec[0:nx * k_exog, 0]),
                        (nx, k_exog), 'F')

        SS = np.reshape(QQSS_vec[(nx * k_exog):((nx + ny) * k_exog), 0],\
                        (ny, k_exog), 'F')

        #Build WW - we don't use this, but Uhlig defines it so we do too.
        WW = sp.vstack((
        hstack((eye(nx), zeros((nx, k_exog)))),
        hstack((dot(RR, la.pinv(PP)), (SS - dot(dot(RR, la.pinv(PP)),
            QQ)))),
        hstack((zeros((k_exog, nx)), eye(k_exog)))))

    return PP, QQ, RR, SS
```

Homework & Lab 12

Suppose that instead of including the jump variables of our our model in the vector X_t above, we separated them into a separate vector, Y_t. In this case we can linearize our system into the following equations:

$$\begin{aligned}
0 &= A\tilde{X}_t + B\tilde{X}_{t-1} + C\tilde{Y}_t + D\tilde{Z}_t \\
0 &= E_t\left\{F\tilde{X}_{t+1} + G\tilde{X}_t + H\tilde{X}_{t-1} + J\tilde{Y}_{t+1} + K\tilde{Y}_t + L\tilde{Z}_{t+1} + M\tilde{Z}_t\right\} \\
\tilde{Z}_t &= N\tilde{Z}_{t-1} + \varepsilon_t
\end{aligned}$$

Assume that C is of full rank. By once again hypothesizing that the transition functions for the model are log-linear, that is they are of the form:

$$\begin{aligned}
\tilde{X}_t &= P\tilde{X}_{t-1} + Q\tilde{Z}_t \\
\tilde{Y}_t &= R\tilde{X}_{t-1} + S\tilde{Z}_t
\end{aligned}$$

Derive algebraic solutions for P, Q, R, and S using techniques similar to those in the notes. Modify the program you wrote in Homework 9 slightly to accommodate separating jump variables from state variables. What advantages might this approach have computationally? In setting up the matricies describing the model?

Answer: We start with the equations that are given. Recursively substituting in the $x_t = Px_{t-1} + Qz_t$, we can put the equations in term of x_{t-1}, and z_t. We get the following equations.

$$0 = (AP + CR + B)x_{t-1} + (Q + RS + D)z_t \tag{9}$$

$$0 = ((FP + JR + G)P + KR + H)x_{t-1} + ((FQ + JS + L)N + (FP + JR + G)Q + KS + M)z_t \tag{10}$$

Since these equations have to hold for all x_{t-1} and z_t then we know that:

$$(AP + CR + B) = 0$$
$$(Q + RS + D) = 0$$
$$((FP + JR + G)P + KR + H) = 0$$
$$((FQ + JS + L)N + (FP + JR + G)Q + KS + M) = 0$$

Thus we can solve $(AP + CR + B) = 0$ by simple linear algebra. We get that $R = -C^{+}(AP + B)$ where C^{+} is the pseudo inverse of C.

Once we know that R is equal to the above then we can plug this back into the 2nd of the initial equations and we get

$$0 = (F - JC^{+}A)P^2 - (JC^{+}B - G + KC^{+}A)P - KC^{+}B + H$$

We can solve this matrix quadratic

Rest of the answer is a Python Program

Lab 13a

Assume $n_X = n_Z = 1$. Write a Python program that will find the numerical values for the derivative of the characterizing equation, taking the parameters as given. Note that you will need to find the steady state value of X first.

Now assume n_X and n_Z take on arbitrary integer values greater than one. Write a Python program that will find the numerical values for the derivatives of the characterizing equations in this case.

Answer: See the code after problem .

Lab 13b

Repeat the second part of the excercise above, where n_X and n_Z take on arbitrary integer values greater than one. However now take numerical values for the the derivatives of the natural logarthims of the characterizing equations.

Answer: A python program implementing these concepts is included below.

```
"""
Created July 18, 2012

Author: Spencer Lyon

Created for "Macroeconomic Theory and Methods" by:
    David Spencer: Brigham Young University
    Kerk Phillips: Brigham Young University
    Rick Evans: Brigham Young University
    Ben Tengensen: Carnegie Mellon University
    Bryan Perry: Massachusetts Institute of Technology

Original MatLab code by Kerk P. (2012) was referenced in creating this
    file.
"""
from numpy import tile, array, empty, ndarray, zeros, log, asarray



def numeric_deriv(func, theta0, nx, ny, nz):
    """
    This function computes the matricies AA-MM in the log-linearizatin of
    the equations in the function 'func'.

    Parameters
    ----------
    func: function
        The function that generates a vector from the dynamic equations
            that are
        to be linearized. This must be written to evaluate to zero in the
        steady state. Often these equations are the Euler equations defining
        the model

    theta0: array, dtype=float
        A vector of steady state values for state parameters. Place the
            values
        of X in the front, then the Y values, followed by the Z's.

    nx: number, int
        The number of elements in X

    ny: number, int
        The number of elements in Y

    nz: number, int
        The number of elements in Z

    Returns
    -------
    AA - MM : 2D array, dtype=float:
        The equaitons described by Uhlig in the log-linearization.
    """
    theta0 = asarray(theta0)
    incr = 1e-7
    T0 = func(theta0) # should be very close to zero.
    length = 3 * nx + 2 * (ny + nz)
    height = T0.size

    dev = tile(theta0.reshape(1, theta0.size), (length, 1))

    for i in range(length):
        dev[i, i] += incr

    bigMat = empty((height, length))
    for i in range(0,length):
        if i < 3 * nx + 2 * ny:
            bigMat[:, i] = theta0[i] * (func(dev[i, :]) - T0) / (1.0 + T0)
        else:
            bigMat[:, i] = (func(dev[i, :]) - T0) / (1.0 + T0)

    bigMat /= incr

    AA = array(bigMat[0:ny, nx:2 * nx])
    BB = array(bigMat[0:ny, 2 * nx:3 * nx])
    CC = array(bigMat[0:ny, 3 * nx + ny:3 * nx + 2 * ny])
    DD = array(bigMat[0:ny, 3 * nx + 2 * ny + nz:length])
    FF = array(bigMat[ny:ny + nx, 0:nx])
    GG = array(bigMat[ny:ny + nx, nx:2 * nx])
    HH = array(bigMat[ny:ny + nx, 2 * nx:3 * nx])
    JJ = array(bigMat[ny:ny + nx, 3 * nx:3 * nx + ny])
    KK = array(bigMat[ny:ny + nx, 3 * nx + ny:3 * nx + 2 * ny])
    LL = array(bigMat[ny:ny + nx, 3 * nx + 2 * ny:3 * nx + 2 * ny + nz])
    MM = array(bigMat[ny:ny + nx, 3 * nx + 2 * ny + nz:length])
    TT = log(1 + T0)

    AA = AA if AA else zeros((ny, nx))
    BB = BB if BB else zeros((ny, nx))
    CC = CC if CC else zeros((ny, ny))
    DD = DD if DD else zeros((ny, nz))
```

Lab 14

For the log-linearized model in section 6.6 of the notes consider the following fundamental functional forms:

$$u(c_t, \ell_t) = \frac{c_t^{1-\gamma} - 1}{1-\gamma} + a\frac{(1-\ell_t)^{1-\gamma} - 1}{1-\gamma}$$
$$F(K_t, L_t, z_t) = K_t^{\alpha}(L_t e^{z_t})^{1-\alpha}$$

Use the following paramter values: $\gamma = 2.5$, $\beta = .98$, $\alpha = .40$, $a = .5$, $\delta = .10$, $\bar{z} = 0$, $\rho_z = .9$ and $\tau = .05$.

Find the values of P and Q in this case if $X_t = \{k_{t-1}, \ell_t\}$.

Answer: The characterizing equations that need to be log-linearized are (3.17) – (3.23). With our functional forms these become the equations below. The first two equations are the elements of the Γ function. The next three equations are definitions used in the first two. The final equation is the exogenous law of motion.

$$c_t^{-\gamma} = \beta E_t\left\{c_{t+1}^{-\gamma}[1 + (r_{t+1} - \delta)(1-\tau)]\right\}$$

$$a(1-\ell_t)^{-\gamma} = c_t^{-\gamma} w_t (1-\tau)$$

$$c_t = (1-\tau)\left[w_t\ell_t + (r_t - \delta)k_t\right] + k_t + T_t - k_{t+1}$$

$$r_t = \alpha k_t^{\alpha-1}(\ell_t e^{z_t})^{1-\alpha}$$

$$w_t = (1-\alpha)k_t^{\alpha}(e^{z_t})^{1-\alpha}(\ell_t)^{-\alpha}$$

$$T_t = \tau\left[w_t\ell_t + (r_t - \delta)k_t\right]$$

$$z_t = (1-\rho_z)\bar{z} + \rho_z z_{t-1} + \epsilon_t^z; \quad \epsilon_t^z \sim \text{i.i.d.}(0, \sigma_z^2)$$

The next step is to log-linearize these equations, except for the law of motion. The log-linearized equations are given below.

The law of motion is:

$$z_t = \rho_z z_{t-1} + \epsilon_t^z; \quad \epsilon_t^z \sim \text{i.i.d.}(0, \sigma_z^2)$$

The expectational equations which define the endogenous state variables are:

$$E_t\left\{\gamma(\tilde{c}_t - \tilde{c}_{t+1}) + \tilde{r}_{t+1}\right\} = 0$$

$$\tilde{w}_t - \gamma(\tilde{c}_t + \tilde{\ell}_t) = 0$$

With the following definitions.

$$\begin{aligned} c_t = & \frac{(1-\tau)\bar{w}\bar{\ell}}{\bar{c}}\tilde{w}_t + \frac{(1-\tau)\bar{w}\bar{\ell}}{\bar{c}}\tilde{\ell}_t + \frac{(1+\bar{r}-\delta)\bar{k}}{\bar{c}}\tilde{r}_t \\ & + \frac{(1+\bar{r}-\delta)\bar{k}}{\bar{c}}\tilde{k}_t + \frac{\bar{T}}{\bar{c}}\tilde{T}_t - \frac{\bar{k}}{\bar{c}}\tilde{k}_{t+1} \end{aligned}$$

$$\tilde{r}_t = (1-\alpha)(\tilde{\ell}_t + z_t - \tilde{k}_t)$$

$$\tilde{w}_t = (1-\alpha)z_t + \alpha(\tilde{k}_t - \tilde{\ell}_t)$$

$$\tilde{T}_t = \frac{\tau\bar{w}\bar{\ell}}{\bar{T}}\tilde{w}_t + \frac{\tau\bar{w}\bar{\ell}}{\bar{T}}\tilde{\ell}_t + \frac{(1+\bar{r}-\delta)\bar{k}}{\bar{T}}\tilde{r}_t + \frac{(1+\bar{r}-\delta)\bar{k}}{\bar{T}}\tilde{k}_t$$

Answer: The matrices P and Q are given below:

$$P = \begin{pmatrix} 0.9153 & 0 \\ -0.19191 & 0 \end{pmatrix} \quad Q = \begin{pmatrix} 0.215. \\ -0.01882 \end{pmatrix}$$

The code used to create the solutions is given below:

```python
#--------------------------- #14: Find P and Q ---------------------------#
from time import time
import numpy as np
import scipy as sp
import math
from numpy import exp, array, zeros
from scipy.optimize import fsolve
import matplotlib.pyplot as plt
from RBCnumerderiv import numeric_deriv
import uhlig

# Just a few house keeping things.
start_time = time()
sp.set_printoptions(linewidth=140, suppress = True, precision = 5)

gamma = 2.5
xsi = 1.5
beta = 0.98
alpha = 0.40
a = 0.5
delta = 0.10
```

```python
zbar = 0.0
tao = 0.05
rho = 0.9

nx, ny, nz = [2, 0, 1]


def state_defs(kt1, kt, lt, zt):
    """
    This function solves for each of the variables defined by the
    model's characterizing equation as a function of the state.

    Parameters
    ----------
    kt1: number, float
        The value of capital stock in the next period.

    kt: number, float
        The value of capital stock in the current period.

    lt: number, float
        The value of labor supplied in the current period.

    zt: number, float
        The value of the productivity shock in the current period.

    Returns
    -------
    arr: array, dtype=float, shape=( x 1)
        The value of each of the definitions as a function of the state
        The values are ordered as follows:
            y: output
            i: investment
            c: consumption
            r: interest rate
            w: wage
            T: government transfers
    """
    y = (kt ** alpha) * (np.exp(zt) * lt ** (1 - alpha))
    i = kt - (1 - delta) * kt
    r = (alpha) * y / kt
    w = (1 - alpha) * y / lt
    T = tao * (w * lt + (r - delta) * kt)
    c = (1 - tao) * (w * lt + (r -delta) * kt) + kt + T - kt1

    return np.array([y, i, c, r, w, T])


def opt_func(guess, want_ss=0):
    """
    This function uses the characterizing equations and Euler equations
    to numerically solve for the steady state. It will be passed to the
    routine in scipy.optimize.fsolve.

    Parameters
    ----------
    guess: array, dtype = float, shape = (2 x 1)
        This is a two element array containing an inital guess for the
        steady state value of capital and labor as the first and second
        elements, respectively.

    Returns
    -------
    Eul_err: array, dtype = float, shape = (2 x 1)
        This is the error in the Euler equations that the current value
        of the parameters yeilds.

    Notes
    -----
    We make the assumption that the error in the Euler equations should
    go to zero in the stead state. That is why this funciton returns
    Eul_err.
    """
    if want_ss == 1:
        kbar, lbar = guess[0:2]
        y, i, c, r, w, T = state_defs(kbar, kbar, lbar, zbar)

        Eul1 = beta * ((r - delta) * (1 - tao) + 1) - 1
        Eul2 = (w * (1 - tao) * (1 - lbar) ** xsi) / (a * (c ** gamma)) - 1

    else:
        kt2, lt1, kt1, lt, kt, lt_1, zt1, zt = guess

        y, i, c, r, w, T = state_defs(kt1, kt, lt, zt)
        y1, i1, c1, r1, w1, T1 = state_defs(kt2, kt1, lt1, zt1)

        Eul1 = w * (1-tao) * (1-lt)**xsi / (a * c**gamma) - 1
        Eul2 = beta * (((c/c1)**gamma) * ((r1-delta) * (1-tao) + 1)) - 1

    return np.array([Eul1, Eul2])

# Solving for the steady state
initial_guess = np.array([4.5, 0.4])
Xbar  = kbar, lbar = fsolve(opt_func, initial_guess, args=(1))

SS_1 = ybar, ibar, cbar, rbar, wbar, Tbar = state_defs(kbar, kbar, lbar,
    zbar)
SS = np.append(Xbar, SS_1)

# Solving for A_M matricies.
theta0 = array([kbar, lbar, kbar, lbar, kbar, lbar, zbar, zbar])

NN = array([rho])
AA, BB, CC, DD, FF, GG, HH, JJ, KK, LL, MM, TT = numeric_deriv(opt_func,
    theta0,
                                                                nx, ny, nz)

PP, QQ, RR, SS = uhlig.solvePQRS(AA, BB, CC, DD, FF, GG,
                                 HH, JJ, KK, LL, MM, NN)
```

Lab 15

Use the law of motion and approximate policy function from homework 14 to generate 1000 artificial time series for an economy where each time series is 250 periods long. Start each simulation off with a starting value for k that is ten percent below the steady state value, and a value of z that is one standard deviation below zero.

Use $\sigma_z^2 = .004$.

For each simulation save the time-series for GDP, consumption and investment. When all 1000 simulations have finished generate a graph for each of these time-series showing the average value over the simulations for each period, and also showing the five and ninety-five percent confidence bands for each series each period.

In addition, calculate the mean, standard deviation, autocorrelation and correlation with output for each series over each simulation and report the average values and standard deviations for these moments over the 1000 simulations.

Answer:

The desired moments are in Table 2

	mean	std	autocorr	corr_y
y	0.242127	0.146601	1.000000	0.895586
i	-0.156114	0.128015	0.845502	0.544935
c	-0.879678	0.148358	0.747940	0.988342

Table 2: Table of moments from the monto carlo simulation

A plot of the mean as well as 95% and 5% confidence bands for each of the time series appears in Figure 5

The code used to generate this solution appears below. Note that this is intended to be a continuation of the solution for the problem that asked for P and Q (problem 14)

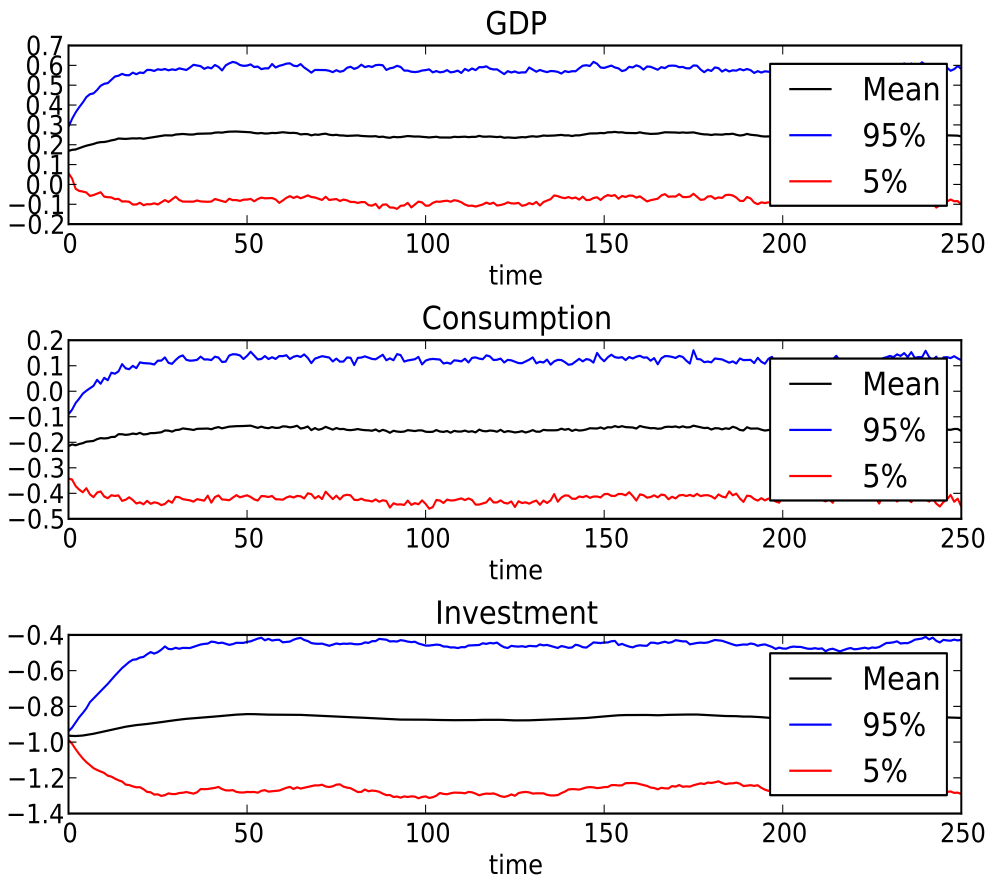

Figure 5: Plot of the mean values as well as 95% and 5% confidence bands obtained from a Monte Carlo simulation. Plots for GDP, investment, and consumption are included.

```
#--------------------------- #15: Monte Carlo Sims -----------------------#
# Make sure we are dealing with numpy arrays
PP = np.asarray(PP)
QQ = np.asarray(QQ).squeeze()
RR = np.asarray(RR)
SS = np.asarray(SS)
nobs = 251 # number of observations in each time series.
nmc = 1000 #  Number of time-series to gather.
sig = math.sqrt(0.004) # stdev. of epsilon in shock process.


mcmoments = zeros((3, 4))
mcy = zeros((nobs, nmc))
mcc = zeros((nobs, nmc))
mci = zeros((nobs, nmc))

for m in range(1, nmc + 1):
        if m % 100 == 0:
            print "on simluation: ", m

        # Initializing series for state vars
        Xtilde = zeros((nobs+2, 2))
        Ztilde = zeros((nobs+2))

        # Generating eps shock series.
        eps = np.random.normal(0, sig, nobs + 2)

        # Generate time series for X and Z
        Xtilde[0, :] = np.log((Xbar * np.exp(-.1)) / Xbar)
        Ztilde[0] = - sig - zbar

        for t in range(1, nobs + 2):
            Ztilde[t] = Ztilde[t - 1] * NN + eps[t]
            Xtilde[t, :] = np.dot(Xtilde[t - 1., :], PP.T) + Ztilde[t] * QQ

        # Transform X, Z in to k, l, z
        MCk = kbar * np.exp(Xtilde[:,0])
        MCl = lbar * np.exp(Xtilde[:,1])
        MCz = zbar + Ztilde

        # Generate the rest of the time series
        MCy, MCi, MCc, MCr, MCw, MCT = state_defs(MCk[1:], MCk[:-1],
                                              MCl[:-1], MCz[:-1])

        # Gather data and throw out non-existent first observation.
        MCdata = np.log(np.hstack((MCy.reshape(MCy.size,1),
                                  MCc.reshape(MCc.size,1),
                                  MCi.reshape(MCi.size,1))))
        MCdata = MCdata[1:,:]

        # Get the moments we are asked for
        moments = zeros((3,4))
        means = np.mean(MCdata, axis=0)
        stdevs = np.std(MCdata, axis=0)
        correls = np.corrcoef(MCdata, rowvar=0)
        MCdata2 = np.hstack((MCdata[:-1,:], MCdata[1:, :]))
        autocorrs = np.corrcoef(MCdata2, rowvar=0)
        for i in range(3):
            moments[i,:] = array([means[i], stdevs[i], correls[i, 0],
                autocorrs[i,i + 3]])

        # Save y, c, i time series.
        mcmoments = mcmoments * (m - 1) / m + moments / m
        mcy[:, m - 1] = MCdata[:,0]
        mcc[:, m - 1] = MCdata[:,1]
        mci[:, m - 1] = MCdata[:,2]

    moments_df = pd.DataFrame(moments, index=['y', 'i', 'c'],
                            columns=['mean', 'std', 'autocorr', 'corr_y'])

    # let pandas format the latex table
    print(moments_df.to_latex())

    # Get mean at each time period.
    meany = np.mean(mcy, axis = 1)
    meanc = np.mean(mcc, axis = 1)
```

```
meani = np.mean(mci, axis = 1)

# Sort the vectors.
sorty = np.sort(mcy.T, axis=0)
sortc = np.sort(mcc.T, axis=0)
sorti = np.sort(mci.T, axis=0)

# Find upper and lower indicies.
upp = np.ceil(0.95 * nmc)
low = np.floor(0.05 * nmc) + 1

# Generate upper and lower bands from sorted data.
uppery = sorty[upp,:].T
lowery = sorty[low,:].T
upperc = sortc[upp,:].T
lowerc = sortc[low,:].T
upperi = sorti[upp,:].T
loweri = sorti[low,:].T


# Plot the stuff.
fig1 = plt.figure(1)
fig1.subplots_adjust(hspace=0.65)

plt.subplot(3, 1, 1)
plt.plot(range(meany.size), meany, 'k-',
        range(uppery.size), uppery, 'b-',
        range(lowery.size), lowery, 'r-')
plt.legend(['Mean', '95%', '5%'], loc=5)
plt.title('GDP')
plt.xlabel('time')

plt.subplot(3, 1, 2)
plt.plot(range(meanc.size), meanc, 'k-',
        range(upperc.size), upperc, 'b-',
        range(lowerc.size), lowerc, 'r-')
plt.legend(['Mean', '95%', '5%'], loc=5)
plt.title('Consumption')
plt.xlabel('time')

plt.subplot(3, 1, 3)
plt.plot(range(meani.size), meani, 'k-',
        range(upperi.size), upperi, 'b-',
        range(loweri.size), loweri, 'r-')
plt.legend(['Mean', '95%', '5%'], loc=5)
plt.title('Investment')
plt.xlabel('time')
```

Lab 16

Using the same setup as homework 14, generate impulse response functions for GDP, consumption and investment with lags from zero to forty periods.

Answer:

A plot of the impulse response functions appears in Figure 6

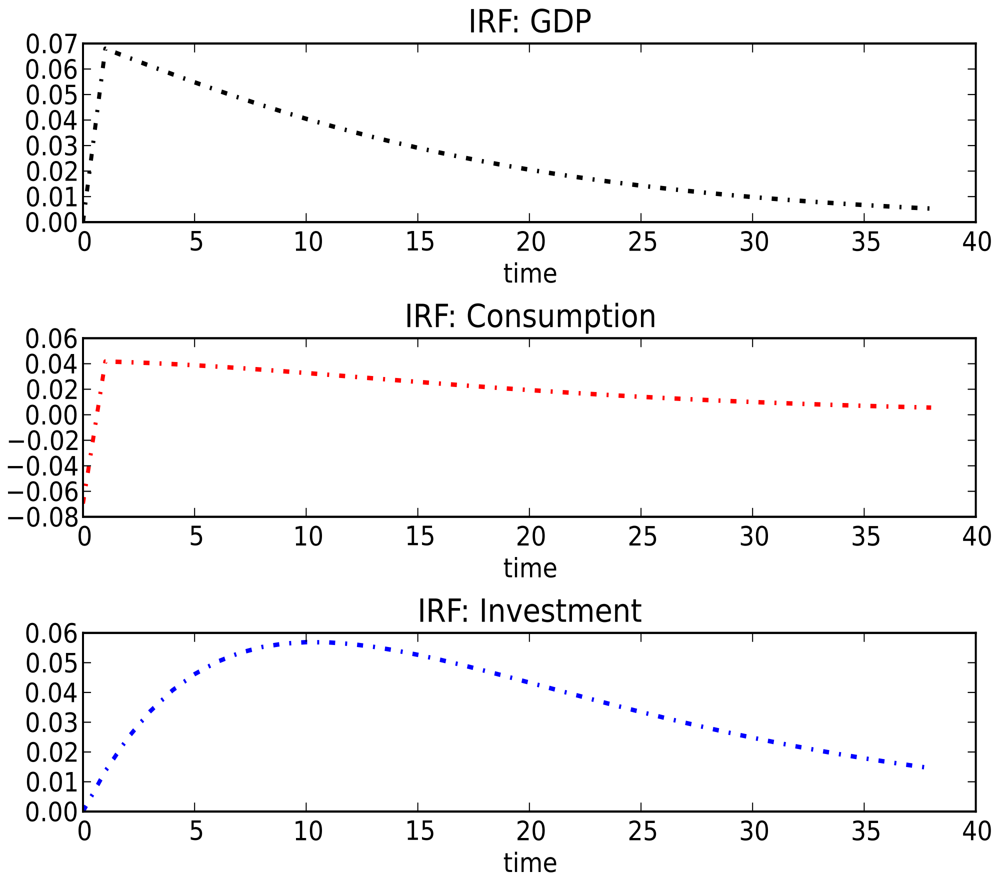

Figure 6: Impulse response functions for GDP, Investment, and Consumption in response to a 1 standard deviation shock in period 3.

The code used to generate this solution appears below. Note that this is intended to be a continuation of the solution for the problem that asked for P and Q (problem 14).

```python
plt.savefig('dsge15.eps', format='eps', dpi=1000)
plt.show()

#------------------------- #16: Impulse Response--------------------------#
Iobs = 41
PP = array(PP)
QQ = array(QQ)
# Instantiate X and Z matrices.
IXtilde = zeros((Iobs, 2))
Iz = zeros(Iobs)

# generate epsilon process.
Ieps = zeros(Iobs)
Ieps[2] = sig

# Get X, Z time-series.
IXtilde[0,:] = 0.0
Iz[0] = 0.
for t in range(1, Iobs):
    Iz[t] = rho * Iz[t - 1] + Ieps[t]
    IXtilde[t,:] = np.dot(IXtilde[t - 1, :], PP.T) + Iz[t] * QQ

# Transform X to k and l
Ik = kbar * np.exp(IXtilde[:,0])
Il = lbar * np.exp(IXtilde[:,1])

# Get other time-series
Iy, Ii, Ic, Ir, Iw, IT = state_defs(Ik[1:], Ik[:-1], Il[:-1], Iz[:-1])

# converting back to y, c, i and deleting non-existent first observation.
Iy = np.log(Iy / ybar)[1:]
Ic = np.log(Ic / cbar)[1:]
Ii = np.log(Ii / ibar)[1:]


# Plot the rest of the stuff and say goodbye!
fig2 = plt.figure(2)
fig2.subplots_adjust(hspace=0.65)

plt.subplot(3,1,1)
plt.plot(range(Iy.size), Iy, 'k-.', linewidth=2)
plt.title('IRF: GDP')
plt.xlabel('time')

plt.subplot(3,1,2)
plt.plot(range(Ic.size), Ic, 'r-.', linewidth=2)
plt.title('IRF: Consumption')
plt.xlabel('time')

plt.subplot(3,1,3)
plt.plot(range(Ii.size), Ii, 'b-.', linewidth=2)
plt.title('IRF: Investment')
plt.xlabel('time')

plt.savefig('dsge16.eps', format='eps', dpi=1000)
plt.show()
```