Advanced Dynamic Programming

Thomas J. Sargent and John Stachurski

May 03, 2024

CONTENTS

This website presents a set of lectures on advanced topics in dynamic programming.

Part I

Dynamic Programming Squared

CHAPTER

ONE

OPTIMAL UNEMPLOYMENT INSURANCE

1.1 Overview

This lecture describes a model of optimal unemployment insurance created by Shavell and Weiss (1979) [SW79].

We use recursive techniques of Hopenhayn and Nicolini (1997) [HN97] to compute optimal insurance plans for Shavell and Weiss's model.

Hopenhayn and Nicolini's model is a generalization of Shavell and Weiss's along dimensions that we'll soon describe.

1.2 Shavell and Weiss's Model

An unemployed worker orders stochastic processes of consumption and search effort $\{c_t, a_t\}_{t=0}^{\infty}$ according to

$$E \sum_{t=0}^{\infty} \beta^t \left[u(c_t) - a_t \right] \tag{1.1}$$

where $\beta \in (0, 1)$ and $u(c)$ is strictly increasing, twice differentiable, and strictly concave.

We assume that $u(0)$ is well defined.

We require that $c_t \geq 0$ and $a_t \geq 0$.

All jobs are alike and pay wage $w > 0$ units of the consumption good each period forever.

An unemployed worker searches with effort a and with probability $p(a)$ receives a permanent job at the beginning of the next period.

Furthermore, $a = 0$ when the worker is employed.

The probability of finding a job is $p(a)$ where p is an increasing, strictly concave, and twice differentiable function of a that satisfies $p(a) \in [0, 1]$ for $a \geq 0$, $p(0) = 0$.

The consumption good is nonstorable.

An unemployed worker has no savings and cannot borrow or lend.

An **insurance agency** or **planner** is the unemployed worker's only source of consumption smoothing over time and across states.

Once a worker has found a job, he is beyond the planner's grasp.

- This is Shavell and Weiss's assumption, but not Hopenhayn and Nicolini's.
- Hopenhayn and Nicolini allow the unemployment insurance agency to impose history-dependent taxes on previously unemployed workers.

- Since there is no incentive problem after the worker has found a job, it is optimal for the agency to provide an employed worker with a constant level of consumption.
- Hence, Hopenhayn and Nicolini's insurance agency imposes a permanent per-period history-dependent tax on a previously unemployed but presently employed worker.

1.2.1 Autarky

As a benchmark, we first study the fate of an unemployed worker who has no access to unemployment insurance.

Because employment is an absorbing state for the worker, we work backward from that state.

Let V^e be the expected sum of discounted one-period utilities of an employed worker.

Once the worker is employed, $a = 0$, making his period utility be $u(c) - a = u(w)$ forever.

Therefore,

$$V^e = \frac{u(w)}{(1-\beta)}. \tag{1.2}$$

Now let V^u be the expected discounted present value of utility for an unemployed worker who chooses consumption, effort pair (c, a) optimally.

It satisfies the Bellman equation

$$V^u = \max_{a \geq 0} \biggl\{ u(0) - a + \beta \left[p(a) V^e + (1 - p(a)) V^u \right] \biggr\}. \tag{1.3}$$

The first-order condition for a maximum is

$$\beta p'(a) \left[V^e - V^u \right] \leq 1, \tag{1.4}$$

with equality if $a > 0$.

Since there is no state variable in this infinite horizon problem, there is a time-invariant optimal search intensity a and an associated value of being unemployed V^u.

Let $V_{\text{aut}} = V^u$ solve Bellman equation (1.3).

Equations (1.3) and (1.4) form the basis for an iterative algorithm for computing $V^u = V_{\text{aut}}$.

- Let V^u_j be the estimate of V_{aut} at the jth iteration.
- Use this value in equation (1.4) and solve for an estimate of effort a_j.
- Use this value in a version of equation (1.3) with V^u_j on the right side to compute V^u_{j+1}.
- Iterate to convergence.

1.2.2 Full Information

Another benchmark model helps set the stage for the model with private information that we ultimately want to study.

In this model, the unemployment agency has full information about the unemployed work.

We study optimal provision of insurance with full information.

An insurance agency can set both the consumption and search effort of an unemployed person.

The agency wants to design an unemployment insurance contract to give the unemployed worker expected discounted utility $V > V_{\text{aut}}$.

The planner wants to deliver value V efficiently, meaning in a way that minimizes expected discounted cost, using β as the discount factor.

We formulate the optimal insurance problem recursively.

Let $C(V)$ be the expected discounted cost of giving the worker expected discounted utility V.

The cost function is strictly convex because a higher V implies a lower marginal utility of the worker; that is, additional expected utils can be awarded to the worker only at an increasing marginal cost in terms of the consumption good.

Given V, the planner assigns first-period pair (c, a) and promised continuation value V^u, should the worker be unlucky and not find a job.

(c, a, V^u) are chosen to be functions of V and to satisfy the Bellman equation

$$C(V) = \min_{c,a,V^u} \Big\{ c + \beta[1 - p(a)]C(V^u) \Big\}, \tag{1.5}$$

where minimization is subject to the promise-keeping constraint

$$V \leq u(c) - a + \beta \left\{ p(a)V^e + [1 - p(a)]V^u \right\}. \tag{1.6}$$

Here V^e is given by equation (1.2), which reflects the assumption that once the worker is employed, he is beyond the reach of the unemployment insurance agency.

The right side of Bellman equation (1.5) is attained by policy functions $c = c(V), a = a(V)$, and $V^u = V^u(V)$.

The promise-keeping constraint, equation (1.6), asserts that the 3-tuple (c, a, V^u) attains at least V.

Let θ be a Lagrange multiplier on constraint (1.6).

At an interior solution, the first-order conditions with respect to c, a, and V^u, respectively, are

$$\begin{aligned} \theta &= \frac{1}{u'(c)}\,, \\ C(V^u) &= \theta \left[\frac{1}{\beta p'(a)} - (V^e - V^u) \right]\,, \\ C'(V^u) &= \theta\,. \end{aligned} \tag{1.7}$$

The envelope condition $C'(V) = \theta$ and the third equation of (1.7) imply that $C'(V^u) = C'(V)$.

Strict convexity of C then implies that $V^u = V$

Applied repeatedly over time, $V^u = V$ makes the continuation value remain constant during the entire spell of unemployment.

The first equation of (1.7) determines c, and the second equation of (1.7) determines a, both as functions of promised value V.

That $V^u = V$ then implies that c and a are held constant during the unemployment spell.

Thus, the unemployed worker's consumption c and search effort a are both fully smoothed during the unemployment spell.

But the worker's consumption is not smoothed across states of employment and unemployment unless $V = V^e$.

1.2.3 Incentive Problem

The preceding efficient insurance scheme requires that the insurance agency control both c and a.

It will not do for the insurance agency simply to announce c and then allow the worker to choose a.

Here is why.

The agency delivers a value V^u higher than the autarky value V_{aut} by doing two things.

It **increases** the unemployed worker's consumption c and **decreases** his search effort a.

But the prescribed search effort is **higher** than what the worker would choose if he were to be guaranteed consumption level c while he remains unemployed.

This follows from the first two equations of (1.7) and the fact that the insurance scheme is costly, $C(V^u) > 0$, which imply $[\beta p'(a)]^{-1} > (V^e - V^u)$.

But look at the worker's first-order condition (1.4) under autarky.

It implies that if search effort $a > 0$, then $[\beta p'(a)]^{-1} = [V^e - V^u]$, which is inconsistent with the preceding inequality $[\beta p'(a)]^{-1} > (V^e - V^u)$ that prevails when $a > 0$ under the social insurance arrangement.

If he were free to choose a, the worker would therefore want to fulfill (1.4), either at equality so long as $a > 0$, or by setting $a = 0$ otherwise.

Starting from the a associated with the social insurance scheme, he would establish the desired equality in (1.4) by *lowering* a, thereby decreasing the term $[\beta p'(a)]^{-1}$ (which also lowers $(V^e - V^u)$ when the value of being unemployed V^u increases).

If an equality can be established before a reaches zero, this would be the worker's preferred search effort; otherwise the worker would find it optimal to accept the insurance payment, set $a = 0$, and never work again.

Thus, since the worker does not take the cost of the insurance scheme into account, he would choose a search effort below the socially optimal one.

The efficient contract relies on the agency's ability to control *both* the unemployed worker's consumption *and* his search effort.

1.3 Private Information

Following Shavell and Weiss (1979) [SW79] and Hopenhayn and Nicolini (1997) [HN97], now assume that the unemployment insurance agency cannot observe or enforce a, though it can observe and control c.

The worker is free to choose a, which puts expression (1.4), the worker's first-order condition under autarky, back in the picture.

- We are assuming that the worker's best response to the unemployment insurance arrangement is completely characterized by the first-order condition (1.4), an instance of the so-called first-order approach to incentive problems.

Given a contract, the individual will choose search effort according to first-order condition (1.4).

This fact leads the insurance agency to design the unemployment insurance contract to respect this restriction.

Thus, the recursive contract design problem is now to minimize the right side of equation (1.5) subject to expression (1.6) and the incentive constraint (1.4).

Since the restrictions (1.4) and (1.6) are not linear and generally do not define a convex set, it becomes difficult to provide conditions under which the solution to the dynamic programming problem results in a convex function $C(V)$.

- Sometimes this complication can be handled by convexifying the constraint set through the introduction of lotteries.

- A common finding is that optimal plans do not involve lotteries, because convexity of the constraint set is a sufficient but not necessary condition for convexity of the cost function.
- Following Hopenhayn and Nicolini (1997) [HN97], we therefore proceed under the assumption that $C(V)$ is strictly convex in order to characterize the optimal solution.

Let η be the multiplier on constraint (1.4), while θ continues to denote the multiplier on constraint (1.6).

But now we replace the weak inequality in (1.6) by an equality.

The unemployment insurance agency cannot award a higher utility than V because that might violate an incentive-compatibility constraint for exerting the proper search effort in earlier periods.

At an interior solution, first-order conditions with respect to c, a, and V^u, respectively, are

$$\begin{aligned} \theta &= \frac{1}{u'(c)}\,, \\ C(V^u) &= \theta\left[\frac{1}{\beta p'(a)} - (V^e - V^u)\right] - \eta\frac{p''(a)}{p'(a)}(V^e - V^u) \\ &= -\eta\frac{p''(a)}{p'(a)}(V^e - V^u)\,, \\ C'(V^u) &= \theta - \eta\frac{p'(a)}{1-p(a)}\,, \end{aligned} \tag{1.8}$$

where the second equality in the second equation in (1.8) follows from strict equality of the incentive constraint (1.4) when $a > 0$.

As long as the insurance scheme is associated with costs, so that $C(V^u) > 0$, first-order condition in the second equation of (1.8) implies that the multiplier η is strictly positive.

The first-order condition in the second equation of the third equality in (1.8) and the envelope condition $C'(V) = \theta$ together allow us to conclude that $C'(V^u) < C'(V)$.

Convexity of C then implies that $V^u < V$.

After we have also used the first equation of (1.8), it follows that in order to provide the proper incentives, the consumption of the unemployed worker must decrease as the duration of the unemployment spell lengthens.

It also follows from (1.4) at equality that search effort a rises as V^u falls, i.e., it rises with the duration of unemployment.

The duration dependence of benefits is designed to provide incentives to search.

To see this, from the third equation of (1.8), notice how the conclusion that consumption falls with the duration of unemployment depends on the assumption that more search effort raises the prospect of finding a job, i.e., that $p'(a) > 0$.

If $p'(a) = 0$, then the third equation of (1.8) and the strict convexity of C imply that $V^u = V$.

Thus, when $p'(a) = 0$, there is no reason for the planner to make consumption fall with the duration of unemployment.

1.3.1 Computational Details

It is useful to note that there are natural lower and upper bounds to the set of continuation values V^u.

The lower bound is the expected lifetime utility in autarky, V_{aut}.

To compute the upper bound, represent condition (1.4) as

$$V^u \geq V^e - [\beta p'(a)]^{-1},$$

with equality if $a > 0$.

If there is zero search effort, then $V^u \geq V^e - [\beta p'(0)]^{-1}$.

Therefore, to rule out zero search effort we require

$$V^u < V^e - [\beta p'(0)]^{-1}.$$

(Remember that $p''(a) < 0$.)

This step gives our upper bound for V^u.

To formulate the Bellman equation numerically, we suggest using the constraints to eliminate c and a as choice variables, thereby reducing the Bellman equation to a minimization over the one choice variable V^u.

First express the promise-keeping constraint (1.6) at equality as

$$u(c) = V + a - \beta\{p(a)V^e + [1-p(a)]V^u\}$$

so that consumption is

$$c = u^{-1}\left(V + a - \beta[p(a)V^e + (1-p(a))V^u]\right). \tag{1.9}$$

Similarly, solving the inequality (1.4) for a leads to

$$a = \max\left\{0, p'^{-1}\left(\frac{1}{\beta(V^e - V^u)}\right)\right\}. \tag{1.10}$$

When we specialize (1.10) to the functional form for $p(a)$ used by Hopenhayn and Nicolini, we obtain

$$a = \max\left\{0, \frac{\log[r\beta(V^e - V^u)]}{r}\right\}. \tag{1.11}$$

Formulas (1.9) and (1.11) express (c, a) as functions of V and the continuation value V^u.

Using these functions allows us to write the Bellman equation in $C(V)$ as

$$C(V) = \min_{V^u}\{c + \beta[1-p(a)]C(V^u)\} \tag{1.12}$$

where c and a are given by equations (1.9) and (1.11).

1.3.2 Python Computations

We'll approximate the planner's optimal cost function with cubic splines.

To do this, we'll load some useful modules

```python
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
```

We first create a class to set up a particular parametrization.

```python
class params_instance:

    def __init__(self,
                 r,
                 β = 0.999,
                 σ = 0.500,
                 w = 100,
                 n_grid = 50):
```

(continues on next page)

(continued from previous page)

```
        self.β,self.σ,self.w,self.r = β,σ,w,r
        self.n_grid = n_grid
        uw = self.w**(1-self.σ)/(1-self.σ)  #Utility from consuming all wage
        self.Ve = uw/(1-β)
```

1.3.3 Parameter Values

For the other parameters we have just loaded in the above Python code, we'll set brate the net interest rate r to match the hazard rate – the probability of finding a job in one period – in US data.

In particular, we seek an r so that in autarky `p(a(r)) = 0.1`, where `a` is the optimal search effort.

First, we create some helper functions.

```
# The probability of finding a job given search effort, a and interest rate r.
def p(a,r):
    return 1-np.exp(-r*a)

def invp_prime(x,r):
    return -np.log(x/r)/r

def p_prime(a,r):
    return r*np.exp(-r*a)

# The utiliy function
def u(self,c):
    return (c**(1-self.σ))/(1-self.σ)

def u_inv(self,x):
    return ((1-self.σ)*x)**(1/(1-self.σ))
```

Recall that under autarky the value for an unemployed worker satisfies the Bellman equation

$$V^u = \max_a \{u(0) - a + \beta\,[p_r(a)V^e + (1 - p_r(a))V^u]\} \tag{1.13}$$

At the optimal choice of a, we have the first order condition for this problem as:

$$\beta p_r'(a)[V^e - V^u] \leq 1 \tag{1.14}$$

with equality when a >0.

Given an interest rate $\bar{r}$, we can solve the autarky problem as follows:

1. Guess $V^u \in \mathbb{R}^+$
2. Given V^u, use the FOC (1.14) to calculate the implied optimal search effort a
3. Evaluate the difference between the LHS and RHS of the Bellman equation (1.13)
4. Update guess for V^u accordingly, then return to 2) and repeat until the Bellman equation is satisfied.

For a given r and guess V^u, the function `Vu_error` calculates the error in the Bellman equation under the optimal search intensity.

We'll soon use this as an input to computing V^u.

```
# The error in the Bellman equation that requires equality at
# the optimal choices.
def Vu_error(self,Vu,r):
    β= self.β
    Ve = self.Ve

    a = invp_prime(1/(β*(Ve-Vu)),r)
    error = u(self,0) -a + β*(p(a,r)*Ve + (1-p(a,r))*Vu) - Vu
    return error
```

Since the calibration exercise is to match the hazard rate under autarky to the data, we must find an interest rate r to match `p(a,r) = 0.1`.

The function below `r_error` calculates, for a given guess of r the difference between the model implied equilibrium hazard rate and 0.1.

This will be used to solve for the a calibrated r^*.

```
# The error of our p(a^*) relative to our calibration target
def r_error(self,r):
    β = self.β
    Ve = self.Ve

    Vu_star = sp.optimize.fsolve(Vu_error_Λ,15000,args = (r))
    a_star = invp_prime(1/(β*(Ve-Vu_star)),r) # Assuming a>0
    return    p(a_star,r) - 0.1
```

Now, let us create an instance of the model with our parametrization

```
params = params_instance(r = 1e-2)
# Create some lambda functions useful for fsolve function
Vu_error_Λ =  lambda Vu,r: Vu_error(params,Vu,r)
r_error_Λ =  lambda r: r_error(params,r)
```

We want to compute an r that is consistent with the hazard rate 0.1 in autarky.

To do so, we will use a bisection strategy.

```
r_calibrated = sp.optimize.brentq(r_error_Λ,1e-10,1-1e-10)
print(f"Interest rate to match 0.1 hazard rate: r = {r_calibrated}")

Vu_aut = sp.optimize.fsolve(Vu_error_Λ,15000,args = (r_calibrated))[0]
a_aut = invp_prime(1/(params.β*(params.Ve-Vu_aut)),r_calibrated)

print(f"Check p at r: {p(a_aut,r_calibrated)}")
```

```
Interest rate to match 0.1 hazard rate: r = 0.0003431409393866592
Check p at r: 0.10000000000001996
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.11/site-packages/scipy/optimize/
↪_minpack_py.py:177: RuntimeWarning: The iteration is not making good progress,␣
↪as measured by the
  improvement from the last five Jacobian evaluations.
  warnings.warn(msg, RuntimeWarning)
```

Now that we have calibrated our interest rate r, we can continue with solving the model with private information.

1.3.4 Computation under Private Information

Our approach to solving the full model is a variant on Judd (1998) [Jud98], who uses a polynomial to approximate the value function and a numerical optimizer to perform the optimization at each iteration.

In contrast, we will use cubic splines to interpolate across a pre-set grid of points to approximate the value function. For further details of the Judd (1998) [Jud98] method, see [LS18], Section 5.7.

Our strategy involves finding a function $C(V)$ – the expected cost of giving the worker value V – that satisfies the Bellman equation:

$$C(V) = \min_{c,a,V^u} \{c + \beta [1 - p(a)] C(V^u)\} \tag{1.15}$$

To solve this model, notice that in equations (1.9) and (1.11), we have analytical solutions of c and a in terms of (at most) promised value V and V^u (and other parameters).

We can substitute these equations for c and a and obtain the functional equation (1.12) that we want to solve.

```
def calc_c(self,Vu,V,a):
    '''
    Calculates the optimal consumption choice coming from the constraint of the␣
↪insurer's problem
    (which is also a Bellman equation)
    '''
    β,Ve,r = self.β,self.Ve,self.r

    c = u_inv(self,V + a - β*(p(a,r)*Ve + (1-p(a,r))*Vu))
    return c

def calc_a(self,Vu):
    '''
    Calculates the optimal effort choice coming from the worker's effort optimality␣
↪condition.
    '''

    r,β,Ve = self.r,self.β,self.Ve

    a_temp = np.log(r*β*(Ve - Vu))/r
    a = max(0,a_temp)
    return a
```

With these analytical solutions for optimal c and a in hand, we can reduce the minimization to (1.12) in the single variable V^u.

With this in hand, we have our algorithm.

1.3.5 Algorithm

1. Fix a set of grid points $grid_V$ for V and Vu_{grid} for V^u
2. Guess a function $C_0(V)$ that is evaluated at a grid $grid_V$.
3. For each point in $grid_V$ find the V^u that minimizes the expression on right side of (1.12). We find the minimum by evaluating the right side of (1.12) at each point in Vu_{grid} and then finding the minimum using cubic splines.
4. Evaluating the minimum across all points in $grid_V$ gives you another function $C_1(V)$.
5. If $C_0(V)$ and $C_1(V)$ are sufficiently different, then repeat steps 3-4 again. Otherwise, we are done.
6. Thus, the iterations are $C_{j+1}(V) = \min_{c,a,V^u}\{c - \beta[1 - p(a)]C_j(V)\}$.

The function `iterate_C` below executes step 3 in the above algorithm.

```python
# Operator iterate_C that calculates the next iteration of the cost function.
def iterate_C(self,C_old,Vu_grid):

    '''
    We solve the model by minimising the value function across a grid of possible␣
↪promised values.
    '''
    β,r,n_grid = self.β,self.r,self.n_grid

    C_new = np.zeros(n_grid)
    cons_star = np.zeros(n_grid)
    a_star = np.zeros(n_grid)
    V_star = np.zeros(n_grid)

    C_new2 = np.zeros(n_grid)
    V_star2 = np.zeros(n_grid)

    for V_i in range(n_grid):
        C_Vi_temp = np.zeros(n_grid)
        cons_Vi_temp = np.zeros(n_grid)
        a_Vi_temp = np.zeros(n_grid)

        for Vu_i in range(n_grid):
            a_i = calc_a(self,Vu_grid[Vu_i])
            c_i = calc_c(self,Vu_grid[Vu_i],Vu_grid[V_i],a_i)

            C_Vi_temp[Vu_i] = c_i + β*(1-p(a_i,r))*C_old[Vu_i]
            cons_Vi_temp[Vu_i] = c_i
            a_Vi_temp[Vu_i] = a_i

        # Interpolate across the grid to get better approximation of the minimum
        C_Vi_temp_interp = sp.interpolate.interp1d(Vu_grid,C_Vi_temp, kind = 'cubic')
        cons_Vi_temp_interp = sp.interpolate.interp1d(Vu_grid,cons_Vi_temp, kind =
↪'cubic')
        a_Vi_temp_interp = sp.interpolate.interp1d(Vu_grid,a_Vi_temp, kind = 'cubic')

        res = sp.optimize.minimize_scalar(C_Vi_temp_interp,method='bounded',bounds =␣
↪(Vu_min,Vu_max))
        V_star[V_i] = res.x
        C_new[V_i] = res.fun

        # Save the associated consumpton and search policy functions as well
        cons_star[V_i] = cons_Vi_temp_interp(V_star[V_i])
        a_star[V_i] = a_Vi_temp_interp(V_star[V_i])

    return C_new,V_star,cons_star,a_star
```

The below code executes steps 4 and 5 in the Algorithm until convergence to a function $C^*(V)$.

```python
def solve_incomplete_info_model(self,Vu_grid,Vu_aut,tol = 1e-6,max_iter = 10000):
    iter = 0
    error = 1

    C_init = np.ones(self.n_grid)*0
    C_old = np.copy(C_init)
```

(continues on next page)

(continued from previous page)

```
    while iter<max_iter and error >tol:
        C_new,V_new,cons_star,a_star = iterate_C(self,C_old,Vu_grid)
        error = np.max(np.abs(C_new - C_old))

        #Only print the iterations every 50 steps
        if iter % 50 ==0:
            print(f"Iteration: {iter}, error:{error}")
        C_old = np.copy(C_new)
        iter+=1

    return C_new,V_new,cons_star,a_star
```

1.4 Outcomes

Using the above functions, we create another instance of the parameters with the correctly calibrated interest rate, r.

```
##? Create another instance with the correct r now
params = params_instance(r = r_calibrated)

#Set up grid
Vu_min = Vu_aut
Vu_max = params.Ve - 1/(params.β*p_prime(0,params.r))
Vu_grid = np.linspace(Vu_min,Vu_max,params.n_grid)

#Solve model
C_star,V_star,cons_star,a_star = solve_incomplete_info_model(params,Vu_grid,Vu_aut,
 ↪tol = 1e-6,max_iter = 10000) #,cons_star,a_star

# Since we have the policy functions in grid form, we will interpolate them to be␣
 ↪able to
# evaluate any promised value
cons_star_interp = sp.interpolate.interp1d(Vu_grid,cons_star)
a_star_interp = sp.interpolate.interp1d(Vu_grid,a_star)
V_star_interp = sp.interpolate.interp1d(Vu_grid,V_star)
```

```
Iteration: 0, error:72.95964854907824
```

```
Iteration: 50, error:12.222761762480786
```

```
Iteration: 100, error:0.12875960366727668
```

```
Iteration: 150, error:0.0009402349710398994
```

```
Iteration: 200, error:6.115462838351959e-06
```

1.4.1 Replacement Ratios and Continuation Values

We want to graph the replacement ratio (c/w) and search effort a as functions of the duration of unemployment.

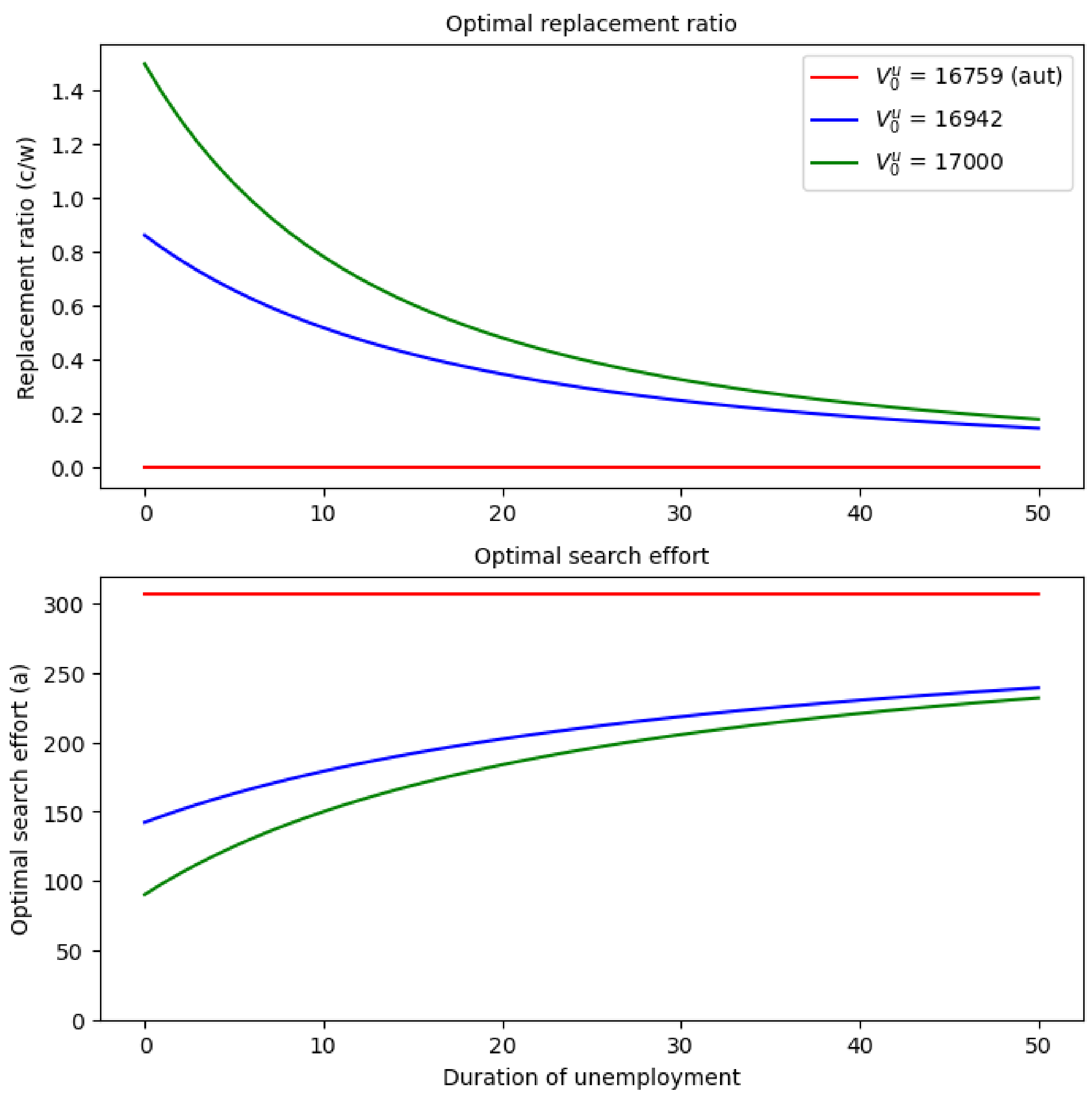
We'll do this for three levels of V_0, the lowest being the autarky value V_{aut}.

We accomplish this by using the optimal policy functions `V_star`, `cons_star` and `a_star` computed above as well the following iterative procedure:

```
# Replacement ratio and effort as a function of unemployment duration
T_max = 52
Vu_t = np.empty((T_max,3))
cons_t = np.empty((T_max-1,3))
a_t = np.empty((T_max-1,3))

# Calculate the replacement ratios depending on different initial
# promised values
Vu_0_hold = np.array([Vu_aut,16942,17000])
```

```
for i,Vu_0, in enumerate(Vu_0_hold):
    Vu_t[0,i] = Vu_0
    for t in range(1,T_max):
        cons_t[t-1,i] = cons_star_interp(Vu_t[t-1,i])
        a_t[t-1,i] = a_star_interp(Vu_t[t-1,i])
        Vu_t[t,i] = V_star_interp(Vu_t[t-1,i])
```

```
fontSize = 10
plt.rc('font', size=fontSize)          # controls default text sizes
plt.rc('axes', titlesize=fontSize)     # fontsize of the axes title
plt.rc('axes', labelsize=fontSize)    # fontsize of the x and y labels
plt.rc('xtick', labelsize=fontSize)    # fontsize of the tick labels
plt.rc('ytick', labelsize=fontSize)    # fontsize of the tick labels
plt.rc('legend', fontsize=fontSize)    # legend fontsize

f1 = plt.figure(figsize = (8,8))
plt.subplot(2,1,1)
plt.plot(range(T_max-1),cons_t[:,0]/params.w,label = '$V^u_0$ = 16759 (aut)',color =
 ↪'red')
plt.plot(range(T_max-1),cons_t[:,1]/params.w,label = '$V^u_0$ = 16942',color = 'blue')
plt.plot(range(T_max-1),cons_t[:,2]/params.w,label = '$V^u_0$ = 17000',color = 'green
 ↪')
plt.ylabel("Replacement ratio (c/w)")
plt.legend()
plt.title("Optimal replacement ratio")

plt.subplot(2,1,2)
plt.plot(range(T_max-1),a_t[:,0],color = 'red')
plt.plot(range(T_max-1),a_t[:,1],color = 'blue')
plt.plot(range(T_max-1),a_t[:,2],color = 'green')
plt.ylim(0,320)
plt.ylabel("Optimal search effort (a)")
plt.xlabel("Duration of unemployment")
plt.title("Optimal search effort")
plt.show()
```



For an initial promised value $V^u = V_{\text{aut}}$, the planner chooses the autarky level of 0 for the replacement ratio and instructs the worker to search at the autarky search intensity, regardless of the duration of unemployment

But for $V^u > V_{\text{aut}}$, the planner makes the replacement ratio decline and search effort increase with the duration of unemployment.

1.4.2 Interpretations

The downward slope of the replacement ratio when $V^u > V_{\text{aut}}$ is a consequence of the the planner's limited information about the worker's search effort.

By providing the worker with a duration-dependent schedule of replacement ratios, the planner induces the worker in effect to reveal his/her search effort to the planner.

We saw earlier that with full information, the planner would smooth consumption over an unemployment spell by keeping the replacement ratio constant.

With private information, the planner can't observe the worker's search effort and therefore makes the replacement ratio fall.

Evidently, search effort rise as the duration of unemployment increases, especially early in an unemployment spell.

There is a **carrot-and-stick** aspect to the replacement rate and search effort schedules:

- the **carrot** occurs in the forms of high compensation and low search effort early in an unemployment spell.
- the **stick** occurs in the low compensation and high effort later in the spell.

We shall encounter a related carrot-and-stick feature in our other lectures about dynamic programming squared.

The planner offers declining benefits and induces increased search effort as the duration of an unemployment spell rises in order to provide an unemployed worker with proper incentives, not to punish an unlucky worker who has been unemployed for a long time.

The planner believes that a worker who has been unemployed a long time is unlucky, not that he has done anything wrong (i.e., has not lived up to the contract).

Indeed, the contract is designed to induce the unemployed workers to search in the way the planner expects.

The falling consumption and rising search effort of the unlucky ones with long unemployment spells are simply costs that have to be paid in order to provide proper incentives.

CHAPTER

TWO

STACKELBERG PLANS

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

2.1 Overview

This lecture formulates and computes a plan that a **Stackelberg leader** uses to manipulate forward-looking decisions of a **Stackelberg follower** that depend on continuation sequences of decisions made once and for all by the Stackelberg leader at time 0.

To facilitate computation and interpretation, we formulate things in a context that allows us to apply dynamic programming for linear-quadratic models.

Technically, our calculations are closely related to ones described this lecture.

From the beginning, we carry along a linear-quadratic model of duopoly in which firms face adjustment costs that make them want to forecast actions of other firms that influence future prices.

Let's start with some standard imports:

```
import numpy as np
import numpy.linalg as la
import quantecon as qe
from quantecon import LQ
import matplotlib.pyplot as plt
%matplotlib inline
```

2.2 Duopoly

Time is discrete and is indexed by $t = 0, 1, \ldots$.

Two firms produce a single good whose demand is governed by the linear inverse demand curve

$$p_t = a_0 - a_1(q_{1t} + q_{2t})$$

where q_{it} is output of firm i at time t and a_0 and a_1 are both positive.

q_{10}, q_{20} are given numbers that serve as initial conditions at time 0.

By incurring a cost equal to

$$\gamma v_{it}^2, \quad \gamma > 0,$$

firm i can change its output according to

$$q_{it+1} = q_{it} + v_{it}$$

Firm i's profits at time t equal

$$\pi_{it} = p_t q_{it} - \gamma v_{it}^2$$

Firm i wants to maximize the present value of its profits

$$\sum_{t=0}^{\infty} \beta^t \pi_{it}$$

where $\beta \in (0, 1)$ is a time discount factor.

2.2.1 Stackelberg Leader and Follower

Each firm $i = 1, 2$ chooses a sequence $\vec{q}_i \equiv \{q_{it+1}\}_{t=0}^{\infty}$ once and for all at time 0.

We let firm 2 be a **Stackelberg leader** and firm 1 be a **Stackelberg follower**.

The leader firm 2 goes first and chooses $\{q_{2t+1}\}_{t=0}^{\infty}$ once and for all at time 0.

Knowing that firm 2 has chosen $\{q_{2t+1}\}_{t=0}^{\infty}$, the follower firm 1 goes second and chooses $\{q_{1t+1}\}_{t=0}^{\infty}$ once and for all at time 0.

In choosing $\vec{q}_2$, firm 2 takes into account that firm 1 will base its choice of $\vec{q}_1$ on firm 2's choice of $\vec{q}_2$.

2.2.2 Statement of Leader's and Follower's Problems

We can express firm 1's problem as

$$\max_{\vec{q}_1} \Pi_1(\vec{q}_1; \vec{q}_2)$$

where the appearance behind the semi-colon indicates that $\vec{q}_2$ is given.

Firm 1's problem induces the best response mapping

$$\vec{q}_1 = B(\vec{q}_2)$$

(Here B maps a sequence into a sequence)

The Stackelberg leader's problem is

$$\max_{\vec{q}_2} \Pi_2(B(\vec{q}_2), \vec{q}_2)$$

whose maximizer is a sequence $\vec{q}_2$ that depends on the initial conditions q_{10}, q_{20} and the parameters of the model a_0, a_1, γ.

This formulation captures key features of the model

- Both firms make once-and-for-all choices at time 0.
- This is true even though both firms are choosing sequences of quantities that are indexed by **time**.
- The Stackelberg leader chooses first **within time** 0, knowing that the Stackelberg follower will choose second **within time** 0.

While our abstract formulation reveals the timing protocol and equilibrium concept well, it obscures details that must be addressed when we want to compute and interpret a Stackelberg plan and the follower's best response to it.

To gain insights about these things, we study them in more detail.

2.2.3 Firms' Problems

Firm 1 acts as if firm 2's sequence $\{q_{2t+1}\}_{t=0}^{\infty}$ is given and beyond its control.

Firm 2 knows that firm 1 chooses second and takes this into account in choosing $\{q_{2t+1}\}_{t=0}^{\infty}$.

In the spirit of *working backward*, we study firm 1's problem first, taking $\{q_{2t+1}\}_{t=0}^{\infty}$ as given.

We can formulate firm 1's optimum problem in terms of the Lagrangian

$$L = \sum_{t=0}^{\infty} \beta^t \{a_0 q_{1t} - a_1 q_{1t}^2 - a_1 q_{1t} q_{2t} - \gamma v_{1t}^2 + \lambda_t [q_{1t} + v_{1t} - q_{1t+1}]\}$$

Firm 1 seeks a maximum with respect to $\{q_{1t+1}, v_{1t}\}_{t=0}^{\infty}$ and a minimum with respect to $\{\lambda_t\}_{t=0}^{\infty}$.

We approach this problem using methods described in [LS18], chapter 2, appendix A and [Sar87], chapter IX.

First-order conditions for this problem are

$$\begin{aligned}
\frac{\partial L}{\partial q_{1t}} &= a_0 - 2a_1 q_{1t} - a_1 q_{2t} + \lambda_t - \beta^{-1}\lambda_{t-1} = 0, \quad t \geq 1 \\
\frac{\partial L}{\partial v_{1t}} &= -2\gamma v_{1t} + \lambda_t = 0, \quad t \geq 0
\end{aligned}$$

These first-order conditions and the constraint $q_{1t+1} = q_{1t} + v_{1t}$ can be rearranged to take the form

$$\begin{aligned} v_{1t} &= \beta v_{1t+1} + \frac{\beta a_0}{2\gamma} - \frac{\beta a_1}{\gamma} q_{1t+1} - \frac{\beta a_1}{2\gamma} q_{2t+1} \\ q_{t+1} &= q_{1t} + v_{1t} \end{aligned}$$

We can substitute the second equation into the first equation to obtain

$$(q_{1t+1} - q_{1t}) = \beta(q_{1t+2} - q_{1t+1}) + c_0 - c_1 q_{1t+1} - c_2 q_{2t+1}$$

where $c_0 = \frac{\beta a_0}{2\gamma}, c_1 = \frac{\beta a_1}{\gamma}, c_2 = \frac{\beta a_1}{2\gamma}$.

This equation can in turn be rearranged to become

$$-q_{1t} + (1 + \beta + c_1) q_{1t+1} - \beta q_{1t+2} = c_0 - c_2 q_{2t+1} \tag{2.1}$$

Equation (2.1) is a second-order difference equation in the sequence $\vec{q}_1$ whose solution we want.

It satisfies **two boundary conditions:**

- an initial condition that $q_{1,0}$, which is given
- a terminal condition requiring that $\lim_{T \to +\infty} \beta^T q_{1t}^2 < +\infty$

Using the lag operators described in [Sar87], chapter IX, difference equation (2.1) can be written as

$$\beta(1 - \frac{1 + \beta + c_1}{\beta} L + \beta^{-1} L^2) q_{1t+2} = -c_0 + c_2 q_{2t+1}$$

The polynomial in the lag operator on the left side can be **factored** as

$$(1 - \frac{1 + \beta + c_1}{\beta} L + \beta^{-1} L^2) = (1 - \delta_1 L)(1 - \delta_2 L) \tag{2.2}$$

where $0 < \delta_1 < 1 < \frac{1}{\sqrt{\beta}} < \delta_2$.

Because $\delta_2 > \frac{1}{\sqrt{\beta}}$ the operator $(1 - \delta_2 L)$ contributes an **unstable** component if solved **backwards** but a **stable** component if solved **forwards**.

Mechanically, write

$$(1 - \delta_2 L) = -\delta_2 L(1 - \delta_2^{-1} L^{-1})$$

and compute the following inverse operator

$$\left[-\delta_2 L(1 - \delta_2^{-1} L^{-1})\right]^{-1} = -\delta_2 (1 - \delta_2{}^{-1})^{-1} L^{-1}$$

Operating on both sides of equation (2.2) with β^{-1} times this inverse operator gives the follower's decision rule for setting q_{1t+1} in the **feedback-feedforward** form

$$q_{1t+1} = \delta_1 q_{1t} - c_0 \delta_2^{-1} \beta^{-1} \frac{1}{1 - \delta_2^{-1}} + c_2 \delta_2^{-1} \beta^{-1} \sum_{j=0}^{\infty} \delta_2^j q_{2t+j+1}, \quad t \geq 0 \tag{2.3}$$

The problem of the Stackelberg leader firm 2 is to choose the sequence $\{q_{2t+1}\}_{t=0}^{\infty}$ to maximize its discounted profits

$$\sum_{t=0}^{\infty} \beta^t \{(a_0 - a_1(q_{1t} + q_{2t})) q_{2t} - \gamma(q_{2t+1} - q_{2t})^2\}$$

subject to the sequence of constraints (2.3) for $t \geq 0$.

We can put a sequence $\{\theta_t\}_{t=0}^{\infty}$ of Lagrange multipliers on the sequence of equations (2.3) and formulate the following Lagrangian for the Stackelberg leader firm 2's problem

$$\begin{aligned}\tilde{L} = & \sum_{t=0}^{\infty} \beta^t \{(a_0 - a_1(q_{1t} + q_{2t}))q_{2t} - \gamma(q_{2t+1} - q_{2t})^2\} \\ & + \sum_{t=0}^{\infty} \beta^t \theta_t \{\delta_1 q_{1t} - c_0 \delta_2^{-1} \beta^{-1} \frac{1}{1-\delta_2^{-1}} + c_2 \delta_2^{-1} \beta^{-1} \sum_{j=0}^{\infty} \delta_2^{-j} q_{2t+j+1} - q_{1t+1}\}\end{aligned} \tag{2.4}$$

subject to initial conditions for q_{1t}, q_{2t} at $t = 0$.

Remarks: We have formulated the Stackelberg problem in a space of sequences.

The max-min problem associated with firm 2's Lagrangian (2.4) is unpleasant because the time t component of firm 2's payoff function depends on the entire future of its choices of $\{q_{2t+j}\}_{j=0}^{\infty}$.

This renders a direct attack on the problem in the space of sequences cumbersome.

Therefore, below we will formulate the Stackelberg leader's problem recursively.

We'll proceed by putting our duopoly model into a broader class of models with the same general structure.

2.3 Stackelberg Problem

We formulate a class of linear-quadratic Stackelberg leader-follower problems of which our duopoly model is an instance.

We use the optimal linear regulator (a.k.a. the linear-quadratic dynamic programming problem described in LQ Dynamic Programming problems) to represent a Stackelberg leader's problem recursively.

Let z_t be an $n_z \times 1$ vector of **natural state variables**.

Let x_t be an $n_x \times 1$ vector of endogenous forward-looking variables that are physically free to jump at t.

In our duopoly example $x_t = v_{1t}$, the time t decision of the Stackelberg **follower**.

Let u_t be a vector of decisions chosen by the Stackelberg leader at t.

The z_t vector is inherited from the past.

But x_t is a decision made by the Stackelberg follower at time t that is the follower's best response to the choice of an entire sequence of decisions made by the Stackelberg leader at time $t = 0$.

Let

$$y_t = \begin{bmatrix} z_t \\ x_t \end{bmatrix}$$

Represent the Stackelberg leader's one-period loss function as

$$r(y, u) = y'Ry + u'Qu$$

Subject to an initial condition for z_0, but not for x_0, the Stackelberg leader wants to maximize

$$-\sum_{t=0}^{\infty} \beta^t r(y_t, u_t) \tag{2.5}$$

The Stackelberg leader faces the model

$$\begin{bmatrix} I & 0 \\ G_{21} & G_{22} \end{bmatrix} \begin{bmatrix} z_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} \hat{A}_{11} & \hat{A}_{12} \\ \hat{A}_{21} & \hat{A}_{22} \end{bmatrix} \begin{bmatrix} z_t \\ x_t \end{bmatrix} + \hat{B} u_t \tag{2.6}$$

We assume that the matrix $\begin{bmatrix} I & 0 \\ G_{21} & G_{22} \end{bmatrix}$ on the left side of equation (2.6) is invertible, so that we can multiply both sides by its inverse to obtain

$$\begin{bmatrix} z_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} z_t \\ x_t \end{bmatrix} + Bu_t \tag{2.7}$$

or

$$y_{t+1} = Ay_t + Bu_t \tag{2.8}$$

2.3.1 Interpretation of Second Block of Equations

The Stackelberg follower's best response mapping is summarized by the second block of equations of (2.7).

In particular, these equations are the first-order conditions of the Stackelberg follower's optimization problem (i.e., its Euler equations).

These Euler equations summarize the forward-looking aspect of the follower's behavior and express how its time t decision depends on the leader's actions at times $s \geq t$.

When combined with a stability condition to be imposed below, the Euler equations summarize the follower's best response to the sequence of actions by the leader.

The Stackelberg leader maximizes (2.5) by choosing sequences $\{u_t, x_t, z_{t+1}\}_{t=0}^{\infty}$ subject to (2.8) and an initial condition for z_0.

Note that we have an initial condition for z_0 but not for x_0.

x_0 is among the variables to be chosen at time 0 by the Stackelberg leader.

The Stackelberg leader uses its understanding of the responses restricted by (2.8) to manipulate the follower's decisions.

2.3.2 More Mechanical Details

For any vector a_t, define $\vec{a}_t = [a_t, a_{t+1} \ldots]$.

Define a feasible set of $(\vec{y}_1, \vec{u}_0)$ sequences

$$\Omega(y_0) = \{(\vec{y}_1, \vec{u}_0) : y_{t+1} = Ay_t + Bu_t, \forall t \geq 0\}$$

Please remember that the follower's system of Euler equations is embedded in the system of dynamic equations $y_{t+1} = Ay_t + Bu_t$.

Note that the definition of $\Omega(y_0)$ treats y_0 as given.

Although it is taken as given in $\Omega(y_0)$, eventually, the x_0 component of y_0 is to be chosen by the Stackelberg leader.

2.3.3 Two Subproblems

Once again we use backward induction.

We express the Stackelberg problem in terms of **two subproblems**.

Subproblem 1 is solved by a **continuation Stackelberg leader** at each date $t \geq 0$.

Subproblem 2 is solved by the **Stackelberg leader** at $t = 0$.

The two subproblems are designed

- to respect the timing protocol in which the follower chooses $\vec{q}_1$ after seeing $\vec{q}_2$ chosen by the leader
- to make the leader choose $\vec{q}_2$ while respecting that $\vec{q}_1$ will be the follower's best response to $\vec{q}_2$
- to represent the leader's problem recursively by artfully choosing the leader's state variables and the control variables available to the leader

Subproblem 1

$$v(y_0) = \max_{(\vec{y}_1, \vec{u}_0) \in \Omega(y_0)} - \sum_{t=0}^{\infty} \beta^t r(y_t, u_t)$$

Subproblem 2

$$w(z_0) = \max_{x_0} v(y_0)$$

Subproblem 1 takes the vector of forward-looking variables x_0 as given.

Subproblem 2 optimizes over x_0.

The value function $w(z_0)$ tells the value of the Stackelberg plan as a function of the vector of natural state variables z_0 at time 0.

2.4 Two Bellman Equations

We now describe Bellman equations for $v(y)$ and $w(z_0)$.

Subproblem 1

The value function $v(y)$ in subproblem 1 satisfies the Bellman equation

$$v(y) = \max_{u, y^*} \{-r(y,u) + \beta v(y^*)\} \tag{2.9}$$

where the maximization is subject to

$$y^* = Ay + Bu$$

and y^* denotes next period's value.

Substituting $v(y) = -y'Py$ into Bellman equation (2.9) gives

$$-y'Py = \max_{u,y^*} \{-y'Ry - u'Qu - \beta y^{*\prime} P y^*\}$$

which as in lecture linear regulator gives rise to the algebraic matrix Riccati equation

$$P = R + \beta A'PA - \beta^2 A'PB(Q + \beta B'PB)^{-1} B'PA$$

and the optimal decision rule coefficient vector

$$F = \beta(Q + \beta B'PB)^{-1} B'PA$$

where the optimal decision rule is

$$u_t = -F y_t$$

Subproblem 2

We find an optimal x_0 by equating to zero the gradient of $v(y_0)$ with respect to x_0:

$$-2P_{21} z_0 - 2P_{22} x_0 = 0,$$

which implies that

$$x_0 = -P_{22}^{-1} P_{21} z_0 \tag{2.10}$$

2.5 Stackelberg Plan for Duopoly

Now let's map our duopoly model into the above setup.

We formulate a state vector

$$y_t = \begin{bmatrix} z_t \\ x_t \end{bmatrix}$$

where for our duopoly model

$$z_t = \begin{bmatrix} 1 \\ q_{2t} \\ q_{1t} \end{bmatrix}, \quad x_t = v_{1t},$$

where $x_t = v_{1t}$ is the time t decision of the follower firm 1, u_t is the time t decision of the leader firm 2 and

$$v_{1t} = q_{1t+1} - q_{1t}, \quad u_t = q_{2t+1} - q_{2t}.$$

For our duopoly model, initial conditions for the natural state variables in z_t are

$$z_0 = \begin{bmatrix} 1 \\ q_{20} \\ q_{10} \end{bmatrix}$$

while $x_0 = v_{10} = q_{11} - q_{10}$ is a choice variable for the Stackelberg leader firm 2, one that will ultimately be chosen according an optimal rule prescribed by (2.10) for subproblem 2 above.

That the Stackelberg leader firm 2 chooses $x_0 = v_{10}$ is subtle.

Of course, $x_0 = v_{10}$ emerges from the feedback-feedforward solution (2.3) of firm 1's system of Euler equations, so that it is actually firm 1 that sets x_0.

But firm 2 manipulates firm 1's choice through firm 2's choice of the sequence $\vec{q}_{2,1} = \{q_{2t+1}\}_{t=0}^{\infty}$.

2.5.1 Calculations to Prepare Duopoly Model

Now we'll proceed to cast our duopoly model within the framework of the more general linear-quadratic structure described above.

That will allow us to compute a Stackelberg plan simply by enlisting a Riccati equation to solve a linear-quadratic dynamic program.

As emphasized above, firm 1 acts as if firm 2's decisions $\{q_{2t+1}, v_{2t}\}_{t=0}^{\infty}$ are given and beyond its control.

2.5.2 Firm 1's Problem

We again formulate firm 1's optimum problem in terms of the Lagrangian

$$L = \sum_{t=0}^{\infty} \beta^t \{a_0 q_{1t} - a_1 q_{1t}^2 - a_1 q_{1t} q_{2t} - \gamma v_{1t}^2 + \lambda_t [q_{1t} + v_{1t} - q_{1t+1}]\}$$

Firm 1 seeks a maximum with respect to $\{q_{1t+1}, v_{1t}\}_{t=0}^{\infty}$ and a minimum with respect to $\{\lambda_t\}_{t=0}^{\infty}$.

First-order conditions for this problem are

$$\begin{aligned} \frac{\partial L}{\partial q_{1t}} &= a_0 - 2a_1 q_{1t} - a_1 q_{2t} + \lambda_t - \beta^{-1}\lambda_{t-1} = 0, \quad t \geq 1 \\ \frac{\partial L}{\partial v_{1t}} &= -2\gamma v_{1t} + \lambda_t = 0, \quad t \geq 0 \end{aligned}$$

These first-order order conditions and the constraint $q_{1t+1} = q_{1t} + v_{1t}$ can be rearranged to take the form

$$\begin{aligned} v_{1t} &= \beta v_{1t+1} + \frac{\beta a_0}{2\gamma} - \frac{\beta a_1}{\gamma} q_{1t+1} - \frac{\beta a_1}{2\gamma} q_{2t+1} \\ q_{t+1} &= q_{1t} + v_{1t} \end{aligned}$$

We use these two equations as components of the following linear system that confronts a Stackelberg continuation leader at time t

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{\beta a_0}{2\gamma} & -\frac{\beta a_1}{2\gamma} & -\frac{\beta a_1}{\gamma} & \beta \end{bmatrix} \begin{bmatrix} 1 \\ q_{2t+1} \\ q_{1t+1} \\ v_{1t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ q_{2t} \\ q_{1t} \\ v_{1t} \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} v_{2t}$$

Time t revenues of firm 2 are $\pi_{2t} = a_0 q_{2t} - a_1 q_{2t}^2 - a_1 q_{1t} q_{2t}$ which evidently equal

$$z_t' R_1 z_t \equiv \begin{bmatrix} 1 \\ q_{2t} \\ q_{1t} \end{bmatrix}' \begin{bmatrix} 0 & \frac{a_0}{2} & 0 \\ \frac{a_0}{2} & -a_1 & -\frac{a_1}{2} \\ 0 & -\frac{a_1}{2} & 0 \end{bmatrix} \begin{bmatrix} 1 \\ q_{2t} \\ q_{1t} \end{bmatrix}$$

If we set $Q = \gamma$, then firm 2's period t profits can then be written

$$y_t' R y_t - Q v_{2t}^2$$

where

$$y_t = \begin{bmatrix} z_t \\ x_t \end{bmatrix}$$

with $x_t = v_{1t}$ and

$$R = \begin{bmatrix} R_1 & 0 \\ 0 & 0 \end{bmatrix}$$

We'll report results of implementing this code soon.

But first, we want to represent the Stackelberg leader's optimal choices recursively.

It is important to do this for several reasons:

- properly to interpret a representation of the Stackelberg leader's choice as a sequence of history-dependent functions
- to formulate a recursive version of the follower's choice problem

First, let's get a recursive representation of the Stackelberg leader's choice of $\vec{q}_2$ for our duopoly model.

2.6 Recursive Representation of Stackelberg Plan

In order to attain an appropriate representation of the Stackelberg leader's history-dependent plan, we will employ what amounts to a version of the **Big K, little k** device often used in macroeconomics by distinguishing z_t, which depends partly on decisions x_t of the followers, from another vector $\check{z}_t$, which does not.

We will use $\check{z}_t$ and its history $\check{z}^t = [\check{z}_t, \check{z}_{t-1}, \dots, \check{z}_0]$ to describe the sequence of the Stackelberg leader's decisions that the Stackelberg follower takes as given.

Thus, we let $\check{y}_t' = [\check{z}_t' \quad \check{x}_t']$ with initial condition $\check{z}_0 = z_0$ given.

That we distinguish $\check{z}_t$ from z_t is part and parcel of the **Big K, little k** device in this instance.

We have demonstrated that a Stackelberg plan for $\{u_t\}_{t=0}^\infty$ has a recursive representation

$$\begin{aligned} \check{x}_0 &= -P_{22}^{-1} P_{21} z_0 \\ u_t &= -F \check{y}_t, \quad t \geq 0 \\ \check{y}_{t+1} &= (A - BF) \check{y}_t, \quad t \geq 0 \end{aligned}$$

From this representation, we can deduce the sequence of functions $\sigma = \{\sigma_t(\check{z}^t)\}_{t=0}^\infty$ that comprise a Stackelberg plan.

For convenience, let $\check{A} \equiv A - BF$ and partition $\check{A}$ conformably to the partition $y_t = \begin{bmatrix} \check{z}_t \\ \check{x}_t \end{bmatrix}$ as

$$\begin{bmatrix} \check{A}_{11} & \check{A}_{12} \\ \check{A}_{21} & \check{A}_{22} \end{bmatrix}$$

Let $H_0^0 \equiv -P_{22}^{-1} P_{21}$ so that $\check{x}_0 = H_0^0 \check{z}_0$.

Then iterations on $\check{y}_{t+1} = \check{A} \check{y}_t$ starting from initial condition $\check{y}_0 = \begin{bmatrix} \check{z}_0 \\ H_0^0 \check{z}_0 \end{bmatrix}$ imply that for $t \geq 1$

$$\check{x}_t = \sum_{j=1}^t H_j^t \check{z}_{t-j}$$

where

$$\begin{aligned} H_1^t &= \check{A}_{21} \\ H_2^t &= \check{A}_{22} \check{A}_{21} \\ \vdots \quad & \quad \vdots \\ H_{t-1}^t &= \check{A}_{22}^{t-2} \check{A}_{21} \\ H_t^t &= \check{A}_{22}^{t-1} (\check{A}_{21} + \check{A}_{22} H_0^0) \end{aligned}$$

An optimal decision rule for the Stackelberg leader's choice of u_t is

$$u_t = -F \check{y}_t \equiv - \begin{bmatrix} F_z & F_x \end{bmatrix} \begin{bmatrix} \check{z}_t \\ x_t \end{bmatrix}$$

or

$$u_t = -F_z \check{z}_t - F_x \sum_{j=1}^t H_j^t z_{t-j} = \sigma_t(\check{z}^t) \tag{2.11}$$

Representation (2.11) confirms that whenever $F_x \neq 0$, the typical situation, the time t component σ_t of a Stackelberg plan is **history-dependent**, meaning that the Stackelberg leader's choice u_t depends not just on $\check{z}_t$ but on components of $\check{z}^{t-1}$.

2.6.1 Comments and Interpretations

Because we set $\check{z}_0 = z_0$, it will turn out that $z_t = \check{z}_t$ for all $t \geq 0$.

Then why did we distinguish $\check{z}_t$ from z_t?

The answer is that if we want to present to the Stackelberg **follower** a history-dependent representation of the Stackelberg **leader's** sequence $\vec{q}_2$, we must use representation (2.11) cast in terms of the history $\check{z}^t$ and **not** a corresponding representation cast in terms of z^t.

2.7 Dynamic Programming and Time Consistency of Follower's Problem

Given the sequence $\vec{q}_2$ chosen by the Stackelberg leader in our duopoly model, it turns out that the Stackelberg **follower's** problem is recursive in the *natural* state variables that confront a follower at any time $t \geq 0$.

This means that the follower's plan is time consistent.

To verify these claims, we'll formulate a recursive version of a follower's problem that builds on our recursive representation of the Stackelberg leader's plan and our use of the **Big K, little k** idea.

2.7.1 Recursive Formulation of a Follower's Problem

We now use what amounts to another "Big K, little k" trick (see rational expectations equilibrium) to formulate a recursive version of a follower's problem cast in terms of an ordinary Bellman equation.

Firm 1, the follower, faces $\{q_{2t}\}_{t=0}^{\infty}$ as a given quantity sequence chosen by the leader and believes that its output price at t satisfies

$$p_t = a_0 - a_1(q_{1t} + q_{2t}), \quad t \geq 0$$

Our challenge is to represent $\{q_{2t}\}_{t=0}^{\infty}$ as a given sequence.

To do so, recall that under the Stackelberg plan, firm 2 sets output according to the q_{2t} component of

$$y_{t+1} = \begin{bmatrix} 1 \\ q_{2t} \\ q_{1t} \\ x_t \end{bmatrix}$$

which is governed by

$$y_{t+1} = (A - BF)y_t$$

To obtain a recursive representation of a $\{q_{2t}\}$ sequence that is exogenous to firm 1, we define a state $\tilde{y}_t$

$$\tilde{y}_t = \begin{bmatrix} 1 \\ q_{2t} \\ \tilde{q}_{1t} \\ \tilde{x}_t \end{bmatrix}$$

that evolves according to

$$\tilde{y}_{t+1} = (A - BF)\tilde{y}_t$$

subject to the initial condition $\tilde{q}_{10} = q_{10}$ and $\tilde{x}_0 = x_0$ where $x_0 = -P_{22}^{-1}P_{21}$ as stated above.

Firm 1's state vector is

$$X_t = \begin{bmatrix} \tilde{y}_t \\ q_{1t} \end{bmatrix}$$

It follows that the follower firm 1 faces law of motion

$$\begin{bmatrix} \tilde{y}_{t+1} \\ q_{1t+1} \end{bmatrix} = \begin{bmatrix} A - BF & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \tilde{y}_t \\ q_{1t} \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} x_t \tag{2.12}$$

This specification assures that from the point of the view of firm 1, q_{2t} is an exogenous process.

Here

- $\tilde{q}_{1t}, \tilde{x}_t$ play the role of **Big K**
- q_{1t}, x_t play the role of **little k**

The time t component of firm 1's objective is

$$\tilde{X}_t'\tilde{R}x_t - x_t^2\tilde{Q} = \begin{bmatrix} 1 \\ q_{2t} \\ \tilde{q}_{1t} \\ \tilde{x}_t \\ q_{1t} \end{bmatrix}' \begin{bmatrix} 0 & 0 & 0 & 0 & \frac{a_0}{2} \\ 0 & 0 & 0 & 0 & -\frac{a_1}{2} \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \frac{a_0}{2} & -\frac{a_1}{2} & 0 & 0 & -a_1 \end{bmatrix} \begin{bmatrix} 1 \\ q_{2t} \\ \tilde{q}_{1t} \\ \tilde{x}_t \\ q_{1t} \end{bmatrix} - \gamma x_t^2$$

Firm 1's optimal decision rule is

$$x_t = -\tilde{F}X_t$$

and its state evolves according to

$$\tilde{X}_{t+1} = (\tilde{A} - \tilde{B}\tilde{F})X_t$$

under its optimal decision rule.

Later we shall compute $\tilde{F}$ and verify that when we set

$$X_0 = \begin{bmatrix} 1 \\ q_{20} \\ q_{10} \\ x_0 \\ q_{10} \end{bmatrix}$$

we recover

$$x_0 = -\tilde{F}\tilde{X}_0,$$

which will verify that we have properly set up a recursive representation of the follower's problem facing the Stackelberg leader's $\vec{q}_2$.

2.7.2 Time Consistency of Follower's Plan

The follower can solve its problem using dynamic programming because its problem is recursive in what for it are the **natural state variables**, namely

$$\begin{bmatrix} 1 \\ q_{2t} \\ \tilde{q}_{1t} \\ \tilde{x}_t \end{bmatrix}$$

It follows that the follower's plan is time consistent.

2.8 Computing Stackelberg Plan

Here is our code to compute a Stackelberg plan via the linear-quadratic dynamic program describe above.

Let's use it to compute the Stackelberg plan.

```python
# Parameters
a0 = 10
a1 = 2
β = 0.96
γ = 120
n = 300
tol0 = 1e-8
tol1 = 1e-16
tol2 = 1e-2

βs = np.ones(n)
βs[1:] = β
βs = βs.cumprod()
```

```python
# In LQ form
Alhs = np.eye(4)

# Euler equation coefficients
Alhs[3, :] = β * a0 / (2 * γ), -β * a1 / (2 * γ), -β * a1 / γ, β

Arhs = np.eye(4)
Arhs[2, 3] = 1

Alhsinv = la.inv(Alhs)

A = Alhsinv @ Arhs

B = Alhsinv @ np.array([[0, 1, 0, 0]]).T

R = np.array([[0,      -a0 / 2,      0, 0],
              [-a0 / 2,     a1, a1 / 2, 0],
              [0,       a1 / 2,      0, 0],
              [0,            0,      0, 0]])

Q = np.array([[γ]])

# Solve using QE's LQ class
# LQ solves minimization problems which is why the sign of R and Q was changed
lq = LQ(Q, R, A, B, beta=β)
P, F, d = lq.stationary_values(method='doubling')

P22 = P[3:, 3:]
P21 = P[3:, :3]
P22inv = la.inv(P22)
H_0_0 = -P22inv @ P21

# Simulate forward

π_leader = np.zeros(n)

z0 = np.array([[1, 1, 1]]).T
x0 = H_0_0 @ z0
y0 = np.vstack((z0, x0))

yt, ut = lq.compute_sequence(y0, ts_length=n)[:2]
```

(continues on next page)

(continued from previous page)

```
π_matrix = (R + F. T @ Q @ F)

for t in range(n):
    π_leader[t] = -(yt[:, t].T @ π_matrix @ yt[:, t])

# Display policies
print("Computed policy for Continuation Stackelberg leader\n")
print(f"F = {F}")
```

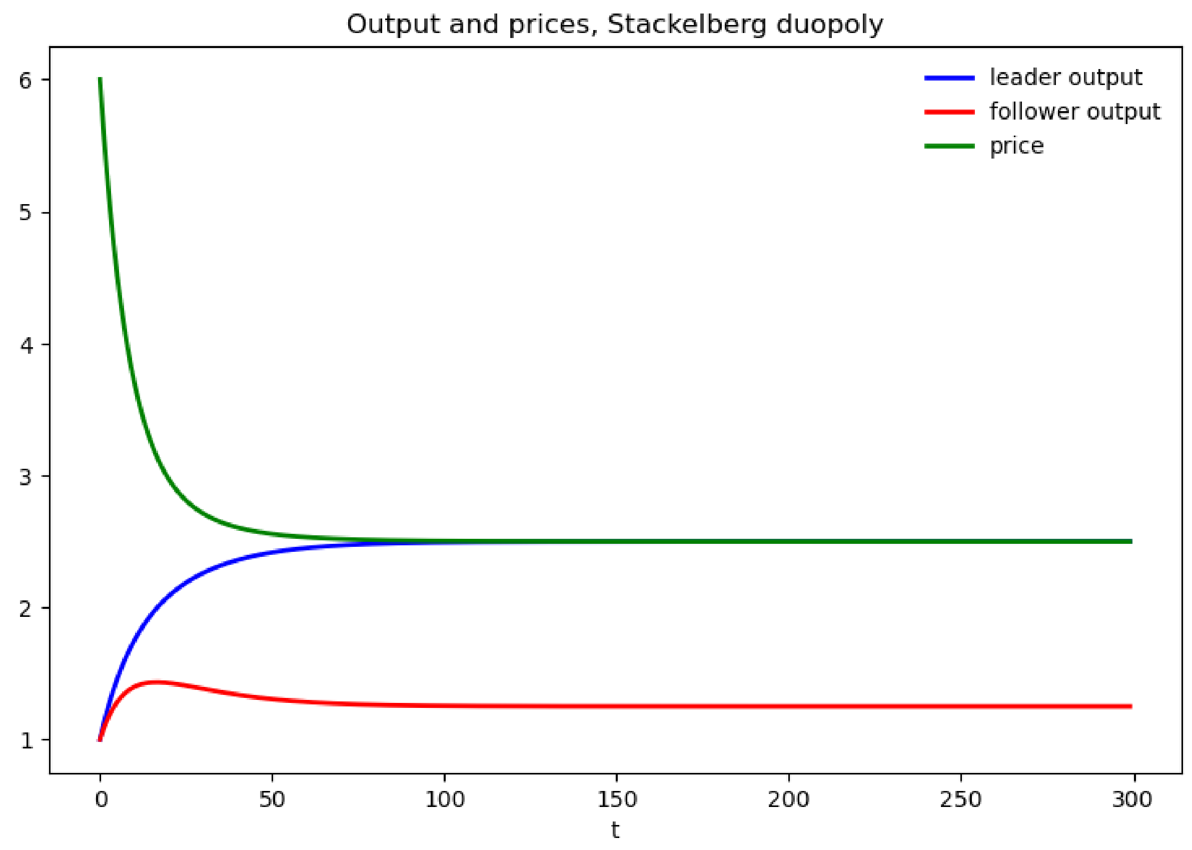
```
Computed policy for Continuation Stackelberg leader

F = [[-1.58004454  0.29461313  0.67480938  6.53970594]]
```

2.9 Time Series for Price and Quantities

Now let's use the code to compute and display outcomes as a Stackelberg plan unfolds.

The following code plots quantities chosen by the Stackelberg leader and follower, together with the equilibrium output price.

```
q_leader = yt[1, :-1]
q_follower = yt[2, :-1]
q = q_leader + q_follower       # Total output, Stackelberg
p = a0 - a1 * q                 # Price, Stackelberg

fig, ax = plt.subplots(figsize=(9, 5.8))
ax.plot(range(n), q_leader, 'b-', lw=2, label='leader output')
ax.plot(range(n), q_follower, 'r-', lw=2, label='follower output')
ax.plot(range(n), p, 'g-', lw=2, label='price')
ax.set_title('Output and prices, Stackelberg duopoly')
ax.legend(frameon=False)
ax.set_xlabel('t')
plt.show()
```



2.9.1 Value of Stackelberg Leader

We'll compute the value $w(x_0)$ attained by the Stackelberg leader, where x_0 is given by the maximizer (2.10) of subproblem 2.

We'll compute it two ways and get the same answer.

In addition to being a useful check on the accuracy of our coding, computing things in these two ways helps us think about the structure of the problem.

```
v_leader_forward = np.sum(βs * π_leader)
v_leader_direct = -yt[:, 0].T @ P @ yt[:, 0]

# Display values
print("Computed values for the Stackelberg leader at t=0:\n")
print(f"v_leader_forward(forward sim) = {v_leader_forward:.4f}")
print(f"v_leader_direct (direct) = {v_leader_direct:.4f}")
```

```
Computed values for the Stackelberg leader at t=0:

v_leader_forward(forward sim) = 150.0316
v_leader_direct (direct) = 150.0324
```

```
# Manually checks whether P is approximately a fixed point
P_next = (R + F.T @ Q @ F + β * (A - B @ F).T @ P @ (A - B @ F))
(P - P_next < tol0).all()
```

```
True
```

```
# Manually checks whether two different ways of computing the
# value function give approximately the same answer
v_expanded = -((y0.T @ R @ y0 + ut[:, 0].T @ Q @ ut[:, 0] +
               β * (y0.T @ (A - B @ F).T @ P @ (A - B @ F) @ y0)))
(v_leader_direct - v_expanded < tol0)[0, 0]
```

```
True
```

2.10 Time Inconsistency of Stackelberg Plan

In the code below we compare two values

- the continuation value $v(y_t) = -y_t' P y_t$ earned by a **continuation Stackelberg leader** who inherits state y_t at t
- the value $w(\hat{x}_t)$ of a **reborn Stackelberg leader** who, at date t along the Stackelberg plan, inherits state z_t at t but who discards x_t from the time t continuation of the original Stackelberg plan and **resets** it to $\hat{x}_t = -P_{22}^{-1} P_{21} z_t$

The difference between these two values is a tell-tale sign of the time inconsistency of the Stackelberg plan

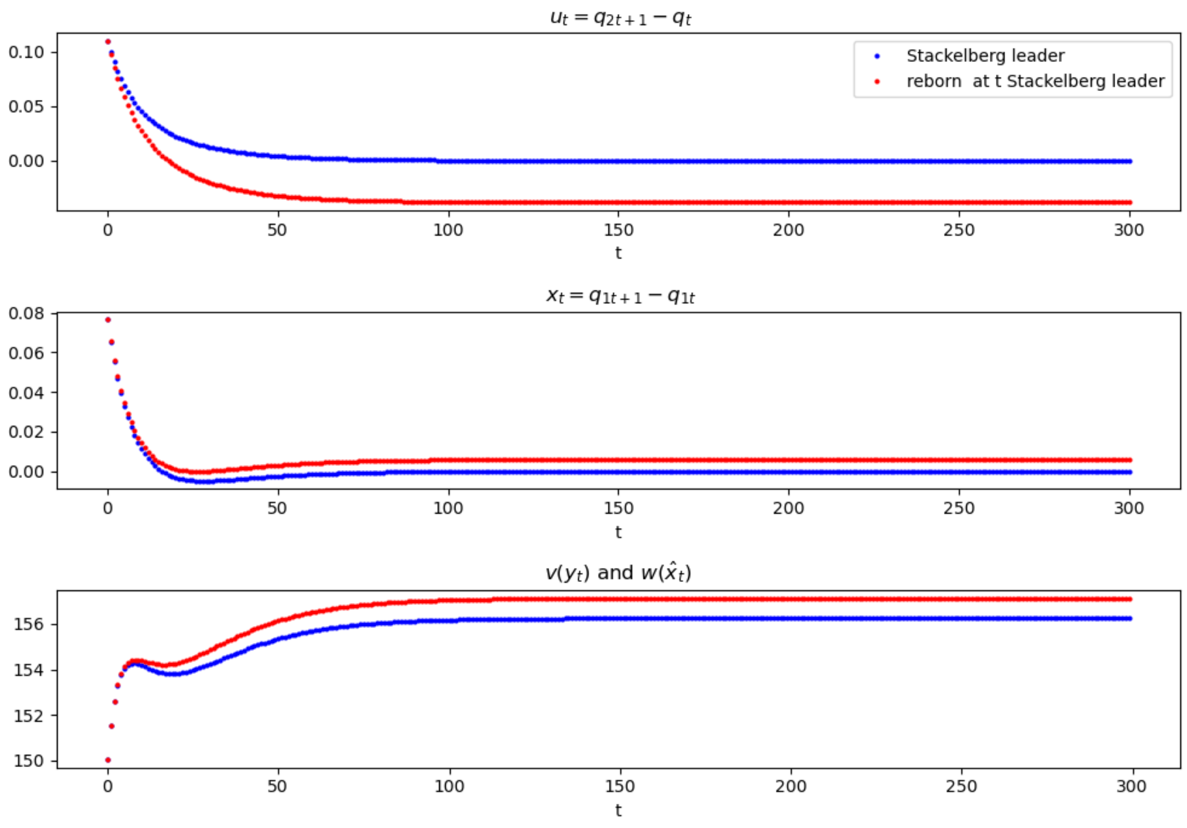
```
# Compute value function over time with a reset at time t
vt_leader = np.zeros(n)
vt_reset_leader = np.empty_like(vt_leader)

yt_reset = yt.copy()
yt_reset[-1, :] = (H_0_0 @ yt[:3, :])

for t in range(n):
    vt_leader[t] = -yt[:, t].T @ P @ yt[:, t]
    vt_reset_leader[t] = -yt_reset[:, t].T @ P @ yt_reset[:, t]
```

```
fig, axes = plt.subplots(3, 1, figsize=(10, 7))

axes[0].plot(range(n+1), (- F @ yt).flatten(), 'bo',
    label='Stackelberg leader', ms=2)
axes[0].plot(range(n+1), (- F @ yt_reset).flatten(), 'ro',
    label='reborn  at t Stackelberg leader', ms=2)
axes[0].set(title=r' $u_{t} = q_{2t+1} - q_t$', xlabel='t')
axes[0].legend()

axes[1].plot(range(n+1), yt[3, :], 'bo', ms=2)
axes[1].plot(range(n+1), yt_reset[3, :], 'ro', ms=2)
axes[1].set(title=r' $x_{t} = q_{1t+1} - q_{1t}$', xlabel='t')

axes[2].plot(range(n), vt_leader, 'bo', ms=2)
axes[2].plot(range(n), vt_reset_leader, 'ro', ms=2)
axes[2].set(title=r'$v(y_{t})$ and $w(\hat x_t)$', xlabel='t')
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```



The figure above shows

- in the third panel that for $t \geq 1$ the **reborn at** t Stackelberg leader's's value $w(\hat{x}_0)$ exceeds the continuation value $v(y_t)$ of the time 0 Stackelberg leader
- in the first panel that for $t \geq 1$ the **reborn at** t Stackelberg leader wants to reduce his output below that prescribed by the time 0 Stackelberg leader
- in the second panel that for $t \geq 1$ the **reborn at** t Stackelberg leader wants to increase the output of the follower firm 2 below that prescribed by the time 0 Stackelberg leader

Taken together, these outcomes express the time inconsistency of the original time 0 Stackelberg leaders's plan.

2.11 Recursive Formulation of Follower's Problem

We now formulate and compute the recursive version of the follower's problem.

We check that the recursive **Big** K **, little** k formulation of the follower's problem produces the same output path $\vec{q}_1$ that we computed when we solved the Stackelberg problem

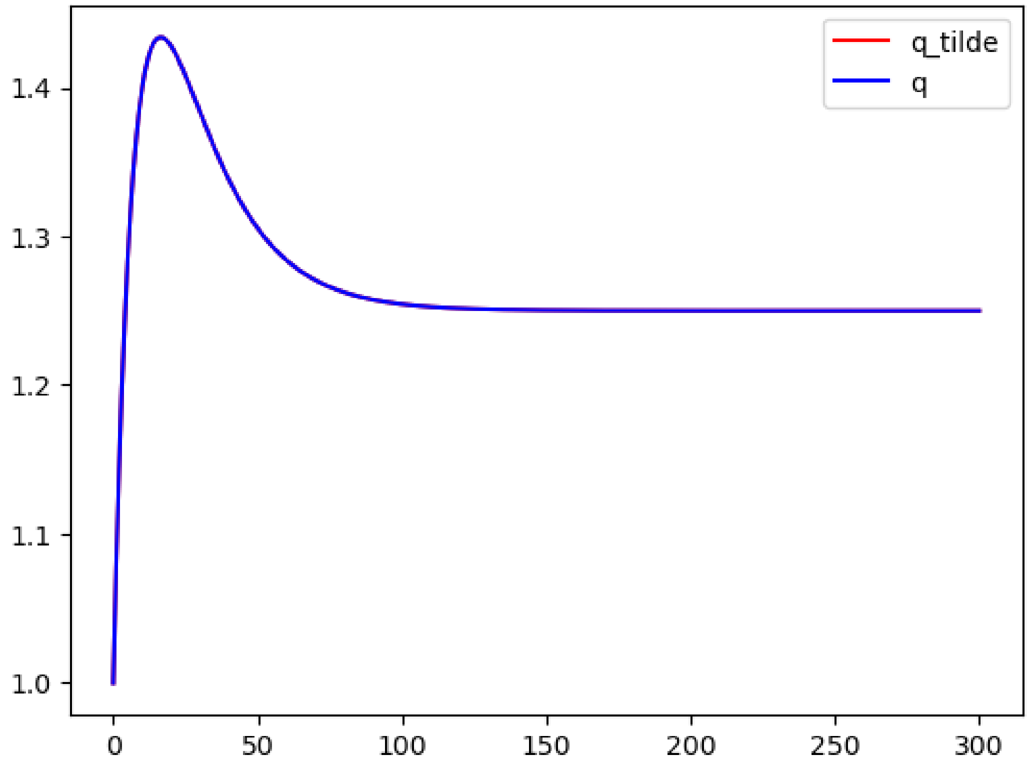
```
A_tilde = np.eye(5)
A_tilde[:4, :4] = A - B @ F
```

(continues on next page)

(continued from previous page)

```
R_tilde = np.array([[0,            0, 0,     0, -a0 / 2],
                    [0,            0, 0,     0,  a1 / 2],
                    [0,            0, 0,     0,       0],
                    [0,            0, 0,     0,       0],
                    [-a0 / 2, a1 / 2, 0,     0,      a1]])

Q_tilde = Q
B_tilde = np.array([[0, 0, 0, 0, 1]]).T

lq_tilde = LQ(Q_tilde, R_tilde, A_tilde, B_tilde, beta=β)
P_tilde, F_tilde, d_tilde = lq_tilde.stationary_values(method='doubling')

y0_tilde = np.vstack((y0, y0[2]))
yt_tilde = lq_tilde.compute_sequence(y0_tilde, ts_length=n)[0]
```

```
# Checks that the recursive formulation of the follower's problem gives
# the same solution as the original Stackelberg problem
fig, ax = plt.subplots()
ax.plot(yt_tilde[4], 'r', label="q_tilde")
ax.plot(yt_tilde[2], 'b', label="q")
ax.legend()
plt.show()
```



Note: Variables with `_tilde` are obtained from solving the follower's problem – those without are from the Stackelberg problem

```
# Maximum absolute difference in quantities over time between
# the first and second solution methods
np.max(np.abs(yt_tilde[4] - yt_tilde[2]))
```

```
4.440892098500626e-16
```

```
# x0 == x0_tilde
yt[:, 0][-1] - (yt_tilde[:, 1] - yt_tilde[:, 0])[-1] < tol0
```

```
True
```

2.11.1 Explanation of Alignment

If we inspect coefficients in the decision rule $-\tilde{F}$, we should be able to spot why the follower chooses to set $x_t = \tilde{x}_t$ when it sets $x_t = -\tilde{F}X_t$ in the recursive formulation of the follower problem.

Can you spot what features of $\tilde{F}$ imply this?

Hint: Remember the components of X_t

```
# Policy function in the follower's problem
F_tilde.round(4)
```

```
array([[ 0.    , -0.    , -0.1032, -1.    ,  0.1032]])
```

```
# Value function in the Stackelberg problem
P
```

```
array([[  963.54083615,  -194.60534465,  -511.62197962, -5258.22585724],
       [ -194.60534465,    37.3535753 ,    81.97712513,   784.76471234],
       [ -511.62197962,    81.97712513,   247.34333344,  2517.05126111],
       [-5258.22585724,   784.76471234,  2517.05126111, 25556.16504097]])
```

```
# Value function in the follower's problem
P_tilde
```

```
array([[-1.81991134e+01,  2.58003020e+00,  1.56048755e+01,
         1.51229815e+02, -5.00000000e+00],
       [ 2.58003020e+00, -9.69465925e-01, -5.26007958e+00,
        -5.09764310e+01,  1.00000000e+00],
       [ 1.56048755e+01, -5.26007958e+00, -3.22759027e+01,
        -3.12791908e+02, -1.23823802e+01],
       [ 1.51229815e+02, -5.09764310e+01, -3.12791908e+02,
        -3.03132584e+03, -1.20000000e+02],
       [-5.00000000e+00,  1.00000000e+00, -1.23823802e+01,
        -1.20000000e+02,  1.43823802e+01]])
```

```
# Manually check that P is an approximate fixed point
(P  - ((R + F.T @ Q @ F) + β * (A - B @ F).T @ P @ (A - B @ F)) < tol0).all()
```

```
True
```

```
# Compute `P_guess` using `F_tilde_star`
F_tilde_star = -np.array([[0, 0, 0, 1, 0]])
P_guess = np.zeros((5, 5))

for i in range(1000):
    P_guess = ((R_tilde + F_tilde_star.T @ Q @ F_tilde_star) +
               β * (A_tilde - B_tilde @ F_tilde_star).T @ P_guess
               @ (A_tilde - B_tilde @ F_tilde_star))
```

```
# Value function in the follower's problem
-(y0_tilde.T @ P_tilde @ y0_tilde)[0, 0]
```

```
112.65590740578115
```

```
# Value function with `P_guess`
-(y0_tilde.T @ P_guess @ y0_tilde)[0, 0]
```

```
112.65590740578136
```

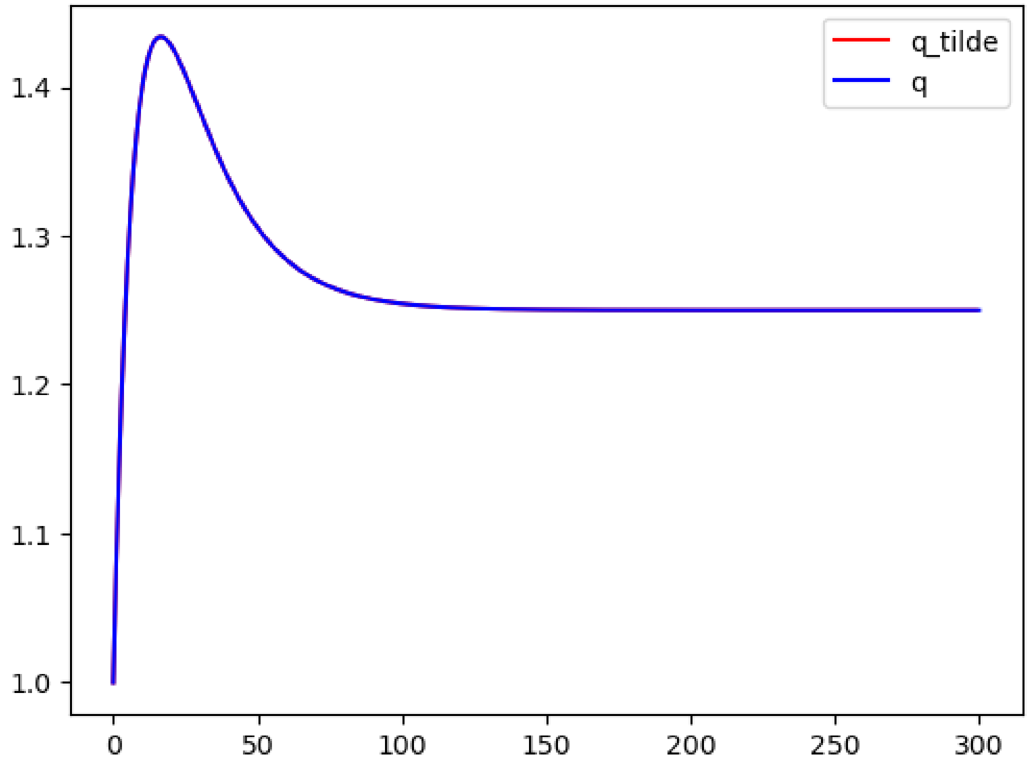
```
# Compute policy using policy iteration algorithm
F_iter = (β * la.inv(Q + β * B_tilde.T @ P_guess @ B_tilde)
          @ B_tilde.T @ P_guess @ A_tilde)

for i in range(100):
    # Compute P_iter
    P_iter = np.zeros((5, 5))
    for j in range(1000):
        P_iter = ((R_tilde + F_iter.T @ Q @ F_iter) + β
                  * (A_tilde - B_tilde @ F_iter).T @ P_iter
                  @ (A_tilde - B_tilde @ F_iter))

    # Update F_iter
    F_iter = (β * la.inv(Q + β * B_tilde.T @ P_iter @ B_tilde)
              @ B_tilde.T @ P_iter @ A_tilde)

dist_vec = (P_iter - ((R_tilde + F_iter.T @ Q @ F_iter)
            + β * (A_tilde - B_tilde @ F_iter).T @ P_iter
            @ (A_tilde - B_tilde @ F_iter)))

if np.max(np.abs(dist_vec)) < 1e-8:
    dist_vec2 = (F_iter - (β * la.inv(Q + β * B_tilde.T @ P_iter @ B_tilde)
                 @ B_tilde.T @ P_iter @ A_tilde))

    if np.max(np.abs(dist_vec2)) < 1e-8:
        F_iter
    else:
        print("The policy didn't converge: try increasing the number of \
```

(continues on next page)

(continued from previous page)

```
            outer loop iterations")
else:
    print("`P_iter` didn't converge: try increasing the number of inner \
        loop iterations")
```

```
# Simulate the system using `F_tilde_star` and check that it gives the
# same result as the original solution

yt_tilde_star = np.zeros((n, 5))
yt_tilde_star[0, :] = y0_tilde.flatten()

for t in range(n-1):
    yt_tilde_star[t+1, :] = (A_tilde - B_tilde @ F_tilde_star) \
        @ yt_tilde_star[t, :]

fig, ax = plt.subplots()
ax.plot(yt_tilde_star[:, 4], 'r', label="q_tilde")
ax.plot(yt_tilde[2], 'b', label="q")
ax.legend()
plt.show()
```



```
# Maximum absolute difference
np.max(np.abs(yt_tilde_star[:, 4] - yt_tilde[2, :-1]))
```

```
0.0
```

2.12 Markov Perfect Equilibrium

The **state** vector is

$$z_t = \begin{bmatrix} 1 \\ q_{2t} \\ q_{1t} \end{bmatrix}$$

and the state transition dynamics are

$$z_{t+1} = Az_t + B_1 v_{1t} + B_2 v_{2t}$$

where A is a 3×3 identity matrix and

$$B_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad B_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

The Markov perfect decision rules are

$$v_{1t} = -F_1 z_t, \quad v_{2t} = -F_2 z_t$$

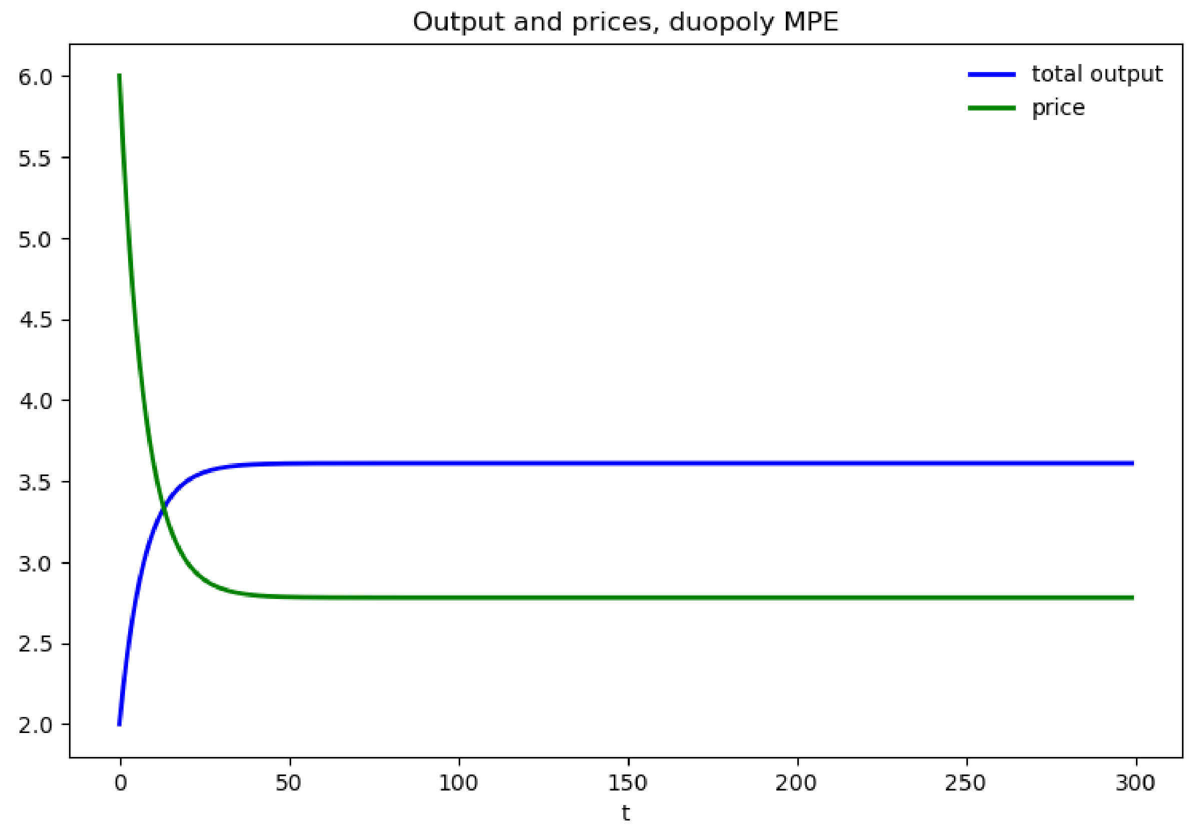
and in the Markov perfect equilibrium, the state evolves according to

$$z_{t+1} = (A - B_1 F_1 - B_2 F_2) z_t$$

```
# In LQ form
A = np.eye(3)
B1 = np.array([[0], [0], [1]])
B2 = np.array([[0], [1], [0]])

R1 = np.array([[0,            0, -a0 / 2],
               [0,            0,  a1 / 2],
               [-a0 / 2, a1 / 2,      a1]])

R2 = np.array([[0,       -a0 / 2,      0],
               [-a0 / 2,      a1, a1 / 2],
               [0,         a1 / 2,      0]])

Q1 = Q2 = γ
S1 = S2 = W1 = W2 = M1 = M2 = 0.0

# Solve using QE's nnash function
F1, F2, P1, P2 = qe.nnash(A, B1, B2, R1, R2, Q1,
                          Q2, S1, S2, W1, W2, M1,
                          M2, beta=β, tol=tol1)

# Simulate forward
AF = A - B1 @ F1 - B2 @ F2
z = np.empty((3, n))
z[:, 0] = 1, 1, 1
for t in range(n-1):
    z[:, t+1] = AF @ z[:, t]

# Display policies
print("Computed policies for firm 1 and firm 2:\n")
print(f"F1 = {F1}")
print(f"F2 = {F2}")
```

```
Computed policies for firm 1 and firm 2:

F1 = [[-0.22701363  0.03129874  0.09447113]]
F2 = [[-0.22701363  0.09447113  0.03129874]]
```

```
q1 = z[1, :]
q2 = z[2, :]
q = q1 + q2       # Total output, MPE
p = a0 - a1 * q   # Price, MPE

fig, ax = plt.subplots(figsize=(9, 5.8))
ax.plot(range(n), q, 'b-', lw=2, label='total output')
ax.plot(range(n), p, 'g-', lw=2, label='price')
ax.set_title('Output and prices, duopoly MPE')
ax.legend(frameon=False)
ax.set_xlabel('t')
plt.show()
```



```
# Computes the maximum difference between the two quantities of the two firms
np.max(np.abs(q1 - q2))
```

```
8.881784197001252e-16
```

```
# Compute values
u1 = (- F1 @ z).flatten()
u2 = (- F2 @ z).flatten()

π_1 = p * q1 - γ * (u1) ** 2
π_2 = p * q2 - γ * (u2) ** 2

v1_forward = np.sum(βs * π_1)
v2_forward = np.sum(βs * π_2)

v1_direct = (- z[:, 0].T @ P1 @ z[:, 0])
v2_direct = (- z[:, 0].T @ P2 @ z[:, 0])

# Display values
print("Computed values for firm 1 and firm 2:\n")
print(f"v1(forward sim) = {v1_forward:.4f}; v1 (direct) = {v1_direct:.4f}")
print(f"v2 (forward sim) = {v2_forward:.4f}; v2 (direct) = {v2_direct:.4f}")
```

```
Computed values for firm 1 and firm 2:

v1(forward sim) = 133.3303; v1 (direct) = 133.3296
v2 (forward sim) = 133.3303; v2 (direct) = 133.3296
```

```
# Sanity check
Λ1 = A - B2 @ F2
lq1 = qe.LQ(Q1, R1, Λ1, B1, beta=β)
P1_ih, F1_ih, d = lq1.stationary_values()

v2_direct_alt = - z[:, 0].T @ lq1.P @ z[:, 0] + lq1.d

(np.abs(v2_direct - v2_direct_alt) < tol2).all()
```

```
True
```

2.13 Comparing Markov Perfect Equilibrium and Stackelberg Outcome

It is enlightening to compare equlibrium values for firms 1 and 2 under two alternative settings:

- A Markov perfect equilibrium like that described in this lecture
- A Stackelberg equilbrium

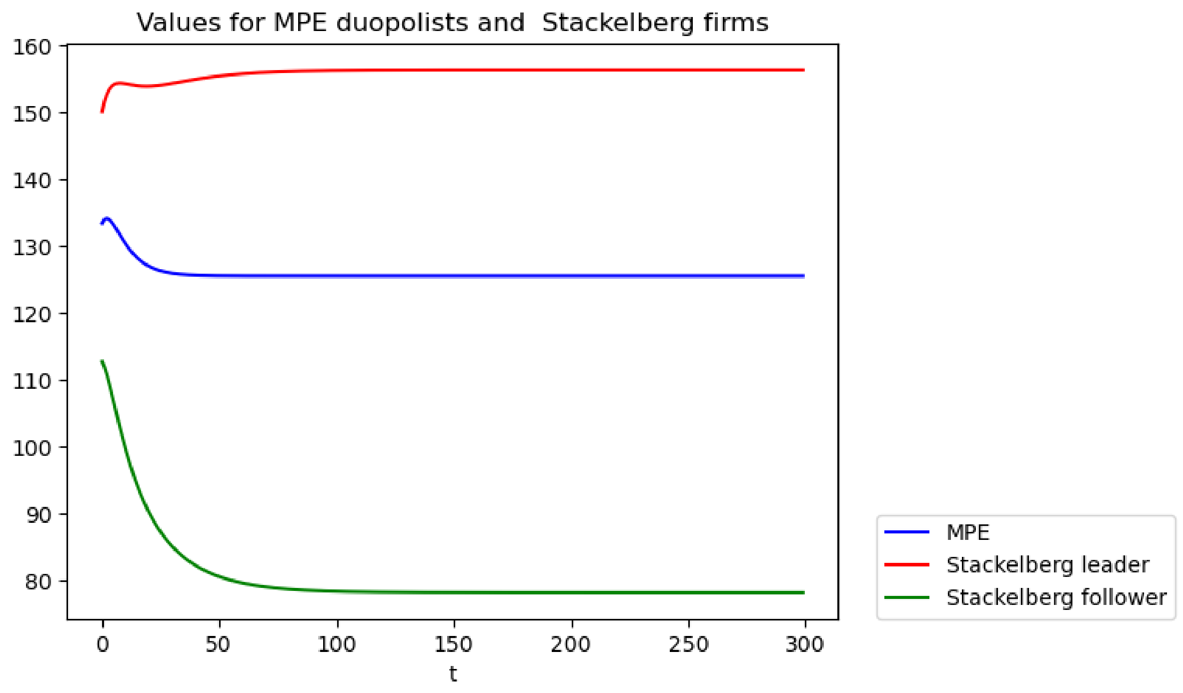
The following code performs the required computations, then plots the continuation values.

```
vt_MPE = np.zeros(n)
vt_follower = np.zeros(n)

for t in range(n):
    vt_MPE[t] = -z[:, t].T @ P1 @ z[:, t]
    vt_follower[t] = -yt_tilde[:, t].T @ P_tilde @ yt_tilde[:, t]
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots()
ax.plot(vt_MPE, 'b', label='MPE')
ax.plot(vt_leader, 'r', label='Stackelberg leader')
ax.plot(vt_follower, 'g', label='Stackelberg follower')
ax.set_title(r'Values for MPE duopolists and  Stackelberg firms')
ax.set_xlabel('t')
ax.legend(loc=(1.05, 0))
plt.show()
```



```
# Display values
print("Computed values:\n")
print(f"vt_leader(y0)  = {vt_leader[0]:.4f}")
print(f"vt_follower(y0)  = {vt_follower[0]:.4f}")
print(f"vt_MPE(y0)  = {vt_MPE[0]:.4f}")
```

```
Computed values:

vt_leader(y0)  = 150.0324
vt_follower(y0)  = 112.6559
vt_MPE(y0)  = 133.3296
```

```
# Compute the difference in total value between the Stackelberg and the MPE
vt_leader[0] + vt_follower[0] - 2 * vt_MPE[0]
```

```
-3.9709425620890784
```

CHAPTER

THREE

RAMSEY PLANS, TIME INCONSISTENCY, SUSTAINABLE PLANS

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

3.1 Overview

This lecture describes a linear-quadratic version of a model that Guillermo Calvo [Cal78] used to illustrate the **time inconsistency** of optimal government plans.

Like Chang [Cha98], we use the model as a laboratory in which to explore the consequences of different timing protocols for government decision making.

The model focuses attention on intertemporal tradeoffs between

- welfare benefits that anticipated deflation generates by increasing a representative agent's liquidity as measured by his or her real money balances, and
- costs associated with distorting taxes that must be used to withdraw money from the economy in order to generate anticipated deflation

The model features

- rational expectations
- costly government actions at all dates $t \geq 1$ that increase household utilities at dates before t
- two Bellman equations, one that expresses the private sector's expectation of future inflation as a function of current and future government actions, another that describes the value function of a Ramsey planner

A theme of this lecture is that timing protocols affect outcomes.

We'll use ideas from papers by Cagan [Cag56], Calvo [Cal78], Stokey [Sto89], [Sto91], Chari and Kehoe [CK90], Chang [Cha98], and Abreu [Abr88] as well as from chapter 19 of [LS18].

In addition, we'll use ideas from linear-quadratic dynamic programming described in Linear Quadratic Control as applied to Ramsey problems in *Stackelberg problems*.

We specify the model in a way that allows us to use linear-quadratic dynamic programming to compute an optimal government plan under a timing protocol in which a government chooses an infinite sequence of money supply growth rates once and for all at time 0.

We'll start with some imports:

```
import numpy as np
from quantecon import LQ
import matplotlib.pyplot as plt
%matplotlib inline
```

3.2 The Model

There is no uncertainty.

Let:

- p_t be the log of the price level
- m_t be the log of nominal money balances
- $\theta_t = p_{t+1} - p_t$ be the net rate of inflation between t and $t+1$
- $\mu_t = m_{t+1} - m_t$ be the net rate of growth of nominal balances

The demand for real balances is governed by a perfect foresight version of the Cagan [Cag56] demand function:

$$m_t - p_t = -\alpha(p_{t+1} - p_t)\ ,\ \alpha > 0 \tag{3.1}$$

for $t \geq 0$.

Equation (3.1) asserts that the demand for real balances is inversely related to the public's expected rate of inflation, which here equals the actual rate of inflation.

(When there is no uncertainty, an assumption of **rational expectations** implies **perfect foresight**).

(See [Sar77] for a rational expectations version of the model when there is uncertainty.)

Subtracting the demand function at time t from the demand function at $t+1$ gives:

$$\mu_t - \theta_t = -\alpha\theta_{t+1} + \alpha\theta_t$$

or

$$\theta_t = \frac{\alpha}{1+\alpha}\theta_{t+1} + \frac{1}{1+\alpha}\mu_t \tag{3.2}$$

Because $\alpha > 0$, $0 < \frac{\alpha}{1+\alpha} < 1$.

Definition: For a scalar b_t, let L^2 be the space of sequences $\{b_t\}_{t=0}^\infty$ satisfying

$$\sum_{t=0}^\infty b_t^2 < +\infty$$

We say that a sequence that belongs to L^2 is **square summable**.

When we assume that the sequence $\vec{\mu} = \{\mu_t\}_{t=0}^\infty$ is square summable and we require that the sequence $\vec{\theta} = \{\theta_t\}_{t=0}^\infty$ is square summable, the linear difference equation (3.2) can be solved forward to get:

$$\theta_t = \frac{1}{1+\alpha}\sum_{j=0}^\infty \left(\frac{\alpha}{1+\alpha}\right)^j \mu_{t+j} \tag{3.3}$$

Insight: In the spirit of Chang [Cha98], note that equations (3.1) and (3.3) show that θ_t intermediates how choices of $\mu_{t+j},\ j = 0, 1, \ldots$ impinge on time t real balances $m_t - p_t = -\alpha\theta_t$.

We shall use this insight to help us simplify and analyze government policy problems.

That future rates of money creation influence earlier rates of inflation creates optimal government policy problems in which timing protocols matter.

We can rewrite the model as:

$$\begin{bmatrix} 1 \\ \theta_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1+\alpha}{\alpha} \end{bmatrix} \begin{bmatrix} 1 \\ \theta_t \end{bmatrix} + \begin{bmatrix} 0 \\ -\frac{1}{\alpha} \end{bmatrix} \mu_t$$

or

$$x_{t+1} = Ax_t + B\mu_t \tag{3.4}$$

We write the model in the state-space form (3.4) even though θ_0 is to be determined by our model and so is not an initial condition, as it ordinarily would be in the state-space model described in our lecture on Linear Quadratic Control.

We write the model in the form (3.4) because we want to apply an approach described in our lecture on *Stackelberg problems*.

We assume that a government believes that a representative household's utility of real balances at time t is:

$$U(m_t - p_t) = a_0 + a_1(m_t - p_t) - \frac{a_2}{2}(m_t - p_t)^2, \quad a_0 > 0, a_1 > 0, a_2 > 0 \tag{3.5}$$

The "bliss level" of real balances is then $\frac{a_1}{a_2}$.

The money demand function (3.1) and the utility function (3.5) imply that utility maximizing or bliss level of real balances is attained when:

$$\theta_t = \theta^* = -\frac{a_1}{a_2 \alpha}$$

Below, we introduce the discount factor $\beta \in (0, 1)$ that a government uses to discount its future utilities.

(If we set parameters so that $\theta^* = \log(\beta)$, then we can regard a recommendation to set $\theta_t = \theta^*$ as a "poor man's Friedman rule" that attains Milton Friedman's **optimal quantity of money**.)

Via equation (3.3), a government plan $\vec{\mu} = \{\mu_t\}_{t=0}^\infty$ leads to a sequence of inflation outcomes $\vec{\theta} = \{\theta_t\}_{t=0}^\infty$.

We assume that social costs $\frac{c}{2}\mu_t^2$ are incurred at t when the government changes the stock of nominal money balances at rate μ_t.

Therefore, the one-period welfare function of a benevolent government is:

$$-s(\theta_t, \mu_t) \equiv -r(x_t, \mu_t) = \begin{bmatrix} 1 \\ \theta_t \end{bmatrix}' \begin{bmatrix} a_0 & -\frac{a_1 \alpha}{2} \\ -\frac{a_1 \alpha}{2} & -\frac{a_2 \alpha^2}{2} \end{bmatrix} \begin{bmatrix} 1 \\ \theta_t \end{bmatrix} - \frac{c}{2}\mu_t^2 = -x_t' R x_t - Q \mu_t^2 \tag{3.6}$$

A benevolent government's time 0 value is

$$v_0 = -\sum_{t=0}^\infty \beta^t r(x_t, \mu_t) = -\sum_{t=0}^\infty \beta^t s(\theta_t, \mu_t) \tag{3.7}$$

We can represent the dependence of v_0 on $(\vec{\theta}, \vec{\mu})$ recursively via the difference equation

$$v_t = -s(\theta_t, \mu_t) + \beta v_{t+1} \tag{3.8}$$

where the government's time t continuation value v_t satisfies

$$v_t = -\sum_{j=0}^\infty \beta^j s(\theta_{t+j}, \mu_{t+j}).$$

3.3 Structure

The following structure is induced by private agents' behavior as summarized by the demand function for money (3.1) that leads to equation (3.3), which tells how future settings of μ affect the current value of θ.

Equation (3.3) maps a **policy** sequence of money growth rates $\vec{\mu} = \{\mu_t\}_{t=0}^\infty \in L^2$ into an inflation sequence $\vec{\theta} = \{\theta_t\}_{t=0}^\infty \in L^2$.

These, in turn, induce a discounted value to a government sequence $\vec{v} = \{v_t\}_{t=0}^\infty \in L^2$ that satisfies the recursion

$$v_t = -s(\theta_t, \mu_t) + \beta v_{t+1}$$

where we have called $s(\theta_t, \mu_t) = r(x_t, \mu_t)$, as above.

Thus, a triple of sequences $(\vec{\mu}, \vec{\theta}, \vec{v})$ depends on a sequence $\vec{\mu} \in L^2$.

At this point $\vec{\mu} \in L^2$ is an arbitrary exogenous policy.

A theory of government decisions will make $\vec{\mu}$ endogenous, i.e., a theoretical **output** instead of an **input**.

3.4 Intertemporal Structure

Criterion function (3.7) and the constraint system (3.4) exhibit the following structure:

- Setting $\mu_t \neq 0$ imposes costs $\frac{c}{2}\mu_t^2$ at time t and at no other times; but
- The money growth rate μ_t affects the government's one-period utilities at all dates $s = 0, 1, \ldots, t$.

This structure sets the stage for the emergence of a time-inconsistent optimal government plan under a Ramsey timing protocol, also called a Stackelberg timing protocol.

We'll eventually study outcomes under a Ramsey timing protocol.

But we'll also study the consequences of other timing protocols.

3.5 Four Models of Government Policy

We consider four models of policymakers that differ in

- what a policymaker is allowed to choose, either a sequence $\vec{\mu}$ or just μ_t in a single period t.
- when a policymaker chooses, either once and for all at time 0, or at some time or times $t \geq 0$.
- what a policymaker assumes about how its choice of μ_t affects private agents' expectations about earlier and later inflation rates.

In two of our models, a single policymaker chooses a sequence $\{\mu_t\}_{t=0}^{\infty}$ once and for all, taking into account how μ_t affects household one-period utilities at dates $s = 0, 1, \ldots, t-1$

- these two models thus employ a **Ramsey** or **Stackelberg** timing protocol.

In two other models, there is a sequence of policymakers, each of whom sets μ_t at one t only.

- Each such policymaker ignores effects that its choice of μ_t has on household one-period utilities at dates $s = 0, 1, \ldots, t-1$.

The four models differ with respect to timing protocols, constraints on government choices, and government policymakers' beliefs about how their decisions affect private agents' beliefs about future government decisions.

The models are distinguished by having either

- A single Ramsey planner chooses a sequence $\{\mu_t\}_{t=0}^{\infty}$ once and for all at time 0; or
- A single Ramsey planner chooses a sequence $\{\mu_t\}_{t=0}^{\infty}$ once and for all at time 0 subject to the constraint that $\mu_t = \mu$ for all $t \geq 0$; or
- A sequence of separate policymakers chooses μ_t for $t = 0, 1, 2, \ldots$
  - a time t policymaker chooses μ_t only and forecasts that future government decisions are unaffected by its choice; or
- A sequence of separate policymakers chooses μ_t for $t = 0, 1, 2, \ldots$
  - a time t policymaker chooses only μ_t but believes that its choice of μ_t shapes private agents' beliefs about future rates of money creation and inflation, and through them, future government actions.

The relationship between outcomes in the first (Ramsey) timing protocol and the fourth timing protocol and belief structure is the subject of a literature on **sustainable** or **credible** public policies (Chari and Kehoe [CK90] [Sto89], and Stokey [Sto91]).

We'll discuss that topic later in this lecture.

3.6 A Ramsey Planner

First, we consider a Ramsey planner that chooses $\{\mu_t, \theta_t\}_{t=0}^{\infty}$ to maximize (3.7) subject to the law of motion (3.4).

We can split this problem into two stages, as in *Stackelberg problems* and [LS18] Chapter 19.

In the first stage, we take the initial inflation rate θ_0 as given, and then solve the resulting LQ dynamic programming problem.

In the second stage, we maximize over the initial inflation rate θ_0.

Define a feasible set of $(\vec{x}_1, \vec{\mu}_0)$ sequences, both of which must belong to L^2:

$$\Omega(x_0) = \{(\vec{x}_1, \vec{\mu}_0) : x_{t+1} = Ax_t + B\mu_t \ , \ \forall t \geq 0\}$$

3.6.1 Subproblem 1

The value function

$$J(x_0) = \max_{(\vec{x}_1, \vec{\mu}_0) \in \Omega(x_0)} -\sum_{t=0}^{\infty} \beta^t r(x_t, \mu_t)$$

satisfies the Bellman equation

$$J(x) = \max_{\mu, x'} \{-r(x, \mu) + \beta J(x')\}$$

subject to:

$$x' = Ax + B\mu$$

As in *Stackelberg problems*, we map this problem into a linear-quadratic control problem and deduce an optimal value function $J(x)$.

Guessing that $J(x) = -x'Px$ and substituting into the Bellman equation gives rise to the algebraic matrix Riccati equation:

$$P = R + \beta A'PA - \beta^2 A'PB(Q + \beta B'PB)^{-1}B'PA$$

and an optimal decision rule

$$\mu_t = -Fx_t$$

where

$$F = \beta(Q + \beta B'PB)^{-1}B'PA$$

The QuantEcon LQ class solves for F and P given inputs Q, R, A, B, and β.

3.6.2 Subproblem 2

The value of the Ramsey problem is

$$V = \max_{x_0} J(x_0)$$

where V is the maximum value of v_0 defined in equation (3.7).

The value function

$$J(x_0) = -\begin{bmatrix} 1 & \theta_0 \end{bmatrix} \begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{bmatrix} \begin{bmatrix} 1 \\ \theta_0 \end{bmatrix} = -P_{11} - 2P_{21}\theta_0 - P_{22}\theta_0^2$$

Maximizing this with respect to θ_0 yields the FOC:

$$-2P_{21} - 2P_{22}\theta_0 = 0$$

which implies

$$\theta_0^* = -\frac{P_{21}}{P_{22}}$$

3.6.3 Representation of Ramsey Plan

The preceding calculations indicate that we can represent a Ramsey plan $\vec{\mu}$ recursively with the following system created in the spirit of Chang [Cha98]:

$$\begin{aligned} \theta_0 &= \theta_0^* \\ \mu_t &= b_0 + b_1\theta_t \\ \theta_{t+1} &= d_0 + d_1\theta_t \end{aligned} \tag{3.9}$$

To interpret this system, think of the sequence $\{\theta_t\}_{t=0}^{\infty}$ as a sequence of synthetic **promised inflation rates**.

At this point, we can think of these promised inflation rates just as computational devices for generating a sequence $\vec{\mu}$ of money growth rates that are to be substituted into equation (3.3) to form **actual** rates of inflation.

But it can be verified that if we substitute a plan $\vec{\mu} = \{\mu_t\}_{t=0}^{\infty}$ that satisfies these equations into equation (3.3), we obtain the same sequence $\vec{\theta}$ generated by the system (3.9).

(Here an application of the Big K, little k trick could once again be enlightening.)

Thus, our construction of a Ramsey plan guarantees that **promised inflation** equals **actual inflation**.

3.6.4 Multiple roles of θ_t

The inflation rate θ_t plays three roles simultaneously:

- In equation (3.3), θ_t is the actual rate of inflation between t and $t+1$.
- In equation (3.2) and (3.3), θ_t is also the public's expected rate of inflation between t and $t+1$.
- In system (3.9), θ_t is a promised rate of inflation chosen by the Ramsey planner at time 0.

That the same variable θ_t takes on these multiple roles brings insights about commitment and forward guidance, following versus leading the market, and dynamic or time inconsistency.

3.6.5 Time Inconsistency

As discussed in *Stackelberg problems* and *Optimal taxation with state-contingent debt*, a continuation Ramsey plan is not a Ramsey plan.

This is a concise way of characterizing the time inconsistency of a Ramsey plan.

The time inconsistency of a Ramsey plan has motivated other models of government decision making that alter either

- the timing protocol and/or
- assumptions about how government decision makers think their decisions affect private agents' beliefs about future government decisions

3.7 A Constrained-to-a-Constant-Growth-Rate Ramsey Government

We now consider a peculiar model of optimal government behavior.

We created this model in order to highlight an aspect of an optimal government policy associated with its time inconsistency, namely, the feature that optimal settings of the policy instrument vary over time.

Instead of allowing the Ramsey government to choose different settings of its instrument at different moments, we now assume that at time 0, a Ramsey government at time 0 once and for all chooses a **constant** sequence $\mu_t = \check{\mu}$ for all $t \geq 0$ to maximize

$$U(-\alpha \check{\mu}) - \frac{c}{2} \check{\mu}^2$$

Here we have imposed the perfect foresight outcome implied by equation (3.2) that $\theta_t = \check{\mu}$ when the government chooses a constant μ for all $t \geq 0$.

With the quadratic form (3.5) for the utility function U, the maximizing $\bar{\mu}$ is

$$\check{\mu} = -\frac{\alpha a_1}{\alpha^2 a_2 + c}$$

Summary: We have introduced the constrained-to-a-constant μ government in order to highlight time-variation of μ_t as a telltale sign of time inconsistency of a Ramsey plan.

3.8 Markov Perfect Governments

We now alter the timing protocol by considering a sequence of government policymakers, the time t representative of which chooses μ_t and expects all future governments to set $\mu_{t+j} = \bar{\mu}$.

This assumption mirrors an assumption made in a different setting Markov Perfect Equilibrium.

A government policymaker at t believes that $\bar{\mu}$ is unaffected by its choice of μ_t.

The time t rate of inflation is then:

$$\theta_t = \frac{\alpha}{1+\alpha} \bar{\mu} + \frac{1}{1+\alpha} \mu_t$$

The time t government policymaker then chooses μ_t to maximize:

$$W = U(-\alpha \theta_t) - \frac{c}{2} \mu_t^2 + \beta V(\bar{\mu})$$

where $V(\bar{\mu})$ is the time 0 value v_0 of recursion (3.8) under a money supply growth rate that is forever constant at $\bar{\mu}$.

Substituting for U and θ_t gives:

$$W = a_0 + a_1 \left(-\frac{\alpha^2}{1+\alpha} \bar{\mu} - \frac{\alpha}{1+\alpha} \mu_t\right) - \frac{a_2}{2} \left(\left(-\frac{\alpha^2}{1+\alpha} \bar{\mu} - \frac{\alpha}{1+\alpha} \mu_t\right)^2 - \frac{c}{2} \mu_t^2 + \beta V(\bar{\mu})\right)$$

The first-order necessary condition for μ_t is then:

$$-\frac{\alpha}{1+\alpha} a_1 - a_2 \left(-\frac{\alpha^2}{1+\alpha} \bar{\mu} - \frac{\alpha}{1+\alpha} \mu_t\right)\left(-\frac{\alpha}{1+\alpha}\right) - c \mu_t = 0$$

Rearranging we get:

$$\mu_t = \frac{-a_1}{\frac{1+\alpha}{\alpha} c + \frac{\alpha}{1+\alpha} a_2} - \frac{\alpha^2 a_2}{\left[\frac{1+\alpha}{\alpha} c + \frac{\alpha}{1+\alpha} a_2\right] (1+\alpha)} \bar{\mu}$$

A **Markov Perfect Equilibrium** (MPE) outcome sets $\mu_t = \bar{\mu}$:

$$\mu_t = \bar{\mu} = \frac{-a_1}{\frac{1+\alpha}{\alpha}c + \frac{\alpha}{1+\alpha}a_2 + \frac{\alpha^2}{1+\alpha}a_2}$$

In light of results presented in the previous section, this can be simplified to:

$$\bar{\mu} = -\frac{\alpha a_1}{\alpha^2 a_2 + (1+\alpha)c}$$

3.9 Outcomes under Three Timing Protocols

Below we compute sequences $\{\theta_t, \mu_t\}$ under a Ramsey plan and compare these with the constant levels of θ and μ in a) a Markov Perfect Equilibrium, and b) a Ramsey plan in which the planner is restricted to choose $\mu_t = \check{\mu}$ for all $t \geq 0$.

We denote the Ramsey sequence as θ^R, μ^R and the MPE values as θ^{MPE}, μ^{MPE}.

The bliss level of inflation is denoted by θ^*.

First, we will create a class ChangLQ that solves the models and stores their values

```
class ChangLQ:
    """
    Class to solve LQ Chang model
    """
    def __init__(self, α, α0, α1, α2, c, T=1000, θ_n=200):

        # Record parameters
        self.α, self.α0, self.α1 = α, α0, α1
        self.α2, self.c, self.T, self.θ_n = α2, c, T, θ_n

        # Create β using "Poor Man's Friedman Rule"
        self.β = np.exp(-α1 / (α * α2))

        # Solve the Ramsey Problem #

        # LQ Matrices
        R = -np.array([[α0,            -α1 * α / 2],
                       [-α1 * α/2, -α2 * α**2 / 2]])
        Q = -np.array([[-c / 2]])
        A = np.array([[1, 0], [0, (1 + α) / α]])
        B = np.array([[0], [-1 / α]])

        # Solve LQ Problem (Subproblem 1)
        lq = LQ(Q, R, A, B, beta=self.β)
        self.P, self.F, self.d = lq.stationary_values()

        # Solve Subproblem 2
        self.θ_R = -self.P[0, 1] / self.P[1, 1]

        # Find bliss level of θ
        self.θ_B = np.log(self.β)

        # Solve the Markov Perfect Equilibrium
        self.μ_MPE = -α1 / ((1 + α) / α * c + α / (1 + α)
                      * α2 + α**2 / (1 + α) * α2)
        self.θ_MPE = self.μ_MPE
```

(continues on next page)

(continued from previous page)

```
        self.μ_check = -α * α1 / (α2 * α**2 + c)

        # Calculate value under MPE and Check economy
        self.J_MPE  = (α0 + α1 * (-α * self.μ_MPE) - α2 / 2
                      * (-α * self.μ_MPE)**2 - c/2 * self.μ_MPE**2) / (1 - self.β)
        self.J_check = (α0 + α1 * (-α * self.μ_check) - α2/2
                        * (-α * self.μ_check)**2 - c / 2 * self.μ_check**2) \
                        / (1 - self.β)

        # Simulate Ramsey plan for large number of periods
        θ_series = np.vstack((np.ones((1, T)), np.zeros((1, T))))
        μ_series = np.zeros(T)
        J_series = np.zeros(T)
        θ_series[1, 0] = self.θ_R
        μ_series[0] = -self.F.dot(θ_series[:, 0])
        J_series[0] = -θ_series[:, 0] @ self.P @ θ_series[:, 0].T
        for i in range(1, T):
            θ_series[:, i] = (A - B @ self.F) @ θ_series[:, i-1]
            μ_series[i] = -self.F @ θ_series[:, i]
            J_series[i] = -θ_series[:, i] @ self.P @ θ_series[:, i].T

        self.J_series = J_series
        self.μ_series = μ_series
        self.θ_series = θ_series

        # Find the range of θ in Ramsey plan
        θ_LB = min(θ_series[1, :])
        θ_LB = min(θ_LB, self.θ_B)
        θ_UB = max(θ_series[1, :])
        θ_UB = max(θ_UB, self.θ_MPE)
        θ_range = θ_UB - θ_LB
        self.θ_LB = θ_LB - 0.05 * θ_range
        self.θ_UB = θ_UB + 0.05 * θ_range
        self.θ_range = θ_range

        # Find value function and policy functions over range of θ
        θ_space = np.linspace(self.θ_LB, self.θ_UB, 200)
        J_space = np.zeros(200)
        check_space = np.zeros(200)
        μ_space = np.zeros(200)
        θ_prime = np.zeros(200)
        for i in range(200):
            J_space[i] = - np.array((1, θ_space[i])) \
                        @ self.P @ np.array((1, θ_space[i])).T
            μ_space[i] = - self.F @ np.array((1, θ_space[i]))
            x_prime = (A - B @ self.F) @ np.array((1, θ_space[i]))
            θ_prime[i] = x_prime[1]
            check_space[i] = (α0 + α1 * (-α * θ_space[i]) -
            α2/2 * (-α * θ_space[i])**2 - c/2 * θ_space[i]**2) / (1 - self.β)

        J_LB = min(J_space)
        J_UB = max(J_space)
        J_range = J_UB - J_LB
        self.J_LB = J_LB - 0.05 * J_range
        self.J_UB = J_UB + 0.05 * J_range
        self.J_range = J_range
```

(continues on next page)

(continued from previous page)

```
        self.J_space = J_space
        self.θ_space = θ_space
        self.μ_space = μ_space
        self.θ_prime = θ_prime
        self.check_space = check_space
```

We will create an instance of ChangLQ with the following parameters

```
clq = ChangLQ(α=1, α0=1, α1=0.5, α2=3, c=2)
clq.β
```

```
/tmp/ipykernel_2356/2001568470.py:51: DeprecationWarning: Conversion of an array␣
↪with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you␣
↪extract a single element from your array before performing this operation.␣
↪(Deprecated NumPy 1.25.)
  μ_series[0] = -self.F.dot(θ_series[:, 0])
/tmp/ipykernel_2356/2001568470.py:55: DeprecationWarning: Conversion of an array␣
↪with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you␣
↪extract a single element from your array before performing this operation.␣
↪(Deprecated NumPy 1.25.)
  μ_series[i] = -self.F @ θ_series[:, i]
/tmp/ipykernel_2356/2001568470.py:81: DeprecationWarning: Conversion of an array␣
↪with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you␣
↪extract a single element from your array before performing this operation.␣
↪(Deprecated NumPy 1.25.)
  μ_space[i] = - self.F @ np.array((1, θ_space[i]))
```

```
0.8464817248906141
```

The following code generates a figure that plots the value function from the Ramsey Planner's problem, which is maximized at θ_0^R.

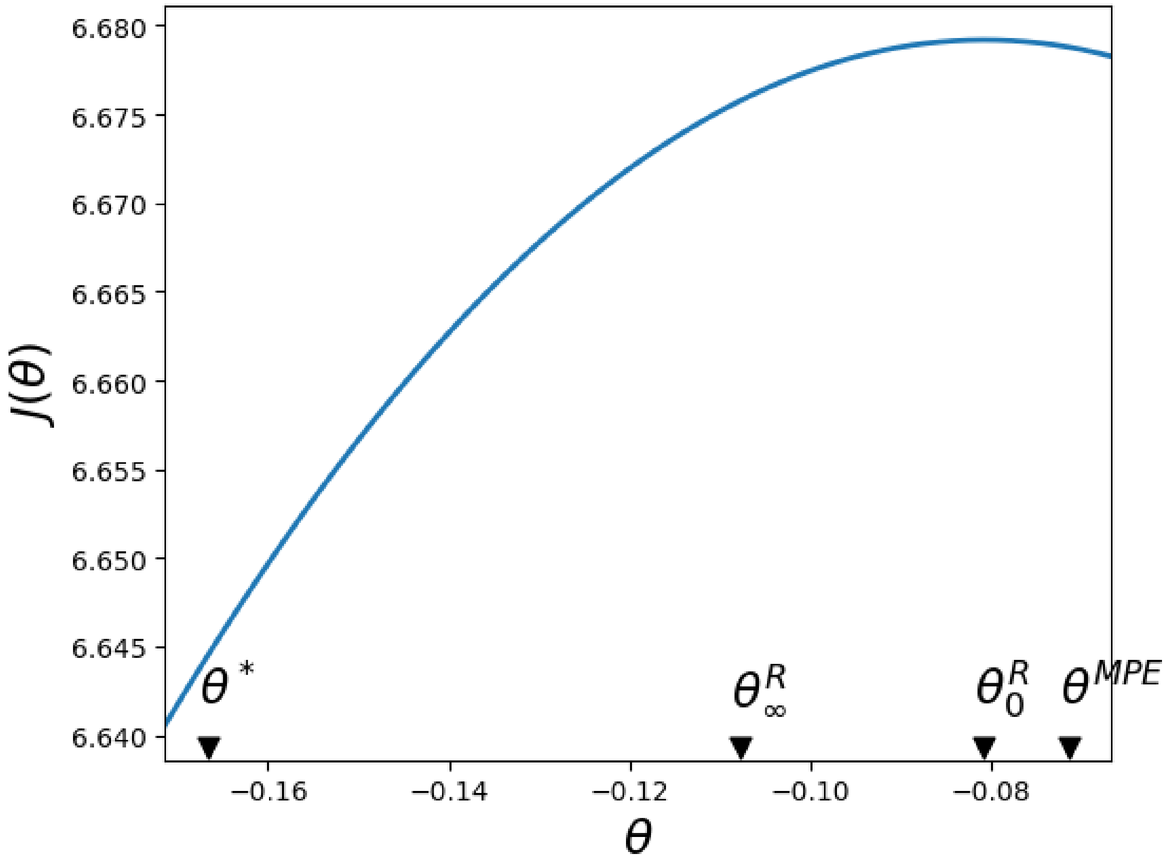
The figure also shows the limiting value θ_∞^R to which the inflation rate θ_t converges under the Ramsey plan and compares it to the MPE value and the bliss value.

```
def plot_value_function(clq):
    """
    Method to plot the value function over the relevant range of θ

    Here clq is an instance of ChangLQ

    """
    fig, ax = plt.subplots()

    ax.set_xlim([clq.θ_LB, clq.θ_UB])
    ax.set_ylim([clq.J_LB, clq.J_UB])

    # Plot value function
    ax.plot(clq.θ_space, clq.J_space, lw=2)
    plt.xlabel(r"$\theta$", fontsize=18)
    plt.ylabel(r"$J(\theta)$", fontsize=18)

    t1 = clq.θ_space[np.argmax(clq.J_space)]
    tR = clq.θ_series[1, -1]
```

(continues on next page)

(continued from previous page)

```
    θ_points = [t1, tR, clq.θ_B, clq.θ_MPE]
    labels = [r"$\theta_0^R$", r"$\theta_\infty^R$",
              r"$\theta^*$", r"$\theta^{MPE}$"]

    # Add points for θs
    for θ, label in zip(θ_points, labels):
        ax.scatter(θ, clq.J_LB + 0.02 * clq.J_range, 60, 'black', 'v')
        ax.annotate(label,
                    xy=(θ, clq.J_LB + 0.01 * clq.J_range),
                    xytext=(θ - 0.01 * clq.θ_range,
                    clq.J_LB + 0.08 * clq.J_range),
                    fontsize=18)
    plt.tight_layout()
    plt.show()

plot_value_function(clq)
```



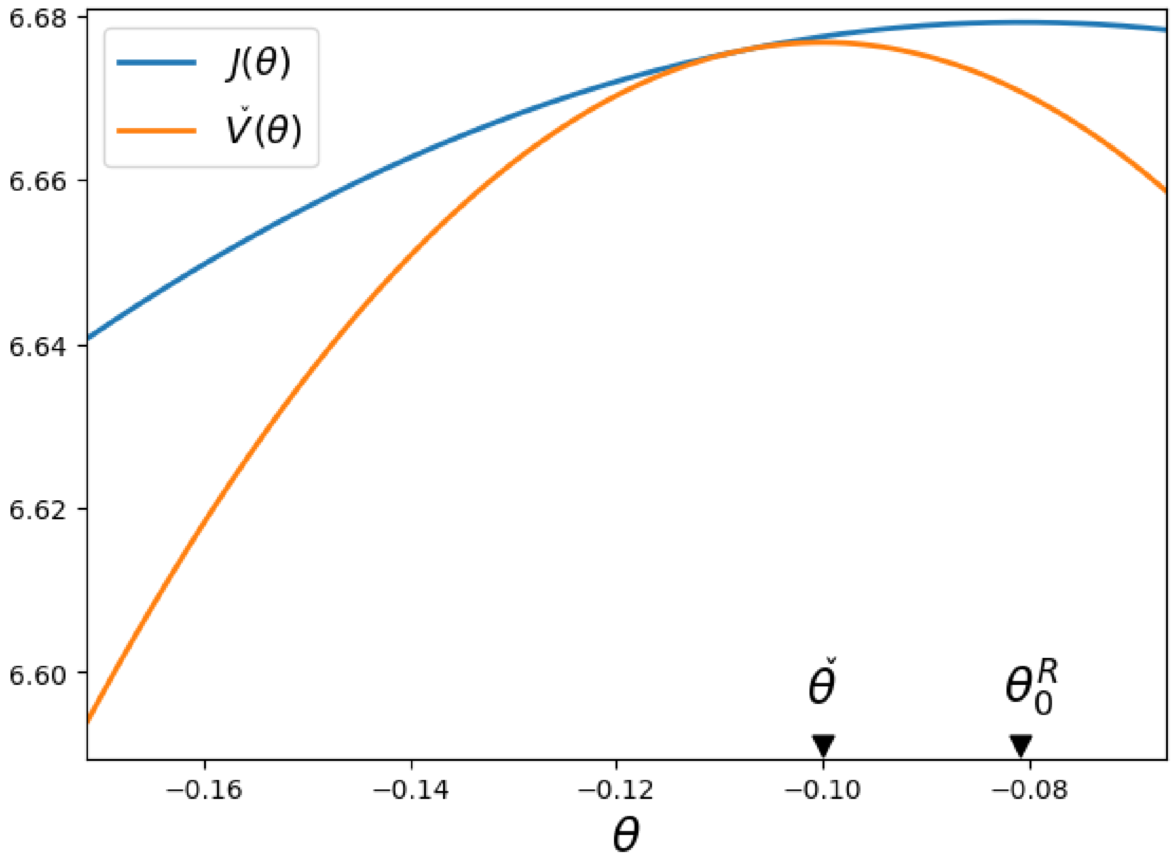
The next code generates a figure that plots the value function from the Ramsey Planner's problem as well as that for a Ramsey planner that must choose a constant μ (that in turn equals an implied constant θ).

```
def compare_ramsey_check(clq):
    """
    Method to compare values of Ramsey and Check

    Here clq is an instance of ChangLQ
```

(continues on next page)

(continued from previous page)

```
    """
    fig, ax = plt.subplots()
    check_min = min(clq.check_space)
    check_max = max(clq.check_space)
    check_range = check_max - check_min
    check_LB = check_min - 0.05 * check_range
    check_UB = check_max + 0.05 * check_range
    ax.set_xlim([clq.θ_LB, clq.θ_UB])
    ax.set_ylim([check_LB, check_UB])
    ax.plot(clq.θ_space, clq.J_space, lw=2, label=r"$J(\theta)$")

    plt.xlabel(r"$\theta$", fontsize=18)
    ax.plot(clq.θ_space, clq.check_space,
            lw=2, label=r"$V^\check(\theta)$")
    plt.legend(fontsize=14, loc='upper left')

    θ_points = [clq.θ_space[np.argmax(clq.J_space)],
                clq.μ_check]
    labels = [r"$\theta_0^R$", r"$\theta^\check$"]

    for θ, label in zip(θ_points, labels):
        ax.scatter(θ, check_LB + 0.02 * check_range, 60, 'k', 'v')
        ax.annotate(label,
                    xy=(θ, check_LB + 0.01 * check_range),
                    xytext=(θ - 0.02 * check_range,
                            check_LB + 0.08 * check_range),
                    fontsize=18)
    plt.tight_layout()
    plt.show()

compare_ramsey_check(clq)
```



The next code generates figures that plot the policy functions for a continuation Ramsey planner.

The left figure shows the choice of θ' chosen by a continuation Ramsey planner who inherits θ.

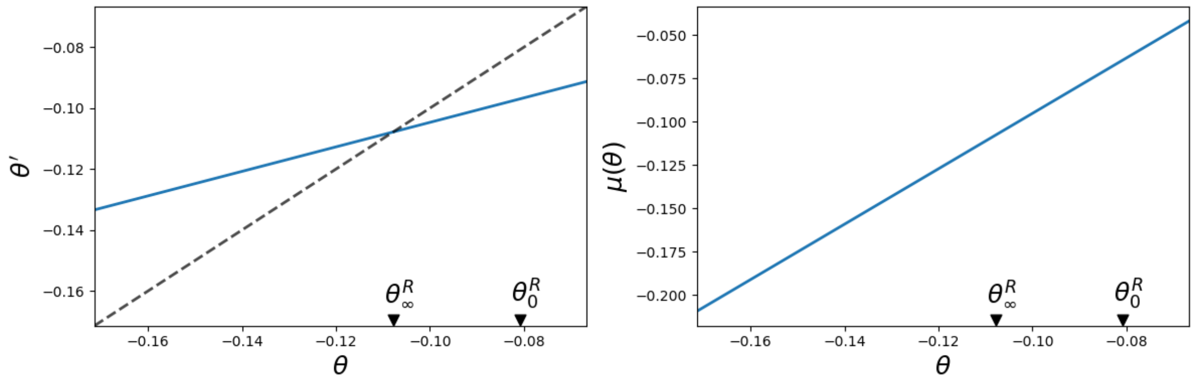
The right figure plots a continuation Ramsey planner's choice of μ as a function of an inherited θ.

```
def plot_policy_functions(clq):
    """
    Method to plot the policy functions over the relevant range of θ

    Here clq is an instance of ChangLQ
    """
    fig, axes = plt.subplots(1, 2, figsize=(12, 4))

    labels = [r"$\theta_0^R$", r"$\theta_\infty^R$"]

    ax = axes[0]
    ax.set_ylim([clq.θ_LB, clq.θ_UB])
    ax.plot(clq.θ_space, clq.θ_prime,
            label=r"$\theta'(\theta)$", lw=2)
    x = np.linspace(clq.θ_LB, clq.θ_UB, 5)
    ax.plot(x, x, 'k--', lw=2, alpha=0.7)
    ax.set_ylabel(r"$\theta'$", fontsize=18)

    θ_points = [clq.θ_space[np.argmax(clq.J_space)],
                clq.θ_series[1, -1]]
```

(continues on next page)

(continued from previous page)

```
    for θ, label in zip(θ_points, labels):
        ax.scatter(θ, clq.θ_LB + 0.02 * clq.θ_range, 60, 'k', 'v')
        ax.annotate(label,
                    xy=(θ, clq.θ_LB + 0.01 * clq.θ_range),
                    xytext=(θ - 0.02 * clq.θ_range,
                            clq.θ_LB + 0.08 * clq.θ_range),
                    fontsize=18)

    ax = axes[1]
    μ_min = min(clq.μ_space)
    μ_max = max(clq.μ_space)
    μ_range = μ_max - μ_min
    ax.set_ylim([μ_min - 0.05 * μ_range, μ_max + 0.05 * μ_range])
    ax.plot(clq.θ_space, clq.μ_space, lw=2)
    ax.set_ylabel(r"$\mu(\theta)$", fontsize=18)

    for ax in axes:
        ax.set_xlabel(r"$\theta$", fontsize=18)
        ax.set_xlim([clq.θ_LB, clq.θ_UB])

    for θ, label in zip(θ_points, labels):
        ax.scatter(θ, μ_min - 0.03 * μ_range, 60, 'black', 'v')
        ax.annotate(label, xy=(θ, μ_min - 0.03 * μ_range),
                    xytext=(θ - 0.02 * clq.θ_range,
                            μ_min + 0.03 * μ_range),
                    fontsize=18)
    plt.tight_layout()
    plt.show()

plot_policy_functions(clq)
```



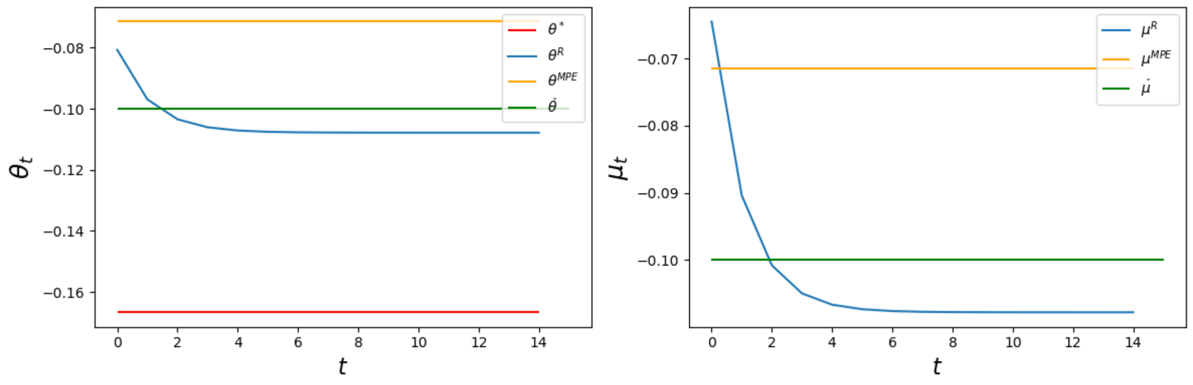
The following code generates a figure that plots sequences of μ and θ in the Ramsey plan and compares these to the constant levels in a MPE and in a Ramsey plan with a government restricted to set μ_t to a constant for all t.

```
def plot_ramsey_MPE(clq, T=15):
    """
    Method to plot Ramsey plan against Markov Perfect Equilibrium

    Here clq is an instance of ChangLQ
    """
    fig, axes = plt.subplots(1, 2, figsize=(12, 4))
```

(continues on next page)

(continued from previous page)

```
    plots = [clq.θ_series[1, 0:T], clq.μ_series[0:T]]
    MPEs = [clq.θ_MPE, clq.μ_MPE]
    labels = [r"\theta", r"\mu"]

    axes[0].hlines(clq.θ_B, 0, T-1, 'r', label=r"$\theta^*$")

    for ax, plot, MPE, label in zip(axes, plots, MPEs, labels):
        ax.plot(plot, label=r"$" + label + "^R$")
        ax.hlines(MPE, 0, T-1, 'orange', label=r"$" + label + "^{MPE}$")
        ax.hlines(clq.μ_check, 0, T, 'g', label=r"$" + label + "^\check$")
        ax.set_xlabel(r"$t$", fontsize=16)
        ax.set_ylabel(r"$" + label + "_t$", fontsize=18)
        ax.legend(loc='upper right')

    plt.tight_layout()
    plt.show()

plot_ramsey_MPE(clq)
```



3.9.1 Time Inconsistency of Ramsey Plan

The variation over time in $\vec{\mu}$ chosen by the Ramsey planner is a symptom of time inconsistency.

- The Ramsey planner reaps immediate benefits from promising lower inflation later to be achieved by costly distorting taxes.
- These benefits are intermediated by reductions in expected inflation that precede the reductions in money creation rates that rationalize them, as indicated by equation (3.3).
- A government authority offered the opportunity to ignore effects on past utilities and to reoptimize at date $t \geq 1$ would, if allowed, want to deviate from a Ramsey plan.

Note: A modified Ramsey plan constructed under the restriction that μ_t must be constant over time is time consistent (see $\check{\mu}$ and $\check{\theta}$ in the above graphs).

3.9.2 Meaning of Time Inconsistency

In settings in which governments actually choose sequentially, many economists regard a time inconsistent plan as implausible because of the incentives to deviate that are presented along the plan.

A way to summarize this *defect* in a Ramsey plan is to say that it is not credible because there endure incentives for policymakers to deviate from it.

For that reason, the Markov perfect equilibrium concept attracts many economists.

- A Markov perfect equilibrium plan is constructed to insure that government policymakers who choose sequentially do not want to deviate from it.

The *no incentive to deviate from the plan* property is what makes the Markov perfect equilibrium concept attractive.

3.9.3 Ramsey Plans Strike Back

Research by Abreu [Abr88], Chari and Kehoe [CK90] [Sto89], and Stokey [Sto91] discovered conditions under which a Ramsey plan can be rescued from the complaint that it is not credible.

They accomplished this by expanding the description of a plan to include expectations about **adverse consequences** of deviating from it that can serve to deter deviations.

We turn to such theories of **sustainable plans** next.

3.10 A Fourth Model of Government Decision Making

This is a model in which

- the government chooses $\{\mu_t\}_{t=0}^\infty$ not once and for all at $t = 0$ but chooses to set μ_t at time t, not before.
- private agents' forecasts of $\{\mu_{t+j+1}, \theta_{t+j+1}\}_{j=0}^\infty$ respond to whether the government at t **confirms** or **disappoints** their forecasts of μ_t brought into period t from period $t-1$.
- the government at each time t understands how private agents' forecasts will respond to its choice of μ_t.
- at each t, the government chooses μ_t to maximize a continuation discounted utility.

3.10.1 A Theory of Government Decision Making

$\vec{\mu}$ is chosen by a sequence of government decision makers, one for each $t \geq 0$.

We assume the following within-period and between-period timing protocol for each $t \geq 0$:

- at time $t-1$, private agents expect that the government will set $\mu_t = \tilde{\mu}_t$, and more generally that it will set $\mu_{t+j} = \tilde{\mu}_{t+j}$ for all $j \geq 0$.
- The forecasts $\{\tilde{\mu}_{t+j}\}_{j\geq 0}$ determine a $\theta_t = \tilde{\theta}_t$ and an associated log of real balances $m_t - p_t = -\alpha\tilde{\theta}_t$ at t.
- Given those expectations and an associated $\theta_t = \tilde{\theta}_t$, at t a government is free to set $\mu_t \in \mathbf{R}$.
- If the government at t **confirms** private agents' expectations by setting $\mu_t = \tilde{\mu}_t$ at time t, private agents expect the continuation government policy $\{\tilde{\mu}_{t+j+1}\}_{j=0}^\infty$ and therefore bring expectation $\tilde{\theta}_{t+1}$ into period $t+1$.
- If the government at t **disappoints** private agents by setting $\mu_t \neq \tilde{\mu}_t$, private agents expect $\{\mu_j^A\}_{j=0}^\infty$ as the continuation policy for $t+1$, i.e., $\{\mu_{t+j+1}\} = \{\mu_j^A\}_{j=0}^\infty$ and therefore expect an associated θ_0^A for $t+1$. Here $\vec{\mu}^A = \{\mu_j^A\}_{j=0}^\infty$ is an alternative government plan to be described below.

3.10.2 Temptation to Deviate from Plan

The government's one-period return function $s(\theta, \mu)$ described in equation (3.6) above has the property that for all θ

$$-s(\theta, 0) \geq -s(\theta, \mu)$$

This inequality implies that whenever the policy calls for the government to set $\mu \neq 0$, the government could raise its one-period payoff by setting $\mu = 0$.

Disappointing private sector expectations in that way would increase the government's **current** payoff but would have adverse consequences for **subsequent** government payoffs because the private sector would alter its expectations about future settings of μ.

The **temporary** gain constitutes the government's temptation to deviate from a plan.

If the government at t is to resist the temptation to raise its current payoff, it is only because it forecasts adverse consequences that its setting of μ_t would bring for continuation government payoffs via alterations in the private sector's expectations.

3.11 Sustainable or Credible Plan

We call a plan $\vec{\mu}$ **sustainable** or **credible** if at each $t \geq 0$ the government chooses to confirm private agents' prior expectation of its setting for μ_t.

The government will choose to confirm prior expectations only if the long-term **loss** from disappointing private sector expectations – coming from the government's understanding of the way the private sector adjusts its expectations in response to having its prior expectations at t disappointed – outweigh the short-term **gain** from disappointing those expectations.

The theory of sustainable or credible plans assumes throughout that private sector expectations about what future governments will do are based on the assumption that governments at times $t \geq 0$ always act to maximize the continuation discounted utilities that describe those governments' purposes.

This aspect of the theory means that credible plans always come in **pairs**:

- a credible (continuation) plan to be followed if the government at t **confirms** private sector expectations
- a credible plan to be followed if the government at t **disappoints** private sector expectations

That credible plans come in pairs threaten to bring an explosion of plans to keep track of

- each credible plan itself consists of two credible plans
- therefore, the number of plans underlying one plan is unbounded

But Dilip Abreu showed how to render manageable the number of plans that must be kept track of.

The key is an object called a **self-enforcing** plan.

3.11.1 Abreu's Self-Enforcing Plan

A plan $\vec{\mu}^A$ (here the superscipt A is for Abreu) is said to be **self-enforcing** if

- the consequence of disappointing private agents' expectations at time j is to **restart** plan $\vec{\mu}^A$ at time $j+1$
- the consequence of restarting the plan is sufficiently adverse that it forever deters all deviations from the plan

More precisely, a government plan $\vec{\mu}^A$ with equilibrium inflation sequence $\vec{\theta}^A$ is **self-enforcing** if

$$\begin{aligned} v_j^A &= -s(\theta_j^A, \mu_j^A) + \beta v_{j+1}^A \\ &\geq -s(\theta_j^A, 0) + \beta v_0^A \equiv v_j^{A,D}, \quad j \geq 0 \end{aligned} \tag{3.10}$$

(Here it is useful to recall that setting $\mu = 0$ is the maximizing choice for the government's one-period return function)

The first line tells the consequences of confirming private agents' expectations by following the plan, while the second line tells the consequences of disappointing private agents' expectations by deviating from the plan.

A consequence of the inequality stated in the definition is that a self-enforcing plan is credible.

Self-enforcing plans can be used to construct other credible plans, including ones with better values.

Thus, where $\vec{v}^A$ is the value associated with a self-enforcing plan $\vec{\mu}^A$, a sufficient condition for another plan $\vec{\mu}$ associated with inflation $\vec{\theta}$ and value $\vec{v}$ to be **credible** is that

$$\begin{aligned} v_j &= -s(\theta_j, \mu_j) + \beta v_{j+1} \\ &\geq -s(\theta_j, 0) + \beta v_0^A \quad \forall j \geq 0 \end{aligned} \tag{3.11}$$

For this condition to be satisfied it is necessary and sufficient that

$$-s(\theta_j, 0) - (-s(\theta_j, \mu_j)) < \beta(v_{j+1} - v_0^A)$$

The left side of the above inequality is the government's **gain** from deviating from the plan, while the right side is the government's **loss** from deviating from the plan.

A government never wants to deviate from a credible plan.

Abreu taught us that key step in constructing a credible plan is first constructing a self-enforcing plan that has a low time 0 value.

The idea is to use the self-enforcing plan as a continuation plan whenever the government's choice at time t fails to confirm private agents' expectation.

We shall use a construction featured in Abreu ([Abr88]) to construct a self-enforcing plan with low time 0 value.

3.11.2 Abreu Carrot-Stick Plan

Abreu ([Abr88]) invented a way to create a self-enforcing plan with a low initial value.

Imitating his idea, we can construct a self-enforcing plan $\vec{\mu}$ with a low time 0 value to the government by insisting that future government decision makers set μ_t to a value yielding low one-period utilities to the household for a long time, after which government decisions thereafter yield high one-period utilities.

- Low one-period utilities early are a **stick**
- High one-period utilities later are a **carrot**

Consider a candidate plan $\vec{\mu}^A$ that sets $\mu_t^A = \bar{\mu}$ (a high positive number) for T_A periods, and then reverts to the Ramsey plan.

Denote this sequence by $\{\mu_t^A\}_{t=0}^\infty$.

The sequence of inflation rates implied by this plan, $\{\theta_t^A\}_{t=0}^\infty$, can be calculated using:

$$\theta_t^A = \frac{1}{1+\alpha} \sum_{j=0}^{\infty} \left(\frac{\alpha}{1+\alpha} \right)^j \mu_{t+j}^A$$

The value of $\{\theta_t^A, \mu_t^A\}_{t=0}^\infty$ at time 0 is

$$v_0^A = - \sum_{t=0}^{T_A - 1} \beta^t s(\theta_t^A, \mu_t^A) + \beta^{T_A} J(\theta_0^R)$$

For an appropriate T_A, this plan can be verified to be self-enforcing and therefore credible.

3.11.3 Example of Self-Enforcing Plan

The following example implements an Abreu stick-and-carrot plan.

The government sets $\mu_t^A = 0.1$ for $t = 0, 1, \ldots, 9$ and then starts the **Ramsey plan**.

We have computed outcomes for this plan.

For this plan, we plot the θ^A, μ^A sequences as well as the implied v^A sequence.

Notice that because the government sets money supply growth high for 10 periods, inflation starts high.

Inflation gradually slowly declines because people expect the government to lower the money growth rate after period 10.

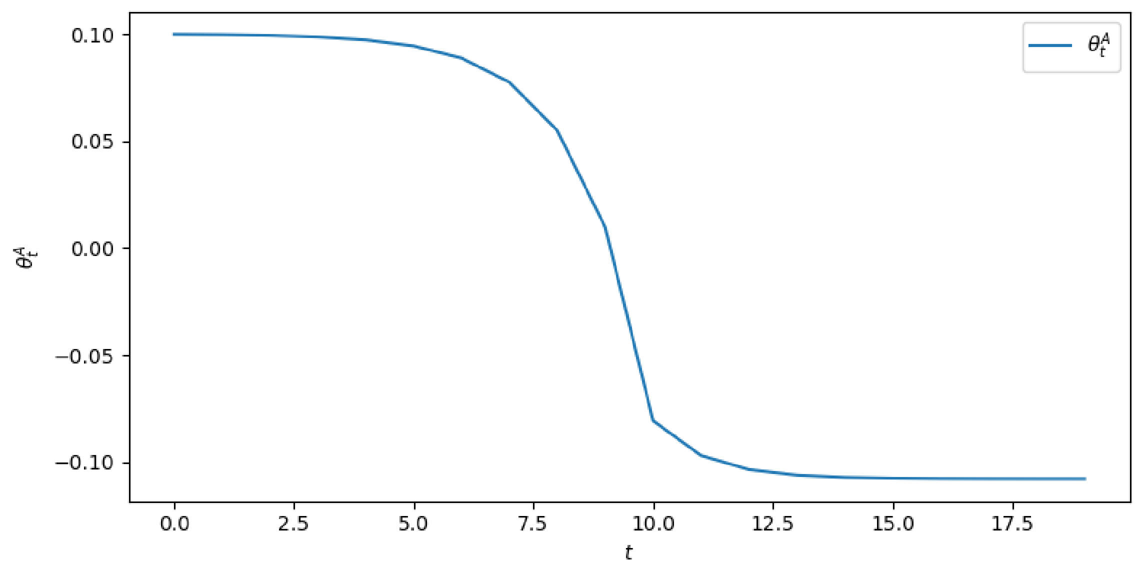
From the 10th period onwards, the inflation rate θ_t^A associated with this **Abreu plan** starts the Ramsey plan from its beginning, i.e., $\theta_{t+10}^A = \theta_t^R \ \ \forall t \geq 0$.

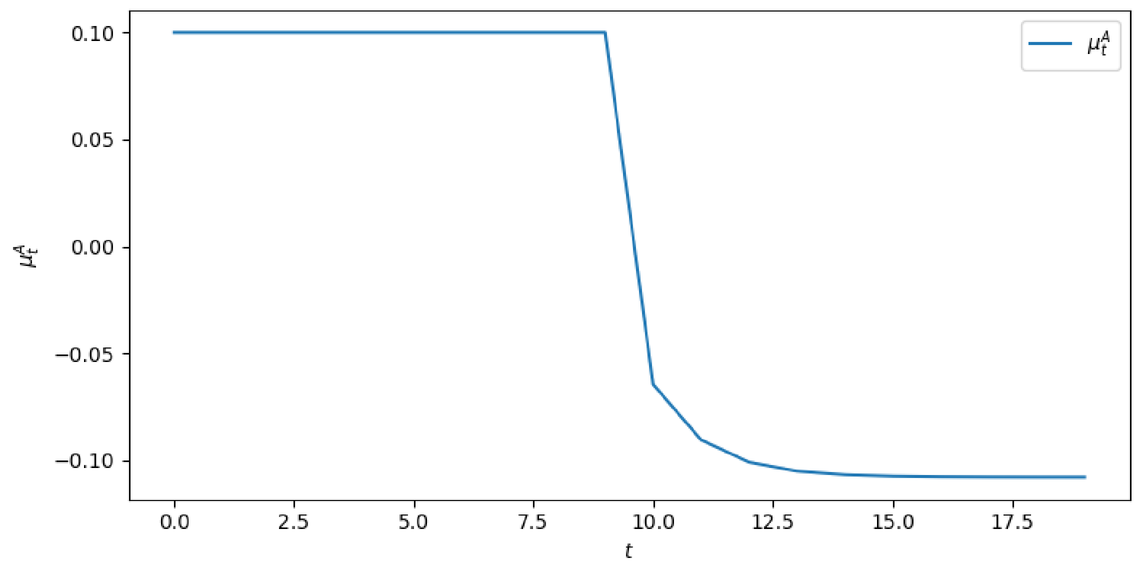
```python
def abreu_plan(clq, T=1000, T_A=10, μ_bar=0.1, T_Plot=20):

    # Append Ramsey μ series to stick μ series
    clq.μ_A = np.append(np.full(T_A, μ_bar), clq.μ_series[:-T_A])

    # Calculate implied stick θ series
    clq.θ_A = np.zeros(T)
    discount = np.zeros(T)
    for t in range(T):
        discount[t] = (clq.α / (1 + clq.α))**t
    for t in range(T):
        length = clq.μ_A[t:].shape[0]
        clq.θ_A[t] = 1 / (clq.α + 1) * sum(clq.μ_A[t:] * discount[0:length])

    # Calculate utility of stick plan
    U_A = np.zeros(T)
    for t in range(T):
        U_A[t] = clq.β**t * (clq.α0 + clq.α1 * (-clq.θ_A[t])
                  - clq.α2 / 2 * (-clq.θ_A[t])**2 - clq.c * clq.μ_A[t]**2)

    clq.V_A = np.zeros(T)
    for t in range(T):
        clq.V_A[t] = sum(U_A[t:] / clq.β**t)

    # Make sure Abreu plan is self-enforcing
    clq.V_dev = np.zeros(T_Plot)
    for t in range(T_Plot):
        clq.V_dev[t] = (clq.α0 + clq.α1 * (-clq.θ_A[t])
```

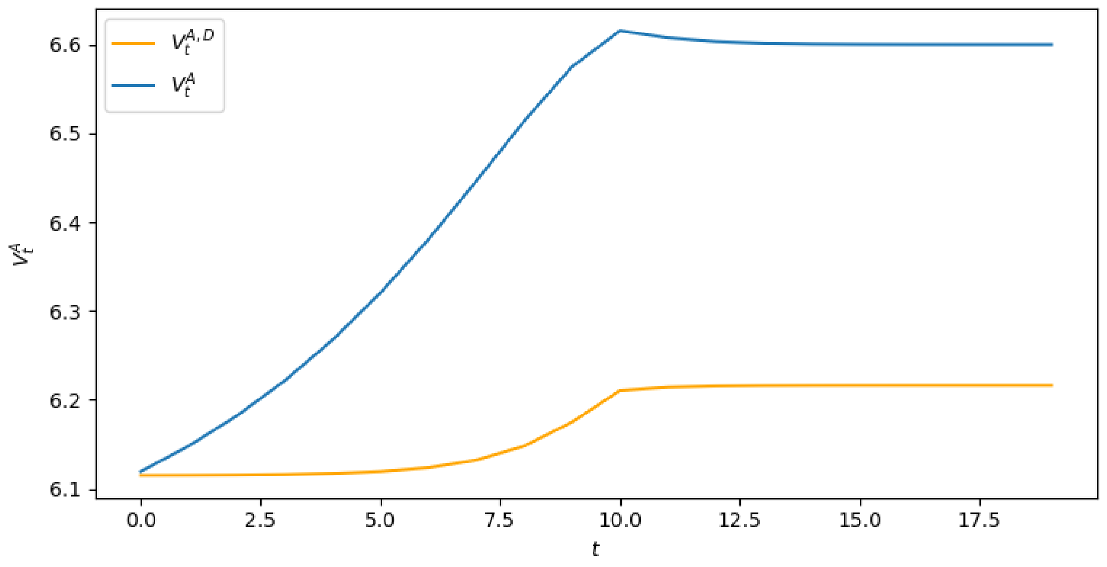
(continues on next page)

(continued from previous page)

```
                        - clq.α2 / 2 * (-clq.θ_A[t])**2) \
                        + clq.β * clq.V_A[0]

    fig, axes = plt.subplots(3, 1, figsize=(8, 12))

    axes[2].plot(clq.V_dev[0:T_Plot], label="$V^{A, D}_t$", c="orange")

    plots = [clq.θ_A, clq.μ_A, clq.V_A]
    labels = [r"$\theta_t^A$", r"$\mu_t^A$", r"$V^A_t$"]

    for plot, ax, label in zip(plots, axes, labels):
        ax.plot(plot[0:T_Plot], label=label)
        ax.set(xlabel="$t$", ylabel=label)
        ax.legend()

    plt.tight_layout()
    plt.show()

abreu_plan(clq)
```


θ_t^A
θ_t^A
t
0.10
0.05
0.00
−0.05
−0.10
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


μ_t^A
μ_t^A
t
0.10
0.05
0.00
−0.05
−0.10
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


$V_t^{A,D}$
V_t^A
V_t^A
t
6.6
6.5
6.4
6.3
6.2
6.1
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

To confirm that the plan $\vec{\mu}^A$ is **self-enforcing**, we plot an object that we call $V_t^{A,D}$, defined in the key inequality in the second line of equation (3.10) above.

$V_t^{A,D}$ is the value at t of deviating from the self-enforcing plan $\vec{\mu}^A$ by setting $\mu_t = 0$ and then restarting the plan at v_0^A at $t+1$:

$$v_t^{A,D} = -s(\theta_j, 0) + \beta v_0^A$$

In the above graph $v_t^A > v_t^{A,D}$, which confirms that $\vec{\mu}^A$ is a self-enforcing plan.

We can also verify the inequalities required for $\vec{\mu}^A$ to be self-confirming numerically as follows

```
np.all(clq.V_A[0:20] > clq.V_dev[0:20])
```

```
True
```

Given that plan $\vec{\mu}^A$ is self-enforcing, we can check that the Ramsey plan $\vec{\mu}^R$ is credible by verifying that:

$$v_t^R \geq -s(\theta_t^R, 0) + \beta v_0^A, \quad \forall t \geq 0$$

```
def check_ramsey(clq, T=1000):
    # Make sure Ramsey plan is sustainable
    R_dev = np.zeros(T)
    for t in range(T):
        R_dev[t] = (clq.α0 + clq.α1 * (-clq.θ_series[1, t])
                    - clq.α2 / 2 * (-clq.θ_series[1, t])**2) \
                    + clq.β * clq.V_A[0]

    return np.all(clq.J_series > R_dev)

check_ramsey(clq)
```

```
True
```

3.11.4 Recursive Representation of a Sustainable Plan

We can represent a sustainable plan recursively by taking the continuation value v_t as a state variable.

We form the following 3-tuple of functions:

$$\begin{aligned} \hat{\mu}_t &= \nu_\mu(v_t) \\ \theta_t &= \nu_\theta(v_t) \\ v_{t+1} &= \nu_v(v_t, \mu_t) \end{aligned} \tag{3.12}$$

In addition to these equations, we need an initial value v_0 to characterize a sustainable plan.

The first equation of (3.12) tells the recommended value of $\hat{\mu}_t$ as a function of the promised value v_t.

The second equation of (3.12) tells the inflation rate as a function of v_t.

The third equation of (3.12) updates the continuation value in a way that depends on whether the government at t confirms private agents' expectations by setting μ_t equal to the recommended value $\hat{\mu}_t$, or whether it disappoints those expectations.

3.12 Whose Credible Plan is it?

A credible government plan $\vec{\mu}$ plays multiple roles.

- It is a sequence of actions chosen by the government.
- It is a sequence of private agents' forecasts of government actions.

Thus, $\vec{\mu}$ is both a government policy and a collection of private agents' forecasts of government policy.

Does the government *choose* policy actions or does it simply *confirm* prior private sector forecasts of those actions?

An argument in favor of the *government chooses* interpretation comes from noting that the theory of credible plans builds in a theory that the government each period chooses the action that it wants.

An argument in favor of the *simply confirm* interpretation is gathered from staring at the key inequality (3.11) that defines a credible policy.

3.13 Comparison of Equilibrium Values

We have computed plans for

- an ordinary (unrestricted) Ramsey planner who chooses a sequence $\{\mu_t\}_{t=0}^{\infty}$ at time 0
- a Ramsey planner restricted to choose a constant μ for all $t \geq 0$
- a Markov perfect sequence of governments

Below we compare equilibrium time zero values for these three.

We confirm that the value delivered by the unrestricted Ramsey planner exceeds the value delivered by the restricted Ramsey planner which in turn exceeds the value delivered by the Markov perfect sequence of governments.

```
clq.J_series[0]
```

```
6.67918822960449
```

```
clq.J_check
```

```
6.676729524674898
```

```
clq.J_MPE
```

```
6.663435886995107
```

We have also computed **credible plans** for a government or sequence of governments that choose sequentially.

These include

- a **self-enforcing** plan that gives a low initial value v_0.
- a better plan – possibly one that attains values associated with Ramsey plan – that is not self-enforcing.

3.14 Note on Dynamic Programming Squared

The theory deployed in this lecture is an application of what we nickname **dynamic programming squared**.

The nickname refers to the feature that a value satisfying one Bellman equation appears as an argument in a second Bellman equation.

Thus, our models have involved two Bellman equations:

- equation (3.1) expresses how θ_t depends on μ_t and θ_{t+1}
- equation (3.4) expresses how value v_t depends on (μ_t, θ_t) and v_{t+1}

A value θ from one Bellman equation appears as an argument of a second Bellman equation for another value v.

CHAPTER

FOUR

OPTIMAL TAXATION WITH STATE-CONTINGENT DEBT

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

4.1 Overview

This lecture describes a celebrated model of optimal fiscal policy by Robert E. Lucas, Jr., and Nancy Stokey [LS83].

The model revisits classic issues about how to pay for a war.

Here a *war* means a more or less temporary surge in an exogenous government expenditure process.

The model features

- a government that must finance an exogenous stream of government expenditures with either
  - a flat rate tax on labor, or
  - purchases and sales from a full array of Arrow state-contingent securities
- a representative household that values consumption and leisure
- a linear production function mapping labor into a single good
- a Ramsey planner who at time $t = 0$ chooses a plan for taxes and trades of Arrow securities for all $t \geq 0$

After first presenting the model in a space of sequences, we shall represent it recursively in terms of two Bellman equations formulated along lines that we encountered in *Dynamic Stackelberg models*.

As in *Dynamic Stackelberg models*, to apply dynamic programming we shall define the state vector artfully.

In particular, we shall include forward-looking variables that summarize optimal responses of private agents to a Ramsey plan.

See Optimal taxation for analysis within a linear-quadratic setting.

Let's start with some standard imports:

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import root
from quantecon import MarkovChain
from quantecon.optimize.nelder_mead import nelder_mead
from numba import njit, prange, float64
from numba.experimental import jitclass
```

4.2 A Competitive Equilibrium with Distorting Taxes

At time $t \geq 0$ a random variable s_t belongs to a time-invariant set $S = [1, 2, \ldots, S]$.

For $t \geq 0$, a history $s^t = [s_t, s_{t-1}, \ldots, s_0]$ of an exogenous state s_t has joint probability density $\pi_t(s^t)$.

We begin by assuming that government purchases $g_t(s^t)$ at time $t \geq 0$ depend on s^t.

Let $c_t(s^t)$, $\ell_t(s^t)$, and $n_t(s^t)$ denote consumption, leisure, and labor supply, respectively, at history s^t and date t.

A representative household is endowed with one unit of time that can be divided between leisure ℓ_t and labor n_t:

$$n_t(s^t) + \ell_t(s^t) = 1 \tag{4.1}$$

Output equals $n_t(s^t)$ and can be divided between $c_t(s^t)$ and $g_t(s^t)$

$$c_t(s^t) + g_t(s^t) = n_t(s^t) \tag{4.2}$$

A representative household's preferences over $\{c_t(s^t), \ell_t(s^t)\}_{t=0}^{\infty}$ are ordered by

$$\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) u[c_t(s^t), \ell_t(s^t)] \tag{4.3}$$

where the utility function u is increasing, strictly concave, and three times continuously differentiable in both arguments.

The technology pins down a pre-tax wage rate to unity for all t, s^t.

The government imposes a flat-rate tax $\tau_t(s^t)$ on labor income at time t, history s^t.

There are complete markets in one-period Arrow securities.

One unit of an Arrow security issued at time t at history s^t and promising to pay one unit of time $t+1$ consumption in state s_{t+1} costs $p_{t+1}(s_{t+1}|s^t)$.

The government issues one-period Arrow securities each period.

The government has a sequence of budget constraints whose time $t \geq 0$ component is

$$g_t(s^t) = \tau_t(s^t) n_t(s^t) + \sum_{s_{t+1}} p_{t+1}(s_{t+1}|s^t) b_{t+1}(s_{t+1}|s^t) - b_t(s_t|s^{t-1}) \tag{4.4}$$

where

- $p_{t+1}(s_{t+1}|s^t)$ is a competitive equilibrium price of one unit of consumption at date $t+1$ in state s_{t+1} at date t and history s^t.
- $b_t(s_t|s^{t-1})$ is government debt falling due at time t, history s^t.

Government debt $b_0(s_0)$ is an exogenous initial condition.

The representative household has a sequence of budget constraints whose time $t \geq 0$ component is

$$c_t(s^t) + \sum_{s_{t+1}} p_t(s_{t+1}|s^t) b_{t+1}(s_{t+1}|s^t) = [1 - \tau_t(s^t)] \, n_t(s^t) + b_t(s_t|s^{t-1}) \quad \forall t \geq 0 \tag{4.5}$$

A **government policy** is an exogenous sequence $\{g(s_t)\}_{t=0}^{\infty}$, a tax rate sequence $\{\tau_t(s^t)\}_{t=0}^{\infty}$, and a government debt sequence $\{b_{t+1}(s^{t+1})\}_{t=0}^{\infty}$.

A **feasible allocation** is a consumption-labor supply plan $\{c_t(s^t), n_t(s^t)\}_{t=0}^{\infty}$ that satisfies (4.2) at all t, s^t.

A **price system** is a sequence of Arrow security prices $\{p_{t+1}(s_{t+1}|s^t)\}_{t=0}^{\infty}$.

The household faces the price system as a price-taker and takes the government policy as given.

The household chooses $\{c_t(s^t), \ell_t(s^t)\}_{t=0}^{\infty}$ to maximize (4.3) subject to (4.5) and (4.1) for all t, s^t.

A **competitive equilibrium with distorting taxes** is a feasible allocation, a price system, and a government policy such that

- Given the price system and the government policy, the allocation solves the household's optimization problem.
- Given the allocation, government policy, and price system, the government's budget constraint is satisfied for all t, s^t.

Note: There are many competitive equilibria with distorting taxes.

They are indexed by different government policies.

The **Ramsey problem** or **optimal taxation problem** is to choose a competitive equilibrium with distorting taxes that maximizes (4.3).

4.2.1 Arrow-Debreu Version of Price System

We find it convenient sometimes to work with the Arrow-Debreu price system that is implied by a sequence of Arrow securities prices.

Let $q_t^0(s^t)$ be the price at time 0, measured in time 0 consumption goods, of one unit of consumption at time t, history s^t.

The following recursion relates Arrow-Debreu prices $\{q_t^0(s^t)\}_{t=0}^{\infty}$ to Arrow securities prices $\{p_{t+1}(s_{t+1}|s^t)\}_{t=0}^{\infty}$

$$q_{t+1}^0(s^{t+1}) = p_{t+1}(s_{t+1}|s^t) q_t^0(s^t) \quad s.t. \quad q_0^0(s^0) = 1 \tag{4.6}$$

Arrow-Debreu prices are useful when we want to compress a sequence of budget constraints into a single intertemporal budget constraint, as we shall find it convenient to do below.

4.2.2 Primal Approach

We apply a popular approach to solving a Ramsey problem, called the *primal approach*.

The idea is to use first-order conditions for household optimization to eliminate taxes and prices in favor of quantities, then pose an optimization problem cast entirely in terms of quantities.

After Ramsey quantities have been found, taxes and prices can then be unwound from the allocation.

The primal approach uses four steps:

1. Obtain first-order conditions of the household's problem and solve them for $\{q_t^0(s^t), \tau_t(s^t)\}_{t=0}^\infty$ as functions of the allocation $\{c_t(s^t), n_t(s^t)\}_{t=0}^\infty$.
2. Substitute these expressions for taxes and prices in terms of the allocation into the household's present-value budget constraint.
   - This intertemporal constraint involves only the allocation and is regarded as an *implementability constraint*.
3. Find the allocation that maximizes the utility of the representative household (4.3) subject to the feasibility constraints (4.1) and (4.2) and the implementability condition derived in step 2.
   - This optimal allocation is called the **Ramsey allocation**.
4. Use the Ramsey allocation together with the formulas from step 1 to find taxes and prices.

4.2.3 The Implementability Constraint

By sequential substitution of one one-period budget constraint (4.5) into another, we can obtain the household's present-value budget constraint:

$$\sum_{t=0}^{\infty}\sum_{s^t} q_t^0(s^t)c_t(s^t) = \sum_{t=0}^{\infty}\sum_{s^t} q_t^0(s^t)[1-\tau_t(s^t)]n_t(s^t) + b_0 \tag{4.7}$$

$\{q_t^0(s^t)\}_{t=1}^\infty$ can be interpreted as a time 0 Arrow-Debreu price system.

To approach the Ramsey problem, we study the household's optimization problem.

First-order conditions for the household's problem for $\ell_t(s^t)$ and $b_t(s_{t+1}|s^t)$, respectively, imply

$$(1-\tau_t(s^t)) = \frac{u_l(s^t)}{u_c(s^t)} \tag{4.8}$$

and

$$p_{t+1}(s_{t+1}|s^t) = \beta\pi(s_{t+1}|s^t)\left(\frac{u_c(s^{t+1})}{u_c(s^t)}\right) \tag{4.9}$$

where $\pi(s_{t+1}|s^t)$ is the probability distribution of s_{t+1} conditional on history s^t.

Equation (4.9) implies that the Arrow-Debreu price system satisfies

$$q_t^0(s^t) = \beta^t\pi_t(s^t)\frac{u_c(s^t)}{u_c(s^0)} \tag{4.10}$$

(The stochastic process $\{q_t^0(s^t)\}$ is an instance of what finance economists call a *stochastic discount factor* process.)

Using the first-order conditions (4.8) and (4.9) to eliminate taxes and prices from (4.7), we derive the *implementability condition*

$$\sum_{t=0}^{\infty}\sum_{s^t}\beta^t\pi_t(s^t)[u_c(s^t)c_t(s^t) - u_\ell(s^t)n_t(s^t)] - u_c(s^0)b_0 = 0 \tag{4.11}$$

The **Ramsey problem** is to choose a feasible allocation that maximizes

$$\sum_{t=0}^{\infty}\sum_{s^t}\beta^t\pi_t(s^t)u[c_t(s^t), 1-n_t(s^t)] \tag{4.12}$$

subject to (4.11).

4.2.4 Solution Details

First, define a "pseudo utility function"

$$V\left[c_t(s^t), n_t(s^t), \Phi\right] = u[c_t(s^t), 1 - n_t(s^t)] + \Phi\left[u_c(s^t)c_t(s^t) - u_\ell(s^t)n_t(s^t)\right] \tag{4.13}$$

where Φ is a Lagrange multiplier on the implementability condition (4.7).

Next form the Lagrangian

$$J = \sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) \Big\{ V[c_t(s^t), n_t(s^t), \Phi] + \theta_t(s^t) \Big[n_t(s^t) - c_t(s^t) - g_t(s_t) \Big] \Big\} - \Phi u_c(0) b_0 \tag{4.14}$$

where $\{\theta_t(s^t); \forall s^t\}_{t\geq0}$ is a sequence of Lagrange multipliers on the feasible conditions (4.2).

Given an initial government debt b_0, we want to maximize J with respect to $\{c_t(s^t), n_t(s^t); \forall s^t\}_{t\geq0}$ and to minimize with respect to Φ and with respect to $\{\theta(s^t); \forall s^t\}_{t\geq0}$.

The first-order conditions for the Ramsey problem for periods $t \geq 1$ and $t = 0$, respectively, are

$$\begin{aligned} c_t(s^t)&\colon (1+\Phi)u_c(s^t) + \Phi\left[u_{cc}(s^t)c_t(s^t) - u_{\ell c}(s^t)n_t(s^t)\right] - \theta_t(s^t) = 0, \quad t \geq 1 \\ n_t(s^t)&\colon -(1+\Phi)u_\ell(s^t) - \Phi\left[u_{c\ell}(s^t)c_t(s^t) - u_{\ell\ell}(s^t)n_t(s^t)\right] + \theta_t(s^t) = 0, \quad t \geq 1 \end{aligned} \tag{4.15}$$

and

$$\begin{aligned} c_0(s^0, b_0)&\colon (1+\Phi)u_c(s^0, b_0) + \Phi\left[u_{cc}(s^0, b_0)c_0(s^0, b_0) - u_{\ell c}(s^0, b_0)n_0(s^0, b_0)\right] - \theta_0(s^0, b_0) \\ &\quad - \Phi u_{cc}(s^0, b_0)b_0 = 0 \\ n_0(s^0, b_0)&\colon -(1+\Phi)u_\ell(s^0, b_0) - \Phi\left[u_{c\ell}(s^0, b_0)c_0(s^0, b_0) - u_{\ell\ell}(s^0, b_0)n_0(s^0, b_0)\right] + \theta_0(s^0, b_0) \\ &\quad + \Phi u_{c\ell}(s^0, b_0)b_0 = 0 \end{aligned} \tag{4.16}$$

Please note how these first-order conditions differ between $t = 0$ and $t \geq 1$.

It is instructive to use first-order conditions (4.15) for $t \geq 1$ to eliminate the multipliers $\theta_t(s^t)$.

For convenience, we suppress the time subscript and the index s^t and obtain

$$\begin{aligned} &(1+\Phi)u_c(c, 1-c-g) + \Phi\big[cu_{cc}(c, 1-c-g) - (c+g)u_{\ell c}(c, 1-c-g)\big] \\ &\quad = (1+\Phi)u_\ell(c, 1-c-g) + \Phi\big[cu_{c\ell}(c, 1-c-g) - (c+g)u_{\ell\ell}(c, 1-c-g)\big] \end{aligned} \tag{4.17}$$

where we have imposed conditions (4.1) and (4.2).

Equation (4.17) is one equation that can be solved to express the unknown c as a function of the exogenous variable g and the Lagrange multiplier Φ.

We also know that time $t = 0$ quantities c_0 and n_0 satisfy

$$\begin{aligned} &(1+\Phi)u_c(c, 1-c-g) + \Phi\big[cu_{cc}(c, 1-c-g) - (c+g)u_{\ell c}(c, 1-c-g)\big] \\ &\quad = (1+\Phi)u_\ell(c, 1-c-g) + \Phi\big[cu_{c\ell}(c, 1-c-g) - (c+g)u_{\ell\ell}(c, 1-c-g)\big] + \Phi(u_{cc} - u_{c,\ell})b_0 \end{aligned} \tag{4.18}$$

Notice that a counterpart to b_0 does *not* appear in (4.17), so c does not *directly* depend on it for $t \geq 1$.

But things are different for time $t = 0$.

An analogous argument for the $t = 0$ equations (4.16) leads to one equation that can be solved for c_0 as a function of the pair $(g(s_0), b_0)$ and the Lagrange multiplier Φ.

These outcomes mean that the following statement would be true even when government purchases are history-dependent functions $g_t(s^t)$ of the history of s^t.

Proposition: If government purchases are equal after two histories s^t and $\tilde{s}^\tau$ for $t, \tau \geq 0$, i.e., if

$$g_t(s^t) = g^\tau(\tilde{s}^\tau) = g$$

then it follows from (4.17) that the Ramsey choices of consumption and leisure, $(c_t(s^t), \ell_t(s^t))$ and $(c_j(\tilde{s}^\tau), \ell_j(\tilde{s}^\tau))$, are identical.

The proposition asserts that the optimal allocation is a function of the currently realized quantity of government purchases g only and does *not* depend on the specific history that preceded that realization of g.

4.2.5 The Ramsey Allocation for a Given Multiplier

Temporarily take Φ as given.

We shall compute $c_0(s^0, b_0)$ and $n_0(s^0, b_0)$ from the first-order conditions (4.16).

Evidently, for $t \geq 1$, c and n depend on the time t realization of g only.

But for $t = 0$, c and n depend on both g_0 and the government's initial debt b_0.

Thus, while b_0 influences c_0 and n_0, there appears no analogous variable b_t that influences c_t and n_t for $t \geq 1$.

The absence of b_t as a direct determinant of the Ramsey allocation for $t \geq 1$ and its presence for $t = 0$ is a symptom of the *time-inconsistency* of a Ramsey plan.

Of course, b_0 affects the Ramsey allocation for $t \geq 1$ *indirectly* through its effect on Φ.

Φ has to take a value that assures that the household and the government's budget constraints are both satisfied at a candidate Ramsey allocation and price system associated with that Φ.

4.2.6 Further Specialization

At this point, it is useful to specialize the model in the following ways.

We assume that s is governed by a finite state Markov chain with states $s \in [1, \dots, S]$ and transition matrix Π, where

$$\Pi(s'|s) = \text{Prob}(s_{t+1} = s'|s_t = s)$$

Also, assume that government purchases g are an exact time-invariant function $g(s)$ of s.

We maintain these assumptions throughout the remainder of this lecture.

4.2.7 Determining the Lagrange Multiplier

We complete the Ramsey plan by computing the Lagrange multiplier Φ on the implementability constraint (4.11).

Government budget balance restricts Φ via the following line of reasoning.

The household's first-order conditions imply

$$(1 - \tau_t(s^t)) = \frac{u_l(s^t)}{u_c(s^t)} \tag{4.19}$$

and the implied one-period Arrow securities prices

$$p_{t+1}(s_{t+1}|s^t) = \beta\Pi(s_{t+1}|s_t)\frac{u_c(s^{t+1})}{u_c(s^t)} \tag{4.20}$$

Substituting from (4.19), (4.20), and the feasibility condition (4.2) into the recursive version (4.5) of the household budget constraint gives

$$\begin{aligned} u_c(s^t)[n_t(s^t) - g_t(s^t)] + \beta \sum_{s_{t+1}} \Pi(s_{t+1}|s_t) u_c(s^{t+1}) b_{t+1}(s_{t+1}|s^t) \\ = u_l(s^t) n_t(s^t) + u_c(s^t) b_t(s_t|s^{t-1}) \end{aligned} \tag{4.21}$$

Define $x_t(s^t) = u_c(s^t) b_t(s_t|s^{t-1})$.

Notice that $x_t(s^t)$ appears on the right side of (4.21) while β times the conditional expectation of $x_{t+1}(s^{t+1})$ appears on the left side.

Hence the equation shares much of the structure of a simple asset pricing equation with x_t being analogous to the price of the asset at time t.

We learned earlier that for a Ramsey allocation $c_t(s^t), n_t(s^t)$, and $b_t(s_t|s^{t-1})$, and therefore also $x_t(s^t)$, are each functions of s_t only, being independent of the history s^{t-1} for $t \geq 1$.

That means that we can express equation (4.21) as

$$u_c(s)[n(s) - g(s)] + \beta \sum_{s'} \Pi(s'|s) x'(s') = u_l(s) n(s) + x(s) \tag{4.22}$$

where s' denotes a next period value of s and $x'(s')$ denotes a next period value of x.

Given $n(s)$ for $s = 1, \ldots, S$, equation (4.22) is easy to solve for $x(s)$ for $s = 1, \ldots, S$.

If we let $\vec{n}, \vec{g}, \vec{x}$ denote $S \times 1$ vectors whose ith elements are the respective n, g, and x values when $s = i$, and let Π be the transition matrix for the Markov state s, then we can express (4.22) as the matrix equation

$$\vec{u}_c(\vec{n} - \vec{g}) + \beta \Pi \vec{x} = \vec{u}_l \vec{n} + \vec{x} \tag{4.23}$$

This is a system of S linear equations in the $S \times 1$ vector x, whose solution is

$$\vec{x} = (I - \beta \Pi)^{-1} [\vec{u}_c(\vec{n} - \vec{g}) - \vec{u}_l \vec{n}] \tag{4.24}$$

In these equations, by $\vec{u}_c \vec{n}$, for example, we mean element-by-element multiplication of the two vectors.

After solving for $\vec{x}$, we can find $b(s_t|s^{t-1})$ in Markov state $s_t = s$ from $b(s) = \frac{x(s)}{u_c(s)}$ or the matrix equation

$$\vec{b} = \frac{\vec{x}}{\vec{u}_c} \tag{4.25}$$

where division here means an element-by-element division of the respective components of the $S \times 1$ vectors $\vec{x}$ and $\vec{u}_c$.

Here is a computational algorithm:

1. Start with a guess for the value for Φ, then use the first-order conditions and the feasibility conditions to compute $c(s_t), n(s_t)$ for $s \in [1, \ldots, S]$ and $c_0(s_0, b_0)$ and $n_0(s_0, b_0)$, given Φ.
   - these are $2(S + 1)$ equations in $2(S + 1)$ unknowns.
2. Solve the S equations (4.24) for the S elements of $\vec{x}$.
   - these depend on Φ.
3. Find a Φ that satisfies

$$u_{c,0} b_0 = u_{c,0}(n_0 - g_0) - u_{l,0} n_0 + \beta \sum_{s=1}^{S} \Pi(s|s_0) x(s) \tag{4.26}$$

   by gradually raising Φ if the left side of (4.26) exceeds the right side and lowering Φ if the left side is less than the right side.

4. After computing a Ramsey allocation, recover the flat tax rate on labor from (4.8) and the implied one-period Arrow securities prices from (4.9).

In summary, when g_t is a time-invariant function of a Markov state s_t, a Ramsey plan can be constructed by solving $3S + 3$ equations for S components each of $\vec{c}$, $\vec{n}$, and $\vec{x}$ together with n_0, c_0, and Φ.

4.2.8 Time Inconsistency

Let $\{\tau_t(s^t)\}_{t=0}^{\infty}, \{b_{t+1}(s_{t+1}|s^t)\}_{t=0}^{\infty}$ be a time 0, state s_0 Ramsey plan.

Then $\{\tau_j(s^j)\}_{j=t}^{\infty}, \{b_{j+1}(s_{j+1}|s^j)\}_{j=t}^{\infty}$ is a time t, history s^t continuation of a time 0, state s_0 Ramsey plan.

A time t, history s^t Ramsey plan is a Ramsey plan that starts from initial conditions $s^t, b_t(s_t|s^{t-1})$.

A time t, history s^t continuation of a time 0, state 0 Ramsey plan is *not* a time t, history s^t Ramsey plan.

The means that a Ramsey plan is *not time consistent*.

Another way to say the same thing is that a Ramsey plan is *time inconsistent*.

The reason is that a continuation Ramsey plan takes $u_{ct} b_t(s_t|s^{t-1})$ as given, not $b_t(s_t|s^{t-1})$.

We shall discuss this more below.

4.2.9 Specification with CRRA Utility

In our calculations below and in a *subsequent lecture* based on an *extension* of the Lucas-Stokey model by Aiyagari, Marcet, Sargent, and Seppälä (2002) [AMSSeppala02], we shall modify the one-period utility function assumed above.

(We adopted the preceding utility specification because it was the one used in the original Lucas-Stokey paper [LS83]. We shall soon revert to that specification in a subsequent section.)

We will modify their specification by instead assuming that the representative agent has utility function

$$u(c, n) = \frac{c^{1-\sigma}}{1-\sigma} - \frac{n^{1+\gamma}}{1+\gamma}$$

where $\sigma > 0$, $\gamma > 0$.

We continue to assume that

$$c_t + g_t = n_t$$

We eliminate leisure from the model.

We also eliminate Lucas and Stokey's restriction that $\ell_t + n_t \leq 1$.

We replace these two things with the assumption that labor $n_t \in [0, +\infty]$.

With these adjustments, the analysis of Lucas and Stokey prevails once we make the following replacements

$$\begin{aligned}
u_\ell(c, \ell) &\sim -u_n(c, n) \\
u_c(c, \ell) &\sim u_c(c, n) \\
u_{\ell,\ell}(c, \ell) &\sim u_{nn}(c, n) \\
u_{c,c}(c, \ell) &\sim u_{c,c}(c, n) \\
u_{c,\ell}(c, \ell) &\sim 0
\end{aligned}$$

With these understandings, equations (4.17) and (4.18) simplify in the case of the CRRA utility function.

They become

$$(1+\Phi)[u_c(c)+u_n(c+g)]+\Phi[cu_{cc}(c)+(c+g)u_{nn}(c+g)]=0 \tag{4.27}$$

and

$$(1+\Phi)[u_c(c_0)+u_n(c_0+g_0)]+\Phi[c_0u_{cc}(c_0)+(c_0+g_0)u_{nn}(c_0+g_0)]-\Phi u_{cc}(c_0)b_0=0 \tag{4.28}$$

In equation (4.27), it is understood that c and g are each functions of the Markov state s.

In addition, the time $t=0$ budget constraint is satisfied at c_0 and initial government debt b_0:

$$b_0+g_0=\tau_0(c_0+g_0)+\beta\sum_{s=1}^{S}\Pi(s|s_0)\frac{u_c(s)}{u_{c,0}}b_1(s) \tag{4.29}$$

where τ_0 is the time $t=0$ tax rate.

In equation (4.29), it is understood that

$$\tau_0=1-\frac{u_{l,0}}{u_{c,0}}$$

4.2.10 Sequence Implementation

The above steps are implemented in a class called SequentialLS

```python
class SequentialLS:

    '''
    Class that takes a preference object, state transition matrix,
    and state contingent government expenditure plan as inputs, and
    solves the sequential allocation problem described above.
    It returns optimal allocations about consumption and labor supply,
    as well as the multiplier on the implementability constraint Φ.
    '''

    def __init__(self,
                 pref,
                 π=np.full((2, 2), 0.5),
                 g=np.array([0.1, 0.2])):

        # Initialize from pref object attributes
        self.β, self.π, self.g = pref.β, π, g
        self.mc = MarkovChain(self.π)
        self.S = len(π)  # Number of states
        self.pref = pref

        # Find the first best allocation
        self.find_first_best()

    def FOC_first_best(self, c, g):
        '''
        First order conditions that characterize
        the first best allocation.
        '''
```

(continues on next page)

(continued from previous page)

```
        pref = self.pref
        Uc, Ul = pref.Uc, pref.Ul

        n = c + g
        l = 1 - n

        return Uc(c, l) - Ul(c, l)

    def find_first_best(self):
        '''
        Find the first best allocation
        '''
        S, g = self.S, self.g

        res = root(self.FOC_first_best, np.full(S, 0.5), args=(g,))

        if (res.fun > 1e-10).any():
            raise Exception('Could not find first best')

        self.cFB = res.x
        self.nFB = self.cFB + g

    def FOC_time1(self, c, Φ, g):
        '''
        First order conditions that characterize
        optimal time 1 allocation problems.
        '''

        pref = self.pref
        Uc, Ucc, Ul, Ull, Ulc = pref.Uc, pref.Ucc, pref.Ul, pref.Ull, pref.Ulc

        n = c + g
        l = 1 - n

        LHS = (1 + Φ) * Uc(c, l) + Φ * (c * Ucc(c, l) - n * Ulc(c, l))
        RHS = (1 + Φ) * Ul(c, l) + Φ * (c * Ulc(c, l) - n * Ull(c, l))

        diff = LHS - RHS

        return diff

    def time1_allocation(self, Φ):
        '''
        Computes optimal allocation for time t >= 1 for a given Φ
        '''
        pref = self.pref
        S, g = self.S, self.g

        # use the first best allocation as intial guess
        res = root(self.FOC_time1, self.cFB, args=(Φ, g))

        if (res.fun > 1e-10).any():
            raise Exception('Could not find LS allocation.')

        c = res.x
        n = c + g
```

(continues on next page)

(continued from previous page)

```
        l = 1 - n

        # Compute x
        I = pref.Uc(c, n) * c - pref.Ul(c, l) * n
        x = np.linalg.solve(np.eye(S) - self.β * self.π, I)

        return c, n, x

    def FOC_time0(self, c0, Φ, g0, b0):
        '''
        First order conditions that characterize
        time 0 allocation problem.
        '''

        pref = self.pref
        Ucc, Ulc = pref.Ucc, pref.Ulc

        n0 = c0 + g0
        l0 = 1 - n0

        diff = self.FOC_time1(c0, Φ, g0)
        diff -= Φ * (Ucc(c0, l0) - Ulc(c0, l0)) * b0

        return diff

    def implementability(self, Φ, b0, s0, cn0_arr):
        '''
        Compute the differences between the RHS and LHS
        of the implementability constraint given Φ,
        initial debt, and initial state.
        '''

        pref, π, g, β = self.pref, self.π, self.g, self.β
        Uc, Ul = pref.Uc, pref.Ul
        g0 = self.g[s0]

        c, n, x = self.time1_allocation(Φ)

        res = root(self.FOC_time0, cn0_arr[0], args=(Φ, g0, b0))
        c0 = res.x
        n0 = c0 + g0
        l0 = 1 - n0

        cn0_arr[:] = c0.item(), n0.item()

        LHS = Uc(c0, l0) * b0
        RHS = Uc(c0, l0) * c0 - Ul(c0, l0) * n0 + β * π[s0] @ x

        return RHS - LHS

    def time0_allocation(self, b0, s0):
        '''
        Finds the optimal time 0 allocation given
        initial government debt b0 and state s0
        '''
```

(continues on next page)

(continued from previous page)

```
        # use the first best allocation as initial guess
        cn0_arr = np.array([self.cFB[s0], self.nFB[s0]])

        res = root(self.implementability, 0., args=(b0, s0, cn0_arr))

        if (res.fun > 1e-10).any():
            raise Exception('Could not find time 0 LS allocation.')

        Φ = res.x[0]
        c0, n0 = cn0_arr

        return Φ, c0, n0

    def τ(self, c, n):
        '''
        Computes τ given c, n
        '''
        pref = self.pref
        Uc, Ul = pref.Uc, pref.Ul

        return 1 - Ul(c, 1-n) / Uc(c, 1-n)

    def simulate(self, b0, s0, T, sHist=None):
        '''
        Simulates planners policies for T periods
        '''
        pref, π, β = self.pref, self.π, self.β
        Uc = pref.Uc

        if sHist is None:
            sHist = self.mc.simulate(T, s0)

        cHist, nHist, Bhist, τHist, ΦHist = np.empty((5, T))
        RHist = np.empty(T-1)

        # Time 0
        Φ, cHist[0], nHist[0] = self.time0_allocation(b0, s0)
        τHist[0] = self.τ(cHist[0], nHist[0])
        Bhist[0] = b0
        ΦHist[0] = Φ

        # Time 1 onward
        for t in range(1, T):
            c, n, x = self.time1_allocation(Φ)
            τ = self.τ(c, n)
            u_c = Uc(c, 1-n)
            s = sHist[t]
            Eu_c = π[sHist[t-1]] @ u_c
            cHist[t], nHist[t], Bhist[t], τHist[t] = c[s], n[s], x[s] / u_c[s], τ[s]
            RHist[t-1] = Uc(cHist[t-1], 1-nHist[t-1]) / (β * Eu_c)
            ΦHist[t] = Φ

        gHist = self.g[sHist]
        yHist = nHist

        return [cHist, nHist, Bhist, τHist, gHist, yHist, sHist, ΦHist, RHist]
```

4.3 Recursive Formulation of the Ramsey Problem

We now temporarily revert to Lucas and Stokey's specification.

We start by noting that $x_t(s^t) = u_c(s^t)b_t(s_t|s^{t-1})$ in equation (4.21) appears to be a purely "forward-looking" variable.

But $x_t(s^t)$ is a natural candidate for a state variable in a recursive formulation of the Ramsey problem, one that records history-dependence and so is `backward-looking`.

4.3.1 Intertemporal Delegation

To express a Ramsey plan recursively, we imagine that a time 0 Ramsey planner is followed by a sequence of continuation Ramsey planners at times $t = 1, 2, \ldots$.

A "continuation Ramsey planner" at time $t \geq 1$ has a different objective function and faces different constraints and state variables than does the Ramsey planner at time $t = 0$.

A key step in representing a Ramsey plan recursively is to regard the marginal utility scaled government debts $x_t(s^t) = u_c(s^t)b_t(s_t|s^{t-1})$ as predetermined quantities that continuation Ramsey planners at times $t \geq 1$ are obligated to attain.

Continuation Ramsey planners do this by choosing continuation policies that induce the representative household to make choices that imply that $u_c(s^t)b_t(s_t|s^{t-1}) = x_t(s^t)$.

A time $t \geq 1$ continuation Ramsey planner faces x_t, s_t as state variables.

A time $t \geq 1$ continuation Ramsey planner delivers x_t by choosing a suitable n_t, c_t pair and a list of s_{t+1}-contingent continuation quantities x_{t+1} to bequeath to a time $t+1$ continuation Ramsey planner.

While a time $t \geq 1$ continuation Ramsey planner faces x_t, s_t as state variables, the time 0 Ramsey planner faces b_0, not x_0, as a state variable.

Furthermore, the Ramsey planner cares about $(c_0(s_0), \ell_0(s_0))$, while continuation Ramsey planners do not.

The time 0 Ramsey planner hands a state-contingent function that make x_1 a function of s_1 to a time 1, state s_1 continuation Ramsey planner.

These lines of delegated authorities and responsibilities across time express the continuation Ramsey planners' obligations to implement their parts of an original Ramsey plan that had been designed once-and-for-all at time 0.

4.3.2 Two Bellman Equations

After s_t has been realized at time $t \geq 1$, the state variables confronting the time t **continuation Ramsey planner** are (x_t, s_t).

- Let $V(x, s)$ be the value of a **continuation Ramsey plan** at $x_t = x, s_t = s$ for $t \geq 1$.
- Let $W(b, s)$ be the value of a **Ramsey plan** at time 0 at $b_0 = b$ and $s_0 = s$.

We work backward by preparing a Bellman equation for $V(x, s)$ first, then a Bellman equation for $W(b, s)$.

4.3.3 The Continuation Ramsey Problem

The Bellman equation for a time $t \geq 1$ continuation Ramsey planner is

$$V(x, s) = \max_{n, \{x'(s')\}} u(n - g(s), 1 - n) + \beta \sum_{s' \in S} \Pi(s'|s) V(x', s') \tag{4.30}$$

where maximization over n and the S elements of $x'(s')$ is subject to the single implementability constraint for $t \geq 1$:

$$x = u_c(n - g(s)) - u_l n + \beta \sum_{s' \in S} \Pi(s'|s) x'(s') \tag{4.31}$$

Here u_c and u_l are today's values of the marginal utilities.

For each given value of x, s, the continuation Ramsey planner chooses n and $x'(s')$ for each $s' \in S$.

Associated with a value function $V(x, s)$ that solves Bellman equation (4.30) are $S + 1$ time-invariant policy functions

$$\begin{aligned} n_t &= f(x_t, s_t), \quad t \geq 1 \\ x_{t+1}(s_{t+1}) &= h(s_{t+1}; x_t, s_t), \; s_{t+1} \in S, \, t \geq 1 \end{aligned} \tag{4.32}$$

4.3.4 The Ramsey Problem

The Bellman equation of the time 0 Ramsey planner is

$$W(b_0, s_0) = \max_{n_0, \{x'(s_1)\}} u(n_0 - g_0, 1 - n_0) + \beta \sum_{s_1 \in S} \Pi(s_1|s_0) V(x'(s_1), s_1) \tag{4.33}$$

where maximization over n_0 and the S elements of $x'(s_1)$ is subject to the time 0 implementability constraint

$$u_{c,0} b_0 = u_{c,0}(n_0 - g_0) - u_{l,0} n_0 + \beta \sum_{s_1 \in S} \Pi(s_1|s_0) x'(s_1) \tag{4.34}$$

coming from restriction (4.26).

Associated with a value function $W(b_0, n_0)$ that solves Bellman equation (4.33) are $S + 1$ time 0 policy functions

$$\begin{aligned} n_0 &= f_0(b_0, s_0) \\ x_1(s_1) &= h_0(s_1; b_0, s_0) \end{aligned} \tag{4.35}$$

Notice the appearance of state variables (b_0, s_0) in the time 0 policy functions for the Ramsey planner as compared to (x_t, s_t) in the policy functions (4.32) for the time $t \geq 1$ continuation Ramsey planners.

The value function $V(x_t, s_t)$ of the time t continuation Ramsey planner equals $E_t \sum_{\tau=t}^{\infty} \beta^{\tau-t} u(c_\tau, l_\tau)$, where consumption and leisure processes are evaluated along the original time 0 Ramsey plan.

4.3.5 First-Order Conditions

Attach a Lagrange multiplier $\Phi_1(x, s)$ to constraint (4.31) and a Lagrange multiplier Φ_0 to constraint (4.26).

Time $t \geq 1$: First-order conditions for the time $t \geq 1$ constrained maximization problem on the right side of the continuation Ramsey planner's Bellman equation (4.30) are

$$\beta \Pi(s'|s) V_x(x', s') - \beta \Pi(s'|s) \Phi_1 = 0 \tag{4.36}$$

for $x'(s')$ and

$$(1 + \Phi_1)(u_c - u_l) + \Phi_1 \left[n(u_{ll} - u_{lc}) + (n - g(s))(u_{cc} - u_{lc}) \right] = 0 \tag{4.37}$$

for n.

Given Φ_1, equation (4.37) is one equation to be solved for n as a function of s (or of $g(s)$).

Equation (4.36) implies $V_x(x', s') = \Phi_1$, while an envelope condition is $V_x(x, s) = \Phi_1$, so it follows that

$$V_x(x', s') = V_x(x, s) = \Phi_1(x, s) \tag{4.38}$$

Time $t = 0$: For the time 0 problem on the right side of the Ramsey planner's Bellman equation (4.33), first-order conditions are

$$V_x(x(s_1), s_1) = \Phi_0 \tag{4.39}$$

for $x(s_1), s_1 \in S$, and

$$\begin{aligned} (1 + \Phi_0)(u_{c,0} - u_{n,0}) + \Phi_0\big[n_0(u_{ll,0} - u_{lc,0}) + (n_0 - g(s_0))(u_{cc,0} - u_{cl,0})\big] \\ - \Phi_0(u_{cc,0} - u_{cl,0})b_0 = 0 \end{aligned} \tag{4.40}$$

Notice similarities and differences between the first-order conditions for $t \geq 1$ and for $t = 0$.

An additional term is present in (4.40) except in three special cases

- $b_0 = 0$, or
- u_c is constant (i.e., preferences are quasi-linear in consumption), or
- initial government assets are sufficiently large to finance all government purchases with interest earnings from those assets so that $\Phi_0 = 0$

Except in these special cases, the allocation and the labor tax rate as functions of s_t differ between dates $t = 0$ and subsequent dates $t \geq 1$.

Naturally, the first-order conditions in this recursive formulation of the Ramsey problem agree with the first-order conditions derived when we first formulated the Ramsey plan in the space of sequences.

4.3.6 State Variable Degeneracy

Equations (4.38) and (4.39) imply that $\Phi_0 = \Phi_1$ and that

$$V_x(x_t, s_t) = \Phi_0 \tag{4.41}$$

for all $t \geq 1$.

When V is concave in x, this implies *state-variable degeneracy* along a Ramsey plan in the sense that for $t \geq 1$, x_t will be a time-invariant function of s_t.

Given Φ_0, this function mapping s_t into x_t can be expressed as a vector $\vec{x}$ that solves equation (4.34) for n and c as functions of g that are associated with $\Phi = \Phi_0$.

4.3.7 Manifestations of Time Inconsistency

While the marginal utility adjusted level of government debt x_t is a key state variable for the continuation Ramsey planners at $t \geq 1$, it is not a state variable at time 0.

The time 0 Ramsey planner faces b_0, not $x_0 = u_{c,0}b_0$, as a state variable.

The discrepancy in state variables faced by the time 0 Ramsey planner and the time $t \geq 1$ continuation Ramsey planners captures the differing obligations and incentives faced by the time 0 Ramsey planner and the time $t \geq 1$ continuation Ramsey planners.

- The time 0 Ramsey planner is obligated to honor government debt b_0 measured in time 0 consumption goods.
- The time 0 Ramsey planner can manipulate the *value* of government debt as measured by $u_{c,0} b_0$.
- In contrast, time $t \geq 1$ continuation Ramsey planners are obligated *not* to alter values of debt, as measured by $u_{c,t} b_t$, that they inherit from a preceding Ramsey planner or continuation Ramsey planner.

When government expenditures g_t are a time-invariant function of a Markov state s_t, a Ramsey plan and associated Ramsey allocation feature marginal utilities of consumption $u_c(s_t)$ that, given Φ, for $t \geq 1$ depend only on s_t, but that for $t = 0$ depend on b_0 as well.

This means that $u_c(s_t)$ will be a time-invariant function of s_t for $t \geq 1$, but except when $b_0 = 0$, a different function for $t = 0$.

This in turn means that prices of one-period Arrow securities $p_{t+1}(s_{t+1}|s_t) = p(s_{t+1}|s_t)$ will be the *same* time-invariant functions of (s_{t+1}, s_t) for $t \geq 1$, but a different function $p_0(s_1|s_0)$ for $t = 0$, except when $b_0 = 0$.

The differences between these time 0 and time $t \geq 1$ objects reflect the Ramsey planner's incentive to manipulate Arrow security prices and, through them, the value of initial government debt b_0.

4.3.8 Recursive Implementation

The above steps are implemented in a class called `RecursiveLS`.

```python
class RecursiveLS:

    '''
    Compute the planner's allocation by solving Bellman
    equation.
    '''

    def __init__(self,
                 pref,
                 x_grid,
                 π=np.full((2, 2), 0.5),
                 g=np.array([0.1, 0.2])):

        self.π, self.g, self.S = π, g, len(π)
        self.pref, self.x_grid = pref, x_grid

        bounds = np.empty((self.S, 2))

        # bound for n
        bounds[0] = 0, 1

        # bound for xprime
        for s in range(self.S-1):
            bounds[s+1] = x_grid.min(), x_grid.max()

        self.bounds = bounds

        # initialization of time 1 value function
        self.V = None

    def time1_allocation(self, V=None, tol=1e-7):
        '''
        Solve the optimal time 1 allocation problem
        by iterating Bellman value function.
```

(continues on next page)

(continued from previous page)

```
        '''

        π, g, S = self.π, self.g, self.S
        pref, x_grid, bounds = self.pref, self.x_grid, self.bounds

        # initial guess of value function
        if V is None:
            V = np.zeros((len(x_grid), S))

        # initial guess of policy
        z = np.empty((len(x_grid), S, S+2))

        # guess of n
        z[:, :, 1] = 0.5

        # guess of xprime
        for s in range(S):
            for i in range(S-1):
                z[:, s, i+2] = x_grid

        while True:
            # value function iteration
            V_new, z_new = T(V, z, pref, π, g, x_grid, bounds)

            if np.max(np.abs(V - V_new)) < tol:
                break

            V = V_new
            z = z_new

        self.V = V_new
        self.z1 = z_new
        self.c1 = z_new[:, :, 0]
        self.n1 = z_new[:, :, 1]
        self.xprime1 = z_new[:, :, 2:]

        return V_new, z_new

    def time0_allocation(self, b0, s0):
        '''
        Find the optimal time 0 allocation by maximization.
        '''

        if self.V is None:
            self.time1_allocation()

        π, g, S = self.π, self.g, self.S
        pref, x_grid, bounds = self.pref, self.x_grid, self.bounds
        V, z1 = self.V, self.z1

        x = 1. # x is arbitrary
        res = nelder_mead(obj_V,
                          z1[0, s0, 1:-1],
                          args=(x, s0, V, pref, π, g, x_grid, b0),
                          bounds=bounds,
                          tol_f=1e-10)
```

(continues on next page)

(continued from previous page)

```
        n0, xprime0 = IC(res.x, x, s0, b0, pref, π, g)
        c0 = n0 - g[s0]
        z0 = np.array([c0, n0, *xprime0])

        self.z0 = z0
        self.n0 = n0
        self.c0 = n0 - g[s0]
        self.xprime0 = xprime0

        return z0

    def τ(self, c, n):
        '''
        Computes τ given c, n
        '''
        pref = self.pref
        uc, ul = pref.Uc(c, 1-n), pref.Ul(c, 1-n)

        return 1 - ul / uc

    def simulate(self, b0, s0, T, sHist=None):
        '''
        Simulates Ramsey plan for T periods
        '''
        pref, π = self.pref, self.π
        Uc = pref.Uc

        if sHist is None:
            sHist = self.mc.simulate(T, s0)

        cHist, nHist, Bhist, τHist, xHist = np.empty((5, T))
        RHist = np.zeros(T-1)

        # Time 0
        self.time0_allocation(b0, s0)
        cHist[0], nHist[0], xHist[0] = self.c0, self.n0, self.xprime0[s0]
        τHist[0] = self.τ(cHist[0], nHist[0])
        Bhist[0] = b0

        # Time 1 onward
        for t in range(1, T):
            s, x = sHist[t], xHist[t-1]
            cHist[t] = np.interp(x, self.x_grid, self.c1[:, s])
            nHist[t] = np.interp(x, self.x_grid, self.n1[:, s])

            τHist[t] = self.τ(cHist[t], nHist[t])

            Bhist[t] = x / Uc(cHist[t], 1-nHist[t])

            c, n = np.empty((2, self.S))
            for sprime in range(self.S):
                c[sprime] = np.interp(x, x_grid, self.c1[:, sprime])
                n[sprime] = np.interp(x, x_grid, self.n1[:, sprime])
            Euc = π[sHist[t-1]] @ Uc(c, 1-n)
            RHist[t-1] = Uc(cHist[t-1], 1-nHist[t-1]) / (self.pref.β * Euc)
```

(continues on next page)

(continued from previous page)

```
            gHist = self.g[sHist]
            yHist = nHist

            if t < T-1:
                sprime = sHist[t+1]
                xHist[t] = np.interp(x, self.x_grid, self.xprime1[:, s, sprime])

        return [cHist, nHist, Bhist, τHist, gHist, yHist, xHist, RHist]

# Helper functions

@njit(parallel=True)
def T(V, z, pref, π, g, x_grid, bounds):
    '''
    One step iteration of Bellman value function.
    '''

    S = len(π)

    V_new = np.empty_like(V)
    z_new = np.empty_like(z)

    for i in prange(len(x_grid)):
        x = x_grid[i]
        for s in prange(S):
            res = nelder_mead(obj_V,
                              z[i, s, 1:-1],
                              args=(x, s, V, pref, π, g, x_grid),
                              bounds=bounds,
                              tol_f=1e-10)

            # optimal policy
            n, xprime = IC(res.x, x, s, None, pref, π, g)
            z_new[i, s, 0] = n - g[s]         # c
            z_new[i, s, 1] = n                # n
            z_new[i, s, 2:] = xprime          # xprime

            V_new[i, s] = res.fun

    return V_new, z_new

@njit
def obj_V(z_sub, x, s, V, pref, π, g, x_grid, b0=None):
    '''
    The objective on the right hand side of the Bellman equation.
    z_sub contains guesses of n and xprime[:-1].
    '''

    S = len(π)
    β, U = pref.β, pref.U

    # find (n, xprime) that satisfies implementability constraint
    n, xprime = IC(z_sub, x, s, b0, pref, π, g)
    c, l = n-g[s], 1-n
```

(continues on next page)

(continued from previous page)

```
    # if xprime[-1] violates bound, return large penalty
    if (xprime[-1] < x_grid.min()):
        return -1e9 * (1 + np.abs(xprime[-1] - x_grid.min()))
    elif (xprime[-1] > x_grid.max()):
        return -1e9 * (1 + np.abs(xprime[-1] - x_grid.max()))

    # prepare Vprime vector
    Vprime = np.empty(S)
    for sprime in range(S):
        Vprime[sprime] = np.interp(xprime[sprime], x_grid, V[:, sprime])

    # compute the objective value
    obj = U(c, l) + β * π[s] @ Vprime

    return obj

@njit
def IC(z_sub, x, s, b0, pref, π, g):
    '''
    Find xprime[-1] that satisfies the implementability condition
    given the guesses of n and xprime[:-1].
    '''

    β, Uc, Ul = pref.β, pref.Uc, pref.Ul

    n = z_sub[0]
    xprime = np.empty(len(π))
    xprime[:-1] = z_sub[1:]

    c, l = n-g[s], 1-n
    uc = Uc(c, l)
    ul = Ul(c, l)

    if b0 is None:
        diff = x
    else:
        diff = uc * b0

    diff -= uc * (n - g[s]) - ul * n + β * π[s][:-1] @ xprime[:-1]
    xprime[-1] = diff / (β * π[s][-1])

    return n, xprime
```

4.4 Examples

We return to the setup with CRRA preferences described above.

4.4.1 Anticipated One-Period War

This example illustrates in a simple setting how a Ramsey planner manages risk.

Government expenditures are known for sure in all periods except one

- For $t < 3$ and $t > 3$ we assume that $g_t = g_l = 0.1$.
- At $t = 3$ a war occurs with probability 0.5.
  - If there is war, $g_3 = g_h = 0.2$
  - If there is no war $g_3 = g_l = 0.1$

We define the components of the state vector as the following six (t, g) pairs: $(0, g_l), (1, g_l), (2, g_l), (3, g_l), (3, g_h), (t \geq 4, g_l)$.

We think of these 6 states as corresponding to $s = 1, 2, 3, 4, 5, 6$.

The transition matrix is

$$\Pi = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Government expenditures at each state are

$$g = \begin{pmatrix} 0.1 \\ 0.1 \\ 0.1 \\ 0.1 \\ 0.2 \\ 0.1 \end{pmatrix}$$

We assume that the representative agent has utility function

$$u(c, n) = \frac{c^{1-\sigma}}{1-\sigma} - \frac{n^{1+\gamma}}{1+\gamma}$$

and set $\sigma = 2$, $\gamma = 2$, and the discount factor $\beta = 0.9$.

Note: For convenience in terms of matching our code, we have expressed utility as a function of n rather than leisure l.

This utility function is implemented in the class `CRRAutility`.

```python
crra_util_data = [
    ('β', float64),
    ('σ', float64),
    ('γ', float64)
]

@jitclass(crra_util_data)
class CRRAutility:

    def __init__(self,
                 β=0.9,
```

(continues on next page)

(continued from previous page)

```
                 σ=2,
                 γ=2):

        self.β, self.σ, self.γ = β, σ, γ

    # Utility function
    def U(self, c, l):
        # Note: `l` should not be interpreted as labor, it is an auxiliary
        # variable used to conveniently match the code and the equations
        # in the lecture
        σ = self.σ
        if σ == 1.:
            U = np.log(c)
        else:
            U = (c**(1 - σ) - 1) / (1 - σ)
        return U - (1-l) ** (1 + self.γ) / (1 + self.γ)

    # Derivatives of utility function
    def Uc(self, c, l):
        return c ** (-self.σ)

    def Ucc(self, c, l):
        return -self.σ * c ** (-self.σ - 1)

    def Ul(self, c, l):
        return (1-l) ** self.γ

    def Ull(self, c, l):
        return -self.γ * (1-l) ** (self.γ - 1)

    def Ucl(self, c, l):
        return 0

    def Ulc(self, c, l):
        return 0
```

We set initial government debt $b_0 = 1$.

We can now plot the Ramsey tax under both realizations of time $t = 3$ government expenditures

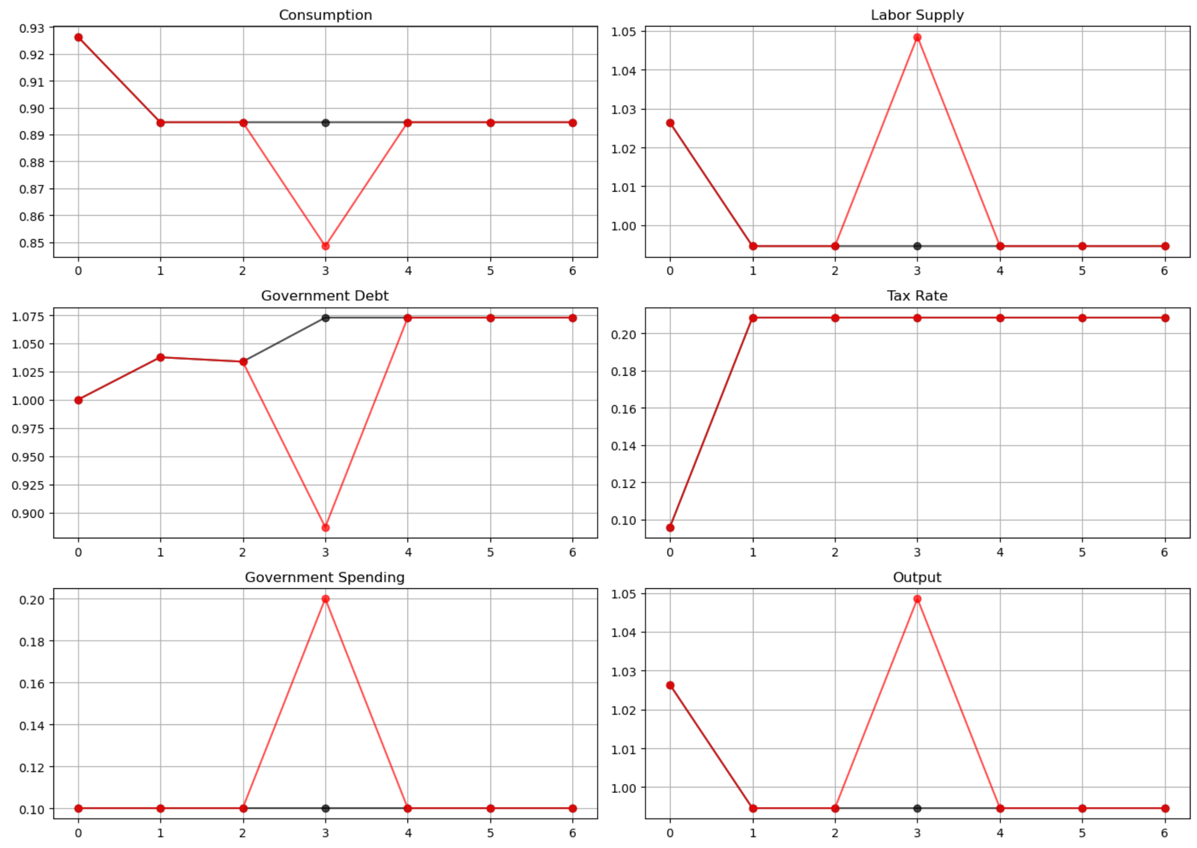
- black when $g_3 = .1$, and
- red when $g_3 = .2$

```
π = np.array([[0, 1, 0,   0,   0,  0],
              [0, 0, 1,   0,   0,  0],
              [0, 0, 0, 0.5, 0.5,  0],
              [0, 0, 0,   0,   0,  1],
              [0, 0, 0,   0,   0,  1],
              [0, 0, 0,   0,   0,  1]])

g = np.array([0.1, 0.1, 0.1, 0.2, 0.1, 0.1])
crra_pref = CRRAutility()

# Solve sequential problem
seq = SequentialLS(crra_pref, π=π, g=g)
sHist_h = np.array([0, 1, 2, 3, 5, 5, 5])
```

(continues on next page)

(continued from previous page)

```python
sHist_l = np.array([0, 1, 2, 4, 5, 5, 5])
sim_seq_h = seq.simulate(1, 0, 7, sHist_h)
sim_seq_l = seq.simulate(1, 0, 7, sHist_l)

fig, axes = plt.subplots(3, 2, figsize=(14, 10))
titles = ['Consumption', 'Labor Supply', 'Government Debt',
          'Tax Rate', 'Government Spending', 'Output']

for ax, title, sim_l, sim_h in zip(axes.flatten(),
                                   titles,
                                   sim_seq_l[:6],
                                   sim_seq_h[:6]):
    ax.set(title=title)
    ax.plot(sim_l, '-ok', sim_h, '-or', alpha=0.7)
    ax.grid()

plt.tight_layout()
plt.show()
```



Tax smoothing

- the tax rate is constant for all $t \geq 1$
  - For $t \geq 1, t \neq 3$, this is a consequence of g_t being the same at all those dates.
  - For $t = 3$, it is a consequence of the special one-period utility function that we have assumed.
  - Under other one-period utility functions, the time $t = 3$ tax rate could be either higher or lower than for dates $t \geq 1, t \neq 3$.

- the tax rate is the same at $t = 3$ for both the high g_t outcome and the low g_t outcome

We have assumed that at $t = 0$, the government owes positive debt b_0.

It sets the time $t = 0$ tax rate partly with an eye to reducing the value $u_{c,0}b_0$ of b_0.

It does this by increasing consumption at time $t = 0$ relative to consumption in later periods.

This has the consequence of *lowering* the time $t = 0$ value of the gross interest rate for risk-free loans between periods t and $t + 1$, which equals

$$R_t = \frac{u_{c,t}}{\beta \mathbb{E}_t[u_{c,t+1}]}$$

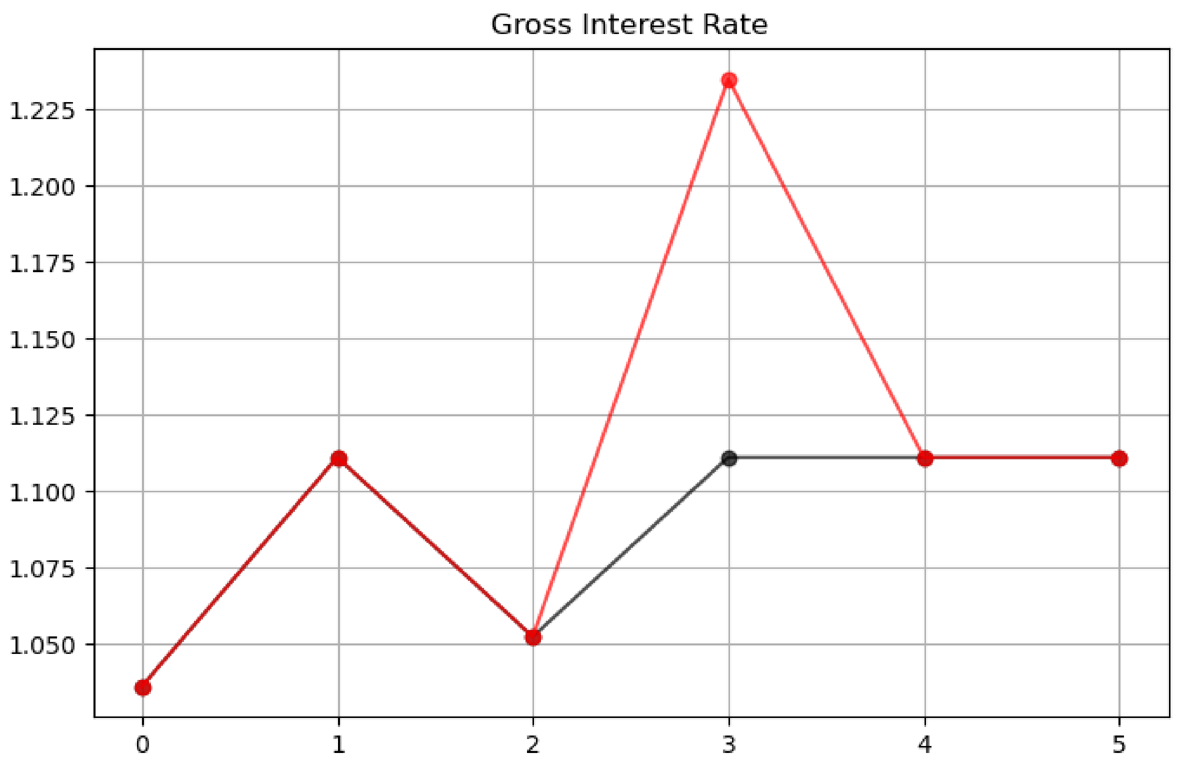
A tax policy that makes time $t = 0$ consumption be higher than time $t = 1$ consumption evidently decreases the risk-free rate one-period interest rate, R_t, at $t = 0$.

Lowering the time $t = 0$ risk-free interest rate makes time $t = 0$ consumption goods cheaper relative to consumption goods at later dates, thereby lowering the value $u_{c,0}b_0$ of initial government debt b_0.

We see this in a figure below that plots the time path for the risk-free interest rate under both realizations of the time $t = 3$ government expenditure shock.

The following plot illustrates how the government lowers the interest rate at time 0 by raising consumption

```
fix, ax = plt.subplots(figsize=(8, 5))
ax.set_title('Gross Interest Rate')
ax.plot(sim_seq_l[-1], '-ok', sim_seq_h[-1], '-or', alpha=0.7)
ax.grid()
plt.show()
```



4.4.2 Government Saving

At time $t = 0$ the government evidently *dissaves* since $b_1 > b_0$.

- This is a consequence of it setting a *lower* tax rate at $t = 0$, implying more consumption at $t = 0$.

At time $t = 1$, the government evidently *saves* since it has set the tax rate sufficiently high to allow it to set $b_2 < b_1$.

- Its motive for doing this is that it anticipates a likely war at $t = 3$.

At time $t = 2$ the government trades state-contingent Arrow securities to hedge against war at $t = 3$.

- It purchases a security that pays off when $g_3 = g_h$.
- It sells a security that pays off when $g_3 = g_l$.
- These purchases are designed in such a way that regardless of whether or not there is a war at $t = 3$, the government will begin period $t = 4$ with the *same* government debt.
- The time $t = 4$ debt level can be serviced with revenues from the constant tax rate set at times $t \geq 1$.

At times $t \geq 4$ the government rolls over its debt, knowing that the tax rate is set at a level that raises enough revenue to pay for government purchases and interest payments on its debt.

4.4.3 Time 0 Manipulation of Interest Rate

We have seen that when $b_0 > 0$, the Ramsey plan sets the time $t = 0$ tax rate partly with an eye toward lowering a risk-free interest rate for one-period loans between times $t = 0$ and $t = 1$.

By lowering this interest rate, the plan makes time $t = 0$ goods cheap relative to consumption goods at later times.

By doing this, it lowers the value of time $t = 0$ debt that it has inherited and must finance.

4.4.4 Time 0 and Time-Inconsistency

In the preceding example, the Ramsey tax rate at time 0 differs from its value at time 1.

To explore what is going on here, let's simplify things by removing the possibility of war at time $t = 3$.

The Ramsey problem then includes no randomness because $g_t = g_l$ for all t.

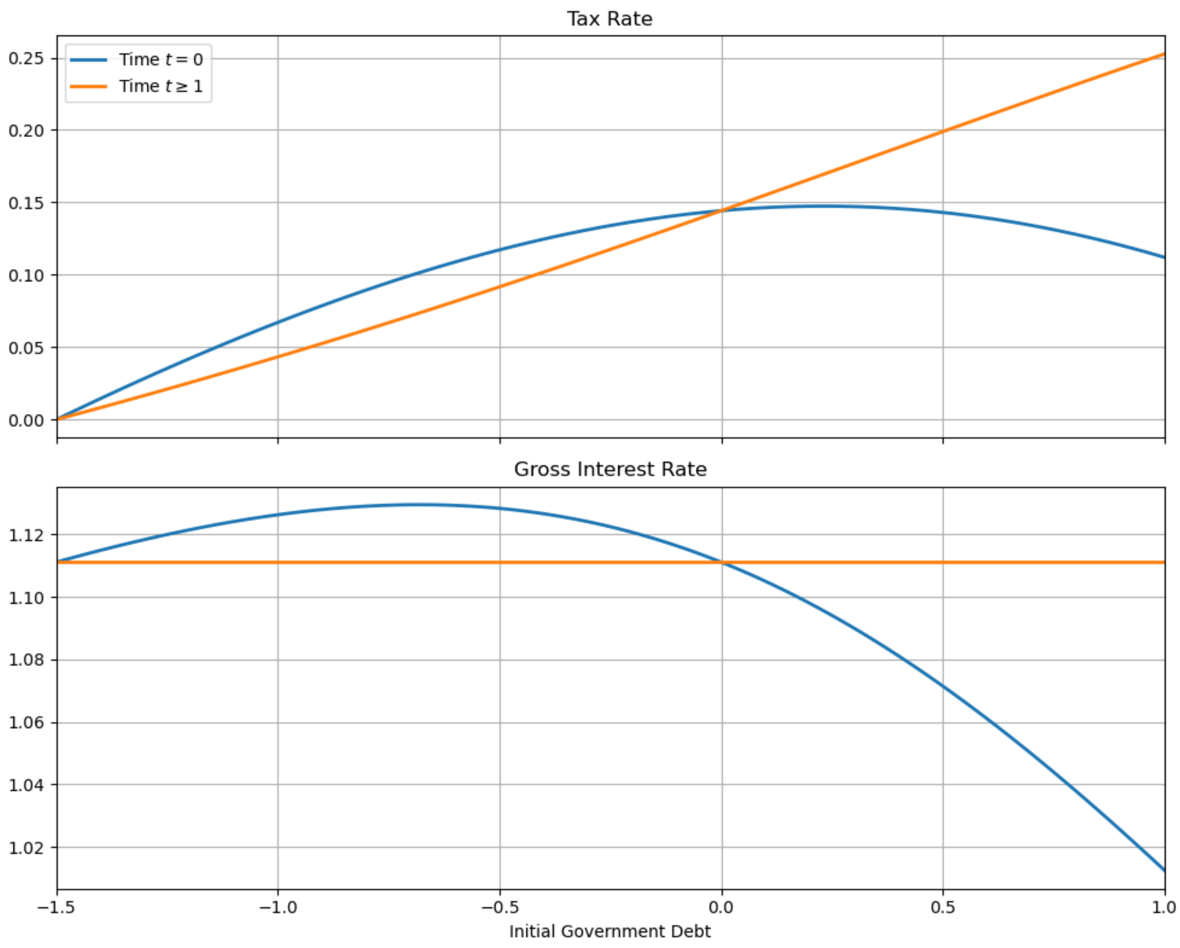
The figure below plots the Ramsey tax rates and gross interest rates at time $t = 0$ and time $t \geq 1$ as functions of the initial government debt (using the sequential allocation solution and a CRRA utility function defined above)

```
tax_seq = SequentialLS(CRRAutility(), g=np.array([0.15]), π=np.ones((1, 1)))

n = 100
tax_policy = np.empty((n, 2))
interest_rate = np.empty((n, 2))
gov_debt = np.linspace(-1.5, 1, n)

for i in range(n):
    tax_policy[i] = tax_seq.simulate(gov_debt[i], 0, 2)[3]
    interest_rate[i] = tax_seq.simulate(gov_debt[i], 0, 3)[-1]

fig, axes = plt.subplots(2, 1, figsize=(10,8), sharex=True)
titles = ['Tax Rate', 'Gross Interest Rate']

for ax, title, plot in zip(axes, titles, [tax_policy, interest_rate]):
```

(continues on next page)

(continued from previous page)

```
    ax.plot(gov_debt, plot[:, 0], gov_debt, plot[:, 1], lw=2)
    ax.set(title=title, xlim=(min(gov_debt), max(gov_debt)))
    ax.grid()

axes[0].legend(('Time $t=0$', 'Time $t \geq 1$'))
axes[1].set_xlabel('Initial Government Debt')

fig.tight_layout()
plt.show()
```



The figure indicates that if the government enters with positive debt, it sets a tax rate at $t = 0$ that is less than all later tax rates.

By setting a lower tax rate at $t = 0$, the government raises consumption, which reduces the *value* $u_{c,0}b_0$ of its initial debt.

It does this by increasing c_0 and thereby lowering $u_{c,0}$.

Conversely, if $b_0 < 0$, the Ramsey planner sets the tax rate at $t = 0$ higher than in subsequent periods.

A side effect of lowering time $t = 0$ consumption is that it lowers the one-period interest rate at time $t = 0$ below that of subsequent periods.

There are only two values of initial government debt at which the tax rate is constant for all $t \geq 0$.

The first is $b_0 = 0$

- Here the government can't use the $t = 0$ tax rate to alter the value of the initial debt.

The second occurs when the government enters with sufficiently large assets that the Ramsey planner can achieve first best and sets $\tau_t = 0$ for all t.

It is only for these two values of initial government debt that the Ramsey plan is time-consistent.

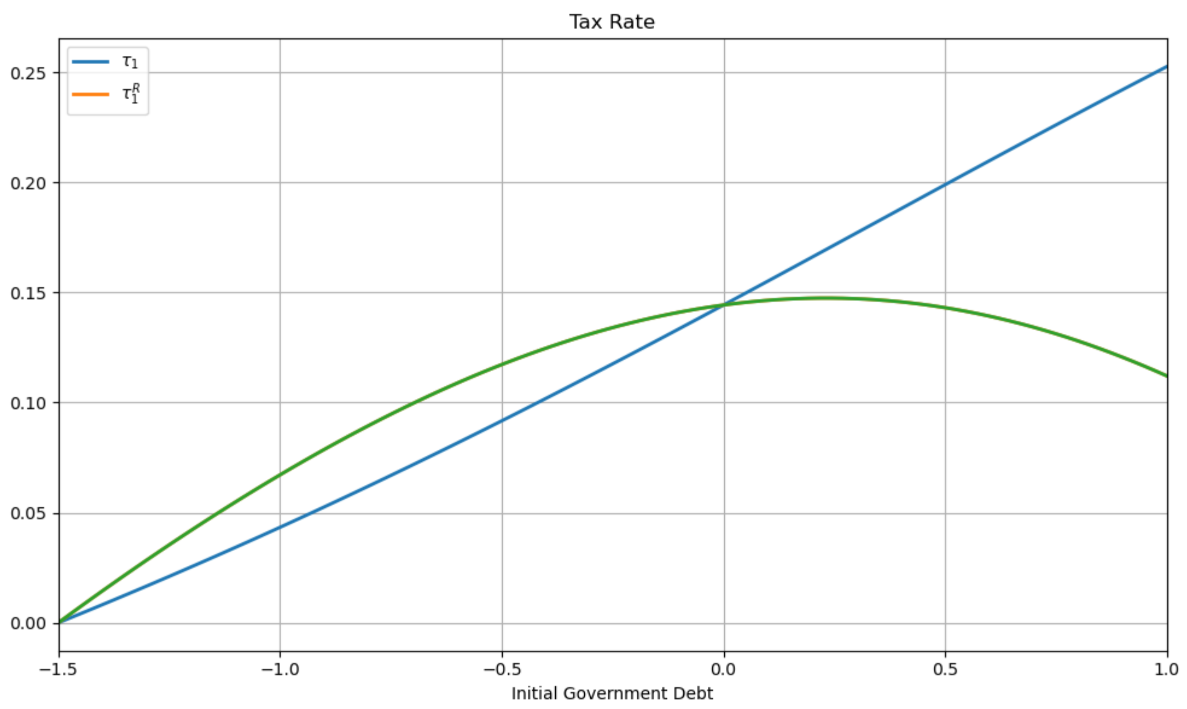
Another way of saying this is that, except for these two values of initial government debt, a continuation of a Ramsey plan is not a Ramsey plan.

To illustrate this, consider a Ramsey planner who starts with an initial government debt b_1 associated with one of the Ramsey plans computed above.

Call τ_1^R the time $t = 0$ tax rate chosen by the Ramsey planner confronting this value for initial government debt government.

The figure below shows both the tax rate at time 1 chosen by our original Ramsey planner and what a new Ramsey planner would choose for its time $t = 0$ tax rate

```
tax_seq = SequentialLS(CRRAutility(), g=np.array([0.15]), π=np.ones((1, 1)))

n = 100
tax_policy = np.empty((n, 2))
τ_reset = np.empty((n, 2))
gov_debt = np.linspace(-1.5, 1, n)

for i in range(n):
    tax_policy[i] = tax_seq.simulate(gov_debt[i], 0, 2)[3]
    τ_reset[i] = tax_seq.simulate(gov_debt[i], 0, 1)[3]

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(gov_debt, tax_policy[:, 1], gov_debt, τ_reset, lw=2)
ax.set(xlabel='Initial Government Debt', title='Tax Rate',
       xlim=(min(gov_debt), max(gov_debt)))
ax.legend((r'$\tau_1$', r'$\tau_1^R$'))
ax.grid()

fig.tight_layout()
plt.show()
```



The tax rates in the figure are equal for only two values of initial government debt.

4.4.5 Tax Smoothing and non-CRRA Preferences

The complete tax smoothing for $t \geq 1$ in the preceding example is a consequence of our having assumed CRRA preferences.

To see what is driving this outcome, we begin by noting that the Ramsey tax rate for $t \geq 1$ is a time-invariant function $\tau(\Phi, g)$ of the Lagrange multiplier on the implementability constraint and government expenditures.

For CRRA preferences, we can exploit the relations $U_{cc}c = -\sigma U_c$ and $U_{nn}n = \gamma U_n$ to derive

$$\frac{(1 + (1 - \sigma)\Phi)U_c}{(1 + (1 - \gamma)\Phi)U_n} = 1$$

from the first-order conditions.

This equation immediately implies that the tax rate is constant.

For other preferences, the tax rate may not be constant.

For example, let the period utility function be

$$u(c, n) = \log(c) + 0.69 \log(1 - n)$$

We will create a new class LogUtility to represent this utility function

```
log_util_data = [
    ('β', float64),
    ('ψ', float64)
]

@jitclass(log_util_data)
```

(continues on next page)

(continued from previous page)

```
class LogUtility:

    def __init__(self,
                 β=0.9,
                 ψ=0.69):

        self.β, self.ψ = β, ψ

    # Utility function
    def U(self, c, l):
        return np.log(c) + self.ψ * np.log(l)

    # Derivatives of utility function
    def Uc(self, c, l):
        return 1 / c

    def Ucc(self, c, l):
        return -c**(-2)

    def Ul(self, c, l):
        return self.ψ / l

    def Ull(self, c, l):
        return -self.ψ / l**2

    def Ucl(self, c, l):
        return 0

    def Ulc(self, c, l):
        return 0
```

Also, suppose that g_t follows a two-state IID process with equal probabilities attached to g_l and g_h.

To compute the tax rate, we will use both the sequential and recursive approaches described above.

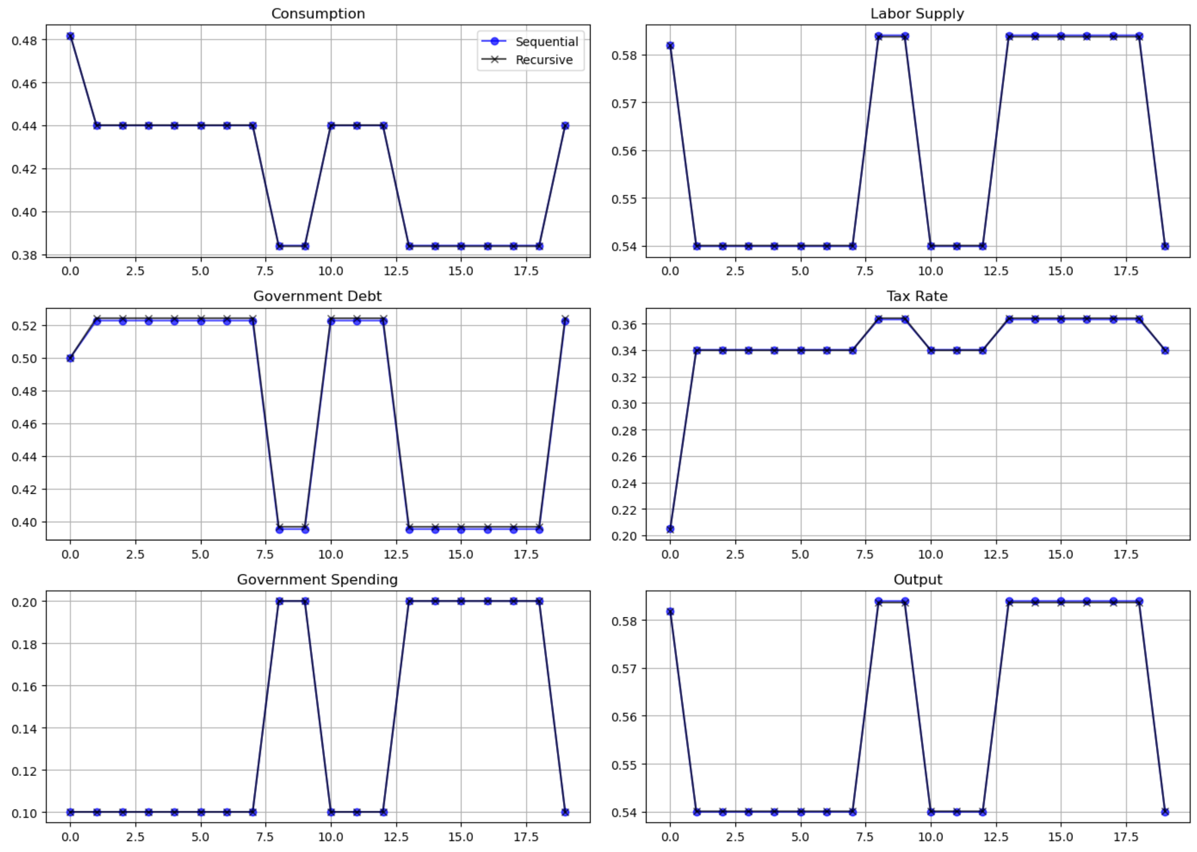
The figure below plots a sample path of the Ramsey tax rate

```
log_example = LogUtility()
# Solve sequential problem
seq_log = SequentialLS(log_example)

# Initialize grid for value function iteration and solve
x_grid = np.linspace(-3., 3., 200)

# Solve recursive problem
rec_log = RecursiveLS(log_example, x_grid)

T_length = 20
sHist = np.array([0, 0, 0, 0, 0,
                  0, 0, 0, 1, 1,
                  0, 0, 0, 1, 1,
                  1, 1, 1, 1, 0])

# Simulate
sim_seq = seq_log.simulate(0.5, 0, T_length, sHist)
sim_rec = rec_log.simulate(0.5, 0, T_length, sHist)
```

(continues on next page)

(continued from previous page)

```
fig, axes = plt.subplots(3, 2, figsize=(14, 10))
titles = ['Consumption', 'Labor Supply', 'Government Debt',
          'Tax Rate', 'Government Spending', 'Output']

for ax, title, sim_s, sim_b in zip(axes.flatten(), titles, sim_seq[:6], sim_rec[:6]):
                                   ax.plot(sim_s, '-ob', sim_b, '-xk', alpha=0.7)
                                   ax.set(title=title)
                                   ax.grid()

axes.flatten()[0].legend(('Sequential', 'Recursive'))
fig.tight_layout()
plt.show()
```



As should be expected, the recursive and sequential solutions produce almost identical allocations.

Unlike outcomes with CRRA preferences, the tax rate is not perfectly smoothed.

Instead, the government raises the tax rate when g_t is high.

4.4.6 Further Comments

A *related lecture* describes an extension of the Lucas-Stokey model by Aiyagari, Marcet, Sargent, and Seppälä (2002) [AMSSeppala02].

In the AMSS economy, only a risk-free bond is traded.

That lecture compares the recursive representation of the Lucas-Stokey model presented in this lecture with one for an AMSS economy.

By comparing these recursive formulations, we shall glean a sense in which the dimension of the state is lower in the Lucas Stokey model.

Accompanying that difference in dimension will be different dynamics of government debt.

CHAPTER

FIVE

OPTIMAL TAXATION WITHOUT STATE-CONTINGENT DEBT

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
!pip install interpolation
```

5.1 Overview

Let's start with following imports:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import root
from interpolation.splines import eval_linear, UCGrid, nodes
from quantecon import optimize, MarkovChain
from numba import njit, prange, float64
from numba.experimental import jitclass
```

In *an earlier lecture*, we described a model of optimal taxation with state-contingent debt due to Robert E. Lucas, Jr., and Nancy Stokey [LS83].

Aiyagari, Marcet, Sargent, and Seppälä [AMSSeppala02] (hereafter, AMSS) studied optimal taxation in a model without state-contingent debt.

In this lecture, we

- describe assumptions and equilibrium concepts
- solve the model
- implement the model numerically

- conduct some policy experiments
- compare outcomes with those in a corresponding complete-markets model

We begin with an introduction to the model.

5.2 Competitive Equilibrium with Distorting Taxes

Many but not all features of the economy are identical to those of *the Lucas-Stokey economy*.

Let's start with things that are identical.

For $t \geq 0$, a history of the state is represented by $s^t = [s_t, s_{t-1}, \dots, s_0]$.

Government purchases $g(s)$ are an exact time-invariant function of s.

Let $c_t(s^t)$, $\ell_t(s^t)$, and $n_t(s^t)$ denote consumption, leisure, and labor supply, respectively, at history s^t at time t.

Each period a representative household is endowed with one unit of time that can be divided between leisure ℓ_t and labor n_t:

$$n_t(s^t) + \ell_t(s^t) = 1 \tag{5.1}$$

Output equals $n_t(s^t)$ and can be divided between consumption $c_t(s^t)$ and $g(s_t)$

$$c_t(s^t) + g(s_t) = n_t(s^t) \tag{5.2}$$

Output is not storable.

The technology pins down a pre-tax wage rate to unity for all t, s^t.

A representative household's preferences over $\{c_t(s^t), \ell_t(s^t)\}_{t=0}^{\infty}$ are ordered by

$$\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) u[c_t(s^t), \ell_t(s^t)] \tag{5.3}$$

where

- $\pi_t(s^t)$ is a joint probability distribution over the sequence s^t, and
- the utility function u is increasing, strictly concave, and three times continuously differentiable in both arguments.

The government imposes a flat rate tax $\tau_t(s^t)$ on labor income at time t, history s^t.

Lucas and Stokey assumed that there are complete markets in one-period Arrow securities; also see smoothing models.

It is at this point that AMSS [AMSSeppala02] modify the Lucas and Stokey economy.

AMSS allow the government to issue only one-period risk-free debt each period.

Ruling out complete markets in this way is a step in the direction of making total tax collections behave more like that prescribed in Robert Barro (1979) [Bar79] than they do in Lucas and Stokey (1983) [LS83].

5.2.1 Risk-free One-Period Debt Only

In period t and history s^t, let

- $b_{t+1}(s^t)$ be the amount of the time $t+1$ consumption good that at time t, history s^t the government promised to pay
- $R_t(s^t)$ be the gross interest rate on risk-free one-period debt between periods t and $t+1$
- $T_t(s^t)$ be a non-negative lump-sum *transfer* to the representative household[1]

That $b_{t+1}(s^t)$ is the same for all realizations of s_{t+1} captures its *risk-free* character.

The market value at time t of government debt maturing at time $t+1$ equals $b_{t+1}(s^t)$ divided by $R_t(s^t)$.

The government's budget constraint in period t at history s^t is

$$\begin{aligned} b_t(s^{t-1}) &= \tau_t^n(s^t)n_t(s^t) - g(s_t) - T_t(s^t) + \frac{b_{t+1}(s^t)}{R_t(s^t)} \\ &\equiv z_t(s^t) + \frac{b_{t+1}(s^t)}{R_t(s^t)}, \end{aligned} \tag{5.4}$$

where $z_t(s^t)$ is the net-of-interest government surplus.

To rule out Ponzi schemes, we assume that the government is subject to a **natural debt limit** (to be discussed in a forthcoming lecture).

The consumption Euler equation for a representative household able to trade only one-period risk-free debt with one-period gross interest rate $R_t(s^t)$ is

$$\frac{1}{R_t(s^t)} = \sum_{s^{t+1}|s^t} \beta\pi_{t+1}(s^{t+1}|s^t)\frac{u_c(s^{t+1})}{u_c(s^t)}$$

Substituting this expression into the government's budget constraint (5.4) yields:

$$b_t(s^{t-1}) = z_t(s^t) + \beta \sum_{s^{t+1}|s^t} \pi_{t+1}(s^{t+1}|s^t)\frac{u_c(s^{t+1})}{u_c(s^t)}\; b_{t+1}(s^t) \tag{5.5}$$

Components of $z_t(s^t)$ on the right side depend on s^t, but the left side is required to depend only on s^{t-1} .

This is what it means for one-period government debt to be risk-free.

Therefore, the right side of equation (5.5) also has to depend only on s^{t-1}.

This requirement will give rise to **measurability constraints** on the Ramsey allocation to be discussed soon.

If we replace $b_{t+1}(s^t)$ on the right side of equation (5.5) by the right side of next period's budget constraint (associated with a particular realization s_t) we get

$$b_t(s^{t-1}) = z_t(s^t) + \sum_{s^{t+1}|s^t} \beta\pi_{t+1}(s^{t+1}|s^t)\frac{u_c(s^{t+1})}{u_c(s^t)}\;\left[z_{t+1}(s^{t+1}) + \frac{b_{t+2}(s^{t+1})}{R_{t+1}(s^{t+1})}\right]$$

After making similar repeated substitutions for all future occurrences of government indebtedness, and by invoking a natural debt limit, we arrive at:

$$b_t(s^{t-1}) = \sum_{j=0}^{\infty}\sum_{s^{t+j}|s^t} \beta^j\pi_{t+j}(s^{t+j}|s^t)\frac{u_c(s^{t+j})}{u_c(s^t)}\; z_{t+j}(s^{t+j}) \tag{5.6}$$

[1] In an allocation that solves the Ramsey problem and that levies distorting taxes on labor, why would the government ever want to hand revenues back to the private sector? It would not in an economy with state-contingent debt, since any such allocation could be improved by lowering distortionary taxes rather than handing out lump-sum transfers. But, without state-contingent debt there can be circumstances when a government would like to make lump-sum transfers to the private sector.

Notice how the conditioning sets in equation (5.6) differ: they are s^{t-1} on the left side and s^t on the right side.

Now let's

- substitute the resource constraint into the net-of-interest government surplus, and
- use the household's first-order condition $1 - \tau_t^n(s^t) = u_\ell(s^t)/u_c(s^t)$ to eliminate the labor tax rate

so that we can express the net-of-interest government surplus $z_t(s^t)$ as

$$z_t(s^t) = \left[1 - \frac{u_\ell(s^t)}{u_c(s^t)}\right][c_t(s^t) + g(s_t)] - g(s_t) - T_t(s^t) \,. \tag{5.7}$$

If we substitute appropriate versions of the right side of (5.7) for $z_{t+j}(s^{t+j})$ into equation (5.6), we obtain a sequence of *implementability constraints* on a Ramsey allocation in an AMSS economy.

Expression (5.6) at time $t = 0$ and initial state s^0 was also an *implementability constraint* on a Ramsey allocation in a Lucas-Stokey economy:

$$b_0(s^{-1}) = \mathbb{E}_0 \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^j)}{u_c(s^0)} \, z_j(s^j) \tag{5.8}$$

Indeed, it was the *only* implementability constraint there.

But now we also have a large number of additional implementability constraints

$$b_t(s^{t-1}) = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} \, z_{t+j}(s^{t+j}) \tag{5.9}$$

Equation (5.9) must hold for each s^t for each $t \geq 1$.

5.2.2 Comparison with Lucas-Stokey Economy

The expression on the right side of (5.9) in the Lucas-Stokey (1983) economy would equal the present value of a continuation stream of government net-of-interest surpluses evaluated at what would be competitive equilibrium Arrow-Debreu prices at date t.

In the Lucas-Stokey economy, that present value is measurable with respect to s^t.

In the AMSS economy, the restriction that government debt be risk-free imposes that that same present value must be measurable with respect to s^{t-1}.

In a language used in the literature on incomplete markets models, it can be said that the AMSS model requires that at each (t, s^t) what would be the present value of continuation government net-of-interest surpluses in the Lucas-Stokey model must belong to the **marketable subspace** of the AMSS model.

5.2.3 Ramsey Problem Without State-contingent Debt

After we have substituted the resource constraint into the utility function, we can express the Ramsey problem as being to choose an allocation that solves

$$\max_{\{c_t(s^t), b_{t+1}(s^t)\}} \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t u\left(c_t(s^t), 1 - c_t(s^t) - g(s_t)\right)$$

where the maximization is subject to

$$\mathbb{E}_0 \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^j)}{u_c(s^0)} \, z_j(s^j) \geq b_0(s^{-1}) \tag{5.10}$$

and

$$\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} \; z_{t+j}(s^{t+j}) = b_t(s^{t-1}) \quad \forall \, t, s^t \tag{5.11}$$

given $b_0(s^{-1})$.

Lagrangian Formulation

Let $\gamma_0(s^0)$ be a non-negative Lagrange multiplier on constraint (5.10).

As in the Lucas-Stokey economy, this multiplier is strictly positive when the government must resort to distortionary taxation; otherwise it equals zero.

A consequence of the assumption that there are no markets in state-contingent securities and that a market exists only in a risk-free security is that we have to attach a stochastic process $\{\gamma_t(s^t)\}_{t=1}^{\infty}$ of Lagrange multipliers to the implementability constraints (5.11).

Depending on how the constraints bind, these multipliers can be positive or negative:

$$\begin{aligned} \gamma_t(s^t) \; &\geq \; (\leq) \; 0 \quad \text{if the constraint binds in the following direction} \\ \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} \; z_{t+j}(s^{t+j}) \; &\geq \; (\leq) \; b_t(s^{t-1}) \end{aligned}$$

A negative multiplier $\gamma_t(s^t) < 0$ means that if we could relax constraint (5.11), we would like to *increase* the beginning-of-period indebtedness for that particular realization of history s^t.

That would let us reduce the beginning-of-period indebtedness for some other history[2].

These features flow from the fact that the government cannot use state-contingent debt and therefore cannot allocate its indebtedness efficiently across future states.

5.2.4 Some Calculations

It is helpful to apply two transformations to the Lagrangian.

Multiply constraint (5.10) by $u_c(s^0)$ and the constraints (5.11) by $\beta^t u_c(s^t)$.

Then a Lagrangian for the Ramsey problem can be represented as

$$\begin{aligned} J &= \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \Big\{ u\left(c_t(s^t), 1 - c_t(s^t) - g(s_t)\right) \\ &\qquad + \gamma_t(s^t) \Big[\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j u_c(s^{t+j}) \, z_{t+j}(s^{t+j}) - u_c(s^t) \, b_t(s^{t-1}) \Big\} \\ &= \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \Big\{ u\left(c_t(s^t), 1 - c_t(s^t) - g(s_t)\right) \\ &\qquad + \Psi_t(s^t) \, u_c(s^t) \, z_t(s^t) - \gamma_t(s^t) \, u_c(s^t) \, b_t(s^{t-1}) \Big\} \end{aligned} \tag{5.12}$$

where

$$\Psi_t(s^t) = \Psi_{t-1}(s^{t-1}) + \gamma_t(s^t) \quad \text{and} \quad \Psi_{-1}(s^{-1}) = 0 \tag{5.13}$$

[2] From the first-order conditions for the Ramsey problem, there exists another realization $\tilde{s}^t$ with the same history up until the previous period, i.e., $\tilde{s}^{t-1} = s^{t-1}$, but where the multiplier on constraint (5.11) takes a positive value, so $\gamma_t(\tilde{s}^t) > 0$.

In (5.12), the second equality uses the law of iterated expectations and Abel's summation formula (also called *summation by parts*, see this page).

First-order conditions with respect to $c_t(s^t)$ can be expressed as

$$
\begin{aligned}
u_c(s^t) - u_\ell(s^t) + \Psi_t(s^t)\left\{\left[u_{cc}(s^t) - u_{c\ell}(s^t)\right] z_t(s^t) + u_c(s^t)\, z_c(s^t)\right\} \\
- \gamma_t(s^t)\left[u_{cc}(s^t) - u_{c\ell}(s^t)\right] b_t(s^{t-1}) = 0
\end{aligned}
\tag{5.14}
$$

and with respect to $b_t(s^t)$ as

$$
\mathbb{E}_t\left[\gamma_{t+1}(s^{t+1})\, u_c(s^{t+1})\right] = 0 \tag{5.15}
$$

If we substitute $z_t(s^t)$ from (5.7) and its derivative $z_c(s^t)$ into the first-order condition (5.14), we find two differences from the corresponding condition for the optimal allocation in a Lucas-Stokey economy with state-contingent government debt.

1. The term involving $b_t(s^{t-1})$ in the first-order condition (5.14) does not appear in the corresponding expression for the Lucas-Stokey economy.
   - This term reflects the constraint that beginning-of-period government indebtedness must be the same across all realizations of next period's state, a constraint that would not be present if government debt could be state-contingent.
2. The Lagrange multiplier $\Psi_t(s^t)$ in the first-order condition (5.14) may change over time in response to realizations of the state, while the multiplier Φ in the Lucas-Stokey economy is time-invariant.

We need some code from *an earlier lecture* on optimal taxation with state-contingent debt sequential allocation implementation:

```python
class SequentialLS:

    '''
    Class that takes a preference object, state transition matrix,
    and state contingent government expenditure plan as inputs, and
    solves the sequential allocation problem described above.
    It returns optimal allocations about consumption and labor supply,
    as well as the multiplier on the implementability constraint Φ.
    '''

    def __init__(self,
                 pref,
                 π=np.full((2, 2), 0.5),
                 g=np.array([0.1, 0.2])):

        # Initialize from pref object attributes
        self.β, self.π, self.g = pref.β, π, g
        self.mc = MarkovChain(self.π)
        self.S = len(π)  # Number of states
        self.pref = pref

        # Find the first best allocation
        self.find_first_best()

    def FOC_first_best(self, c, g):
        '''
        First order conditions that characterize
        the first best allocation.
        '''
```

(continues on next page)

(continued from previous page)

```
        pref = self.pref
        Uc, Ul = pref.Uc, pref.Ul

        n = c + g
        l = 1 - n

        return Uc(c, l) - Ul(c, l)

    def find_first_best(self):
        '''
        Find the first best allocation
        '''
        S, g = self.S, self.g

        res = root(self.FOC_first_best, np.full(S, 0.5), args=(g,))

        if (res.fun > 1e-10).any():
            raise Exception('Could not find first best')

        self.cFB = res.x
        self.nFB = self.cFB + g

    def FOC_time1(self, c, Φ, g):
        '''
        First order conditions that characterize
        optimal time 1 allocation problems.
        '''

        pref = self.pref
        Uc, Ucc, Ul, Ull, Ulc = pref.Uc, pref.Ucc, pref.Ul, pref.Ull, pref.Ulc

        n = c + g
        l = 1 - n

        LHS = (1 + Φ) * Uc(c, l) + Φ * (c * Ucc(c, l) - n * Ulc(c, l))
        RHS = (1 + Φ) * Ul(c, l) + Φ * (c * Ulc(c, l) - n * Ull(c, l))

        diff = LHS - RHS

        return diff

    def time1_allocation(self, Φ):
        '''
        Computes optimal allocation for time t >= 1 for a given Φ
        '''
        pref = self.pref
        S, g = self.S, self.g

        # use the first best allocation as intial guess
        res = root(self.FOC_time1, self.cFB, args=(Φ, g))

        if (res.fun > 1e-10).any():
            raise Exception('Could not find LS allocation.')

        c = res.x
```

(continues on next page)

(continued from previous page)

```
        n = c + g
        l = 1 - n

        # Compute x
        I = pref.Uc(c, n) * c - pref.Ul(c, l) * n
        x = np.linalg.solve(np.eye(S) - self.β * self.π, I)

        return c, n, x

    def FOC_time0(self, c0, Φ, g0, b0):
        '''
        First order conditions that characterize
        time 0 allocation problem.
        '''

        pref = self.pref
        Ucc, Ulc = pref.Ucc, pref.Ulc

        n0 = c0 + g0
        l0 = 1 - n0

        diff = self.FOC_time1(c0, Φ, g0)
        diff -= Φ * (Ucc(c0, l0) - Ulc(c0, l0)) * b0

        return diff

    def implementability(self, Φ, b0, s0, cn0_arr):
        '''
        Compute the differences between the RHS and LHS
        of the implementability constraint given Φ,
        initial debt, and initial state.
        '''

        pref, π, g, β = self.pref, self.π, self.g, self.β
        Uc, Ul = pref.Uc, pref.Ul
        g0 = self.g[s0]

        c, n, x = self.time1_allocation(Φ)

        res = root(self.FOC_time0, cn0_arr[0], args=(Φ, g0, b0))
        c0 = res.x
        n0 = c0 + g0
        l0 = 1 - n0

        cn0_arr[:] = c0.item(), n0.item()

        LHS = Uc(c0, l0) * b0
        RHS = Uc(c0, l0) * c0 - Ul(c0, l0) * n0 + β * π[s0] @ x

        return RHS - LHS

    def time0_allocation(self, b0, s0):
        '''
        Finds the optimal time 0 allocation given
        initial government debt b0 and state s0
        '''
```

(continues on next page)

(continued from previous page)

```
        # use the first best allocation as initial guess
        cn0_arr = np.array([self.cFB[s0], self.nFB[s0]])

        res = root(self.implementability, 0., args=(b0, s0, cn0_arr))

        if (res.fun > 1e-10).any():
            raise Exception('Could not find time 0 LS allocation.')

        Φ = res.x[0]
        c0, n0 = cn0_arr

        return Φ, c0, n0

    def τ(self, c, n):
        '''
        Computes τ given c, n
        '''
        pref = self.pref
        Uc, Ul = pref.Uc, pref.Ul

        return 1 - Ul(c, 1-n) / Uc(c, 1-n)

    def simulate(self, b0, s0, T, sHist=None):
        '''
        Simulates planners policies for T periods
        '''
        pref, π, β = self.pref, self.π, self.β
        Uc = pref.Uc

        if sHist is None:
            sHist = self.mc.simulate(T, s0)

        cHist, nHist, Bhist, τHist, ΦHist = np.empty((5, T))
        RHist = np.empty(T-1)

        # Time 0
        Φ, cHist[0], nHist[0] = self.time0_allocation(b0, s0)
        τHist[0] = self.τ(cHist[0], nHist[0])
        Bhist[0] = b0
        ΦHist[0] = Φ

        # Time 1 onward
        for t in range(1, T):
            c, n, x = self.time1_allocation(Φ)
            τ = self.τ(c, n)
            u_c = Uc(c, 1-n)
            s = sHist[t]
            Eu_c = π[sHist[t-1]] @ u_c
            cHist[t], nHist[t], Bhist[t], τHist[t] = c[s], n[s], x[s] / u_c[s], τ[s]
            RHist[t-1] = Uc(cHist[t-1], 1-nHist[t-1]) / (β * Eu_c)
            ΦHist[t] = Φ

        gHist = self.g[sHist]
        yHist = nHist

        return [cHist, nHist, Bhist, τHist, gHist, yHist, sHist, ΦHist, RHist]
```

To analyze the AMSS model, we find it useful to adopt a recursive formulation using techniques like those in our lectures on *dynamic Stackelberg models* and *optimal taxation with state-contingent debt*.

5.3 Recursive Version of AMSS Model

We now describe a recursive formulation of the AMSS economy.

We have noted that from the point of view of the Ramsey planner, the restriction to one-period risk-free securities

- leaves intact the single implementability constraint on allocations (5.8) from the Lucas-Stokey economy, but
- adds measurability constraints (5.6) on functions of tails of allocations at each time and history

We now explore how these constraints alter Bellman equations for a time 0 Ramsey planner and for time $t \geq 1$, history s^t continuation Ramsey planners.

5.3.1 Recasting State Variables

In the AMSS setting, the government faces a sequence of budget constraints

$$\tau_t(s^t)n_t(s^t) + T_t(s^t) + b_{t+1}(s^t)/R_t(s^t) = g_t + b_t(s^{t-1})$$

where $R_t(s^t)$ is the gross risk-free rate of interest between t and $t+1$ at history s^t and $T_t(s^t)$ are non-negative transfers.

Throughout this lecture, we shall set transfers to zero (for some issues about the limiting behavior of debt, this is possibly an important difference from AMSS [AMSSeppala02], who restricted transfers to be non-negative).

In this case, the household faces a sequence of budget constraints

$$b_t(s^{t-1}) + (1-\tau_t(s^t))n_t(s^t) = c_t(s^t) + b_{t+1}(s^t)/R_t(s^t) \tag{5.16}$$

The household's first-order conditions are $u_{c,t} = \beta R_t \mathbb{E}_t u_{c,t+1}$ and $(1-\tau_t)u_{c,t} = u_{l,t}$.

Using these to eliminate R_t and τ_t from budget constraint (5.16) gives

$$b_t(s^{t-1}) + \frac{u_{l,t}(s^t)}{u_{c,t}(s^t)}n_t(s^t) = c_t(s^t) + \frac{\beta(\mathbb{E}_t u_{c,t+1})b_{t+1}(s^t)}{u_{c,t}(s^t)} \tag{5.17}$$

or

$$u_{c,t}(s^t)b_t(s^{t-1}) + u_{l,t}(s^t)n_t(s^t) = u_{c,t}(s^t)c_t(s^t) + \beta(\mathbb{E}_t u_{c,t+1})b_{t+1}(s^t) \tag{5.18}$$

Now define

$$x_t \equiv \beta b_{t+1}(s^t)\mathbb{E}_t u_{c,t+1} = u_{c,t}(s^t)\frac{b_{t+1}(s^t)}{R_t(s^t)} \tag{5.19}$$

and represent the household's budget constraint at time t, history s^t as

$$\frac{u_{c,t}x_{t-1}}{\beta\mathbb{E}_{t-1}u_{c,t}} = u_{c,t}c_t - u_{l,t}n_t + x_t \tag{5.20}$$

for $t \geq 1$.

5.3.2 Measurability Constraints

Write equation (5.18) as

$$b_t(s^{t-1}) = c_t(s^t) - \frac{u_{l,t}(s^t)}{u_{c,t}(s^t)} n_t(s^t) + \frac{\beta(\mathbb{E}_t u_{c,t+1}) b_{t+1}(s^t)}{u_{c,t}} \tag{5.21}$$

The right side of equation (5.21) expresses the time t value of government debt in terms of a linear combination of terms whose individual components are measurable with respect to s^t.

The sum of terms on the right side of equation (5.21) must equal $b_t(s^{t-1})$.

That implies that it has to be *measurable* with respect to s^{t-1}.

Equations (5.21) are the *measurability constraints* that the AMSS model adds to the single time 0 implementation constraint imposed in the Lucas and Stokey model.

5.3.3 Two Bellman Equations

Let $\Pi(s|s_-)$ be a Markov transition matrix whose entries tell probabilities of moving from state s_- to state s in one period.

Let

- $V(x_-, s_-)$ be the continuation value of a continuation Ramsey plan at $x_{t-1} = x_-, s_{t-1} = s_-$ for $t \geq 1$
- $W(b, s)$ be the value of the Ramsey plan at time 0 at $b_0 = b$ and $s_0 = s$

We distinguish between two types of planners:

For $t \geq 1$, the value function for a **continuation Ramsey planner** satisfies the Bellman equation

$$V(x_-, s_-) = \max_{\{n(s), x(s)\}} \sum_s \Pi(s|s_-) \left[u(n(s) - g(s), 1 - n(s)) + \beta V(x(s), s) \right] \tag{5.22}$$

subject to the following collection of implementability constraints, one for each $s \in S$:

$$\frac{u_c(s) x_-}{\beta \sum_{\tilde{s}} \Pi(\tilde{s}|s_-) u_c(\tilde{s})} = u_c(s)(n(s) - g(s)) - u_l(s) n(s) + x(s) \tag{5.23}$$

A continuation Ramsey planner at $t \geq 1$ takes $(x_{t-1}, s_{t-1}) = (x_-, s_-)$ as given and before s is realized chooses $(n_t(s_t), x_t(s_t)) = (n(s), x(s))$ for $s \in S$.

The **Ramsey planner** takes (b_0, s_0) as given and chooses (n_0, x_0).

The value function $W(b_0, s_0)$ for the time $t = 0$ Ramsey planner satisfies the Bellman equation

$$W(b_0, s_0) = \max_{n_0, x_0} u(n_0 - g_0, 1 - n_0) + \beta V(x_0, s_0) \tag{5.24}$$

where maximization is subject to

$$u_{c,0} b_0 = u_{c,0}(n_0 - g_0) - u_{l,0} n_0 + x_0 \tag{5.25}$$

5.3.4 Martingale Supercedes State-Variable Degeneracy

Let $\mu(s|s\_)\Pi(s|s\_)$ be a Lagrange multiplier on the constraint (5.23) for state s.

After forming an appropriate Lagrangian, we find that the continuation Ramsey planner's first-order condition with respect to $x(s)$ is

$$\beta V_x(x(s), s) = \mu(s|s\_) \tag{5.26}$$

Applying an envelope theorem to Bellman equation (5.22) gives

$$V_x(x\_, s\_) = \sum_s \Pi(s|s\_)\mu(s|s\_)\frac{u_c(s)}{\beta \sum_{\tilde{s}} \Pi(\tilde{s}|s\_)u_c(\tilde{s})} \tag{5.27}$$

Equations (5.26) and (5.27) imply that

$$V_x(x\_, s\_) = \sum_s \left(\Pi(s|s\_)\frac{u_c(s)}{\sum_{\tilde{s}} \Pi(\tilde{s}|s\_)u_c(\tilde{s})}\right) V_x(x, s) \tag{5.28}$$

Equation (5.28) states that $V_x(x, s)$ is a *risk-adjusted martingale*.

Saying that $V_x(x, s)$ is a risk-adjusted martingale means that $V_x(x, s)$ is a martingale with respect to the probability distribution over s^t sequences that are generated by the *twisted* transition probability matrix:

$$\check{\Pi}(s|s\_) \equiv \Pi(s|s\_)\frac{u_c(s)}{\sum_{\tilde{s}} \Pi(\tilde{s}|s\_)u_c(\tilde{s})}$$

Exercise 5.3.1

Please verify that $\check{\Pi}(s|s\_)$ is a valid Markov transition density, i.e., that its elements are all non-negative and that for each $s\_$, the sum over s equals unity.

5.3.5 Absence of State Variable Degeneracy

Along a Ramsey plan, the state variable $x_t = x_t(s^t, b_0)$ becomes a function of the history s^t and initial government debt b_0.

In *Lucas-Stokey model*, we found that

- a counterpart to $V_x(x, s)$ is time-invariant and equal to the Lagrange multiplier on the Lucas-Stokey implementability constraint
- time invariance of $V_x(x, s)$ is the source of a key feature of the Lucas-Stokey model, namely, **state variable degeneracy** in which x_t is an exact time-invariant function of s_t.

That $V_x(x, s)$ varies over time according to a twisted martingale means that there is no state-variable degeneracy in the AMSS model.

In the AMSS model, both x and s are needed to describe the state.

This property of the AMSS model transmits a twisted martingale component to consumption, employment, and the tax rate.

5.3.6 Digression on Non-negative Transfers

Throughout this lecture, we have imposed that transfers $T_t = 0$.

AMSS [AMSSeppala02] instead imposed a nonnegativity constraint $T_t \geq 0$ on transfers.

They also considered a special case of quasi-linear preferences, $u(c, l) = c + H(l)$.

In this case, $V_x(x, s) \leq 0$ is a non-positive martingale.

By the *martingale convergence theorem* $V_x(x, s)$ converges almost surely.

Furthermore, when the Markov chain $\Pi(s|s\_)$ and the government expenditure function $g(s)$ are such that g_t is perpetually random, $V_x(x, s)$ almost surely converges to zero.

For quasi-linear preferences, the first-order condition for maximizing (5.22) subject to (5.23) with respect to $n(s)$ becomes

$$(1 - \mu(s|s\_))(1 - u_l(s)) + \mu(s|s\_)n(s)u_{ll}(s) = 0$$

When $\mu(s|s\_) = \beta V_x(x(s), x)$ converges to zero, in the limit $u_l(s) = 1 = u_c(s)$, so that $\tau(x(s), s) = 0$.

Thus, in the limit, if g_t is perpetually random, the government accumulates sufficient assets to finance all expenditures from earnings on those assets, returning any excess revenues to the household as non-negative lump-sum transfers.

5.3.7 Code

The recursive formulation is implemented as follows

```
class AMSS:
    # WARNING: THE CODE IS EXTREMELY SENSITIVE TO CHOCIES OF PARAMETERS.
    # DO NOT CHANGE THE PARAMETERS AND EXPECT IT TO WORK

    def __init__(self, pref, β, Π, g, x_grid, bounds_v):
        self.β, self.Π, self.g = β, Π, g
        self.x_grid = x_grid
        self.n = x_grid[0][2]
        self.S = len(Π)
        self.bounds = bounds_v
        self.pref = pref

        self.T_v, self.T_w = bellman_operator_factory(Π, β, x_grid, g,
                                                      bounds_v)

        self.V_solved = False
        self.W_solved = False

    def compute_V(self, V, σ_v_star, tol_vfi, maxitr, print_itr):

        T_v = self.T_v

        self.success = False

        V_new = np.zeros_like(V)

        Δ = 1.0
        for itr in range(maxitr):
            T_v(V, V_new, σ_v_star, self.pref)
```

(continues on next page)

(continued from previous page)

```
            Δ = np.max(np.abs(V_new - V))

            if Δ < tol_vfi:
                self.V_solved = True
                print('Successfully completed VFI after %i iterations'
                      % (itr+1))
                break

            if (itr + 1) % print_itr == 0:
                print('Error at iteration %i : ' % (itr + 1), Δ)

            V[:] = V_new[:]

        self.V = V
        self.σ_v_star = σ_v_star

        return V, σ_v_star

    def compute_W(self, b_0, W, σ_w_star):
        T_w = self.T_w
        V = self.V

        T_w(W, σ_w_star, V, b_0, self.pref)

        self.W = W
        self.σ_w_star = σ_w_star
        self.W_solved = True
        print('Succesfully solved the time 0 problem.')

        return W, σ_w_star

    def solve(self, V, σ_v_star,  b_0, W, σ_w_star, tol_vfi=1e-7,
              maxitr=1000, print_itr=10):
        print("===============")
        print("Solve time 1 problem")
        print("===============")
        self.compute_V(V, σ_v_star, tol_vfi, maxitr, print_itr)
        print("===============")
        print("Solve time 0 problem")
        print("===============")
        self.compute_W(b_0, W, σ_w_star)

    def simulate(self, s_hist, b_0):
        if not (self.V_solved and self.W_solved):
            msg = "V and W need to be successfully computed before simulation."
            raise ValueError(msg)

        pref = self.pref
        x_grid, g, β, S = self.x_grid, self.g, self.β, self.S
        σ_v_star, σ_w_star = self.σ_v_star, self.σ_w_star

        T = len(s_hist)
        s_0 = s_hist[0]

        # Pre-allocate
        n_hist = np.zeros(T)
```

(continues on next page)

(continued from previous page)

```
        x_hist = np.zeros(T)
        c_hist = np.zeros(T)
        τ_hist = np.zeros(T)
        b_hist = np.zeros(T)
        g_hist = np.zeros(T)

        # Compute t = 0
        l_0, T_0 = σ_w_star[s_0]
        c_0 = (1 - l_0) - g[s_0]
        x_0 = (-pref.Uc(c_0, l_0) * (c_0 - T_0 - b_0) +
               pref.Ul(c_0, l_0) * (1 - l_0))

        n_hist[0] = (1 - l_0)
        x_hist[0] = x_0
        c_hist[0] = c_0
        τ_hist[0] = 1 - pref.Ul(c_0, l_0) / pref.Uc(c_0, l_0)
        b_hist[0] = b_0
        g_hist[0] = g[s_0]

        # Compute t > 0
        for t in range(T - 1):
            x_ = x_hist[t]
            s_ = s_hist[t]
            l = np.zeros(S)
            T = np.zeros(S)
            for s in range(S):
                x_arr = np.array([x_])
                l[s] = eval_linear(x_grid, σ_v_star[s_, :, s], x_arr)
                T[s] = eval_linear(x_grid, σ_v_star[s_, :, S+s], x_arr)

            c = (1 - l) - g
            u_c = pref.Uc(c, l)
            Eu_c = Π[s_] @ u_c

            x = u_c * x_ / (β * Eu_c) - u_c * (c - T) + pref.Ul(c, l) * (1 - l)

            c_next = c[s_hist[t+1]]
            l_next = l[s_hist[t+1]]

            x_hist[t+1] = x[s_hist[t+1]]
            n_hist[t+1] = 1 - l_next
            c_hist[t+1] = c_next
            τ_hist[t+1] = 1 - pref.Ul(c_next, l_next) / pref.Uc(c_next, l_next)
            b_hist[t+1] = x_ / (β * Eu_c)
            g_hist[t+1] = g[s_hist[t+1]]

        return c_hist, n_hist, b_hist, τ_hist, g_hist, n_hist


def obj_factory(Π, β, x_grid, g):
    S = len(Π)

    @njit
    def obj_V(σ, state, V, pref):
        # Unpack state
        s_, x_ = state
```

(continues on next page)

(continued from previous page)

```
        l = σ[:S]
        T = σ[S:]

        c = (1 - l) - g
        u_c = pref.Uc(c, l)
        Eu_c = Π[s_] @ u_c
        x = u_c * x_ / (β * Eu_c) - u_c * (c - T) + pref.Ul(c, l) * (1 - l)

        V_next = np.zeros(S)

        for s in range(S):
            V_next[s] = eval_linear(x_grid, V[s], np.array([x[s]]))

        out = Π[s_] @ (pref.U(c, l) + β * V_next)

        return out

    @njit
    def obj_W(σ, state, V, pref):
        # Unpack state
        s_, b_0 = state
        l, T = σ

        c = (1 - l) - g[s_]
        x = -pref.Uc(c, l) * (c - T - b_0) + pref.Ul(c, l) * (1 - l)

        V_next = eval_linear(x_grid, V[s_], np.array([x]))

        out = pref.U(c, l) + β * V_next

        return out

    return obj_V, obj_W


def bellman_operator_factory(Π, β, x_grid, g, bounds_v):
    obj_V, obj_W = obj_factory(Π, β, x_grid, g)
    n = x_grid[0][2]
    S = len(Π)
    x_nodes = nodes(x_grid)

    @njit(parallel=True)
    def T_v(V, V_new, σ_star, pref):
        for s_ in prange(S):
            for x_i in prange(n):
                state = (s_, x_nodes[x_i])
                x0 = σ_star[s_, x_i]
                res = optimize.nelder_mead(obj_V, x0, bounds=bounds_v,
                                           args=(state, V, pref))

                if res.success:
                    V_new[s_, x_i] = res.fun
                    σ_star[s_, x_i] = res.x
                else:
                    print("Optimization routine failed.")
```

(continues on next page)

(continued from previous page)

```
    bounds_w = np.array([[-9.0, 1.0], [0., 10.]])

    def T_w(W, σ_star, V, b_0, pref):
        for s_ in prange(S):
            state = (s_, b_0)
            x0 = σ_star[s_]
            res = optimize.nelder_mead(obj_W, x0, bounds=bounds_w,
                                       args=(state, V, pref))

            W[s_] = res.fun
            σ_star[s_] = res.x

    return T_v, T_w
```

5.4 Examples

We now turn to some examples.

5.4.1 Anticipated One-Period War

In our lecture on *optimal taxation with state-contingent debt* we studied how the government manages uncertainty in a simple setting.

As in that lecture, we assume the one-period utility function

$$u(c, n) = \frac{c^{1-\sigma}}{1-\sigma} - \frac{n^{1+\gamma}}{1+\gamma}$$

Note: For convenience in matching our computer code, we have expressed utility as a function of n rather than leisure l.

We first consider a government expenditure process that we studied earlier in a lecture on *optimal taxation with state-contingent debt*.

Government expenditures are known for sure in all periods except one.

- For $t < 3$ or $t > 3$ we assume that $g_t = g_l = 0.1$.
- At $t = 3$ a war occurs with probability 0.5.
  - If there is war, $g_3 = g_h = 0.2$.
  - If there is no war $g_3 = g_l = 0.1$.

A useful trick is to define components of the state vector as the following six (t, g) pairs:

$$(0, g_l), (1, g_l), (2, g_l), (3, g_l), (3, g_h), (t \geq 4, g_l)$$

We think of these 6 states as corresponding to $s = 1, 2, 3, 4, 5, 6$.

The transition matrix is

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

The government expenditure at each state is

$$g = \begin{pmatrix} 0.1 \\ 0.1 \\ 0.1 \\ 0.1 \\ 0.2 \\ 0.1 \end{pmatrix}$$

We assume the same utility parameters as in the *Lucas-Stokey economy*.

This utility function is implemented in the following class.

```python
crra_util_data = [
    ('β', float64),
    ('σ', float64),
    ('γ', float64)
]

@jitclass(crra_util_data)
class CRRAutility:

    def __init__(self,
                 β=0.9,
                 σ=2,
                 γ=2):

        self.β, self.σ, self.γ = β, σ, γ

    # Utility function
    def U(self, c, l):
        # Note: `l` should not be interpreted as labor, it is an auxiliary
        # variable used to conveniently match the code and the equations
        # in the lecture
        σ = self.σ
        if σ == 1.:
            U = np.log(c)
        else:
            U = (c**(1 - σ) - 1) / (1 - σ)
        return U - (1-l) ** (1 + self.γ) / (1 + self.γ)

    # Derivatives of utility function
    def Uc(self, c, l):
        return c ** (-self.σ)

    def Ucc(self, c, l):
        return -self.σ * c ** (-self.σ - 1)

    def Ul(self, c, l):
```

(continues on next page)

(continued from previous page)

```
        return (1-l) ** self.γ

    def Ull(self, c, l):
        return -self.γ * (1-l) ** (self.γ - 1)

    def Ucl(self, c, l):
        return 0

    def Ulc(self, c, l):
        return 0
```

The following figure plots Ramsey plans under complete and incomplete markets for both possible realizations of the state at time $t = 3$.

Ramsey outcomes and policies when the government has access to state-contingent debt are represented by black lines and by red lines when there is only a risk-free bond.

Paths with circles are histories in which there is peace, while those with triangle denote war.

```
# WARNING: DO NOT EXPECT THE CODE TO WORK IF YOU CHANGE PARAMETERS
σ = 2
γ = 2
β = 0.9
Π = np.array([[0, 1, 0,   0,   0,  0],
              [0, 0, 1,   0,   0,  0],
              [0, 0, 0, 0.5, 0.5,  0],
              [0, 0, 0,   0,   0,  1],
              [0, 0, 0,   0,   0,  1],
              [0, 0, 0,   0,   0,  1]])
g = np.array([0.1, 0.1, 0.1, 0.2, 0.1, 0.1])

x_min = -1.5555
x_max = 17.339
x_num = 300

x_grid = UCGrid((x_min, x_max, x_num))

crra_pref = CRRAutility(β=β, σ=σ, γ=γ)

S = len(Π)
bounds_v = np.vstack([np.hstack([np.full(S, -10.), np.zeros(S)]),
                      np.hstack([np.ones(S) - g, np.full(S, 10.)])]).T

amss_model = AMSS(crra_pref, β, Π, g, x_grid, bounds_v)
```

```
# WARNING: DO NOT EXPECT THE CODE TO WORK IF YOU CHANGE PARAMETERS
V = np.zeros((len(Π), x_num))
V[:] = -nodes(x_grid).T ** 2

σ_v_star = np.ones((S, x_num, S * 2))
σ_v_star[:, :, :S] = 0.0

W = np.empty(len(Π))
b_0 = 1.0
σ_w_star = np.ones((S, 2))
σ_w_star[:, 0] = -0.05
```

```
%%time

amss_model.solve(V, σ_v_star, b_0, W, σ_w_star)
```

```
================
Solve time 1 problem
================
```

```
Error at iteration 10 :  1.110064840137854
```

```
Error at iteration 20 :  0.30784885876438395
```

```
Error at iteration 30 :  0.03221851531398379
```

```
Error at iteration 40 :  0.014347598008733087
```

```
Error at iteration 50 :  0.0031219444631354065
```

```
Error at iteration 60 :  0.0010783647355108172
```

```
Error at iteration 70 :  0.0003761255356202753
```

```
Error at iteration 80 :  0.0001318127597098595
```

```
Error at iteration 90 :  4.650031579878089e-05
```

```
Error at iteration 100 :  1.801377708510188e-05
```

```
Error at iteration 110 :  6.175872600877597e-06
```

```
Error at iteration 120 :  2.4450291853383987e-06
```

```
Error at iteration 130 :  1.0836745989450947e-06
```

```
Error at iteration 140 :  5.682877084467464e-07
```

```
Error at iteration 150 :  3.567560966644123e-07
```

```
Error at iteration 160 :  2.5837734796141376e-07
```

```
Error at iteration 170 :  2.047536575844333e-07
Error at iteration 180 :  1.7066849622437985e-07
```

```
Error at iteration 190 :  1.4622035848788073e-07
```

```
Error at iteration 200 :  1.27387780324284e-07
```

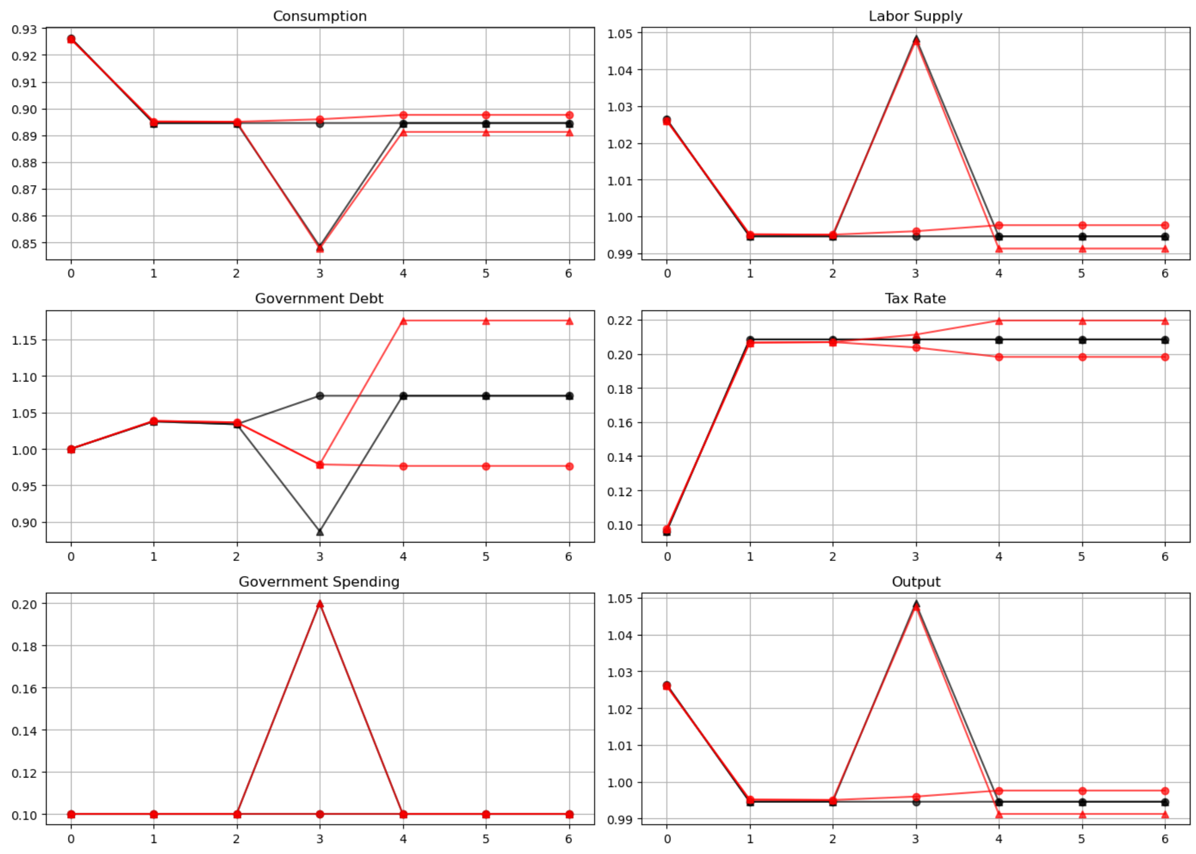
```
Error at iteration 210 :  1.1226231499961159e-07
```

```
Successfully completed VFI after 220 iterations
===============
Solve time 0 problem
===============
```

```
Succesfully solved the time 0 problem.
CPU times: user 2min 5s, sys: 1.82 s, total: 2min 7s
Wall time: 1min 26s
```

```
# Solve the LS model
ls_model = SequentialLS(crra_pref, g=g, π=Π)
```

```
# WARNING: DO NOT EXPECT THE CODE TO WORK IF YOU CHANGE PARAMETERS
s_hist_h = np.array([0, 1, 2, 3, 5, 5, 5])
s_hist_l = np.array([0, 1, 2, 4, 5, 5, 5])

sim_h_amss = amss_model.simulate(s_hist_h, b_0)
sim_l_amss = amss_model.simulate(s_hist_l, b_0)

sim_h_ls = ls_model.simulate(b_0, 0, 7, s_hist_h)
sim_l_ls = ls_model.simulate(b_0, 0, 7, s_hist_l)

fig, axes = plt.subplots(3, 2, figsize=(14, 10))
titles = ['Consumption', 'Labor Supply', 'Government Debt',
          'Tax Rate', 'Government Spending', 'Output']

for ax, title, ls_l, ls_h, amss_l, amss_h in zip(axes.flatten(), titles,
                                                  sim_l_ls, sim_h_ls,
                                                  sim_l_amss, sim_h_amss):
    ax.plot(ls_l, '-ok', ls_h, '-^k', amss_l, '-or', amss_h, '-^r',
            alpha=0.7)
    ax.set(title=title)
    ax.grid()

plt.tight_layout()
plt.show()
```



How a Ramsey planner responds to war depends on the structure of the asset market.

If it is able to trade state-contingent debt, then at time $t = 2$

- the government **purchases** an Arrow security that pays off when $g_3 = g_h$
- the government **sells** an Arrow security that pays off when $g_3 = g_l$
- the Ramsey planner designs these purchases and sales designed so that, regardless of whether or not there is a war at $t = 3$, the government begins period $t = 4$ with the *same* government debt

This pattern facilities smoothing tax rates across states.

The government without state-contingent debt cannot do this.

Instead, it must enter time $t = 3$ with the same level of debt falling due whether there is peace or war at $t = 3$.

The risk-free rate between time 2 and time 3 is unusually **low** because at time 2 consumption at time 3 is expected to be unusually **low**.

A **low** risk-free rate of return on government debt between time 2 and time 3 allows the government to enter period 3 with **lower** government debt than it entered period 2.

To finance a war at time 3 it raises taxes and issues more debt to carry into perpetual peace that begins in period 4.

To service the additional debt burden, it raises taxes in all future periods.

The absence of state-contingent debt leads to an important difference in the optimal tax policy.

When the Ramsey planner has access to state-contingent debt, the optimal tax policy is history independent

- the tax rate is a function of the current level of government spending only, given the Lagrange multiplier on the implementability constraint

Without state-contingent debt, the optimal tax rate is history dependent.

- A war at time $t = 3$ causes a permanent **increase** in the tax rate.
- Peace at time $t = 3$ causes a permanent **reduction** in the tax rate.

Perpetual War Alert

History dependence occurs more dramatically in a case in which the government perpetually faces the prospect of war.

This case was studied in the final example of the lecture on *optimal taxation with state-contingent debt*.

There, each period the government faces a constant probability, 0.5, of war.

In addition, this example features the following preferences

$$u(c, n) = \log(c) + 0.69 \log(1 - n)$$

In accordance, we will re-define our utility function.

```
log_util_data = [
    ('β', float64),
    ('ψ', float64)
]

@jitclass(log_util_data)
class LogUtility:

    def __init__(self,
                 β=0.9,
                 ψ=0.69):

        self.β, self.ψ = β, ψ

    # Utility function
    def U(self, c, l):
        return np.log(c) + self.ψ * np.log(l)

    # Derivatives of utility function
    def Uc(self, c, l):
        return 1 / c

    def Ucc(self, c, l):
        return -c**(-2)

    def Ul(self, c, l):
        return self.ψ / l

    def Ull(self, c, l):
        return -self.ψ / l**2

    def Ucl(self, c, l):
        return 0

    def Ulc(self, c, l):
        return 0
```

With these preferences, Ramsey tax rates will vary even in the Lucas-Stokey model with state-contingent debt.

The figure below plots optimal tax policies for both the economy with state-contingent debt (circles) and the economy with only a risk-free bond (triangles).

```
# WARNING: DO NOT EXPECT THE CODE TO WORK IF YOU CHANGE PARAMETERS
ψ = 0.69
Π = np.full((2, 2), 0.5)
β = 0.9
g = np.array([0.1, 0.2])

x_min = -3.4107
x_max = 3.709
x_num = 300

x_grid = UCGrid((x_min, x_max, x_num))
log_pref = LogUtility(β=β, ψ=ψ)

S = len(Π)
bounds_v = np.vstack([np.zeros(2 * S), np.hstack([1 - g, np.ones(S)]) ]).T

V = np.zeros((len(Π), x_num))
V[:] = -(nodes(x_grid).T + x_max) ** 2 / 14

σ_v_star = 1 - np.full((S, x_num, S * 2), 0.55)

W = np.empty(len(Π))
b_0 = 0.5
σ_w_star = 1 - np.full((S, 2), 0.55)

amss_model = AMSS(log_pref, β, Π, g, x_grid, bounds_v)
```

```
%%time

amss_model.solve(V, σ_v_star, b_0, W, σ_w_star, tol_vfi=3e-5, maxitr=3000,
                 print_itr=100)
```

```
===============
Solve time 1 problem
===============
```

```
Error at iteration 100 :  0.0011569123052908026
```

```
Error at iteration 200 :  0.0005024948171925558
```

```
Error at iteration 300 :  0.0002995649778405607
```

```
Error at iteration 400 :  0.00020753209923363158
```

```
Error at iteration 500 :  0.00015556566848218267
```

```
Error at iteration 600 :  0.0001228034492957164
Error at iteration 700 :  0.00010068689697462219
```

```
Error at iteration 800 :  8.474340939912395e-05
```

```
Error at iteration 900 :  7.290920770763876e-05
```

```
Error at iteration 1000 :  6.375694017535238e-05
```

```
Error at iteration 1100 :  5.642689428775327e-05
```

```
Error at iteration 1200 :  5.045426282634935e-05
```

```
Error at iteration 1300 :  4.561168914030134e-05
```

```
Error at iteration 1400 :  4.150059282892471e-05
```

```
Error at iteration 1500 :  3.799110186264443e-05
```

```
Error at iteration 1600 :  3.5163266918658564e-05
```

```
Error at iteration 1700 :  3.263979350620616e-05
```

```
Error at iteration 1800 :  3.0359381506528393e-05
```

```
Successfully completed VFI after 1818 iterations
===============
Solve time 0 problem
===============
```

```
Succesfully solved the time 0 problem.
CPU times: user 1min 58s, sys: 1.05 s, total: 1min 59s
Wall time: 1min 28s
```

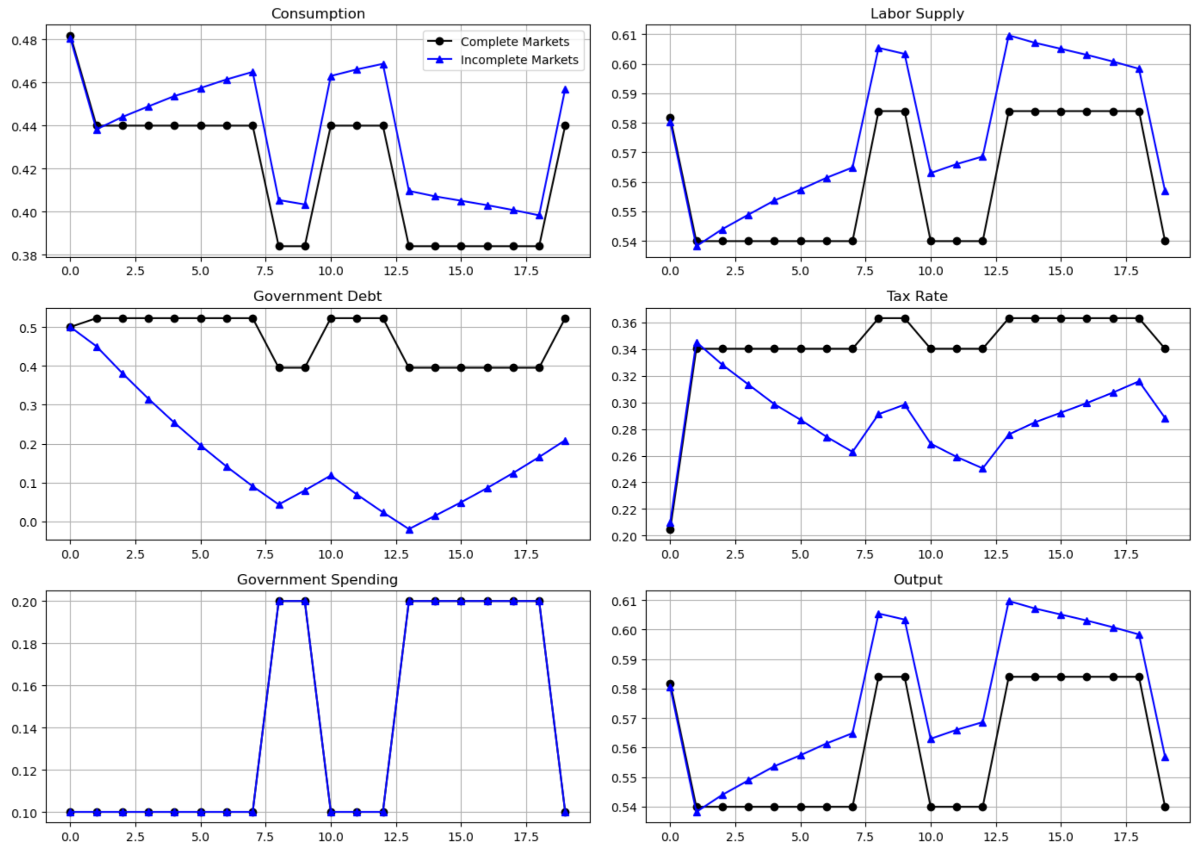
```python
ls_model = SequentialLS(log_pref, g=g, π=Π)  # Solve sequential problem
```

```python
# WARNING: DO NOT EXPECT THE CODE TO WORK IF YOU CHANGE PARAMETERS
s_hist = np.array([0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
                   0, 0, 0, 1, 1, 1, 1, 1, 1, 0])

T = len(s_hist)

sim_amss = amss_model.simulate(s_hist, b_0)
sim_ls = ls_model.simulate(0.5, 0, T, s_hist)

titles = ['Consumption', 'Labor Supply', 'Government Debt',
```

(continues on next page)

(continued from previous page)

```
          'Tax Rate', 'Government Spending', 'Output']

fig, axes = plt.subplots(3, 2, figsize=(14, 10))

for ax, title, ls, amss in zip(axes.flatten(), titles, sim_ls, sim_amss):
    ax.plot(ls, '-ok', amss, '-^b')
    ax.set(title=title)
    ax.grid()

axes[0, 0].legend(('Complete Markets', 'Incomplete Markets'))
plt.tight_layout()
plt.show()
```



When the government experiences a prolonged period of peace, it is able to reduce government debt and set persistently lower tax rates.

However, the government finances a long war by borrowing and raising taxes.

This results in a drift away from policies with state-contingent debt that depends on the history of shocks.

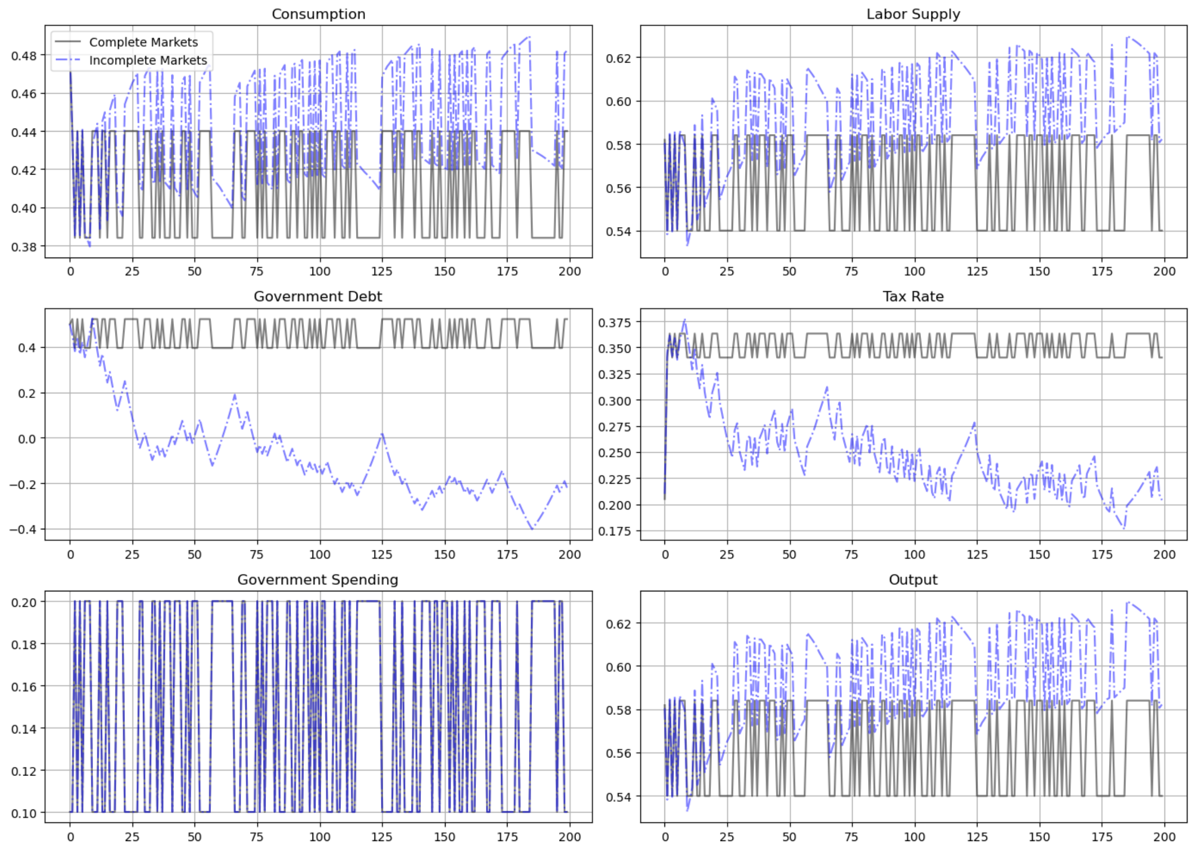
This is even more evident in the following figure that plots the evolution of the two policies over 200 periods.

This outcome reflects the presence of a force for **precautionary saving** that the incomplete markets structure imparts to the Ramsey plan.

In *this subsequent lecture* and *this subsequent lecture*, some ultimate consequences of that force are explored.

```
T = 200
s_0 = 0
mc = MarkovChain(Π)

s_hist_long = mc.simulate(T, init=s_0, random_state=5)
```

```
sim_amss = amss_model.simulate(s_hist_long, b_0)
sim_ls = ls_model.simulate(0.5, 0, T, s_hist_long)

titles = ['Consumption', 'Labor Supply', 'Government Debt',
          'Tax Rate', 'Government Spending', 'Output']


fig, axes = plt.subplots(3, 2, figsize=(14, 10))

for ax, title, ls, amss in zip(axes.flatten(), titles, sim_ls, \
        sim_amss):
    ax.plot(ls, '-k', amss, '-.b', alpha=0.5)
    ax.set(title=title)
    ax.grid()

axes[0, 0].legend(('Complete Markets','Incomplete Markets'))
plt.tight_layout()
plt.show()
```



CHAPTER

SIX

FLUCTUATING INTEREST RATES DELIVER FISCAL INSURANCE

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

6.1 Overview

This lecture extends our investigations of how optimal policies for levying a flat-rate tax on labor income and issuing government debt depend on whether there are complete markets for debt.

A Ramsey allocation and Ramsey policy in the AMSS [AMSSeppala02] model described in *optimal taxation without state-contingent debt* generally differs from a Ramsey allocation and Ramsey policy in the Lucas-Stokey [LS83] model described in *optimal taxation with state-contingent debt*.

This is because the implementability restriction that a competitive equilibrium with a distorting tax imposes on allocations in the Lucas-Stokey model is just one among a set of implementability conditions imposed in the AMSS model.

These additional constraints require that time t components of a Ramsey allocation for the AMSS model be **measurable** with respect to time $t-1$ information.

The measurability constraints imposed by the AMSS model are inherited from the restriction that only one-period risk-free bonds can be traded.

Differences between the Ramsey allocations in the two models indicate that at least some of the **implementability constraints** of the AMSS model of *optimal taxation without state-contingent debt* are violated at the Ramsey allocation of a corresponding [LS83] model with state-contingent debt.

Another way to say this is that differences between the Ramsey allocations of the two models indicate that some of the **measurability constraints** imposed by the AMSS model are violated at the Ramsey allocation of the Lucas-Stokey model.

Nonzero Lagrange multipliers on those constraints make the Ramsey allocation for the AMSS model differ from the Ramsey allocation for the Lucas-Stokey model.

This lecture studies a special AMSS model in which

- The exogenous state variable s_t is governed by a finite-state Markov chain.
- With an arbitrary budget-feasible initial level of government debt, the measurability constraints
  - bind for many periods, but ....
  - eventually, they stop binding evermore, so that ...
  - in the tail of the Ramsey plan, the Lagrange multipliers $\gamma_t(s^t)$ on the AMSS implementability constraints (5.8) are zero.
- After the implementability constraints (5.8) no longer bind in the tail of the AMSS Ramsey plan
  - history dependence of the AMSS state variable x_t vanishes and x_t becomes a time-invariant function of the Markov state s_t.
  - the par value of government debt becomes **constant over time** so that $b_{t+1}(s^t) = \bar{b}$ for $t \geq T$ for a sufficiently large T.
  - $\bar{b} < 0$, so that the tail of the Ramsey plan instructs the government always to make a constant par value of risk-free one-period loans **to** the private sector.
  - the one-period gross interest rate $R_t(s^t)$ on risk-free debt converges to a time-invariant function of the Markov state s_t.
- For a **particular** $b_0 < 0$ (i.e., a positive level of initial government **loans** to the private sector), the measurability constraints **never** bind.
- In this special case
  - the **par value** $b_{t+1}(s_t) = \bar{b}$ of government debt at time t and Markov state s_t is constant across time and states, but ....
  - the **market value** $\frac{\bar{b}}{R_t(s_t)}$ of government debt at time t varies as a time-invariant function of the Markov state s_t.
  - fluctuations in the interest rate make gross earnings on government debt $\frac{\bar{b}}{R_t(s_t)}$ fully insure the gross-of-gross-interest-payments government budget against fluctuations in government expenditures.
  - the state variable x in a recursive representation of a Ramsey plan is a time-invariant function of the Markov state for $t \geq 0$.
- In this special case, the Ramsey allocation in the AMSS model agrees with that in a Lucas-Stokey [LS83] complete markets model in which the same amount of state-contingent debt falls due in all states tomorrow
  - it is a situation in which the Ramsey planner loses nothing from not being able to trade state-contingent debt and being restricted to exchange only risk-free debt debt.
- This outcome emerges only when we initialize government debt at a particular $b_0 < 0$.

In a nutshell, the reason for this striking outcome is that at a particular level of risk-free government **assets**, fluctuations in the one-period risk-free interest rate provide the government with complete insurance against stochastically varying government expenditures.

Let's start with some imports:

```
import matplotlib.pyplot as plt
%matplotlib inline
from scipy.optimize import fsolve, fmin
```

6.2 Forces at Work

The forces driving asymptotic outcomes here are examples of dynamics present in a more general class of incomplete markets models analyzed in [BEGS17] (BEGS).

BEGS provide conditions under which government debt under a Ramsey plan converges to an invariant distribution.

BEGS construct approximations to that asymptotically invariant distribution of government debt under a Ramsey plan.

BEGS also compute an approximation to a Ramsey plan's rate of convergence to that limiting invariant distribution.

We shall use the BEGS approximating limiting distribution and their approximating rate of convergence to help interpret outcomes here.

For a long time, the Ramsey plan puts a nontrivial martingale-like component into the par value of government debt as part of the way that the Ramsey plan imperfectly smooths distortions from the labor tax rate across time and Markov states.

But BEGS show that binding implementability constraints slowly push government debt in a direction designed to let the government use fluctuations in equilibrium interest rates rather than fluctuations in par values of debt to insure against shocks to government expenditures.

- This is a **weak** (but unrelenting) force that, starting from a positive initial debt level, for a long time is dominated by the stochastic martingale-like component of debt dynamics that the Ramsey planner uses to facilitate imperfect tax-smoothing across time and states.
- This weak force slowly drives the par value of government **assets** to a **constant** level at which the government can completely insure against government expenditure shocks while shutting down the stochastic component of debt dynamics.
- At that point, the tail of the par value of government debt becomes a trivial martingale: it is constant over time.

6.3 Logical Flow of Lecture

We present ideas in the following order

- We describe a two-state AMSS economy and generate a long simulation starting from a positive initial government debt.
- We observe that in a long simulation starting from positive government debt, the par value of government debt eventually converges to a constant $\bar{b}$.
- In fact, the par value of government debt converges to the same constant level $\bar{b}$ for alternative realizations of the Markov government expenditure process and for alternative settings of initial government debt b_0.
- We reverse engineer a particular value of initial government debt b_0 (it turns out to be negative) for which the continuation debt moves to $\bar{b}$ immediately.

- We note that for this particular initial debt b_0, the Ramsey allocations for the AMSS economy and the Lucas-Stokey model are identical
  - we verify that the LS Ramsey planner chooses to purchase **identical** claims to time $t+1$ consumption for all Markov states tomorrow for each Markov state today.
- We compute the BEGS approximations to check how accurately they describe the dynamics of the long-simulation.

6.3.1 Equations from Lucas-Stokey (1983) Model

Although we are studying an AMSS [AMSSeppala02] economy, a Lucas-Stokey [LS83] economy plays an important role in the reverse-engineering calculation to be described below.

For that reason, it is helpful to have key equations underlying a Ramsey plan for the Lucas-Stokey economy readily available.

Recall first-order conditions for a Ramsey allocation for the Lucas-Stokey economy.

For $t \geq 1$, these take the form

$$\begin{aligned}(1+\Phi)u_c(c,1-c-g) + \Phi\big[cu_{cc}(c,1-c-g) - (c+g)u_{\ell c}(c,1-c-g)\big] \\ = (1+\Phi)u_\ell(c,1-c-g) + \Phi\big[cu_{c\ell}(c,1-c-g) - (c+g)u_{\ell\ell}(c,1-c-g)\big]\end{aligned} \tag{6.1}$$

There is one such equation for each value of the Markov state s_t.

Given an initial Markov state, the time $t=0$ quantities c_0 and b_0 satisfy

$$\begin{aligned}(1+\Phi)u_c(c,1-c-g) + \Phi\big[cu_{cc}(c,1-c-g) - (c+g)u_{\ell c}(c,1-c-g)\big] \\ = (1+\Phi)u_\ell(c,1-c-g) + \Phi\big[cu_{c\ell}(c,1-c-g) - (c+g)u_{\ell\ell}(c,1-c-g)\big] + \Phi(u_{cc} - u_{c,\ell})b_0\end{aligned} \tag{6.2}$$

In addition, the time $t=0$ budget constraint is satisfied at c_0 and initial government debt b_0

$$b_0 + g_0 = \tau_0(c_0+g_0) + \frac{\bar{b}}{R_0} \tag{6.3}$$

where R_0 is the gross interest rate for the Markov state s_0 that is assumed to prevail at time $t=0$ and τ_0 is the time $t=0$ tax rate.

In equation (6.3), it is understood that

$$\begin{aligned}\tau_0 &= 1 - \frac{u_{l,0}}{u_{c,0}} \\ R_0^{-1} &= \beta\sum_{s=1}^{S}\Pi(s|s_0)\frac{u_c(s)}{u_{c,0}}\end{aligned}$$

It is useful to transform some of the above equations to forms that are more natural for analyzing the case of a CRRA utility specification that we shall use in our example economies.

6.3.2 Specification with CRRA Utility

As in lectures *optimal taxation without state-contingent debt* and *optimal taxation with state-contingent debt*, we assume that the representative agent has utility function

$$u(c,n) = \frac{c^{1-\sigma}}{1-\sigma} - \frac{n^{1+\gamma}}{1+\gamma}$$

and set $\sigma = 2$, $\gamma = 2$, and the discount factor $\beta = 0.9$.

We eliminate leisure from the model and continue to assume that

$$c_t + g_t = n_t$$

The analysis of Lucas and Stokey prevails once we make the following replacements

$$\begin{aligned}
u_\ell(c,\ell) &\sim -u_n(c,n)\\
u_c(c,\ell) &\sim u_c(c,n)\\
u_{\ell,\ell}(c,\ell) &\sim u_{nn}(c,n)\\
u_{c,c}(c,\ell) &\sim u_{c,c}(c,n)\\
u_{c,\ell}(c,\ell) &\sim 0
\end{aligned}$$

With these understandings, equations (6.1) and (6.2) simplify in the case of the CRRA utility function.

They become

$$(1+\Phi)[u_c(c)+u_n(c+g)]+\Phi[cu_{cc}(c)+(c+g)u_{nn}(c+g)]=0 \tag{6.4}$$

and

$$(1+\Phi)[u_c(c_0)+u_n(c_0+g_0)]+\Phi[c_0u_{cc}(c_0)+(c_0+g_0)u_{nn}(c_0+g_0)]-\Phi u_{cc}(c_0)b_0=0 \tag{6.5}$$

In equation (6.4), it is understood that c and g are each functions of the Markov state s.

The CRRA utility function is represented in the following class.

```python
import numpy as np


class CRRAutility:

    def __init__(self,
                 β=0.9,
                 σ=2,
                 γ=2,
                 π=np.full((2, 2), 0.5),
                 G=np.array([0.1, 0.2]),
                 Θ=np.ones(2),
                 transfers=False):

        self.β, self.σ, self.γ = β, σ, γ
        self.π, self.G, self.Θ, self.transfers = π, G, Θ, transfers

    # Utility function
    def U(self, c, n):
        σ = self.σ
        if σ == 1.:
            U = np.log(c)
        else:
            U = (c**(1 - σ) - 1) / (1 - σ)
        return U - n**(1 + self.γ) / (1 + self.γ)

    # Derivatives of utility function
    def Uc(self, c, n):
        return c**(-self.σ)

    def Ucc(self, c, n):
```

(continues on next page)

(continued from previous page)

```
        return -self.σ * c**(-self.σ - 1)

    def Un(self, c, n):
        return -n**self.γ

    def Unn(self, c, n):
        return -self.γ * n**(self.γ - 1)
```

6.4 Example Economy

We set the following parameter values.

The Markov state s_t takes two values, namely, $0, 1$.

The initial Markov state is 0.

The Markov transition matrix is $.5I$ where I is a 2×2 identity matrix, so the s_t process is IID.

Government expenditures $g(s)$ equal $.1$ in Markov state 0 and $.2$ in Markov state 1.

We set preference parameters as follows:

$$\begin{aligned} \beta &= .9 \\ \sigma &= 2 \\ \gamma &= 2 \end{aligned}$$

Here are several classes that do most of the work for us.

The code is mostly taken or adapted from the earlier lectures *optimal taxation without state-contingent debt* and *optimal taxation with state-contingent debt*.

```
import numpy as np
from scipy.optimize import root
from quantecon import MarkovChain


class SequentialAllocation:

    '''
    Class that takes CESutility or BGPutility object as input returns
    planner's allocation as a function of the multiplier on the
    implementability constraint μ.
    '''

    def __init__(self, model):

        # Initialize from model object attributes
        self.β, self.π, self.G = model.β, model.π, model.G
        self.mc, self.Θ = MarkovChain(self.π), model.Θ
        self.S = len(model.π)  # Number of states
        self.model = model

        # Find the first best allocation
        self.find_first_best()
```

(continues on next page)

(continued from previous page)

```
    def find_first_best(self):
        '''
        Find the first best allocation
        '''
        model = self.model
        S, Θ, G = self.S, self.Θ, self.G
        Uc, Un = model.Uc, model.Un

        def res(z):
            c = z[:S]
            n = z[S:]
            return np.hstack([Θ * Uc(c, n) + Un(c, n), Θ * n - c - G])

        res = root(res, np.full(2 * S, 0.5))

        if not res.success:
            raise Exception('Could not find first best')

        self.cFB = res.x[:S]
        self.nFB = res.x[S:]

        # Multiplier on the resource constraint
        self.ΞFB = Uc(self.cFB, self.nFB)
        self.zFB = np.hstack([self.cFB, self.nFB, self.ΞFB])

    def time1_allocation(self, μ):
        '''
        Computes optimal allocation for time t >= 1 for a given μ
        '''
        model = self.model
        S, Θ, G = self.S, self.Θ, self.G
        Uc, Ucc, Un, Unn = model.Uc, model.Ucc, model.Un, model.Unn

        def FOC(z):
            c = z[:S]
            n = z[S:2 * S]
            Ξ = z[2 * S:]
            # FOC of c
            return np.hstack([Uc(c, n) - μ * (Ucc(c, n) * c + Uc(c, n)) - Ξ,
                              Un(c, n) - μ * (Unn(c, n) * n + Un(c, n)) \
                              + Θ * Ξ,  # FOC of n
                              Θ * n - c - G])

        # Find the root of the first-order condition
        res = root(FOC, self.zFB)
        if not res.success:
            raise Exception('Could not find LS allocation.')
        z = res.x
        c, n, Ξ = z[:S], z[S:2 * S], z[2 * S:]

        # Compute x
        I = Uc(c, n) * c + Un(c, n) * n
        x = np.linalg.solve(np.eye(S) - self.β * self.π, I)

        return c, n, x, Ξ
```

(continues on next page)

(continued from previous page)

```
    def time0_allocation(self, B_, s_0):
        '''
        Finds the optimal allocation given initial government debt B_ and
        state s_0
        '''
        model, π, Θ, G, β = self.model, self.π, self.Θ, self.G, self.β
        Uc, Ucc, Un, Unn = model.Uc, model.Ucc, model.Un, model.Unn

        # First order conditions of planner's problem
        def FOC(z):
            μ, c, n, Ξ = z
            xprime = self.time1_allocation(μ)[2]
            return np.hstack([Uc(c, n) * (c - B_) + Un(c, n) * n + β * π[s_0]
                                            @ xprime,
                              Uc(c, n) - μ * (Ucc(c, n)
                                            * (c - B_) + Uc(c, n)) - Ξ,
                              Un(c, n) - μ * (Unn(c, n) * n
                                            + Un(c, n)) + Θ[s_0] * Ξ,
                              (Θ * n - c - G)[s_0]])

        # Find root
        res = root(FOC, np.array(
            [0, self.cFB[s_0], self.nFB[s_0], self.ΞFB[s_0]]))
        if not res.success:
            raise Exception('Could not find time 0 LS allocation.')

        return res.x

    def time1_value(self, μ):
        '''
        Find the value associated with multiplier μ
        '''
        c, n, x, Ξ = self.time1_allocation(μ)
        U = self.model.U(c, n)
        V = np.linalg.solve(np.eye(self.S) - self.β * self.π, U)
        return c, n, x, V

    def T(self, c, n):
        '''
        Computes T given c, n
        '''
        model = self.model
        Uc, Un = model.Uc(c, n), model.Un(c,  n)

        return 1 + Un / (self.Θ * Uc)

    def simulate(self, B_, s_0, T, sHist=None):
        '''
        Simulates planners policies for T periods
        '''
        model, π, β = self.model, self.π, self.β
        Uc = model.Uc

        if sHist is None:
            sHist = self.mc.simulate(T, s_0)
```

(continues on next page)

(continued from previous page)

```
        cHist, nHist, Bhist, THist, μHist = np.zeros((5, T))
        RHist = np.zeros(T - 1)

        # Time 0
        μ, cHist[0], nHist[0], _ = self.time0_allocation(B_, s_0)
        THist[0] = self.T(cHist[0], nHist[0])[s_0]
        Bhist[0] = B_
        μHist[0] = μ

        # Time 1 onward
        for t in range(1, T):
            c, n, x, Ξ = self.time1_allocation(μ)
            T = self.T(c, n)
            u_c = Uc(c, n)
            s = sHist[t]
            Eu_c = π[sHist[t - 1]] @ u_c
            cHist[t], nHist[t], Bhist[t], THist[t] = c[s], n[s], x[s] / u_c[s], \
                                                     T[s]
            RHist[t - 1] = Uc(cHist[t - 1], nHist[t - 1]) / (β * Eu_c)
            μHist[t] = μ

        return [cHist, nHist, Bhist, THist, sHist, μHist, RHist]
```

```
import numpy as np
from scipy.optimize import fmin_slsqp
from scipy.optimize import root
from quantecon import MarkovChain


class RecursiveAllocationAMSS:

    def __init__(self, model, μgrid, tol_diff=1e-7, tol=1e-7):

        self.β, self.π, self.G = model.β, model.π, model.G
        self.mc, self.S = MarkovChain(self.π), len(model.π)  # Number of states
        self.Θ, self.model, self.μgrid = model.Θ, model, μgrid
        self.tol_diff, self.tol = tol_diff, tol

        # Find the first best allocation
        self.solve_time1_bellman()
        self.T.time_0 = True  # Bellman equation now solves time 0 problem

    def solve_time1_bellman(self):
        '''
        Solve the time  1 Bellman equation for calibration model and
        initial grid μgrid0
        '''
        model, μgrid0 = self.model, self.μgrid
        π = model.π
        S = len(model.π)

        # First get initial fit from Lucas Stokey solution.
        # Need to change things to be ex ante
        pp = SequentialAllocation(model)
        interp = interpolator_factory(2, None)
```

(continues on next page)

(continued from previous page)

```
        def incomplete_allocation(μ_, s_):
            c, n, x, V = pp.time1_value(μ_)
            return c, n, π[s_] @ x, π[s_] @ V
        cf, nf, xgrid, Vf, xprimef = [], [], [], [], []
        for s_ in range(S):
            c, n, x, V = zip(*map(lambda μ: incomplete_allocation(μ, s_), μgrid0))
            c, n = np.vstack(c).T, np.vstack(n).T
            x, V = np.hstack(x), np.hstack(V)
            xprimes = np.vstack([x] * S)
            cf.append(interp(x, c))
            nf.append(interp(x, n))
            Vf.append(interp(x, V))
            xgrid.append(x)
            xprimef.append(interp(x, xprimes))
        cf, nf, xprimef = fun_vstack(cf), fun_vstack(nf), fun_vstack(xprimef)
        Vf = fun_hstack(Vf)
        policies = [cf, nf, xprimef]

        # Create xgrid
        x = np.vstack(xgrid).T
        xbar = [x.min(0).max(), x.max(0).min()]
        xgrid = np.linspace(xbar[0], xbar[1], len(μgrid0))
        self.xgrid = xgrid

        # Now iterate on Bellman equation
        T = BellmanEquation(model, xgrid, policies, tol=self.tol)
        diff = 1
        while diff > self.tol_diff:
            PF = T(Vf)

            Vfnew, policies = self.fit_policy_function(PF)
            diff = np.abs((Vf(xgrid) - Vfnew(xgrid)) / Vf(xgrid)).max()

            print(diff)
            Vf = Vfnew

        # Store value function policies and Bellman Equations
        self.Vf = Vf
        self.policies = policies
        self.T = T

    def fit_policy_function(self, PF):
        '''
        Fits the policy functions
        '''
        S, xgrid = len(self.π), self.xgrid
        interp = interpolator_factory(3, 0)
        cf, nf, xprimef, Tf, Vf = [], [], [], [], []
        for s_ in range(S):
            PFvec = np.vstack([PF(x, s_) for x in self.xgrid]).T
            Vf.append(interp(xgrid, PFvec[0, :]))
            cf.append(interp(xgrid, PFvec[1:1 + S]))
            nf.append(interp(xgrid, PFvec[1 + S:1 + 2 * S]))
            xprimef.append(interp(xgrid, PFvec[1 + 2 * S:1 + 3 * S]))
            Tf.append(interp(xgrid, PFvec[1 + 3 * S:]))
```

(continues on next page)

(continued from previous page)

```
        policies = fun_vstack(cf), fun_vstack(
            nf), fun_vstack(xprimef), fun_vstack(Tf)
        Vf = fun_hstack(Vf)
        return Vf, policies

    def T(self, c, n):
        '''
        Computes T given c and n
        '''
        model = self.model
        Uc, Un = model.Uc(c, n), model.Un(c, n)

        return 1 + Un / (self.Θ * Uc)

    def time0_allocation(self, B_, s0):
        '''
        Finds the optimal allocation given initial government debt B_ and
        state s_0
        '''
        PF = self.T(self.Vf)
        z0 = PF(B_, s0)
        c0, n0, xprime0, T0 = z0[1:]
        return c0, n0, xprime0, T0

    def simulate(self, B_, s_0, T, sHist=None):
        '''
        Simulates planners policies for T periods
        '''
        model, π = self.model, self.π
        Uc = model.Uc
        cf, nf, xprimef, Tf = self.policies

        if sHist is None:
            sHist = simulate_markov(π, s_0, T)

        cHist, nHist, Bhist, xHist, THist, THist, μHist = np.zeros((7, T))
        # Time 0
        cHist[0], nHist[0], xHist[0], THist[0] = self.time0_allocation(B_, s_0)
        THist[0] = self.T(cHist[0], nHist[0])[s_0]
        Bhist[0] = B_
        μHist[0] = self.Vf[s_0](xHist[0])

        # Time 1 onward
        for t in range(1, T):
            s_, x, s = sHist[t - 1], xHist[t - 1], sHist[t]
            c, n, xprime, T = cf[s_, :](x), nf[s_, :](
                x), xprimef[s_, :](x), Tf[s_, :](x)

            T = self.T(c, n)[s]
            u_c = Uc(c, n)
            Eu_c = π[s_, :] @ u_c

            μHist[t] = self.Vf[s](xprime[s])

            cHist[t], nHist[t], Bhist[t], THist[t] = c[s], n[s], x / Eu_c, T
            xHist[t], THist[t] = xprime[s], T[s]
```

(continues on next page)

(continued from previous page)

```
        return [cHist, nHist, Bhist, THist, THist, μHist, sHist, xHist]


class BellmanEquation:
    '''
    Bellman equation for the continuation of the Lucas-Stokey Problem
    '''

    def __init__(self, model, xgrid, policies0, tol, maxiter=1000):

        self.β, self.π, self.G = model.β, model.π, model.G
        self.S = len(model.π)  # Number of states
        self.Θ, self.model, self.tol = model.Θ, model, tol
        self.maxiter = maxiter

        self.xbar = [min(xgrid), max(xgrid)]
        self.time_0 = False

        self.z0 = {}
        cf, nf, xprimef = policies0

        for s_ in range(self.S):
            for x in xgrid:
                self.z0[x, s_] = np.hstack([cf[s_, :](x),
                                            nf[s_, :](x),
                                            xprimef[s_, :](x),
                                            np.zeros(self.S)])

        self.find_first_best()

    def find_first_best(self):
        '''
        Find the first best allocation
        '''
        model = self.model
        S, Θ, Uc, Un, G = self.S, self.Θ, model.Uc, model.Un, self.G

        def res(z):
            c = z[:S]
            n = z[S:]
            return np.hstack([Θ * Uc(c, n) + Un(c, n), Θ * n - c - G])

        res = root(res, np.full(2 * S, 0.5))
        if not res.success:
            raise Exception('Could not find first best')

        self.cFB = res.x[:S]
        self.nFB = res.x[S:]
        IFB = Uc(self.cFB, self.nFB) * self.cFB + \
            Un(self.cFB, self.nFB) * self.nFB

        self.xFB = np.linalg.solve(np.eye(S) - self.β * self.π, IFB)

        self.zFB = {}
        for s in range(S):
            self.zFB[s] = np.hstack(
```

(continues on next page)

(continued from previous page)

```
                [self.cFB[s], self.nFB[s], self.π[s] @ self.xFB, 0.])

    def __call__(self, Vf):
        '''
        Given continuation value function next period return value function this
        period return T(V) and optimal policies
        '''
        if not self.time_0:
            def PF(x, s): return self.get_policies_time1(x, s, Vf)
        else:
            def PF(B_, s0): return self.get_policies_time0(B_, s0, Vf)
        return PF

    def get_policies_time1(self, x, s_, Vf):
        '''
        Finds the optimal policies
        '''
        model, β, Θ, G, S, π = self.model, self.β, self.Θ, self.G, self.S, self.π
        U, Uc, Un = model.U, model.Uc, model.Un

        def objf(z):
            c, n, xprime = z[:S], z[S:2 * S], z[2 * S:3 * S]

            Vprime = np.empty(S)
            for s in range(S):
                Vprime[s] = Vf[s](xprime[s])

            return -π[s_] @ (U(c, n) + β * Vprime)

        def objf_prime(x):

            epsilon = 1e-7
            x0 = np.asfarray(x)
            f0 = np.atleast_1d(objf(x0))
            jac = np.zeros([len(x0), len(f0)])
            dx = np.zeros(len(x0))
            for i in range(len(x0)):
                dx[i] = epsilon
                jac[i] = (objf(x0+dx) - f0)/epsilon
                dx[i] = 0.0

            return jac.transpose()

        def cons(z):
            c, n, xprime, T = z[:S], z[S:2 * S], z[2 * S:3 * S], z[3 * S:]
            u_c = Uc(c, n)
            Eu_c = π[s_] @ u_c
            return np.hstack([
                x * u_c / Eu_c - u_c * (c - T) - Un(c, n) * n - β * xprime,
                Θ * n - c - G])

        if model.transfers:
            bounds = [(0., 100)] * S + [(0., 100)] * S + \
                [self.xbar] * S + [(0., 100.)] * S
        else:
            bounds = [(0., 100)] * S + [(0., 100)] * S + \
```

(continues on next page)

(continued from previous page)

```
                [self.xbar] * S + [(0., 0.)] * S
        out, fx, _, imode, smode = fmin_slsqp(objf, self.z0[x, s_],
                                              f_eqcons=cons, bounds=bounds,
                                              fprime=objf_prime, full_output=True,
                                              iprint=0, acc=self.tol, iter=self.
↪maxiter)

        if imode > 0:
            raise Exception(smode)

        self.z0[x, s_] = out
        return np.hstack([-fx, out])

    def get_policies_time0(self, B_, s0, Vf):
        '''
        Finds the optimal policies
        '''
        model, β, Θ, G = self.model, self.β, self.Θ, self.G
        U, Uc, Un = model.U, model.Uc, model.Un

        def objf(z):
            c, n, xprime = z[:-1]

            return -(U(c, n) + β * Vf[s0](xprime))

        def cons(z):
            c, n, xprime, T = z
            return np.hstack([
                -Uc(c, n) * (c - B_ - T) - Un(c, n) * n - β * xprime,
                (Θ * n - c - G)[s0]])

        if model.transfers:
            bounds = [(0., 100), (0., 100), self.xbar, (0., 100.)]
        else:
            bounds = [(0., 100), (0., 100), self.xbar, (0., 0.)]
        out, fx, _, imode, smode = fmin_slsqp(objf, self.zFB[s0], f_eqcons=cons,
                                              bounds=bounds, full_output=True,
                                              iprint=0)

        if imode > 0:
            raise Exception(smode)

        return np.hstack([-fx, out])
```

```
import numpy as np
from scipy.interpolate import UnivariateSpline


class interpolate_wrapper:

    def __init__(self, F):
        self.F = F

    def __getitem__(self, index):
        return interpolate_wrapper(np.asarray(self.F[index]))
```

(continues on next page)

(continued from previous page)

```
    def reshape(self, *args):
        self.F = self.F.reshape(*args)
        return self

    def transpose(self):
        self.F = self.F.transpose()

    def __len__(self):
        return len(self.F)

    def __call__(self, xvec):
        x = np.atleast_1d(xvec)
        shape = self.F.shape
        if len(x) == 1:
            fhat = np.hstack([f(x) for f in self.F.flatten()])
            return fhat.reshape(shape)
        else:
            fhat = np.vstack([f(x) for f in self.F.flatten()])
            return fhat.reshape(np.hstack((shape, len(x))))


class interpolator_factory:

    def __init__(self, k, s):
        self.k, self.s = k, s

    def __call__(self, xgrid, Fs):
        shape, m = Fs.shape[:-1], Fs.shape[-1]
        Fs = Fs.reshape((-1, m))
        F = []
        xgrid = np.sort(xgrid)  # Sort xgrid
        for Fhat in Fs:
            F.append(UnivariateSpline(xgrid, Fhat, k=self.k, s=self.s))
        return interpolate_wrapper(np.array(F).reshape(shape))


def fun_vstack(fun_list):

    Fs = [IW.F for IW in fun_list]
    return interpolate_wrapper(np.vstack(Fs))


def fun_hstack(fun_list):

    Fs = [IW.F for IW in fun_list]
    return interpolate_wrapper(np.hstack(Fs))


def simulate_markov(π, s_0, T):

    sHist = np.empty(T, dtype=int)
    sHist[0] = s_0
    S = len(π)
    for t in range(1, T):
        sHist[t] = np.random.choice(np.arange(S), p=π[sHist[t - 1]])
```

(continues on next page)

(continued from previous page)

```
    return sHist
```

6.5 Reverse Engineering Strategy

We can reverse engineer a value b_0 of initial debt due that renders the AMSS measurability constraints not binding from time $t = 0$ onward.

We accomplish this by recognizing that if the AMSS measurability constraints never bind, then the AMSS allocation and Ramsey plan is equivalent with that for a Lucas-Stokey economy in which for each period $t \geq 0$, the government promises to pay the **same** state-contingent amount $\bar{b}$ in each state tomorrow.

This insight tells us to find a b_0 and other fundamentals for the Lucas-Stokey [LS83] model that make the Ramsey planner want to borrow the same value $\bar{b}$ next period for all states and all dates.

We accomplish this by using various equations for the Lucas-Stokey [LS83] model presented in *optimal taxation with state-contingent debt*.

We use the following steps.

Step 1: Pick an initial Φ.

Step 2: Given that Φ, jointly solve two versions of equation (6.4) for $c(s), s = 1, 2$ associated with the two values for $g(s), s = 1, 2$.

Step 3: Solve the following equation for $\vec{x}$

$$\vec{x} = (I - \beta\Pi)^{-1}[\vec{u}_c(\vec{n} - \vec{g}) - \vec{u}_l\vec{n}] \tag{6.6}$$

Step 4: After solving for $\vec{x}$, we can find $b(s_t|s^{t-1})$ in Markov state $s_t = s$ from $b(s) = \frac{x(s)}{u_c(s)}$ or the matrix equation

$$\vec{b} = \frac{\vec{x}}{\vec{u}_c} \tag{6.7}$$

Step 5: Compute $J(\Phi) = (b(1) - b(2))^2$.

Step 6: Put steps 2 through 6 in a function minimizer and find a Φ that minimizes $J(\Phi)$.

Step 7: At the value of Φ and the value of $\bar{b}$ that emerged from step 6, solve equations (6.5) and (6.3) jointly for c_0, b_0.

6.6 Code for Reverse Engineering

Here is code to do the calculations for us.

```
u = CRRAutility()

def min_Φ(Φ):

    g1, g2 = u.G  # Government spending in s=0 and s=1

    # Solve Φ(c)
    def equations(unknowns, Φ):
        c1, c2 = unknowns
```

(continues on next page)

(continued from previous page)

```
        # First argument of .Uc and second argument of .Un are redundant

        # Set up simultaneous equations
        eq = lambda c, g: (1 + Φ) * (u.Uc(c, 1) - -u.Un(1, c + g)) + \
                            Φ * ((c + g) * u.Unn(1, c + g) + c * u.Ucc(c, 1))

        # Return equation evaluated at s=1 and s=2
        return np.array([eq(c1, g1), eq(c2, g2)]).flatten()

    global c1                   # Update c1 globally
    global c2                   # Update c2 globally

    c1, c2 = fsolve(equations, np.ones(2), args=(Φ))

    uc = u.Uc(np.array([c1, c2]), 1)                            # uc(n - g)
    # ul(n) = -un(c + g)
    ul = -u.Un(1, np.array([c1 + g1, c2 + g2])) * [c1 + g1, c2 + g2]
    # Solve for x
    x = np.linalg.solve(np.eye((2)) - u.β * u.π, uc * [c1, c2] - ul)

    global b                    # Update b globally
    b = x / uc
    loss = (b[0] - b[1])**2

    return loss

Φ_star = fmin(min_Φ, .1, ftol=1e-14)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 24
         Function evaluations: 48
```

To recover and print out $\bar{b}$

```
b_bar = b[0]
b_bar
```

```
-1.0757576567504166
```

To complete the reverse engineering exercise by jointly determining c_0, b_0, we set up a function that returns two simultaneous equations.

```
def solve_cb(unknowns, Φ, b_bar, s=1):

    c0, b0 = unknowns

    g0 = u.G[s-1]

    R_0 = u.β * u.π[s] @ [u.Uc(c1, 1) / u.Uc(c0, 1), u.Uc(c2, 1) / u.Uc(c0, 1)]
    R_0 = 1 / R_0

    τ_0 = 1 + u.Un(1, c0 + g0) / u.Uc(c0, 1)
```

(continues on next page)

(continued from previous page)

```
    eq1 = τ_0 * (c0 + g0) + b_bar / R_0 - b0 - g0
    eq2 = (1 + Φ) * (u.Uc(c0, 1)  + u.Un(1, c0 + g0)) \
          + Φ * (c0 * u.Ucc(c0, 1) + (c0 + g0) * u.Unn(1, c0 + g0)) \
          - Φ * u.Ucc(c0, 1) * b0

    return np.array([eq1, eq2.item()], dtype='float64')
```

To solve the equations for c_0, b_0, we use SciPy's fsolve function

```
c0, b0 = fsolve(solve_cb, np.array([1., -1.], dtype='float64'),
                args=(Φ_star, b[0], 1), xtol=1.0e-12)
c0, b0
```

```
(0.9344994030900681, -1.0386984075517638)
```

Thus, we have reverse engineered an initial $b0 = -1.038698407551764$ that ought to render the AMSS measurability constraints slack.

6.7 Short Simulation for Reverse-engineered: Initial Debt

The following graph shows simulations of outcomes for both a Lucas-Stokey economy and for an AMSS economy starting from initial government debt equal to $b_0 = -1.038698407551764$.

These graphs report outcomes for both the Lucas-Stokey economy with complete markets and the AMSS economy with one-period risk-free debt only.

```
μ_grid = np.linspace(-0.09, 0.1, 100)

log_example = CRRAutility()

log_example.transfers = True                    # Government can use transfers
log_sequential = SequentialAllocation(log_example)  # Solve sequential problem
log_bellman = RecursiveAllocationAMSS(log_example, μ_grid,
                                      tol_diff=1e-10, tol=1e-10)

T = 20
sHist = np.array([0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
                  0, 0, 0, 1, 1, 1, 1, 1, 1, 0])


sim_seq = log_sequential.simulate(-1.03869841, 0, T, sHist)
sim_bel = log_bellman.simulate(-1.03869841, 0, T, sHist)

titles = ['Consumption', 'Labor Supply', 'Government Debt',
          'Tax Rate', 'Government Spending', 'Output']

# Government spending paths
sim_seq[4] = log_example.G[sHist]
sim_bel[4] = log_example.G[sHist]

# Output paths
sim_seq[5] = log_example.Θ[sHist] * sim_seq[1]
```

(continues on next page)

(continued from previous page)

```
sim_bel[5] = log_example.Θ[sHist] * sim_bel[1]

fig, axes = plt.subplots(3, 2, figsize=(14, 10))

for ax, title, seq, bel in zip(axes.flatten(), titles, sim_seq, sim_bel):
    ax.plot(seq, '-ok', bel, '-^b')
    ax.set(title=title)
    ax.grid()

axes[0, 0].legend(('Complete Markets', 'Incomplete Markets'))
plt.tight_layout()
plt.show()
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.11/site-packages/scipy/optimize/
↪_optimize.py:404: RuntimeWarning: Values in x were outside bounds during a␣
↪minimize step, clipping to bounds
  warnings.warn("Values in x were outside bounds during a "
```

```
/tmp/ipykernel_2232/108196118.py:24: RuntimeWarning: divide by zero encountered in␣
↪reciprocal
  U = (c**(1 - σ) - 1) / (1 - σ)
/tmp/ipykernel_2232/108196118.py:29: RuntimeWarning: divide by zero encountered in␣
↪power
  return c**(-self.σ)
/tmp/ipykernel_2232/1277371586.py:249: RuntimeWarning: invalid value encountered␣
↪in divide
  x * u_c / Eu_c - u_c * (c - T) - Un(c, n) * n - β * xprime,
/tmp/ipykernel_2232/1277371586.py:249: RuntimeWarning: invalid value encountered␣
↪in multiply
  x * u_c / Eu_c - u_c * (c - T) - Un(c, n) * n - β * xprime,
```

```
0.04094445433232542
```

```
0.001673211146137493
```

```
0.001484674847917127
```

```
0.001313772136887205
```

```
0.0011814037130420663
```

```
0.001055965336102068
```

```
0.0009446661649946108
```

```
0.0008463807319492324
```

```
0.0007560453788611131
```

```
0.0006756001033938903
```

```
0.000604152845540819
```

```
0.0005396004518747859
```

```
0.00048207169166290613
```

```
0.00043082732064067867
```

```
0.00038481851351225495
```

```
0.000343835217593145
```

```
0.0003072436935049677
```

```
0.0002745009146233244
```

```
0.00024531773293589513
```

```
0.00021923324298642947
```

```
0.00019593539310787213
```

```
0.00017514303481690137
```

```
0.0001565593985003591
```

```
0.00013996737081815812
```

```
0.00012514457789841946
```

```
0.00011190070823325749
```

```
0.0001000702000922041
```

```
8.949728534363834e-05
```

```
8.00497532414663e-05
```

```
7.160585250570457e-05
```

```
6.405840591557493e-05
```

```
5.731160522780524e-05
```

```
5.1279701373366633e-05
```

```
4.588651722582404e-05
```

```
4.106390497232627e-05
```

```
3.6750969979187823e-05
```

```
3.289357328148953e-05
```

```
2.9443322731171715e-05
```

```
2.6356778254647064e-05
```

```
2.3595477005441402e-05
```

```
2.1124867549068547e-05
```

```
1.8914292342161616e-05
```

```
1.6935989661294087e-05
```

```
1.5165570482803087e-05
```

```
1.3581075188566359e-05
```

```
1.2162766163347089e-05
```

```
1.0893227516817513e-05
```

```
9.756678182519297e-06
```

```
8.739234428152772e-06
```

```
7.828320614508025e-06
```

```
7.012602839408298e-06
```

```
6.2821988113865695e-06
```

```
5.628118884533389e-06
```

```
5.0424276120745635e-06
```

```
4.517800318375349e-06
```

```
4.048011435284343e-06
3.6271819852132397e-06
```

```
3.250228025571809e-06
```

```
2.91255521672949e-06
2.6100632205124585e-06
```

```
2.339096372677708e-06
```

```
2.096300057053759e-06
```

```
1.8787856014677842e-06
1.6838896002658147e-06
```

```
1.5092763000475938e-06
1.352790440377663e-06
```

```
1.2125870135921682e-06
1.0869367592654264e-06
```

```
9.74329344948381e-07
8.734258726613521e-07
```

```
7.82979401245993e-07
7.019280421759928e-07
```

```
6.292786681149374e-07
5.641636376342722e-07
```

```
5.058008139530142e-07
4.5348427330256424e-07
```

```
4.0659062310367744e-07
3.6455314441729855e-07
```

```
3.2687002299145745e-07
2.930882045255147e-07
```

```
2.6280345786809706e-07
2.356529429295176e-07
```

```
2.1131168850248635e-07
1.8948851788438695e-07
```

```
1.6992245629426705e-07
1.5237965358488245e-07
```

```
1.3665054480740185e-07
1.2254729288266142e-07
```

```
1.0990157880047098e-07
9.85625196806722e-08
```

```
8.839490296454315e-08
7.927751099544721e-08
```

```
7.110169892267009e-08
6.377012234144897e-08
```

```
5.719543299951795e-08
5.129944108294742e-08
```

```
4.6011930465755267e-08
4.127024907212617e-08
```

```
3.7017901411273995e-08
3.320421136675924e-08
```

```
2.9783836454122435e-08
2.6716185879207155e-08
```

```
2.3964828404060055e-08
2.1497111441656643e-08
```

```
1.928376711102591e-08
1.7298534286134342e-08
```

```
1.5517887041510468e-08
1.3920711115842077e-08
```

```
1.2488086772484325e-08
1.120303914946054e-08
```

```
1.0050349805051883e-08
9.016372957223345e-09
```

```
8.088867717275256e-09
7.256860052028448e-09
```

```
6.5105080491085e-09
5.8409842196277625e-09
```

```
5.240371187393206e-09
4.701571286205833e-09
```

```
4.2182149401635156e-09
3.784594252430241e-09
```

```
3.3955835551064364e-09
3.0465910785331343e-09
```

```
2.7334965385949916e-09
2.4526029798499404e-09
```

```
2.2005967896788517e-09
1.9745023230252437e-09
```

```
1.7716540861495694e-09
1.5896779606666392e-09
```

```
1.4263644656786832e-09
1.279915801041798e-09
```

```
1.1484611488603225e-09
1.0305702313922867e-09
```

```
9.247647878021015e-10
8.298468061604299e-10
```

```
7.446744286173443e-10
6.682506157688693e-10
```

```
5.996765544062293e-10
5.381420956749845e-10
```

```
4.829271458904042e-10
4.3337871811544764e-10
```

```
3.8891892933983235e-10
3.4902066124392655e-10
```

```
3.1321799130111273e-10
2.8109002457092086e-10
```

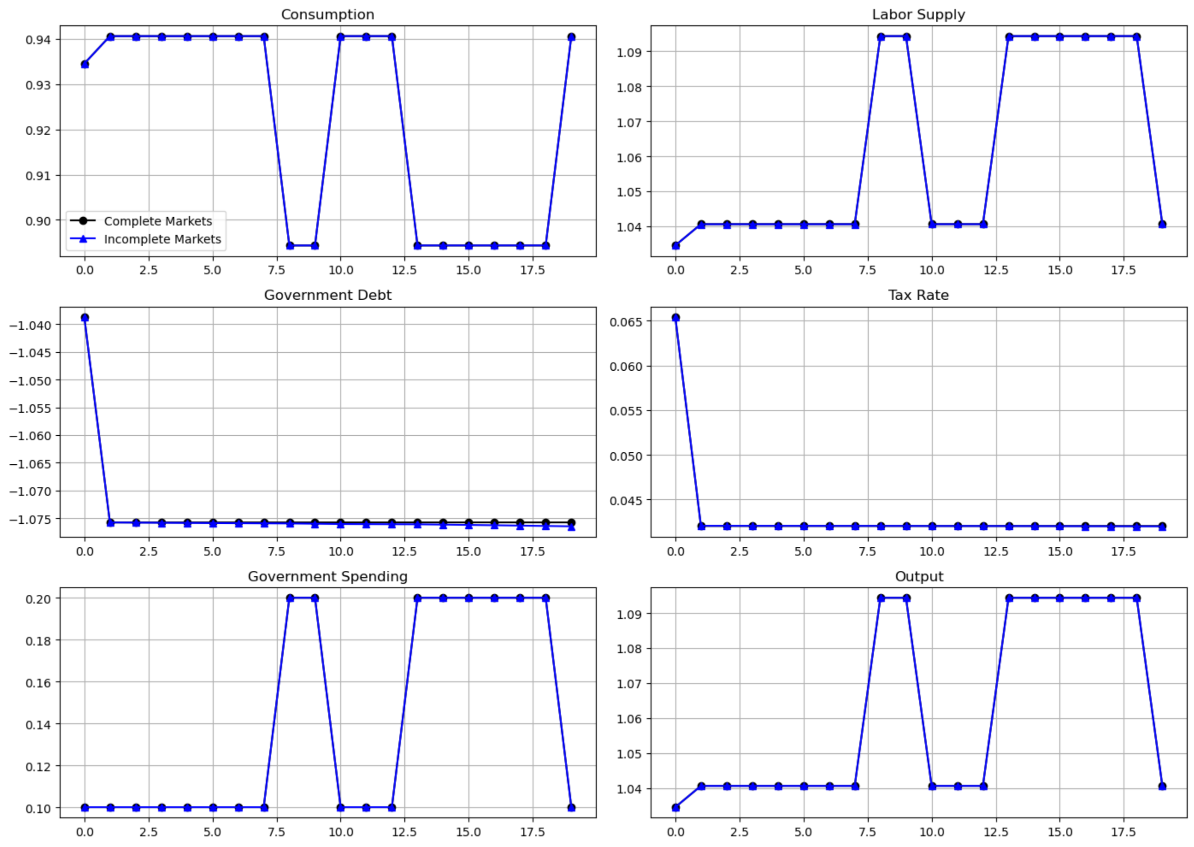
```
2.5225950288597284e-10
2.263868938948011e-10
```

```
2.0316830484184638e-10
1.8233409175417047e-10
```

```
1.6363582056463494e-10
1.4685617665861112e-10
```

```
1.3179940303096093e-10
1.1828486777347211e-10
```

```
1.0615888599012755e-10
9.527490070407684e-11
```



The Ramsey allocations and Ramsey outcomes are **identical** for the Lucas-Stokey and AMSS economies.

This outcome confirms the success of our reverse-engineering exercises.

Notice how for $t \geq 1$, the tax rate is a constant - so is the par value of government debt.

However, output and labor supply are both nontrivial time-invariant functions of the Markov state.

6.8 Long Simulation

The following graph shows the par value of government debt and the flat-rate tax on labor income for a long simulation for our sample economy.

For the **same** realization of a government expenditure path, the graph reports outcomes for two economies

- the gray lines are for the Lucas-Stokey economy with complete markets
- the blue lines are for the AMSS economy with risk-free one-period debt only

For both economies, initial government debt due at time 0 is $b_0 = .5$.

For the Lucas-Stokey complete markets economy, the government debt plotted is $b_{t+1}(s_{t+1})$.

- Notice that this is a time-invariant function of the Markov state from the beginning.

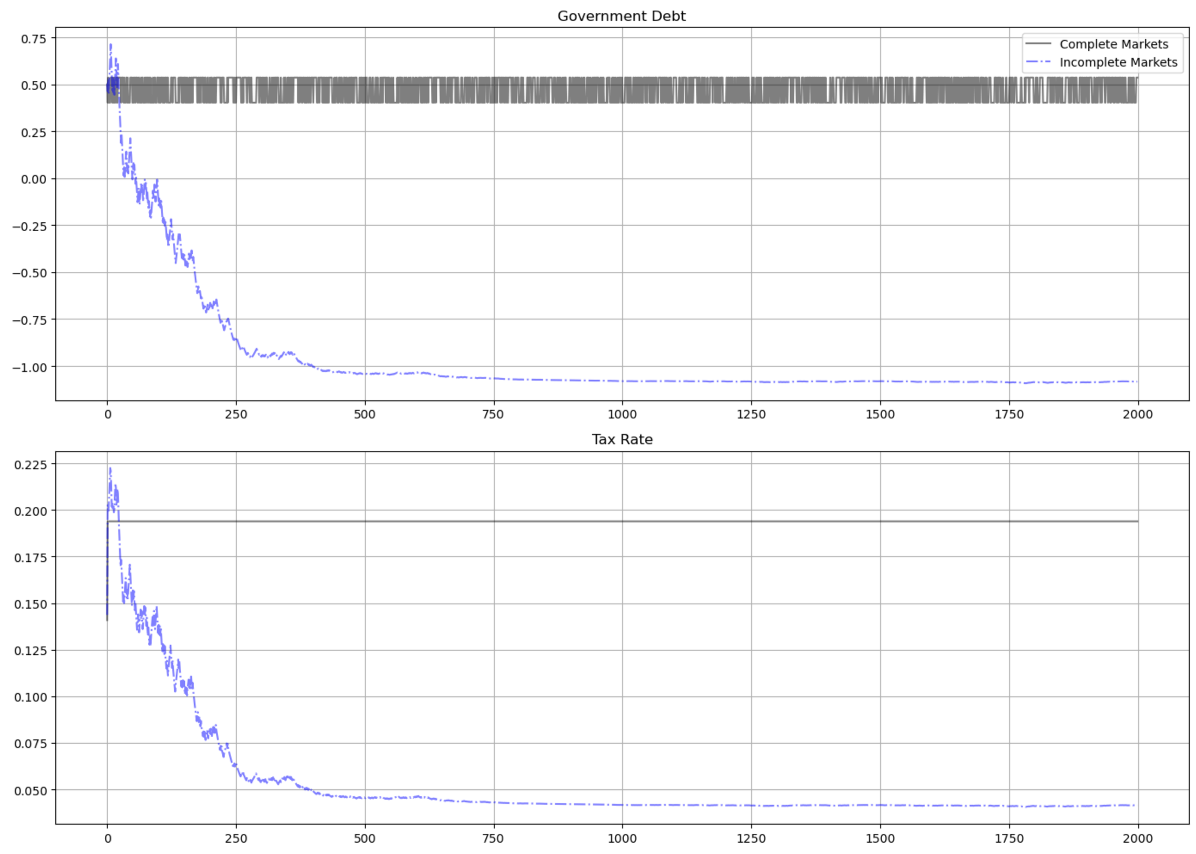
For the AMSS incomplete markets economy, the government debt plotted is $b_{t+1}(s^t)$.

- Notice that this is a martingale-like random process that eventually seems to converge to a constant $\bar{b} \approx -1.07$.
- Notice that the limiting value $\bar{b} < 0$ so that asymptotically the government makes a constant level of risk-free loans to the public.
- In the simulation displayed as well as other simulations we have run, the par value of government debt converges to about 1.07 after between 1400 to 2000 periods.

For the AMSS incomplete markets economy, the marginal tax rate on labor income τ_t converges to a constant

- labor supply and output each converge to time-invariant functions of the Markov state

```
T = 2000  # Set T to 200 periods

sim_seq_long = log_sequential.simulate(0.5, 0, T)
sHist_long = sim_seq_long[-3]
sim_bel_long = log_bellman.simulate(0.5, 0, T, sHist_long)

titles = ['Government Debt', 'Tax Rate']

fig, axes = plt.subplots(2, 1, figsize=(14, 10))

for ax, title, id in zip(axes.flatten(), titles, [2, 3]):
    ax.plot(sim_seq_long[id], '-k', sim_bel_long[id], '-.b', alpha=0.5)
    ax.set(title=title)
    ax.grid()

axes[0].legend(('Complete Markets', 'Incomplete Markets'))
plt.tight_layout()
plt.show()
```



6.8.1 Remarks about Long Simulation

As remarked above, after $b_{t+1}(s^t)$ has converged to a constant, the measurability constraints in the AMSS model cease to bind

- the associated Lagrange multipliers on those implementability constraints converge to zero

This leads us to seek an initial value of government debt b_0 that renders the measurability constraints slack from time $t = 0$ onward

- a tell-tale sign of this situation is that the Ramsey planner in a corresponding Lucas-Stokey economy would instruct the government to issue a constant level of government debt $b_{t+1}(s_{t+1})$ across the two Markov states

We now describe how to find such an initial level of government debt.

6.9 BEGS Approximations of Limiting Debt and Convergence Rate

It is useful to link the outcome of our reverse engineering exercise to limiting approximations constructed by BEGS [BEGS17].

BEGS [BEGS17] used a slightly different notation to represent a generalization of the AMSS model.

We'll introduce a version of their notation so that readers can quickly relate notation that appears in their key formulas to the notation that we have used.

BEGS work with objects $B_t, \mathcal{B}_t, \mathcal{R}_t, \mathcal{X}_t$ that are related to our notation by

$$\begin{aligned}
\mathcal{R}_t &= \frac{u_{c,t}}{u_{c,t-1}} R_{t-1} = \frac{u_{c,t}}{\beta E_{t-1} u_{c,t}} \\
B_t &= \frac{b_{t+1}(s^t)}{R_t(s^t)} \\
b_t(s^{t-1}) &= \mathcal{R}_{t-1} B_{t-1} \\
\mathcal{B}_t &= u_{c,t} B_t = (\beta E_t u_{c,t+1}) b_{t+1}(s^t) \\
\mathcal{X}_t &= u_{c,t}[g_t - \tau_t n_t]
\end{aligned}$$

In terms of their notation, equation (44) of [BEGS17] expresses the time t state s government budget constraint as

$$\mathcal{B}(s) = \mathcal{R}_\tau(s, s_-)\mathcal{B}_- + \mathcal{X}_{\tau(s)}(s) \tag{6.8}$$

where the dependence on τ is to remind us that these objects depend on the tax rate and s_- is last period's Markov state.

BEGS interpret random variations in the right side of (6.8) as a measure of **fiscal risk** composed of

- interest-rate-driven fluctuations in time t effective payments due on the government portfolio, namely, $\mathcal{R}_\tau(s, s_-)\mathcal{B}_-$, and
- fluctuations in the effective government deficit $\mathcal{X}_t$

6.9.1 Asymptotic Mean

BEGS give conditions under which the ergodic mean of $\mathcal{B}_t$ is

$$\mathcal{B}^* = -\frac{\text{cov}^\infty(\mathcal{R}, \mathcal{X})}{\text{var}^\infty(\mathcal{R})} \tag{6.9}$$

where the superscript ∞ denotes a moment taken with respect to an ergodic distribution.

Formula (6.9) presents $\mathcal{B}^*$ as a regression coefficient of $\mathcal{X}_t$ on $\mathcal{R}_t$ in the ergodic distribution.

This regression coefficient emerges as the minimizer for a variance-minimization problem:

$$\mathcal{B}^* = \text{argmin}_{\mathcal{B}} \text{var}(\mathcal{R}\mathcal{B} + \mathcal{X}) \tag{6.10}$$

The minimand in criterion (6.10) is the measure of fiscal risk associated with a given tax-debt policy that appears on the right side of equation (6.8).

Expressing formula (6.9) in terms of our notation tells us that $\bar{b}$ should approximately equal

$$\hat{b} = \frac{\mathcal{B}^*}{\beta E_t u_{c,t+1}} \tag{6.11}$$

6.9.2 Rate of Convergence

BEGS also derive the following approximation to the rate of convergence to $\mathcal{B}^*$ from an arbitrary initial condition.

$$\frac{E_t(\mathcal{B}_{t+1} - \mathcal{B}^*)}{(\mathcal{B}_t - \mathcal{B}^*)} \approx \frac{1}{1 + \beta^2 \text{var}(\mathcal{R})} \tag{6.12}$$

(See the equation above equation (47) in [BEGS17])

6.9.3 Formulas and Code Details

For our example, we describe some code that we use to compute the steady state mean and the rate of convergence to it.

The values of $\pi(s)$ are 0.5, 0.5.

We can then construct $\mathcal{X}(s), \mathcal{R}(s), u_c(s)$ for our two states using the definitions above.

We can then construct $\beta E_{t-1} u_c = \beta \sum_s u_c(s)\pi(s)$, cov$(\mathcal{R}(s), \mathcal{X}(s))$ and var$(\mathcal{R}(s))$ to be plugged into formula (6.11).

We also want to compute var$(\mathcal{X})$.

To compute the variances and covariance, we use the following standard formulas.

Temporarily let $x(s), s = 1, 2$ be an arbitrary random variables.

Then we define

$$\mu_x = \sum_s x(s)\pi(s)$$

$$\text{var}(x) = \left(\sum_s \sum_s x(s)^2 \pi(s)\right) - \mu_x^2$$

$$\text{cov}(x, y) = \left(\sum_s x(s)y(s)\pi(s)\right) - \mu_x \mu_y$$

After we compute these moments, we compute the BEGS approximation to the asymptotic mean $\hat{b}$ in formula (6.11).

After that, we move on to compute $\mathcal{B}^*$ in formula (6.9).

We'll also evaluate the BEGS criterion (6.8) at the limiting value $\mathcal{B}^*$

$$J(\mathcal{B}^*) = \text{var}(\mathcal{R})\left(\mathcal{B}^*\right)^2 + 2\mathcal{B}^* \text{cov}(\mathcal{R}, \mathcal{X}) + \text{var}(\mathcal{X}) \tag{6.13}$$

Here are some functions that we'll use to compute key objects that we want

```python
def mean(x):
    '''Returns mean for x given initial state'''
    x = np.array(x)
    return x @ u.π[s]

def variance(x):
    x = np.array(x)
    return x**2 @ u.π[s] - mean(x)**2

def covariance(x, y):
    x, y = np.array(x), np.array(y)
    return x * y @ u.π[s] - mean(x) * mean(y)
```

Now let's form the two random variables $\mathcal{R}, \mathcal{X}$ appearing in the BEGS approximating formulas

```python
u = CRRAutility()

s = 0
c = [0.940580824225584, 0.8943592757759343]  # Vector for c
g = u.G       # Vector for g
n = c + g     # Total population
τ = lambda s: 1 + u.Un(1, n[s]) / u.Uc(c[s], 1)

```

(continues on next page)

(continued from previous page)

```
R_s = lambda s: u.Uc(c[s], n[s]) / (u.β * (u.Uc(c[0], n[0]) * u.π[0, 0] \
                + u.Uc(c[1], n[1]) * u.π[1, 0]))
X_s = lambda s: u.Uc(c[s], n[s]) * (g[s] - τ(s) * n[s])

R = [R_s(0), R_s(1)]
X = [X_s(0), X_s(1)]

print(f"R, X = {R}, {X}")
```

```
R, X = [1.055169547122964, 1.1670526750992583], [0.06357685646224803, 0.
↪19251010100512958]
```

Now let's compute the ingredient of the approximating limit and the approximating rate of convergence

```
bstar = -covariance(R, X) / variance(R)
div = u.β * (u.Uc(c[0], n[0]) * u.π[s, 0] + u.Uc(c[1], n[1]) * u.π[s, 1])
bhat = bstar / div
bhat
```

```
-1.0757585378303758
```

Print out $\hat{b}$ and $\bar{b}$

```
bhat, b_bar
```

```
(-1.0757585378303758, -1.0757576567504166)
```

So we have

```
bhat - b_bar
```

```
-8.810799592140484e-07
```

These outcomes show that $\hat{b}$ does a remarkably good job of approximating $\bar{b}$.

Next, let's compute the BEGS fiscal criterion that $\hat{b}$ is minimizing

```
Jmin = variance(R) * bstar**2 + 2 * bstar * covariance(R, X) + variance(X)
Jmin
```

```
-9.020562075079397e-17
```

This is *machine zero*, a verification that $\hat{b}$ succeeds in minimizing the nonnegative fiscal cost criterion $J(\mathcal{B}^*)$ defined in BEGS and in equation (6.13) above.

Let's push our luck and compute the mean reversion speed in the formula above equation (47) in [BEGS17].

```
den2 = 1 + (u.β**2) * variance(R)
speedrever = 1/den2
print(f'Mean reversion speed = {speedrever}')
```

```
Mean reversion speed = 0.9974715478249827
```

Now let's compute the implied meantime to get to within 0.01 of the limit

```
ttime = np.log(.01) / np.log(speedrever)
print(f"Time to get within .01 of limit = {ttime}")
```

```
Time to get within .01 of limit = 1819.0360880098472
```

The slow rate of convergence and the implied time of getting within one percent of the limiting value do a good job of approximating our long simulation above.

In *a subsequent lecture* we shall study an extension of the model in which the force highlighted in this lecture causes government debt to converge to a nontrivial distribution instead of the single debt level discovered here.

CHAPTER

SEVEN

FISCAL RISK AND GOVERNMENT DEBT

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

7.1 Overview

This lecture studies government debt in an AMSS economy [AMSSeppala02] of the type described in *Optimal Taxation without State-Contingent Debt*.

We study the behavior of government debt as time $t \to +\infty$.

We use these techniques

- simulations
- a regression coefficient from the tail of a long simulation that allows us to verify that the asymptotic mean of government debt solves a fiscal-risk minimization problem
- an approximation to the **mean** of an ergodic distribution of government debt
- an approximation to the **rate of convergence** to an ergodic distribution of government debt

We apply tools that are applicable to more general incomplete markets economies that are presented on pages 648 - 650 in section III.D of [BEGS17] (BEGS).

We study an AMSS economy [AMSSeppala02] with three Markov states driving government expenditures.

- In a *previous lecture*, we showed that with only two Markov states, it is possible that endogenous interest rate fluctuations eventually can support complete markets allocations and Ramsey outcomes.
- The presence of three states prevents the full spanning that eventually prevails in the two-state example featured in *Fiscal Insurance via Fluctuating Interest Rates*.

The lack of full spanning means that the ergodic distribution of the par value of government debt is nontrivial, in contrast to the situation in *Fiscal Insurance via Fluctuating Interest Rates* in which the ergodic distribution of the par value of government debt is concentrated on one point.

Nevertheless, [BEGS17] (BEGS) establish that, for general settings that include ours, the Ramsey planner steers government assets to a level that comes **as close as possible** to providing full spanning in a precise a sense defined by BEGS that we describe below.

We use code constructed in *Fluctuating Interest Rates Deliver Fiscal Insurance*.

Warning: Key equations in [BEGS17] section III.D carry typos that we correct below.

Let's start with some imports:

```
import matplotlib.pyplot as plt
%matplotlib inline
from scipy.optimize import minimize
```

7.2 The Economy

As in *Optimal Taxation without State-Contingent Debt* and *Optimal Taxation with State-Contingent Debt*, we assume that the representative agent has utility function

$$u(c,n) = \frac{c^{1-\sigma}}{1-\sigma} - \frac{n^{1+\gamma}}{1+\gamma}$$

We work directly with labor supply instead of leisure.

We assume that

$$c_t + g_t = n_t$$

The Markov state s_t takes **three** values, namely, $0, 1, 2$.

The initial Markov state is 0.

The Markov transition matrix is $(1/3)I$ where I is a 3×3 identity matrix, so the s_t process is IID.

Government expenditures $g(s)$ equal $.1$ in Markov state 0, $.2$ in Markov state 1, and $.3$ in Markov state 2.

We set preference parameters

$$\begin{aligned} \beta &= .9 \\ \sigma &= 2 \\ \gamma &= 2 \end{aligned}$$

The following Python code sets up the economy

```
import numpy as np


class CRRAutility:

    def __init__(self,
                 β=0.9,
                 σ=2,
                 γ=2,
```

(continues on next page)

(continued from previous page)

```
                 π=np.full((2, 2), 0.5),
                 G=np.array([0.1, 0.2]),
                 Θ=np.ones(2),
                 transfers=False):

        self.β, self.σ, self.γ = β, σ, γ
        self.π, self.G, self.Θ, self.transfers = π, G, Θ, transfers

    # Utility function
    def U(self, c, n):
        σ = self.σ
        if σ == 1.:
            U = np.log(c)
        else:
            U = (c**(1 - σ) - 1) / (1 - σ)
        return U - n**(1 + self.γ) / (1 + self.γ)

    # Derivatives of utility function
    def Uc(self, c, n):
        return c**(-self.σ)

    def Ucc(self, c, n):
        return -self.σ * c**(-self.σ - 1)

    def Un(self, c, n):
        return -n**self.γ

    def Unn(self, c, n):
        return -self.γ * n**(self.γ - 1)
```

7.2.1 First and Second Moments

We'll want first and second moments of some key random variables below.

The following code computes these moments; the code is recycled from *Fluctuating Interest Rates Deliver Fiscal Insurance*.

```
def mean(x, s):
    '''Returns mean for x given initial state'''
    x = np.array(x)
    return x @ u.π[s]

def variance(x, s):
    x = np.array(x)
    return x**2 @ u.π[s] - mean(x, s)**2

def covariance(x, y, s):
    x, y = np.array(x), np.array(y)
    return x * y @ u.π[s] - mean(x, s) * mean(y, s)
```

7.3 Long Simulation

To generate a long simulation we use the following code.

We begin by showing the code that we used in earlier lectures on the AMSS model.

Here it is

```
import numpy as np
from scipy.optimize import root
from quantecon import MarkovChain


class SequentialAllocation:

    '''
    Class that takes CESutility or BGPutility object as input returns
    planner's allocation as a function of the multiplier on the
    implementability constraint μ.
    '''

    def __init__(self, model):

        # Initialize from model object attributes
        self.β, self.π, self.G = model.β, model.π, model.G
        self.mc, self.Θ = MarkovChain(self.π), model.Θ
        self.S = len(model.π)  # Number of states
        self.model = model

        # Find the first best allocation
        self.find_first_best()

    def find_first_best(self):
        '''
        Find the first best allocation
        '''
        model = self.model
        S, Θ, G = self.S, self.Θ, self.G
        Uc, Un = model.Uc, model.Un

        def res(z):
            c = z[:S]
            n = z[S:]
            return np.hstack([Θ * Uc(c, n) + Un(c, n), Θ * n - c - G])

        res = root(res, np.full(2 * S, 0.5))

        if not res.success:
            raise Exception('Could not find first best')

        self.cFB = res.x[:S]
        self.nFB = res.x[S:]

        # Multiplier on the resource constraint
        self.ΞFB = Uc(self.cFB, self.nFB)
        self.zFB = np.hstack([self.cFB, self.nFB, self.ΞFB])
```

(continues on next page)

(continued from previous page)

```
    def time1_allocation(self, μ):
        '''
        Computes optimal allocation for time t >= 1 for a given μ
        '''
        model = self.model
        S, Θ, G = self.S, self.Θ, self.G
        Uc, Ucc, Un, Unn = model.Uc, model.Ucc, model.Un, model.Unn

        def FOC(z):
            c = z[:S]
            n = z[S:2 * S]
            Ξ = z[2 * S:]
            # FOC of c
            return np.hstack([Uc(c, n) - μ * (Ucc(c, n) * c + Uc(c, n)) - Ξ,
                              Un(c, n) - μ * (Unn(c, n) * n + Un(c, n)) \
                              + Θ * Ξ,  # FOC of n
                              Θ * n - c - G])

        # Find the root of the first-order condition
        res = root(FOC, self.zFB)
        if not res.success:
            raise Exception('Could not find LS allocation.')
        z = res.x
        c, n, Ξ = z[:S], z[S:2 * S], z[2 * S:]

        # Compute x
        I = Uc(c, n) * c + Un(c, n) * n
        x = np.linalg.solve(np.eye(S) - self.β * self.π, I)

        return c, n, x, Ξ

    def time0_allocation(self, B_, s_0):
        '''
        Finds the optimal allocation given initial government debt B_ and
        state s_0
        '''
        model, π, Θ, G, β = self.model, self.π, self.Θ, self.G, self.β
        Uc, Ucc, Un, Unn = model.Uc, model.Ucc, model.Un, model.Unn

        # First order conditions of planner's problem
        def FOC(z):
            μ, c, n, Ξ = z
            xprime = self.time1_allocation(μ)[2]
            return np.hstack([Uc(c, n) * (c - B_) + Un(c, n) * n + β * π[s_0]
                                            @ xprime,
                              Uc(c, n) - μ * (Ucc(c, n)
                                            * (c - B_) + Uc(c, n)) - Ξ,
                              Un(c, n) - μ * (Unn(c, n) * n
                                            + Un(c, n)) + Θ[s_0] * Ξ,
                              (Θ * n - c - G)[s_0]])

        # Find root
        res = root(FOC, np.array(
            [0, self.cFB[s_0], self.nFB[s_0], self.ΞFB[s_0]]))
        if not res.success:
            raise Exception('Could not find time 0 LS allocation.')
```

(continues on next page)

(continued from previous page)

```
        return res.x

    def time1_value(self, μ):
        '''
        Find the value associated with multiplier μ
        '''
        c, n, x, Ξ = self.time1_allocation(μ)
        U = self.model.U(c, n)
        V = np.linalg.solve(np.eye(self.S) - self.β * self.π, U)
        return c, n, x, V

    def T(self, c, n):
        '''
        Computes T given c, n
        '''
        model = self.model
        Uc, Un = model.Uc(c, n), model.Un(c,  n)

        return 1 + Un / (self.Θ * Uc)

    def simulate(self, B_, s_0, T, sHist=None):
        '''
        Simulates planners policies for T periods
        '''
        model, π, β = self.model, self.π, self.β
        Uc = model.Uc

        if sHist is None:
            sHist = self.mc.simulate(T, s_0)

        cHist, nHist, Bhist, THist, μHist = np.zeros((5, T))
        RHist = np.zeros(T - 1)

        # Time 0
        μ, cHist[0], nHist[0], _ = self.time0_allocation(B_, s_0)
        THist[0] = self.T(cHist[0], nHist[0])[s_0]
        Bhist[0] = B_
        μHist[0] = μ

        # Time 1 onward
        for t in range(1, T):
            c, n, x, Ξ = self.time1_allocation(μ)
            T = self.T(c, n)
            u_c = Uc(c, n)
            s = sHist[t]
            Eu_c = π[sHist[t - 1]] @ u_c
            cHist[t], nHist[t], Bhist[t], THist[t] = c[s], n[s], x[s] / u_c[s], \
                                                      T[s]
            RHist[t - 1] = Uc(cHist[t - 1], nHist[t - 1]) / (β * Eu_c)
            μHist[t] = μ

        return [cHist, nHist, Bhist, THist, sHist, μHist, RHist]
```

```
import numpy as np
```

(continues on next page)

(continued from previous page)

```
from scipy.optimize import fmin_slsqp
from scipy.optimize import root
from quantecon import MarkovChain


class RecursiveAllocationAMSS:

    def __init__(self, model, μgrid, tol_diff=1e-7, tol=1e-7):

        self.β, self.π, self.G = model.β, model.π, model.G
        self.mc, self.S = MarkovChain(self.π), len(model.π)  # Number of states
        self.Θ, self.model, self.μgrid = model.Θ, model, μgrid
        self.tol_diff, self.tol = tol_diff, tol

        # Find the first best allocation
        self.solve_time1_bellman()
        self.T.time_0 = True  # Bellman equation now solves time 0 problem

    def solve_time1_bellman(self):
        '''
        Solve the time  1 Bellman equation for calibration model and
        initial grid μgrid0
        '''
        model, μgrid0 = self.model, self.μgrid
        π = model.π
        S = len(model.π)

        # First get initial fit from Lucas Stokey solution.
        # Need to change things to be ex ante
        pp = SequentialAllocation(model)
        interp = interpolator_factory(2, None)

        def incomplete_allocation(μ_, s_):
            c, n, x, V = pp.time1_value(μ_)
            return c, n, π[s_] @ x, π[s_] @ V
        cf, nf, xgrid, Vf, xprimef = [], [], [], [], []
        for s_ in range(S):
            c, n, x, V = zip(*map(lambda μ: incomplete_allocation(μ, s_), μgrid0))
            c, n = np.vstack(c).T, np.vstack(n).T
            x, V = np.hstack(x), np.hstack(V)
            xprimes = np.vstack([x] * S)
            cf.append(interp(x, c))
            nf.append(interp(x, n))
            Vf.append(interp(x, V))
            xgrid.append(x)
            xprimef.append(interp(x, xprimes))
        cf, nf, xprimef = fun_vstack(cf), fun_vstack(nf), fun_vstack(xprimef)
        Vf = fun_hstack(Vf)
        policies = [cf, nf, xprimef]

        # Create xgrid
        x = np.vstack(xgrid).T
        xbar = [x.min(0).max(), x.max(0).min()]
        xgrid = np.linspace(xbar[0], xbar[1], len(μgrid0))
        self.xgrid = xgrid
```

(continues on next page)

(continued from previous page)

```
        # Now iterate on Bellman equation
        T = BellmanEquation(model, xgrid, policies, tol=self.tol)
        diff = 1
        while diff > self.tol_diff:
            PF = T(Vf)

            Vfnew, policies = self.fit_policy_function(PF)
            diff = np.abs((Vf(xgrid) - Vfnew(xgrid)) / Vf(xgrid)).max()

            print(diff)
            Vf = Vfnew

        # Store value function policies and Bellman Equations
        self.Vf = Vf
        self.policies = policies
        self.T = T

    def fit_policy_function(self, PF):
        '''
        Fits the policy functions
        '''
        S, xgrid = len(self.π), self.xgrid
        interp = interpolator_factory(3, 0)
        cf, nf, xprimef, Tf, Vf = [], [], [], [], []
        for s_ in range(S):
            PFvec = np.vstack([PF(x, s_) for x in self.xgrid]).T
            Vf.append(interp(xgrid, PFvec[0, :]))
            cf.append(interp(xgrid, PFvec[1:1 + S]))
            nf.append(interp(xgrid, PFvec[1 + S:1 + 2 * S]))
            xprimef.append(interp(xgrid, PFvec[1 + 2 * S:1 + 3 * S]))
            Tf.append(interp(xgrid, PFvec[1 + 3 * S:]))
        policies = fun_vstack(cf), fun_vstack(
            nf), fun_vstack(xprimef), fun_vstack(Tf)
        Vf = fun_hstack(Vf)
        return Vf, policies

    def T(self, c, n):
        '''
        Computes T given c and n
        '''
        model = self.model
        Uc, Un = model.Uc(c, n), model.Un(c, n)

        return 1 + Un / (self.Θ * Uc)

    def time0_allocation(self, B_, s0):
        '''
        Finds the optimal allocation given initial government debt B_ and
        state s_0
        '''
        PF = self.T(self.Vf)
        z0 = PF(B_, s0)
        c0, n0, xprime0, T0 = z0[1:]
        return c0, n0, xprime0, T0

    def simulate(self, B_, s_0, T, sHist=None):
```

(continues on next page)

(continued from previous page)

```
        '''
        Simulates planners policies for T periods
        '''
        model, π = self.model, self.π
        Uc = model.Uc
        cf, nf, xprimef, Tf = self.policies

        if sHist is None:
            sHist = simulate_markov(π, s_0, T)

        cHist, nHist, Bhist, xHist, THist, THist, μHist = np.zeros((7, T))
        # Time 0
        cHist[0], nHist[0], xHist[0], THist[0] = self.time0_allocation(B_, s_0)
        THist[0] = self.T(cHist[0], nHist[0])[s_0]
        Bhist[0] = B_
        μHist[0] = self.Vf[s_0](xHist[0])

        # Time 1 onward
        for t in range(1, T):
            s_, x, s = sHist[t - 1], xHist[t - 1], sHist[t]
            c, n, xprime, T = cf[s_, :](x), nf[s_, :](
                x), xprimef[s_, :](x), Tf[s_, :](x)

            T = self.T(c, n)[s]
            u_c = Uc(c, n)
            Eu_c = π[s_, :] @ u_c

            μHist[t] = self.Vf[s](xprime[s])

            cHist[t], nHist[t], Bhist[t], THist[t] = c[s], n[s], x / Eu_c, T
            xHist[t], THist[t] = xprime[s], T[s]
        return [cHist, nHist, Bhist, THist, THist, μHist, sHist, xHist]


class BellmanEquation:
    '''
    Bellman equation for the continuation of the Lucas-Stokey Problem
    '''

    def __init__(self, model, xgrid, policies0, tol, maxiter=1000):

        self.β, self.π, self.G = model.β, model.π, model.G
        self.S = len(model.π)  # Number of states
        self.Θ, self.model, self.tol = model.Θ, model, tol
        self.maxiter = maxiter

        self.xbar = [min(xgrid), max(xgrid)]
        self.time_0 = False

        self.z0 = {}
        cf, nf, xprimef = policies0

        for s_ in range(self.S):
            for x in xgrid:
                self.z0[x, s_] = np.hstack([cf[s_, :](x),
                                            nf[s_, :](x),
```

(continues on next page)

(continued from previous page)

```
                                                    xprimef[s_, :](x),
                                                    np.zeros(self.S)])

        self.find_first_best()

    def find_first_best(self):
        '''
        Find the first best allocation
        '''
        model = self.model
        S, Θ, Uc, Un, G = self.S, self.Θ, model.Uc, model.Un, self.G

        def res(z):
            c = z[:S]
            n = z[S:]
            return np.hstack([Θ * Uc(c, n) + Un(c, n), Θ * n - c - G])

        res = root(res, np.full(2 * S, 0.5))
        if not res.success:
            raise Exception('Could not find first best')

        self.cFB = res.x[:S]
        self.nFB = res.x[S:]
        IFB = Uc(self.cFB, self.nFB) * self.cFB + \
            Un(self.cFB, self.nFB) * self.nFB

        self.xFB = np.linalg.solve(np.eye(S) - self.β * self.π, IFB)

        self.zFB = {}
        for s in range(S):
            self.zFB[s] = np.hstack(
                [self.cFB[s], self.nFB[s], self.π[s] @ self.xFB, 0.])

    def __call__(self, Vf):
        '''
        Given continuation value function next period return value function this
        period return T(V) and optimal policies
        '''
        if not self.time_0:
            def PF(x, s): return self.get_policies_time1(x, s, Vf)
        else:
            def PF(B_, s0): return self.get_policies_time0(B_, s0, Vf)
        return PF

    def get_policies_time1(self, x, s_, Vf):
        '''
        Finds the optimal policies
        '''
        model, β, Θ, G, S, π = self.model, self.β, self.Θ, self.G, self.S, self.π
        U, Uc, Un = model.U, model.Uc, model.Un

        def objf(z):
            c, n, xprime = z[:S], z[S:2 * S], z[2 * S:3 * S]

            Vprime = np.empty(S)
            for s in range(S):
```

(continues on next page)

(continued from previous page)

```
                Vprime[s] = Vf[s](xprime[s])

            return -π[s_] @ (U(c, n) + β * Vprime)

        def objf_prime(x):

            epsilon = 1e-7
            x0 = np.asfarray(x)
            f0 = np.atleast_1d(objf(x0))
            jac = np.zeros([len(x0), len(f0)])
            dx = np.zeros(len(x0))
            for i in range(len(x0)):
                dx[i] = epsilon
                jac[i] = (objf(x0+dx) - f0)/epsilon
                dx[i] = 0.0

            return jac.transpose()

        def cons(z):
            c, n, xprime, T = z[:S], z[S:2 * S], z[2 * S:3 * S], z[3 * S:]
            u_c = Uc(c, n)
            Eu_c = π[s_] @ u_c
            return np.hstack([
                x * u_c / Eu_c - u_c * (c - T) - Un(c, n) * n - β * xprime,
                Θ * n - c - G])

        if model.transfers:
            bounds = [(0., 100)] * S + [(0., 100)] * S + \
                [self.xbar] * S + [(0., 100.)] * S
        else:
            bounds = [(0., 100)] * S + [(0., 100)] * S + \
                [self.xbar] * S + [(0., 0.)] * S
        out, fx, _, imode, smode = fmin_slsqp(objf, self.z0[x, s_],
                                              f_eqcons=cons, bounds=bounds,
                                              fprime=objf_prime, full_output=True,
                                              iprint=0, acc=self.tol, iter=self.
↪maxiter)

        if imode > 0:
            raise Exception(smode)

        self.z0[x, s_] = out
        return np.hstack([-fx, out])

    def get_policies_time0(self, B_, s0, Vf):
        '''
        Finds the optimal policies
        '''
        model, β, Θ, G = self.model, self.β, self.Θ, self.G
        U, Uc, Un = model.U, model.Uc, model.Un

        def objf(z):
            c, n, xprime = z[:-1]

            return -(U(c, n) + β * Vf[s0](xprime))
```

(continues on next page)

(continued from previous page)

```
        def cons(z):
            c, n, xprime, T = z
            return np.hstack([
                -Uc(c, n) * (c - B_ - T) - Un(c, n) * n - β * xprime,
                (Θ * n - c - G)[s0]])

        if model.transfers:
            bounds = [(0., 100), (0., 100), self.xbar, (0., 100.)]
        else:
            bounds = [(0., 100), (0., 100), self.xbar, (0., 0.)]
        out, fx, _, imode, smode = fmin_slsqp(objf, self.zFB[s0], f_eqcons=cons,
                                              bounds=bounds, full_output=True,
                                              iprint=0)

        if imode > 0:
            raise Exception(smode)

        return np.hstack([-fx, out])
```

```
import numpy as np
from scipy.interpolate import UnivariateSpline


class interpolate_wrapper:

    def __init__(self, F):
        self.F = F

    def __getitem__(self, index):
        return interpolate_wrapper(np.asarray(self.F[index]))

    def reshape(self, *args):
        self.F = self.F.reshape(*args)
        return self

    def transpose(self):
        self.F = self.F.transpose()

    def __len__(self):
        return len(self.F)

    def __call__(self, xvec):
        x = np.atleast_1d(xvec)
        shape = self.F.shape
        if len(x) == 1:
            fhat = np.hstack([f(x) for f in self.F.flatten()])
            return fhat.reshape(shape)
        else:
            fhat = np.vstack([f(x) for f in self.F.flatten()])
            return fhat.reshape(np.hstack((shape, len(x))))


class interpolator_factory:

    def __init__(self, k, s):
```

(continues on next page)

(continued from previous page)

```
        self.k, self.s = k, s

    def __call__(self, xgrid, Fs):
        shape, m = Fs.shape[:-1], Fs.shape[-1]
        Fs = Fs.reshape((-1, m))
        F = []
        xgrid = np.sort(xgrid)  # Sort xgrid
        for Fhat in Fs:
            F.append(UnivariateSpline(xgrid, Fhat, k=self.k, s=self.s))
        return interpolate_wrapper(np.array(F).reshape(shape))


def fun_vstack(fun_list):

    Fs = [IW.F for IW in fun_list]
    return interpolate_wrapper(np.vstack(Fs))


def fun_hstack(fun_list):

    Fs = [IW.F for IW in fun_list]
    return interpolate_wrapper(np.hstack(Fs))


def simulate_markov(π, s_0, T):

    sHist = np.empty(T, dtype=int)
    sHist[0] = s_0
    S = len(π)
    for t in range(1, T):
        sHist[t] = np.random.choice(np.arange(S), p=π[sHist[t - 1]])

    return sHist
```

Next, we show the code that we use to generate a very long simulation starting from initial government debt equal to $-.5$.

Here is a graph of a long simulation of 102000 periods.

```
μ_grid = np.linspace(-0.09, 0.1, 100)

log_example = CRRAutility(π=np.full((3, 3), 1 / 3),
                          G=np.array([0.1, 0.2, .3]),
                          Θ=np.ones(3))

log_example.transfers = True        # Government can use transfers
log_sequential = SequentialAllocation(log_example)  # Solve sequential problem
log_bellman = RecursiveAllocationAMSS(log_example, μ_grid,
                                      tol=1e-12, tol_diff=1e-10)




T = 102000  # Set T to 102000 periods

sim_seq_long = log_sequential.simulate(0.5, 0, T)
sHist_long = sim_seq_long[-3]
sim_bel_long = log_bellman.simulate(0.5, 0, T, sHist_long)
```

(continues on next page)

(continued from previous page)

```
titles = ['Government Debt', 'Tax Rate']

fig, axes = plt.subplots(2, 1, figsize=(10, 8))

for ax, title, id in zip(axes.flatten(), titles, [2, 3]):
    ax.plot(sim_seq_long[id], '-k', sim_bel_long[id], '-.b', alpha=0.5)
    ax.set(title=title)
    ax.grid()

axes[0].legend(('Complete Markets', 'Incomplete Markets'))
plt.tight_layout()
plt.show()
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.11/site-packages/scipy/optimize/
↪_optimize.py:404: RuntimeWarning: Values in x were outside bounds during a␣
↪minimize step, clipping to bounds
  warnings.warn("Values in x were outside bounds during a "
```

```
/tmp/ipykernel_2277/108196118.py:24: RuntimeWarning: divide by zero encountered in␣
↪reciprocal
  U = (c**(1 - σ) - 1) / (1 - σ)
/tmp/ipykernel_2277/108196118.py:29: RuntimeWarning: divide by zero encountered in␣
↪power
  return c**(-self.σ)
/tmp/ipykernel_2277/1277371586.py:249: RuntimeWarning: invalid value encountered␣
↪in divide
  x * u_c / Eu_c - u_c * (c - T) - Un(c, n) * n - β * xprime,
```

```
/tmp/ipykernel_2277/1277371586.py:249: RuntimeWarning: invalid value encountered␣
↪in multiply
  x * u_c / Eu_c - u_c * (c - T) - Un(c, n) * n - β * xprime,
```

```
0.038266353387659546
```

```
0.0015144378246632448
```

```
0.0013387575049931865
```

```
0.0011833202400662248
```

```
0.0010600307116134505
```

```
0.0009506620324908642
```

```
0.0008518776517238551
```

```
0.0007625857031042564
```

```
0.0006819563061669217
```

```
0.0006094002927240671
```

```
0.0005443007356805235
```

```
0.00048599500343956094
```

```
0.0004338395935928358
```

```
0.00038722730865154364
```

```
0.00034559541217657187
```

```
0.00030842870645340995
```

```
0.00027525901875688697
```

```
0.0002456631291987257
```

```
0.00021925988533911457
```

```
0.0001957069581927878
```

```
0.00017469751641633328
```

```
0.00015595697131045533
```

```
0.00013923987965580473
```

```
0.0001243270476244632
```

```
0.00011102285954170156
```

```
9.915283206080047e-05
```

```
8.856139177373994e-05
```

```
7.910986485356134e-05
```

```
7.067466534026614e-05
```

```
6.314566737649043e-05
```

```
5.6424746008715835e-05
```

```
5.04244714230645e-05
```

```
4.5066942129829506e-05
```

```
4.028274354582181e-05
```

```
3.601001917066026e-05
```

```
3.219364287744318e-05
```

```
2.878448158073308e-05
```

```
2.5738738366349524e-05
```

```
2.3017369974638877e-05
```

```
2.0585562530972924e-05
```

```
1.8412273759209572e-05
```

```
1.6470096733078585e-05
```

```
1.4734148603737835e-05
```

```
1.3182214255360329e-05
```

```
1.1794654716176686e-05
```

```
1.0553942898779478e-05
```

```
9.444436197515114e-06
```

```
8.452171093491432e-06
```

```
7.564681603048501e-06
```

```
6.770836606096674e-06
```

```
6.060699172057158e-06
```

```
5.4253876343226e-06
```

```
4.856977544060761e-06
```

```
4.348382732427091e-06
```

```
3.893276456302588e-06
```

```
3.4860028420224977e-06
```

```
3.1215110784890745e-06
```

```
2.7952840260155024e-06
```

```
2.503284254157189e-06
```

```
2.241904747465382e-06
```

```
2.0079209145832687e-06
```

```
1.7984472260187192e-06
```

```
1.610904141295967e-06
```

```
1.4429883256895489e-06
```

```
1.2926354365994746e-06
```

```
1.1580011940576491e-06
```

```
1.0374362190402233e-06
```

```
9.294651286343194e-07
```

```
8.327660623755013e-07
```

```
7.461585686381671e-07
```

```
6.68586648784756e-07
```

```
5.991017296865946e-07
```

```
5.368606502407216e-07
```

```
4.811037017633464e-07
```

```
4.3115434615062044e-07
```

```
3.8640500348483447e-07
```

```
3.4631274740294855e-07
```

```
3.1039146715661056e-07
```

```
2.782060642970499e-07
```

```
2.493665449692665e-07
```

```
2.235241683944158e-07
```

```
2.0036660045892633e-07
```

```
1.796140357496926e-07
```

```
1.610161234596195e-07
```

```
1.4434845857135709e-07
```

```
1.29410194199688e-07
```

```
1.1602140686642469e-07
```

```
1.04020962175412e-07
```

```
9.326451087350253e-08
```

```
8.362279520562034e-08
```

```
7.49799979528415e-08
```

```
6.723237810210067e-08
```

```
6.028699653820159e-08
```

```
5.4060588066801066e-08
```

```
4.847855517381241e-08
```

```
4.347405660607874e-08
```

```
3.898720608840536e-08
```

```
3.496434157686767e-08
```

```
3.135737680533792e-08
```

```
2.8123222131646282e-08
```

```
2.5223262308472423e-08
```

```
2.2622892571432625e-08
```

```
2.0291098813063476e-08
```

```
1.820008555543109e-08
```

```
1.6324938418135388e-08
```

```
1.4643330672610771e-08
```

```
1.3135245110419445e-08
```

```
1.178274355586975e-08
```

```
1.0569743803546048e-08
```

```
9.48183058751907e-09
```

```
8.506079544395937e-09
```

```
7.630907318911004e-09
```

```
6.845926774203295e-09
```

```
6.141826797773109e-09
```

```
5.510259068441386e-09
```

```
4.943738281315066e-09
```

```
4.435554859709816e-09
```

```
3.979736766026741e-09
```

```
3.5708317622814044e-09
```

```
3.2040044801866767e-09
```

```
2.874916539533131e-09
```

```
2.579680212253616e-09
```

```
2.3148068175021918e-09
```

```
2.077170148801081e-09
```

```
1.8639635474165993e-09
```

```
1.6726726276855955e-09
```

```
1.5010414936033808e-09
```

```
1.3470449992327086e-09
```

```
1.2088698423920761e-09
```

```
1.0848882197883804e-09
```

```
9.736395405805598e-10
```

```
8.738135346705384e-10
```

```
7.842367703299733e-10
```

```
7.03855297579472e-10
```

```
6.317225605423774e-10
```

```
5.669925787732949e-10
```

```
5.089032105148693e-10
```

```
4.5677367318159076e-10
```

```
4.0999013116379334e-10
```

```
3.680044560697966e-10
```

```
3.3032415368561477e-10
```

```
2.96506010211222e-10
```

```
2.6615516244191936e-10
```

```
2.389139399385772e-10
```

```
2.144649644252697e-10
```

```
1.9252092177853976e-10
```

```
1.7282471699749249e-10
```

```
1.551454449875162e-10
```

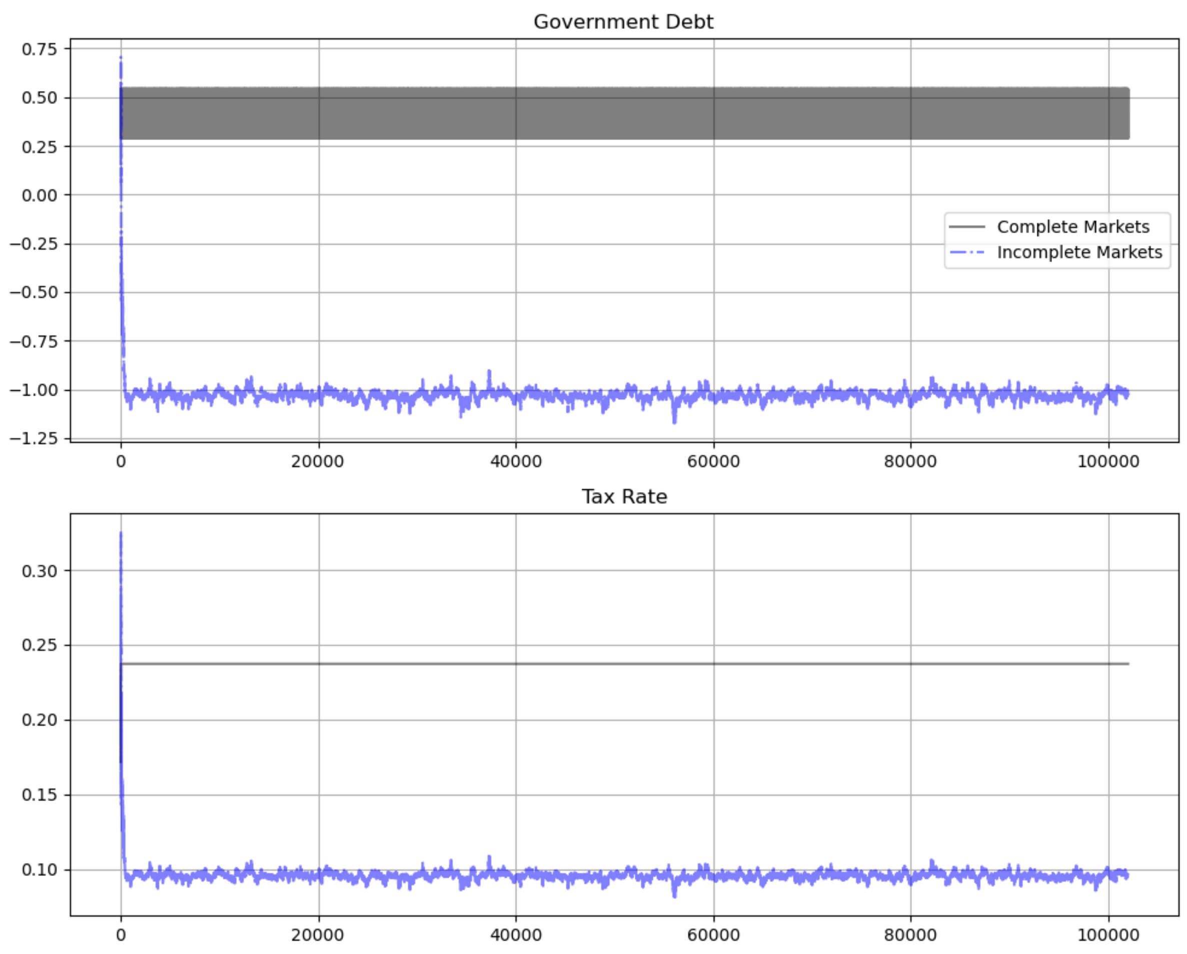
```
1.3927730577138407e-10
```

```
1.2503449048385917e-10
```

```
1.1224916676355658e-10
```

```
1.0077318342152794e-10
```

```
9.047094182757221e-11
```



The long simulation apparently indicates eventual convergence to an ergodic distribution.

It takes about 1000 periods to reach the ergodic distribution – an outcome that is forecast by approximations to rates of convergence that appear in BEGS [BEGS17] and that we discuss in *Fluctuating Interest Rates Deliver Fiscal Insurance*.

Let's discard the first 2000 observations of the simulation and construct the histogram of the par value of government debt.

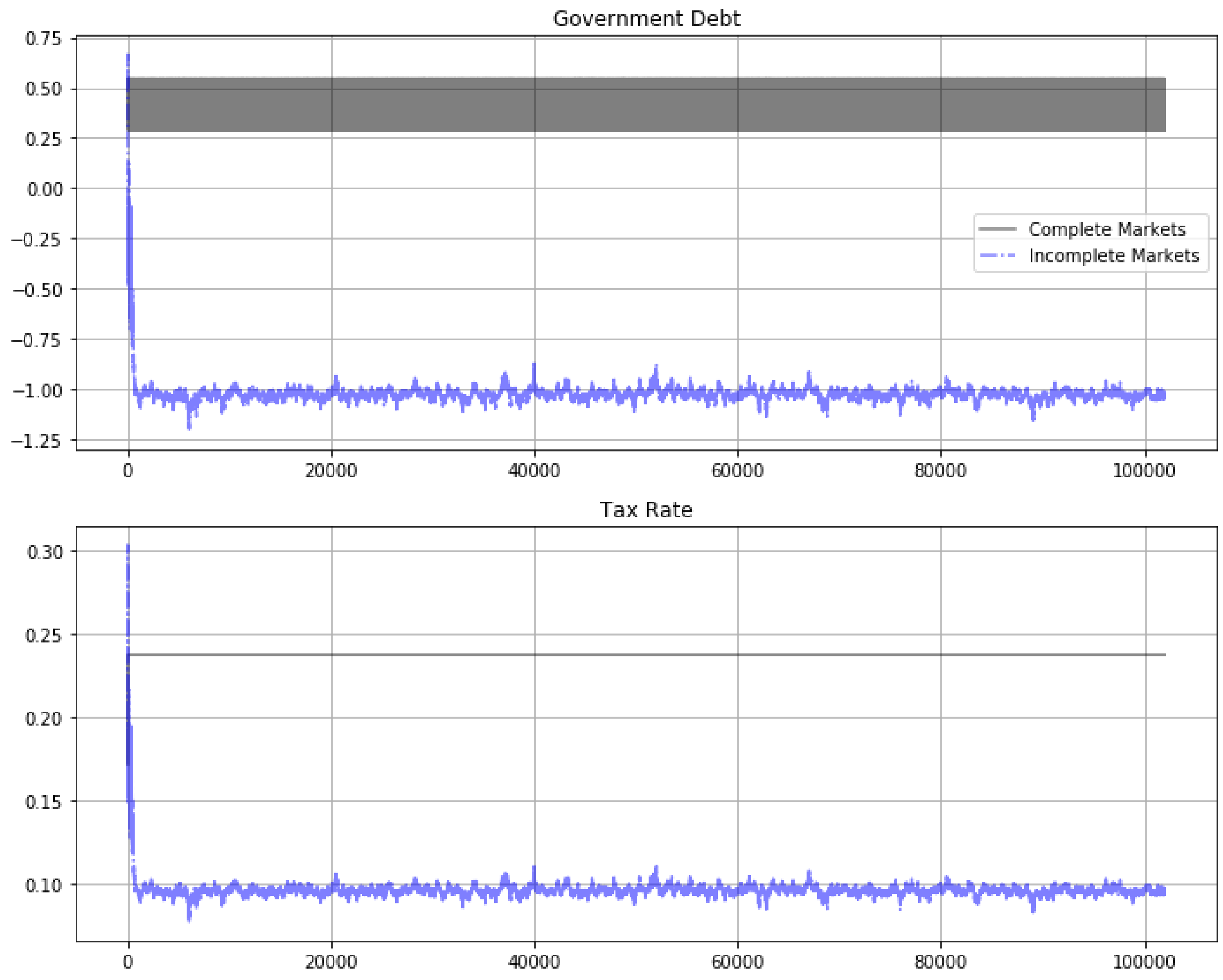
We obtain the following graph for the histogram of the last 100,000 observations on the par value of government debt.

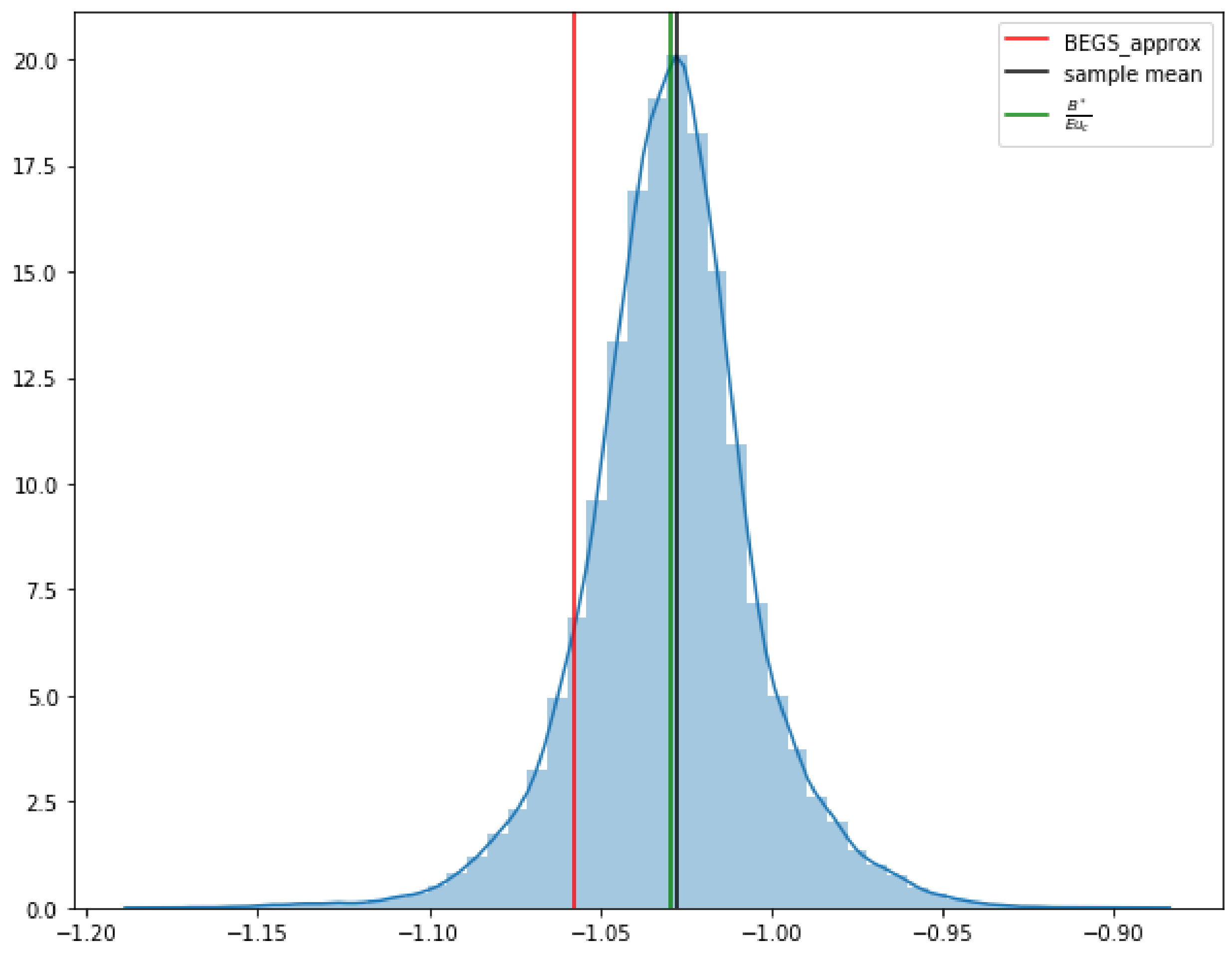
The black vertical line denotes the sample mean for the last 100,000 observations included in the histogram; the green vertical line denotes the value of $\frac{\mathcal{B}^*}{Eu_c}$, associated with a sample from our approximation to the ergodic distribution where $\mathcal{B}^*$ is a regression coefficient to be described below; the red vertical line denotes an approximation by [BEGS17] to the mean of the ergodic distribution that can be computed **before** the ergodic distribution has been approximated, as described below.

Before moving on to discuss the histogram and the vertical lines approximating the ergodic mean of government debt in more detail, the following graphs show government debt and taxes early in the simulation, for periods 1-100 and 101 to 200 respectively.

```
titles = ['Government Debt', 'Tax Rate']

fig, axes = plt.subplots(4, 1, figsize=(10, 15))

for i, id in enumerate([2, 3]):
```

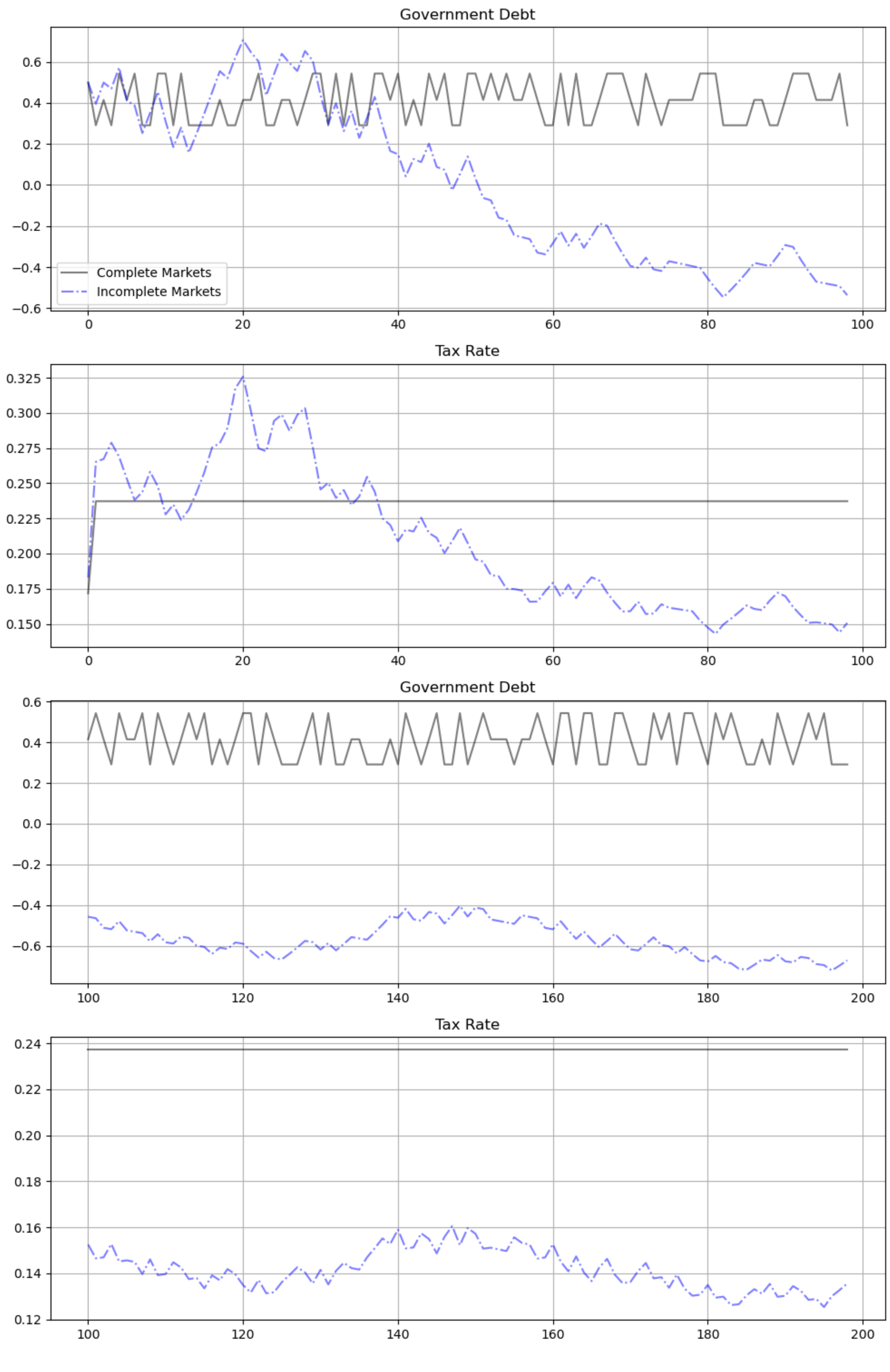
(continues on next page)

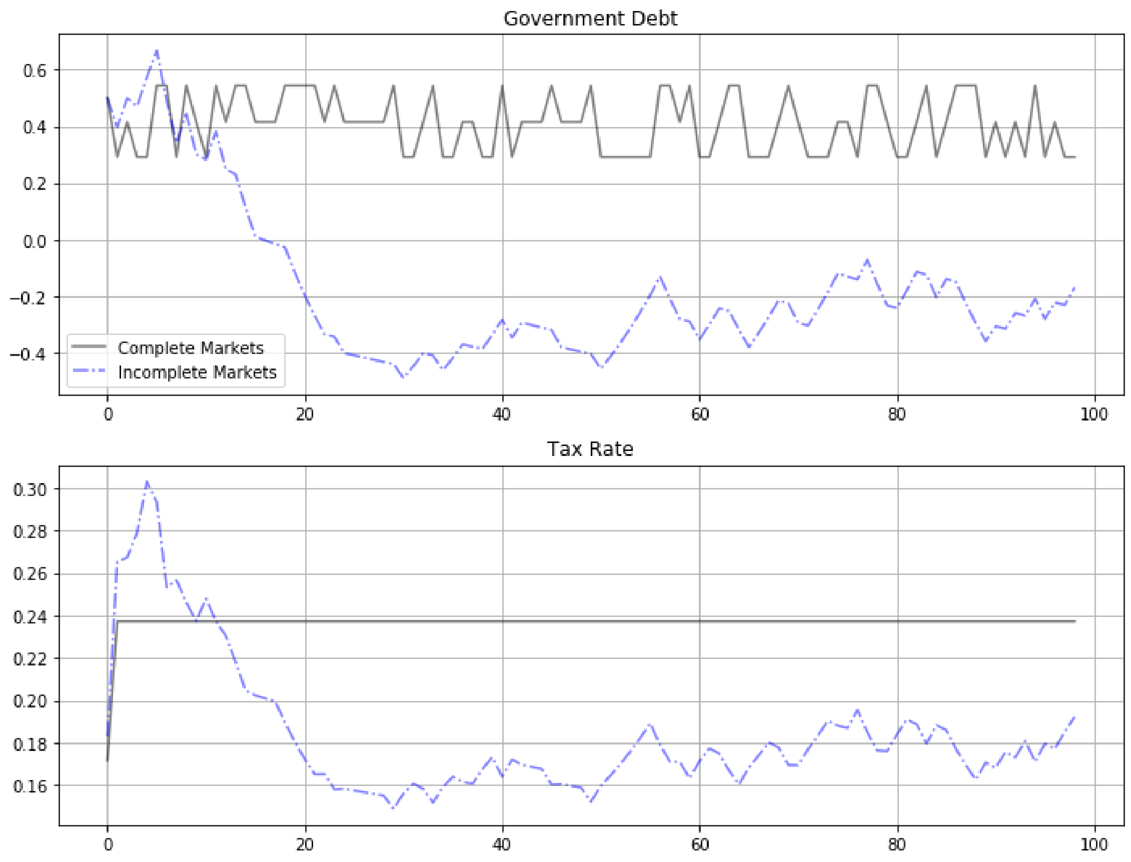

Government Debt
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
0
20000
40000
60000
80000
100000
Complete Markets
Incomplete Markets
Tax Rate
0.30
0.25
0.20
0.15
0.10
0
20000
40000
60000
80000
100000

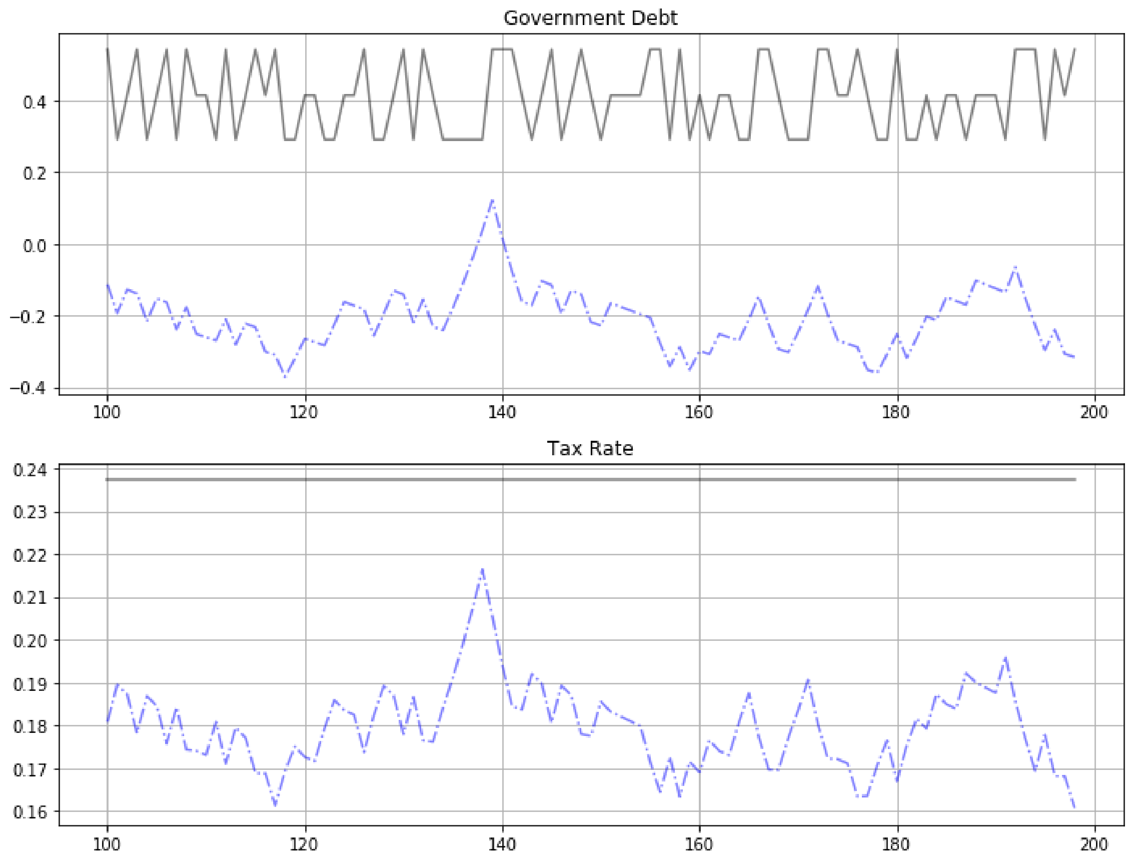

BEGS_approx
sample mean
$\frac{B^*}{Eu_c}$
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−1.20
−1.15
−1.10
−1.05
−1.00
−0.95
−0.90

(continued from previous page)

```
    axes[i].plot(sim_seq_long[id][:99], '-k', sim_bel_long[id][:99],
                 '-.b', alpha=0.5)
    axes[i+2].plot(range(100, 199), sim_seq_long[id][100:199], '-k',
                   range(100, 199), sim_bel_long[id][100:199], '-.b',
                   alpha=0.5)
    axes[i].set(title=titles[i])
    axes[i+2].set(title=titles[i])
    axes[i].grid()
    axes[i+2].grid()

axes[0].legend(('Complete Markets', 'Incomplete Markets'))
plt.tight_layout()
plt.show()
```


Government Debt
Complete Markets
Incomplete Markets
Tax Rate
Government Debt
Tax Rate


Government Debt
0.6
0.4
0.2
0.0
−0.2
−0.4
Complete Markets
Incomplete Markets
0
20
40
60
80
100
Tax Rate
0.30
0.28
0.26
0.24
0.22
0.20
0.18
0.16
0
20
40
60
80
100


Government Debt
0.4
0.2
0.0
−0.2
−0.4
100
120
140
160
180
200
Tax Rate
0.24
0.23
0.22
0.21
0.20
0.19
0.18
0.17
0.16
100
120
140
160
180
200

For the short samples early in our simulated sample of 102,000 observations, fluctuations in government debt and the tax rate conceal the weak but inexorable force that the Ramsey planner puts into both series driving them toward ergodic marginal distributions that are far from these early observations

- early observations are more influenced by the initial value of the par value of government debt than by the ergodic mean of the par value of government debt
- much later observations are more influenced by the ergodic mean and are independent of the par value of initial government debt

7.4 Asymptotic Mean and Rate of Convergence

We apply the results of BEGS [BEGS17] to interpret

- the mean of the ergodic distribution of government debt
- the rate of convergence to the ergodic distribution from an arbitrary initial government debt

We begin by computing objects required by the theory of section III.i of BEGS [BEGS17].

As in *Fiscal Insurance via Fluctuating Interest Rates*, we recall that BEGS [BEGS17] used a particular notation to represent what we can regard as their generalization of an AMSS model.

We introduce some of the [BEGS17] notation so that readers can quickly relate notation that appears in key BEGS formulas to the notation that we have used in previous lectures *here* and *here*.

BEGS work with objects $B_t, \mathcal{B}_t, \mathcal{R}_t, \mathcal{X}_t$ that are related to notation that we used in earlier lectures by

$$
\begin{aligned}
\mathcal{R}_t &= \frac{u_{c,t}}{u_{c,t-1}} R_{t-1} = \frac{u_{c,t}}{\beta E_{t-1} u_{c,t}} \\
B_t &= \frac{b_{t+1}(s^t)}{R_t(s^t)} \\
b_t(s^{t-1}) &= \mathcal{R}_{t-1} B_{t-1} \\
\mathcal{B}_t &= u_{c,t} B_t = (\beta E_t u_{c,t+1}) b_{t+1}(s^t) \\
\mathcal{X}_t &= u_{c,t}[g_t - \tau_t n_t]
\end{aligned}
$$

BEGS [BEGS17] call $\mathcal{X}_t$ the **effective** government deficit and $\mathcal{B}_t$ the **effective** government debt.

Equation (44) of [BEGS17] expresses the time t state s government budget constraint as

$$
\mathcal{B}(s) = \mathcal{R}_\tau(s, s_-)\mathcal{B}_- + \mathcal{X}_\tau(s) \tag{7.1}
$$

where the dependence on τ is meant to remind us that these objects depend on the tax rate; s_- is last period's Markov state.

BEGS interpret random variations in the right side of (7.1) as **fiscal risks** generated by

- interest-rate-driven fluctuations in time t effective payments due on the government portfolio, namely, $\mathcal{R}_\tau(s, s_-)\mathcal{B}_-$, and
- fluctuations in the effective government deficit $\mathcal{X}_t$

7.4.1 Asymptotic Mean

BEGS give conditions under which the ergodic mean of $\mathcal{B}_t$ is approximated by

$$\mathcal{B}^* = -\frac{\text{cov}^\infty(\mathcal{R}_\text{t}, \mathcal{X}_\text{t})}{\text{var}^\infty(\mathcal{R}_\text{t})} \tag{7.2}$$

where the superscript ∞ denotes a moment taken with respect to an ergodic distribution.

Formula (7.2) represents $\mathcal{B}^*$ as a regression coefficient of $\mathcal{X}_t$ on $\mathcal{R}_t$ in the ergodic distribution.

Regression coefficient $\mathcal{B}^*$ solves a variance-minimization problem:

$$\mathcal{B}^* = \text{argmin}_{\mathcal{B}} \text{var}^\infty(\mathcal{R}\mathcal{B} + \mathcal{X}) \tag{7.3}$$

The minimand in criterion (7.3) measures **fiscal risk** associated with a given tax-debt policy that appears on the right side of equation (7.1).

Expressing formula (7.2) in terms of our notation tells us that the ergodic mean of the par value b of government debt in the AMSS model should be approximately

$$\hat{b} = \frac{\mathcal{B}^*}{\beta E(E_t u_{c,t+1})} = \frac{\mathcal{B}^*}{\beta E(u_{c,t+1})} \tag{7.4}$$

where mathematical expectations are taken with respect to the ergodic distribution.

7.4.2 Rate of Convergence

BEGS also derive the following approximation to the rate of convergence to $\mathcal{B}^*$ from an arbitrary initial condition.

$$\frac{E_t(\mathcal{B}_{t+1} - \mathcal{B}^*)}{(\mathcal{B}_t - \mathcal{B}^*)} \approx \frac{1}{1 + \beta^2 \text{var}^\infty(\mathcal{R})} \tag{7.5}$$

(See the equation above equation (47) in BEGS [BEGS17])

7.4.3 More Advanced Topic

The remainder of this lecture is about technical material based on formulas from BEGS [BEGS17].

The topic involves interpreting and extending formula (7.3) for the ergodic mean $\mathcal{B}^*$.

7.4.4 Chicken and Egg

Notice how attributes of the ergodic distribution for $\mathcal{B}_t$ appear on the right side of formula (7.3) for approximating the ergodic mean via $\mathcal{B}^*$.

Therefor, formula (7.3) is not useful for estimating the mean of the ergodic **in advance** of actually approximating the ergodic distribution.

- we need to know the ergodic distribution to compute the right side of formula (7.3)

So the primary use of equation (7.3) is how it **confirms** that the ergodic distribution solves a **fiscal-risk minimization problem**.

As an example, notice how we used the formula for the mean of $\mathcal{B}$ in the ergodic distribution of the special AMSS economy in *Fiscal Insurance via Fluctuating Interest Rates*

- **first** we computed the ergodic distribution using a reverse-engineering construction
- **then** we verified that $\mathcal{B}^*$ agrees with the mean of that distribution

7.4.5 Approximating the Ergodic Mean

BEGS also [BEGS17] propose an approximation to $\mathcal{B}^*$ that can be computed **without** first approximating the ergodic distribution.

To construct the BEGS approximation to $\mathcal{B}^*$, we just follow steps set forth on pages 648 - 650 of section III.D of [BEGS17]

- notation in BEGS might be confusing at first sight, so it is important to stare and digest before computing
- there are also some sign errors in the [BEGS17] text that we'll want to correct here

Here is a step-by-step description of the BEGS [BEGS17] approximation procedure.

7.4.6 Step by Step

Step 1: For a given τ we compute a vector of values $c_\tau(s), s = 1, 2, \dots, S$ that satisfy

$$(1-\tau)c_\tau(s)^{-\sigma} - (c_\tau(s) + g(s))^\gamma = 0$$

This is a nonlinear equation to be solved for $c_\tau(s), s = 1, \dots, S$.

$S = 3$ in our case, but we'll write code for a general integer S.

Typo alert: Please note that there is a sign error in equation (42) of BEGS [BEGS17] – it should be a **minus** rather than a **plus** in the middle.

- We have made the appropriate correction in the above equation.

Step 2: Knowing $c_\tau(s), s = 1, \dots, S$ for a given τ, we want to compute the random variables

$$\mathcal{R}_\tau(s) = \frac{c_\tau(s)^{-\sigma}}{\beta \sum_{s'=1}^{S} c_\tau(s')^{-\sigma}\pi(s')}$$

and

$$\mathcal{X}_\tau(s) = (c_\tau(s) + g(s))^{1+\gamma} - c_\tau(s)^{1-\sigma}$$

each for $s = 1, \dots, S$.

BEGS call $\mathcal{R}_\tau(s)$ the **effective return** on risk-free debt and they call $\mathcal{X}_\tau(s)$ the **effective government deficit**.

Step 3: With the preceding objects in hand, for a given $\mathcal{B}$, we seek a τ that satisfies

$$\mathcal{B} = -\frac{\beta}{1-\beta} E\mathcal{X}_\tau \equiv -\frac{\beta}{1-\beta}\sum_s \mathcal{X}_\tau(s)\pi(s)$$

This equation says that at a constant discount factor β, equivalent government debt $\mathcal{B}$ equals the present value of the mean effective government **surplus**.

Another typo alert: there is a sign error in equation (46) of BEGS [BEGS17] –the left side should be multiplied by -1.

- We have made this correction in the above equation.

For a given $\mathcal{B}$, let a τ that solves the above equation be called $\tau(\mathcal{B})$.

We'll use a Python root solver to find a τ that solves this equation for a given $\mathcal{B}$.

We'll use this function to induce a function $\tau(\mathcal{B})$.

Step 4: With a Python program that computes $\tau(\mathcal{B})$ in hand, next we write a Python function to compute the random variable.

$$J(\mathcal{B})(s) = \mathcal{R}_{\tau(\mathcal{B})}(s)\mathcal{B} + \mathcal{X}_{\tau(\mathcal{B})}(s), \quad s = 1, \dots, S$$

Step 5: Now that we have a way to compute the random variable $J(\mathcal{B})(s), s = 1, \dots, S$, via a composition of Python functions, we can use the population variance function that we defined in the code above to construct a function $\text{var}(J(\mathcal{B}))$.

We put $\text{var}(J(\mathcal{B}))$ into a Python function minimizer and compute

$$\mathcal{B}^* = \text{argmin}_{\mathcal{B}} \text{var}(J(\mathcal{B}))$$

Step 6: Next we take the minimizer $\mathcal{B}^*$ and the Python functions for computing means and variances and compute

$$\text{rate} = \frac{1}{1 + \beta^2 \text{var}(\mathcal{R}_{\tau(\mathcal{B}^*)})}$$

Ultimate outputs of this string of calculations are two scalars

$$(\mathcal{B}^*, \text{rate})$$

Step 7: Compute the divisor

$$div = \beta E u_{c,t+1}$$

and then compute the mean of the par value of government debt in the AMSS model

$$\hat{b} = \frac{\mathcal{B}^*}{div}$$

In the two-Markov-state AMSS economy in *Fiscal Insurance via Fluctuating Interest Rates*, $E_t u_{c,t+1} = E u_{c,t+1}$ in the ergodic distribution.

We have confirmed that this formula very accurately describes a **constant** par value of government debt that

- supports full fiscal insurance via fluctuating interest parameters, and
- is the limit of government debt as $t \to +\infty$

In the three-Markov-state economy of this lecture, the par value of government debt fluctuates in a history-dependent way even asymptotically.

In this economy, $\hat{b}$ given by the above formula approximates the mean of the ergodic distribution of the par value of government debt

so while the approximation circumvents the chicken and egg problem that surrounds
the much better approximation associated with the green vertical line, it does so by enlarging the approximation error

- $\hat{b}$ is represented by the red vertical line plotted in the histogram of the last 100,000 observations of our simulation of the par value of government debt plotted above
- the approximation is fairly accurate but not perfect

7.4.7 Execution

Now let's move on to compute things step by step.

Step 1

```
u = CRRAutility(π=np.full((3, 3), 1 / 3),
                G=np.array([0.1, 0.2, .3]),
                Θ=np.ones(3))

τ = 0.05           # Initial guess of τ (to displays calcs along the way)
S = len(u.G)       # Number of states

def solve_c(c, τ, u):
    return (1 - τ) * c**(-u.σ) - (c + u.G)**u.γ

# .x returns the result from root
c = root(solve_c, np.ones(S), args=(τ, u)).x
c
```

```
array([0.93852387, 0.89231015, 0.84858872])
```

```
root(solve_c, np.ones(S), args=(τ, u))
```

```
 message: The solution converged.
 success: True
  status: 1
     fun: [ 5.618e-10 -4.769e-10  1.175e-11]
       x: [ 9.385e-01  8.923e-01  8.486e-01]
    nfev: 11
    fjac: [[-9.999e-01 -4.954e-03 -1.261e-02]
           [-5.156e-03  9.999e-01  1.610e-02]
           [-1.253e-02 -1.616e-02  9.998e-01]]
       r: [ 4.269e+00  8.685e-02 -6.301e-02 -4.713e+00 -7.433e-02
           -5.508e+00]
     qtf: [ 1.556e-08  1.283e-08  7.899e-11]
```

Step 2

```
n = c + u.G   # Compute labor supply
```

7.4.8 Note about Code

Remember that in our code π is a 3×3 transition matrix.

But because we are studying an IID case, π has identical rows and we need only to compute objects for one row of π.

This explains why at some places below we set $s = 0$ just to pick off the first row of π.

7.4.9 Running the code

Let's take the code out for a spin.

First, let's compute $\mathcal{R}$ and $\mathcal{X}$ according to our formulas

```
def compute_R_X(τ, u, s):
    c = root(solve_c, np.ones(S), args=(τ, u)).x  # Solve for vector of c's
    div = u.β * (u.Uc(c[0], n[0]) * u.π[s, 0]  \
                 +  u.Uc(c[1], n[1]) * u.π[s, 1] \
                 +  u.Uc(c[2], n[2]) * u.π[s, 2])
    R = c**(-u.σ) / (div)
    X = (c + u.G)**(1 + u.γ) - c**(1 - u.σ)
    return R, X
```

```
c**(-u.σ) @ u.π
```

```
array([1.25997521, 1.25997521, 1.25997521])
```

```
u.π
```

```
array([[0.33333333, 0.33333333, 0.33333333],
       [0.33333333, 0.33333333, 0.33333333],
       [0.33333333, 0.33333333, 0.33333333]])
```

We only want unconditional expectations because we are in an IID case.

So we'll set $s = 0$ and just pick off expectations associated with the first row of π

```
s = 0

R, X = compute_R_X(τ, u, s)
```

Let's look at the random variables $\mathcal{R}, \mathcal{X}$

```
R
```

```
array([1.00116313, 1.10755123, 1.22461897])
```

```
mean(R, s)
```

```
1.1111111111111112
```

```
X
```

```
array([0.05457803, 0.18259396, 0.33685546])
```

```
mean(X, s)
```

```
0.19134248445303795
```

```
X @ u.π
```

```
array([0.19134248, 0.19134248, 0.19134248])
```

Step 3

```
def solve_τ(τ, B, u, s):
    R, X = compute_R_X(τ, u, s)
    return ((u.β - 1) / u.β) * B - X @ u.π[s]
```

Note that B is a scalar.

Let's try out our method computing τ

```
s = 0
B = 1.0

τ = root(solve_τ, .1, args=(B, u, s)).x[0]  # Very sensitive to initial value
τ
```

```
0.2740159773695818
```

In the above cell, B is fixed at 1 and τ is to be computed as a function of B.

Note that 0.2 is the initial value for τ in the root-finding algorithm.

Step 4

```
def min_J(B, u, s):
    # Very sensitive to initial value of τ
    τ = root(solve_τ, .5, args=(B, u, s)).x[0]
    R, X = compute_R_X(τ, u, s)
    return variance(R * B + X, s)
```

```
min_J(B, u, s)
```

```
0.035564405653720765
```

Step 6

```
B_star = minimize(min_J, .5, args=(u, s)).x[0]
B_star
```

```
-1.199483167941158
```

```
n = c + u.G  # Compute labor supply
```

```
div = u.β * (u.Uc(c[0], n[0]) * u.π[s, 0]  \
             +  u.Uc(c[1], n[1]) * u.π[s, 1] \
             +  u.Uc(c[2], n[2]) * u.π[s, 2])
```

```
B_hat = B_star/div
B_hat
```

```
-1.0577661126390971
```

```
τ_star = root(solve_τ, 0.05, args=(B_star, u, s)).x[0]
τ_star
```

```
0.09572916798461703
```

```
R_star, X_star = compute_R_X(τ_star, u, s)
R_star, X_star
```

```
(array([0.9998398 , 1.10746593, 1.2260276 ]),
 array([0.0020272 , 0.12464752, 0.27315299]))
```

```
rate = 1 / (1 + u.β**2 * variance(R_star, s))
rate
```

```
0.9931353432732218
```

```
root(solve_c, np.ones(S), args=(τ_star, u)).x
```

```
array([0.9264382 , 0.88027117, 0.83662635])
```

CHAPTER

EIGHT

COMPETITIVE EQUILIBRIA OF A MODEL OF CHANG

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install polytope
```

8.1 Overview

This lecture describes how Chang [Cha98] analyzed **competitive equilibria** and a best competitive equilibrium called a **Ramsey plan**.

He did this by

- characterizing a competitive equilibrium recursively in a way also employed in the *dynamic Stackelberg problems* and *Calvo model* lectures to pose Stackelberg problems in linear economies, and then
- appropriately adapting an argument of Abreu, Pearce, and Stachetti [APS90] to describe key features of the set of competitive equilibria

Roberto Chang [Cha98] chose a model of Calvo [Cal78] as a simple structure that conveys ideas that apply more broadly.

A textbook version of Chang's model appears in chapter 25 of [LS18].

This lecture and *Credible Government Policies in Chang Model* can be viewed as more sophisticated and complete treatments of the topics discussed in *Ramsey plans, time inconsistency, sustainable plans*.

Both this lecture and *Credible Government Policies in Chang Model* make extensive use of an idea to which we apply the nickname **dynamic programming squared**.

In dynamic programming squared problems there are typically two interrelated Bellman equations

- A Bellman equation for a set of agents or followers with value or value function v_a.
- A Bellman equation for a principal or Ramsey planner or Stackelberg leader with value or value function v_p in which v_a appears as an argument.

We encountered problems with this structure in *dynamic Stackelberg problems*, *optimal taxation with state-contingent debt*, and other lectures.

We'll start with some standard imports:

```
import numpy as np
import polytope
import matplotlib.pyplot as plt
%matplotlib inline
```

```
`polytope` failed to import `cvxopt.glpk`.
```

```
will use `scipy.optimize.linprog`
```

8.1.1 The Setting

First, we introduce some notation.

For a sequence of scalars $\vec{z} \equiv \{z_t\}_{t=0}^{\infty}$, let $\vec{z}^t = (z_0, \ldots, z_t)$, $\vec{z}_t = (z_t, z_{t+1}, \ldots)$.

An infinitely lived representative agent and an infinitely lived government exist at dates $t = 0, 1, \ldots$.

The objects in play are

- an initial quantity M_{-1} of nominal money holdings
- a sequence of inverse money growth rates $\vec{h}$ and an associated sequence of nominal money holdings $\vec{M}$
- a sequence of values of money $\vec{q}$
- a sequence of real money holdings $\vec{m}$
- a sequence of total tax collections $\vec{x}$
- a sequence of per capita rates of consumption $\vec{c}$
- a sequence of per capita incomes $\vec{y}$

A benevolent government chooses sequences $(\vec{M}, \vec{h}, \vec{x})$ subject to a sequence of budget constraints and other constraints imposed by competitive equilibrium.

Given tax collection and price of money sequences, a representative household chooses sequences $(\vec{c}, \vec{m})$ of consumption and real balances.

In competitive equilibrium, the price of money sequence $\vec{q}$ clears markets, thereby reconciling decisions of the government and the representative household.

Chang adopts a version of a model that [Cal78] designed to exhibit time-inconsistency of a Ramsey policy in a simple and transparent setting.

By influencing the representative household's expectations, government actions at time t affect components of household utilities for periods s before t.

When setting a path for monetary expansion rates, the government takes into account how the household's anticipations of the government's future actions affect the household's current decisions.

The ultimate source of time inconsistency is that a time 0 Ramsey planner takes these effects into account in designing a plan of government actions for $t \geq 0$.

8.2 Setting

8.2.1 The Household's Problem

A representative household faces a nonnegative value of money sequence $\vec{q}$ and sequences $\vec{y}, \vec{x}$ of income and total tax collections, respectively.

The household chooses nonnegative sequences $\vec{c}, \vec{M}$ of consumption and nominal balances, respectively, to maximize

$$\sum_{t=0}^{\infty} \beta^t \left[u(c_t) + v(q_t M_t)\right] \tag{8.1}$$

subject to

$$q_t M_t \leq y_t + q_t M_{t-1} - c_t - x_t \tag{8.2}$$

and

$$q_t M_t \leq \bar{m} \tag{8.3}$$

Here q_t is the reciprocal of the price level at t, which we can also call the *value of money*.

Chang [Cha98] assumes that

- $u : \mathbb{R}_+ \rightarrow \mathbb{R}$ is twice continuously differentiable, strictly concave, and strictly increasing;
- $v : \mathbb{R}_+ \rightarrow \mathbb{R}$ is twice continuously differentiable and strictly concave;
- $u'(c)_{c \rightarrow 0} = \lim_{m \rightarrow 0} v'(m) = +\infty$;
- there is a finite level $m = m^f$ such that $v'(m^f) = 0$

The household carries real balances out of a period equal to $m_t = q_t M_t$.

Inequality (8.2) is the household's time t budget constraint.

It tells how real balances $q_t M_t$ carried out of period t depend on income, consumption, taxes, and real balances $q_t M_{t-1}$ carried into the period.

Equation (8.3) imposes an exogenous upper bound $\bar{m}$ on the household's choice of real balances, where $\bar{m} \geq m^f$.

8.2.2 Government

The government chooses a sequence of inverse money growth rates with time t component $h_t \equiv \frac{M_{t-1}}{M_t} \in \Pi \equiv [\underline{\pi}, \overline{\pi}]$, where $0 < \underline{\pi} < 1 < \frac{1}{\beta} \leq \overline{\pi}$.

The government faces a sequence of budget constraints with time t component

$$-x_t = q_t(M_t - M_{t-1})$$

which by using the definitions of m_t and h_t can also be expressed as

$$-x_t = m_t(1 - h_t) \tag{8.4}$$

The restrictions $m_t \in [0, \bar{m}]$ and $h_t \in \Pi$ evidently imply that $x_t \in X \equiv [(\underline{\pi} - 1)\bar{m}, (\overline{\pi} - 1)\bar{m}]$.

We define the set $E \equiv [0, \bar{m}] \times \Pi \times X$, so that we require that $(m, h, x) \in E$.

To represent the idea that taxes are distorting, Chang makes the following assumption about outcomes for per capita output:

$$y_t = f(x_t), \tag{8.5}$$

where $f : \mathbb{R} \to \mathbb{R}$ satisfies $f(x) > 0$, is twice continuously differentiable, $f''(x) < 0$, and $f(x) = f(-x)$ for all $x \in \mathbb{R}$, so that subsidies and taxes are equally distorting.

Calvo's and Chang's purpose is not to model the causes of tax distortions in any detail but simply to summarize the *outcome* of those distortions via the function $f(x)$.

A key part of the specification is that tax distortions are increasing in the absolute value of tax revenues.

Ramsey plan: A Ramsey plan is a competitive equilibrium that maximizes (8.1).

Within-period timing of decisions is as follows:

- first, the government chooses h_t and x_t;
- then given $\vec{q}$ and its expectations about future values of x and y's, the household chooses M_t and therefore m_t because $m_t = q_t M_t$;
- then output $y_t = f(x_t)$ is realized;
- finally $c_t = y_t$

This within-period timing confronts the government with choices framed by how the private sector wants to respond when the government takes time t actions that differ from what the private sector had expected.

This consideration will be important in lecture *credible government policies* when we study *credible government policies*.

The model is designed to focus on the intertemporal trade-offs between the welfare benefits of deflation and the welfare costs associated with the high tax collections required to retire money at a rate that delivers deflation.

A benevolent time 0 government can promote utility generating increases in real balances only by imposing sufficiently large distorting tax collections.

To promote the welfare increasing effects of high real balances, the government wants to induce *gradual deflation*.

8.2.3 Household's Problem

Given M_{-1} and $\{q_t\}_{t=0}^{\infty}$, the household's problem is

$$\begin{aligned} \mathcal{L} = \max_{\vec{c}, \vec{M}} \min_{\vec{\lambda}, \vec{\mu}} \sum_{t=0}^{\infty} \beta^t \{ & u(c_t) + v(M_t q_t) + \lambda_t [y_t - c_t - x_t + q_t M_{t-1} - q_t M_t] \\ & + \mu_t [\bar{m} - q_t M_t] \} \end{aligned}$$

First-order conditions with respect to c_t and M_t, respectively, are

$$\begin{aligned} u'(c_t) &= \lambda_t \\ q_t [u'(c_t) - v'(M_t q_t)] &\leq \beta u'(c_{t+1}) q_{t+1}, \quad = \text{ if } M_t q_t < \bar{m} \end{aligned}$$

The last equation expresses Karush-Kuhn-Tucker complementary slackness conditions (see here).

These insist that the inequality is an equality at an interior solution for M_t.

Using $h_t = \frac{M_{t-1}}{M_t}$ and $q_t = \frac{m_t}{M_t}$ in these first-order conditions and rearranging implies

$$m_t [u'(c_t) - v'(m_t)] \leq \beta u'(f(x_{t+1})) m_{t+1} h_{t+1}, \quad = \text{ if } m_t < \bar{m} \tag{8.6}$$

Define the following key variable

$$\theta_{t+1} \equiv u'(f(x_{t+1}))m_{t+1}h_{t+1} \tag{8.7}$$

This is real money balances at time $t+1$ measured in units of marginal utility, which Chang refers to as 'the marginal utility of real balances'.

From the standpoint of the household at time t, equation (8.7) shows that θ_{t+1} intermediates the influences of $(\vec{x}_{t+1}, \vec{m}_{t+1})$ on the household's choice of real balances m_t.

By "intermediates" we mean that the future paths $(\vec{x}_{t+1}, \vec{m}_{t+1})$ influence m_t entirely through their effects on the scalar θ_{t+1}.

The observation that the one dimensional promised marginal utility of real balances θ_{t+1} functions in this way is an important step in constructing a class of competitive equilibria that have a recursive representation.

A closely related observation pervaded the analysis of Stackelberg plans in lecture *dynamic Stackelberg problems*.

8.3 Competitive Equilibrium

Definition:

- A *government policy* is a pair of sequences $(\vec{h}, \vec{x})$ where $h_t \in \Pi \ \forall t \geq 0$.
- A *price system* is a nonnegative value of money sequence $\vec{q}$.
- An *allocation* is a triple of nonnegative sequences $(\vec{c}, \vec{m}, \vec{y})$.

It is required that time t components $(m_t, x_t, h_t) \in E$.

Definition:

Given M_{-1}, a government policy $(\vec{h}, \vec{x})$, price system $\vec{q}$, and allocation $(\vec{c}, \vec{m}, \vec{y})$ are said to be a *competitive equilibrium* if

- $m_t = q_t M_t$ and $y_t = f(x_t)$.
- The government budget constraint is satisfied.
- Given $\vec{q}, \vec{x}, \vec{y}$, $(\vec{c}, \vec{m})$ solves the household's problem.

8.4 Inventory of Objects in Play

Chang constructs the following objects

1. A set Ω of initial marginal utilities of money θ_0
   - Let Ω denote the set of initial promised marginal utilities of money θ_0 associated with competitive equilibria.
   - Chang exploits the fact that a competitive equilibrium consists of a first period outcome (h_0, m_0, x_0) and a continuation competitive equilibrium with marginal utility of money $\theta_1 \in \Omega$.
2. Competitive equilibria that have a recursive representation
   - A competitive equilibrium with a recursive representation consists of an initial θ_0 and a four-tuple of functions (h, m, x, Ψ) mapping θ into this period's (h, m, x) and next period's θ, respectively.

- A competitive equilibrium can be represented recursively by iterating on

$$
\begin{aligned}
h_t &= h(\theta_t) \\
m_t &= m(\theta_t) \\
x_t &= x(\theta_t) \\
\theta_{t+1} &= \Psi(\theta_t)
\end{aligned}
\tag{8.8}
$$

  starting from θ_0

  The range and domain of $\Psi(\cdot)$ are both Ω

3. A recursive representation of a Ramsey plan

   - A recursive representation of a Ramsey plan is a recursive competitive equilibrium $\theta_0, (h, m, x, \Psi)$ that, among all recursive competitive equilibria, maximizes $\sum_{t=0}^\infty \beta^t \left[u(c_t) + v(q_t M_t) \right]$.
   - The Ramsey planner chooses $\theta_0, (h, m, x, \Psi)$ from among the set of recursive competitive equilibria at time 0.
   - Iterations on the function Ψ determine subsequent θ_t's that summarize the aspects of the continuation competitive equilibria that influence the household's decisions.
   - At time 0, the Ramsey planner commits to this implied sequence $\{\theta_t\}_{t=0}^\infty$ and therefore to an associated sequence of continuation competitive equilibria.

4. A characterization of time-inconsistency of a Ramsey plan

   - Imagine that after a 'revolution' at time $t \geq 1$, a new Ramsey planner is given the opportunity to ignore history and solve a brand new Ramsey plan.
   - This new planner would want to reset the θ_t associated with the original Ramsey plan to θ_0.
   - The incentive to reinitialize θ_t associated with this revolution experiment indicates the time-inconsistency of the Ramsey plan.
   - By resetting θ to θ_0, the new planner avoids the costs at time t that the original Ramsey planner must pay to reap the beneficial effects that the original Ramsey plan for $s \geq t$ had achieved via its influence on the household's decisions for $s = 0, \dots, t-1$.

8.5 Analysis

A competitive equilibrium is a triple of sequences $(\vec{m}, \vec{x}, \vec{h}) \in E^\infty$ that satisfies (8.2), (8.3), and (8.6).

Chang works with a set of competitive equilibria defined as follows.

Definition: $CE = \{(\vec{m}, \vec{x}, \vec{h}) \in E^\infty$ such that (8.2), (8.3), and (8.6) are satisfied $\}$.

CE is not empty because there exists a competitive equilibrium with $h_t = 1$ for all $t \geq 1$, namely, an equilibrium with a constant money supply and constant price level.

Chang establishes that CE is also compact.

Chang makes the following key observation that combines ideas of Abreu, Pearce, and Stacchetti [APS90] with insights of Kydland and Prescott [KP80].

Proposition: The continuation of a competitive equilibrium is a competitive equilibrium.

That is, $(\vec{m}, \vec{x}, \vec{h}) \in CE$ implies that $(\vec{m}_t, \vec{x}_t, \vec{h}_t) \in CE \ \forall \ t \geq 1$.

(Lecture *dynamic Stackelberg problems* also used a version of this insight)

We can now state that a **Ramsey problem** is to

$$\max_{(\vec{m},\vec{x},\vec{h})\in E^\infty} \sum_{t=0}^{\infty} \beta^t \left[u(c_t) + v(m_t)\right]$$

subject to restrictions (8.2), (8.3), and (8.6).

Evidently, associated with any competitive equilibrium (m_0, x_0) is an implied value of $\theta_0 = u'(f(x_0))(m_0 + x_0)$.

To bring out a recursive structure inherent in the Ramsey problem, Chang defines the set

$$\Omega = \left\{\theta \in \mathbb{R} \text{ such that } \theta = u'(f(x_0))(m_0 + x_0) \text{ for some } (\vec{m},\vec{x},\vec{h}) \in CE\right\}$$

Equation (8.6) inherits from the household's Euler equation for money holdings the property that the value of m_0 consistent with the representative household's choices depends on $(\vec{h}_1, \vec{m}_1)$.

This dependence is captured in the definition above by making Ω be the set of first period values of θ_0 satisfying $\theta_0 = u'(f(x_0))(m_0 + x_0)$ for first period component (m_0, h_0) of competitive equilibrium sequences $(\vec{m},\vec{x},\vec{h})$.

Chang establishes that Ω is a nonempty and compact subset of $\mathbb{R}_+$.

Next Chang advances:

Definition: $\Gamma(\theta) = \{(\vec{m},\vec{x},\vec{h}) \in CE | \theta = u'(f(x_0))(m_0 + x_0)\}$.

Thus, $\Gamma(\theta)$ is the set of competitive equilibrium sequences $(\vec{m},\vec{x},\vec{h})$ whose first period components (m_0, h_0) deliver the prescribed value θ for first period marginal utility.

If we knew the sets $\Omega, \Gamma(\theta)$, we could use the following two-step procedure to find at least the *value* of the Ramsey outcome to the representative household

1. Find the indirect value function $w(\theta)$ defined as

$$w(\theta) = \max_{(\vec{m},\vec{x},\vec{h})\in\Gamma(\theta)} \sum_{t=0}^{\infty} \beta^t \left[u(f(x_t)) + v(m_t)\right]$$

2. Compute the value of the Ramsey outcome by solving $\max_{\theta\in\Omega} w(\theta)$.

Thus, Chang states the following

Proposition:

$w(\theta)$ satisfies the Bellman equation

$$w(\theta) = \max_{x,m,h,\theta'} \left\{u(f(x)) + v(m) + \beta w(\theta')\right\} \tag{8.9}$$

where maximization is subject to

$$(m, x, h) \in E \text{ and } \theta' \in \Omega \tag{8.10}$$

and

$$\theta = u'(f(x))(m + x) \tag{8.11}$$

and

$$-x = m(1 - h) \tag{8.12}$$

and

$$m \cdot [u'(f(x)) - v'(m)] \leq \beta\theta', \quad = \text{ if } m < \bar{m} \tag{8.13}$$

Before we use this proposition to recover a recursive representation of the Ramsey plan, note that the proposition relies on knowing the set Ω.

To find Ω, Chang uses the insights of Kydland and Prescott [KP80] together with a method based on the Abreu, Pearce, and Stacchetti [APS90] iteration to convergence on an operator B that maps continuation values into values.

We want an operator that maps a continuation θ into a current θ.

Chang lets Q be a nonempty, bounded subset of $\mathbb{R}$.

Elements of the set Q are taken to be candidate values for continuation marginal utilities.

Chang defines an operator

$$B(Q) = \theta \in \mathbb{R} \text{ such that there is } (m, x, h, \theta') \in E \times Q$$

such that (8.11), (8.12), and (8.13) hold.

Thus, $B(Q)$ is the set of first period θ's attainable with $(m, x, h) \in E$ and some $\theta' \in Q$.

Proposition:

1. $Q \subset B(Q)$ implies $B(Q) \subset \Omega$ ('self-generation').
2. $\Omega = B(\Omega)$ ('factorization').

The proposition characterizes Ω as the largest fixed point of B.

It is easy to establish that $B(Q)$ is a monotone operator.

This property allows Chang to compute Ω as the limit of iterations on B provided that iterations begin from a sufficiently large initial set.

8.5.1 Some Useful Notation

Let $\vec{h}^t = (h_0, h_1, \dots, h_t)$ denote a history of inverse money creation rates with time t component $h_t \in \Pi$.

A *government strategy* $\sigma = \{\sigma_t\}_{t=0}^\infty$ is a $\sigma_0 \in \Pi$ and for $t \geq 1$ a sequence of functions $\sigma_t : \Pi^{t-1} \to \Pi$.

Chang restricts the government's choice of strategies to the following space:

$$CE_\pi = \{\vec{h} \in \Pi^\infty : \text{ there is some } (\vec{m}, \vec{x}) \text{ such that } (\vec{m}, \vec{x}, \vec{h}) \in CE\}$$

In words, CE_π is the set of money growth sequences consistent with the existence of competitive equilibria.

Chang observes that CE_π is nonempty and compact.

Definition: σ is said to be *admissible* if for all $t \geq 1$ and after any history $\vec{h}^{t-1}$, the continuation $\vec{h}_t$ implied by σ belongs to CE_π.

Admissibility of σ means that anticipated policy choices associated with σ are consistent with the existence of competitive equilibria after each possible subsequent history.

After any history $\vec{h}^{t-1}$, admissibility restricts the government's choice in period t to the set

$$CE_\pi^0 = \{h \in \Pi : \text{there is } \vec{h} \in CE_\pi \text{ with } h = h_0\}$$

In words, CE_π^0 is the set of all first period money growth rates $h = h_0$, each of which is consistent with the existence of a sequence of money growth rates $\vec{h}$ starting from h_0 in the initial period and for which a competitive equilibrium exists.

Remark: $CE_\pi^0 = \{h \in \Pi : \text{ there is } (m, \theta') \in [0, \bar{m}] \times \Omega \text{ such that } mu'[f((h-1)m) - v'(m)] \leq \beta\theta' \text{ with equality if } m < \bar{m}\}$.

Definition: An *allocation rule* is a sequence of functions $\vec{\alpha} = \{\alpha_t\}_{t=0}^\infty$ such that $\alpha_t : \Pi^t \to [0, \bar{m}] \times X$.

Thus, the time t component of $\alpha_t(h^t)$ is a pair of functions $(m_t(h^t), x_t(h^t))$.

Definition: Given an admissible government strategy σ, an allocation rule α is called *competitive* if given any history $\vec{h}^{t-1}$ and $h_t \in CE_\pi^0$, the continuations of σ and α after $(\vec{h}^{t-1}, h_t)$ induce a competitive equilibrium sequence.

8.5.2 Another Operator

At this point it is convenient to introduce another operator that can be used to compute a Ramsey plan.

For computing a Ramsey plan, this operator is wasteful because it works with a state vector that is bigger than necessary.

We introduce this operator because it helps to prepare the way for Chang's operator called $\tilde{D}(Z)$ that we shall describe in lecture *credible government policies*.

It is also useful because a fixed point of the operator to be defined here provides a good guess for an initial set from which to initiate iterations on Chang's set-to-set operator $\tilde{D}(Z)$ to be described in lecture *credible government policies*.

Let S be the set of all pairs (w, θ) of competitive equilibrium values and associated initial marginal utilities.

Let W be a bounded set of *values* in $\mathbb{R}$.

Let Z be a nonempty subset of $W \times \Omega$.

Think of using pairs (w', θ') drawn from Z as candidate continuation value, θ pairs.

Define the operator

$$D(Z) = \Big\{ (w, \theta) : \text{there is } h \in CE_\pi^0$$

$$\text{and a four-tuple } (m(h), x(h), w'(h), \theta'(h)) \in [0, \bar{m}] \times X \times Z$$

such that

$$w = u(f(x(h))) + v(m(h)) + \beta w'(h) \tag{8.14}$$

$$\theta = u'(f(x(h)))(m(h) + x(h)) \tag{8.15}$$

$$x(h) = m(h)(h-1) \tag{8.16}$$

$$m(h)(u'(f(x(h))) - v'(m(h))) \leq \beta \theta'(h) \tag{8.17}$$

$$\text{with equality if } m(h) < \bar{m} \Big\}$$

It is possible to establish.

Proposition:

1. If $Z \subset D(Z)$, then $D(Z) \subset S$ ('self-generation').
2. $S = D(S)$ ('factorization').

Proposition:

1. Monotonicity of D: $Z \subset Z'$ implies $D(Z) \subset D(Z')$.
2. Z compact implies that $D(Z)$ is compact.

It can be shown that S is compact and that therefore there exists a (w, θ) pair within this set that attains the highest possible value w.

This (w, θ) pair i associated with a Ramsey plan.

Further, we can compute S by iterating to convergence on D provided that one begins with a sufficiently large initial set S_0.

As a very useful by-product, the algorithm that finds the largest fixed point $S = D(S)$ also produces the Ramsey plan, its value w, and the associated competitive equilibrium.

8.6 Calculating all Promise-Value Pairs in CE

Above we have defined the $D(Z)$ operator as:

$$D(Z) = \{(w, \theta) : \exists h \in CE^0_\pi \text{ and } (m(h), x(h), w'(h), \theta'(h)) \in [0, \bar{m}] \times X \times Z$$

such that

$$w = u(f(x(h))) + v(m(h)) + \beta w'(h)$$

$$\theta = u'(f(x(h)))(m(h) + x(h))$$

$$x(h) = m(h)(h - 1)$$

$$m(h)(u'(f(x(h))) - v'(m(h))) \leq \beta\theta'(h) \text{ (with equality if } m(h) < \bar{m})\}$$

We noted that the set S can be found by iterating to convergence on D, provided that we start with a sufficiently large initial set S_0.

Our implementation builds on ideas in this notebook.

To find S we use a numerical algorithm called the *outer hyperplane approximation algorithm*.

It was invented by Judd, Yeltekin, Conklin [JYC03].

This algorithm constructs the smallest convex set that contains the fixed point of the $D(S)$ operator.

Given that we are finding the smallest convex set that contains S, we can represent it on a computer as the intersection of a finite number of half-spaces.

Let H be a set of subgradients, and C be a set of hyperplane levels.

We approximate S by:

$$\tilde{S} = \{(w, \theta) | H \cdot (w, \theta) \leq C\}$$

A key feature of this algorithm is that we discretize the action space, i.e., we create a grid of possible values for m and h (note that x is implied by m and h). This discretization simplifies computation of $\tilde{S}$ by allowing us to find it by solving a sequence of linear programs.

The *outer hyperplane approximation algorithm* proceeds as follows:

1. Initialize subgradients, H, and hyperplane levels, C_0.
2. Given a set of subgradients, H, and hyperplane levels, C_t, for each subgradient $h_i \in H$:
    - Solve a linear program (described below) for each action in the action space.
    - Find the maximum and update the corresponding hyperplane level, $C_{i,t+1}$.
3. If $|C_{t+1} - C_t| > \epsilon$, return to 2.

Step 1 simply creates a large initial set S_0.

Given some set S_t, **Step 2** then constructs the set $S_{t+1} = D(S_t)$. The linear program in Step 2 is designed to construct a set S_{t+1} that is as large as possible while satisfying the constraints of the $D(S)$ operator.

To do this, for each subgradient h_i, and for each point in the action space (m_j, h_j), we solve the following problem:

$$\max_{[w',\theta']} h_i \cdot (w, \theta)$$

subject to

$$H \cdot (w', \theta') \leq C_t$$

$$w = u(f(x_j)) + v(m_j) + \beta w'$$

$$\theta = u'(f(x_j))(m_j + x_j)$$

$$x_j = m_j(h_j - 1)$$

$$m_j(u'(f(x_j)) - v'(m_j)) \leq \beta\theta' \quad (= \text{if } m_j < \bar{m})$$

This problem maximizes the hyperplane level for a given set of actions.

The second part of Step 2 then finds the maximum possible hyperplane level across the action space.

The algorithm constructs a sequence of progressively smaller sets $S_{t+1} \subset S_t \subset S_{t-1} \cdots \subset S_0$.

Step 3 ends the algorithm when the difference between these sets is small enough.

We have created a Python class that solves the model assuming the following functional forms:

$$u(c) = log(c)$$

$$v(m) = \frac{1}{500}(m\bar{m} - 0.5m^2)^{0.5}$$

$$f(x) = 180 - (0.4x)^2$$

The remaining parameters $\{\beta, \bar{m}, \underline{h}, \bar{h}\}$ are then variables to be specified for an instance of the Chang class.

Below we use the class to solve the model and plot the resulting equilibrium set, once with $\beta = 0.3$ and once with $\beta = 0.8$.

(Here we have set the number of subgradients to 10 in order to speed up the code for now - we can increase accuracy by increasing the number of subgradients)

```
"""
Provides a class called ChangModel to solve different
parameterizations of the Chang (1998) model.
"""

import numpy as np
import quantecon as qe
import time

from scipy.spatial import ConvexHull
from scipy.optimize import linprog, minimize, minimize_scalar
from scipy.interpolate import UnivariateSpline
import numpy.polynomial.chebyshev as cheb


class ChangModel:
```

(continues on next page)

(continued from previous page)

```
    """
    Class to solve for the competitive and sustainable sets in the Chang (1998)
    model, for different parameterizations.
    """

    def __init__(self, β, mbar, h_min, h_max, n_h, n_m, N_g):
        # Record parameters
        self.β, self.mbar, self.h_min, self.h_max = β, mbar, h_min, h_max
        self.n_h, self.n_m, self.N_g = n_h, n_m, N_g

        # Create other parameters
        self.m_min = 1e-9
        self.m_max = self.mbar
        self.N_a = self.n_h*self.n_m

        # Utility and production functions
        uc = lambda c: np.log(c)
        uc_p = lambda c: 1/c
        v = lambda m: 1/500 * (mbar * m - 0.5 * m**2)**0.5
        v_p = lambda m: 0.5/500 * (mbar * m - 0.5 * m**2)**(-0.5) * (mbar - m)
        u = lambda h, m: uc(f(h, m)) + v(m)

        def f(h, m):
            x = m * (h - 1)
            f = 180 - (0.4 * x)**2
            return f

        def θ(h, m):
            x = m * (h - 1)
            θ = uc_p(f(h, m)) * (m + x)
            return θ

        # Create set of possible action combinations, A
        A1 = np.linspace(h_min, h_max, n_h).reshape(n_h, 1)
        A2 = np.linspace(self.m_min, self.m_max, n_m).reshape(n_m, 1)
        self.A = np.concatenate((np.kron(np.ones((n_m, 1)), A1),
                                 np.kron(A2, np.ones((n_h, 1)))), axis=1)

        # Pre-compute utility and output vectors
        self.euler_vec = -np.multiply(self.A[:, 1], \
            uc_p(f(self.A[:, 0], self.A[:, 1])) - v_p(self.A[:, 1]))
        self.u_vec = u(self.A[:, 0], self.A[:, 1])
        self.Θ_vec = θ(self.A[:, 0], self.A[:, 1])
        self.f_vec = f(self.A[:, 0], self.A[:, 1])
        self.bell_vec = np.multiply(uc_p(f(self.A[:, 0],
                                   self.A[:, 1])),
                                   np.multiply(self.A[:, 1],
                                   (self.A[:, 0] - 1))) \
                        + np.multiply(self.A[:, 1],
                                      v_p(self.A[:, 1]))

        # Find extrema of (w, θ) space for initial guess of equilibrium sets
        p_vec = np.zeros(self.N_a)
        w_vec = np.zeros(self.N_a)
        for i in range(self.N_a):
            p_vec[i] = self.Θ_vec[i]
```

(continues on next page)

(continued from previous page)

```
            w_vec[i] = self.u_vec[i]/(1 - β)

        w_space = np.array([min(w_vec[~np.isinf(w_vec)]),
                            max(w_vec[~np.isinf(w_vec)])])
        p_space = np.array([0, max(p_vec[~np.isinf(w_vec)])])
        self.p_space = p_space

        # Set up hyperplane levels and gradients for iterations
        def SG_H_V(N, w_space, p_space):
            """
            This function  initializes the subgradients, hyperplane levels,
            and extreme points of the value set by choosing an appropriate
            origin and radius. It is based on a similar function in QuantEcon's
            Games.jl
            """

            # First, create a unit circle. Want points placed on [0, 2π]
            inc = 2 * np.pi / N
            degrees = np.arange(0, 2 * np.pi, inc)

            # Points on circle
            H = np.zeros((N, 2))
            for i in range(N):
                x = degrees[i]
                H[i, 0] = np.cos(x)
                H[i, 1] = np.sin(x)

            # Then calculate origin and radius
            o = np.array([np.mean(w_space), np.mean(p_space)])
            r1 = max((max(w_space) - o[0])**2, (o[0] - min(w_space))**2)
            r2 = max((max(p_space) - o[1])**2, (o[1] - min(p_space))**2)
            r = np.sqrt(r1 + r2)

            # Now calculate vertices
            Z = np.zeros((2, N))
            for i in range(N):
                Z[0, i] = o[0] + r*H.T[0, i]
                Z[1, i] = o[1] + r*H.T[1, i]

            # Corresponding hyperplane levels
            C = np.zeros(N)
            for i in range(N):
                C[i] = np.dot(Z[:, i], H[i, :])

            return C, H, Z

        C, self.H, Z = SG_H_V(N_g, w_space, p_space)
        C = C.reshape(N_g, 1)
        self.c0_c, self.c0_s, self.c1_c, self.c1_s = np.copy(C), np.copy(C), \
            np.copy(C), np.copy(C)
        self.z0_s, self.z0_c, self.z1_s, self.z1_c = np.copy(Z), np.copy(Z), \
            np.copy(Z), np.copy(Z)

        self.w_bnds_s, self.w_bnds_c = (w_space[0], w_space[1]), \
            (w_space[0], w_space[1])
        self.p_bnds_s, self.p_bnds_c = (p_space[0], p_space[1]), \
```

(continues on next page)

(continued from previous page)

```
                (p_space[0], p_space[1])

        # Create dictionaries to save equilibrium set for each iteration
        self.c_dic_s, self.c_dic_c = {}, {}
        self.c_dic_s[0], self.c_dic_c[0] = self.c0_s, self.c0_c

    def solve_worst_spe(self):
        """
        Method to solve for BR(Z). See p.449 of Chang (1998)
        """

        p_vec = np.full(self.N_a, np.nan)
        c = [1, 0]

        # Pre-compute constraints
        aineq_mbar = np.vstack((self.H, np.array([0, -self.β])))
        bineq_mbar = np.vstack((self.c0_s, 0))

        aineq = self.H
        bineq = self.c0_s
        aeq = [[0, -self.β]]

        for j in range(self.N_a):
            # Only try if consumption is possible
            if self.f_vec[j] > 0:
                # If m = mbar, use inequality constraint
                if self.A[j, 1] == self.mbar:
                    bineq_mbar[-1] = self.euler_vec[j]
                    res = linprog(c, A_ub=aineq_mbar, b_ub=bineq_mbar,
                                  bounds=(self.w_bnds_s, self.p_bnds_s))
                else:
                    beq = self.euler_vec[j]
                    res = linprog(c, A_ub=aineq, b_ub=bineq, A_eq=aeq, b_eq=beq,
                                  bounds=(self.w_bnds_s, self.p_bnds_s))
                if res.status == 0:
                    p_vec[j] = self.u_vec[j] + self.β * res.x[0]

        # Max over h and min over other variables (see Chang (1998) p.449)
        self.br_z = np.nanmax(np.nanmin(p_vec.reshape(self.n_m, self.n_h), 0))

    def solve_subgradient(self):
        """
        Method to solve for E(Z). See p.449 of Chang (1998)
        """

        # Pre-compute constraints
        aineq_C_mbar = np.vstack((self.H, np.array([0, -self.β])))
        bineq_C_mbar = np.vstack((self.c0_c, 0))

        aineq_C = self.H
        bineq_C = self.c0_c
        aeq_C = [[0, -self.β]]

        aineq_S_mbar = np.vstack((np.vstack((self.H, np.array([0, -self.β]))),
                                  np.array([-self.β, 0])))
        bineq_S_mbar = np.vstack((self.c0_s, np.zeros((2, 1))))
```

(continues on next page)

(continued from previous page)

```
        aineq_S = np.vstack((self.H, np.array([-self.β, 0])))
        bineq_S = np.vstack((self.c0_s, 0))
        aeq_S = [[0, -self.β]]

        # Update maximal hyperplane level
        for i in range(self.N_g):
            c_a1a2_c, t_a1a2_c = np.full(self.N_a, -np.inf), \
                np.zeros((self.N_a, 2))
            c_a1a2_s, t_a1a2_s = np.full(self.N_a, -np.inf), \
                np.zeros((self.N_a, 2))

            c = [-self.H[i, 0], -self.H[i, 1]]

            for j in range(self.N_a):
                # Only try if consumption is possible
                if self.f_vec[j] > 0:

                    # COMPETITIVE EQUILIBRIA
                    # If m = mbar, use inequality constraint
                    if self.A[j, 1] == self.mbar:
                        bineq_C_mbar[-1] = self.euler_vec[j]
                        res = linprog(c, A_ub=aineq_C_mbar, b_ub=bineq_C_mbar,
                                      bounds=(self.w_bnds_c, self.p_bnds_c))
                    # If m < mbar, use equality constraint
                    else:
                        beq_C = self.euler_vec[j]
                        res = linprog(c, A_ub=aineq_C, b_ub=bineq_C, A_eq = aeq_C,
                                      b_eq = beq_C, bounds=(self.w_bnds_c, \
                                          self.p_bnds_c))
                    if res.status == 0:
                        c_a1a2_c[j] = self.H[i, 0] * (self.u_vec[j] \
                            + self.β * res.x[0]) + self.H[i, 1] * self.Θ_vec[j]
                        t_a1a2_c[j] = res.x

                    # SUSTAINABLE EQUILIBRIA
                    # If m = mbar, use inequality constraint
                    if self.A[j, 1] == self.mbar:
                        bineq_S_mbar[-2] = self.euler_vec[j]
                        bineq_S_mbar[-1] = self.u_vec[j] - self.br_z
                        res = linprog(c, A_ub=aineq_S_mbar, b_ub=bineq_S_mbar,
                                      bounds=(self.w_bnds_s, self.p_bnds_s))
                    # If m < mbar, use equality constraint
                    else:
                        bineq_S[-1] = self.u_vec[j] - self.br_z
                        beq_S = self.euler_vec[j]
                        res = linprog(c, A_ub=aineq_S, b_ub=bineq_S, A_eq = aeq_S,
                                      b_eq = beq_S, bounds=(self.w_bnds_s, \
                                          self.p_bnds_s))
                    if res.status == 0:
                        c_a1a2_s[j] = self.H[i, 0] * (self.u_vec[j] \
                            + self.β*res.x[0]) + self.H[i, 1] * self.Θ_vec[j]
                        t_a1a2_s[j] = res.x

            idx_c = np.where(c_a1a2_c == max(c_a1a2_c))[0][0]
            self.z1_c[:, i] = np.array([self.u_vec[idx_c]
```

(continues on next page)

(continued from previous page)

```
                                    + self.β * t_a1a2_c[idx_c, 0],
                                    self.Θ_vec[idx_c]])

            idx_s = np.where(c_a1a2_s == max(c_a1a2_s))[0][0]
            self.z1_s[:, i] = np.array([self.u_vec[idx_s]
                                    + self.β * t_a1a2_s[idx_s, 0],
                                    self.Θ_vec[idx_s]])

        for i in range(self.N_g):
            self.c1_c[i] = np.dot(self.z1_c[:, i], self.H[i, :])
            self.c1_s[i] = np.dot(self.z1_s[:, i], self.H[i, :])

    def solve_sustainable(self, tol=1e-5, max_iter=250):
        """
        Method to solve for the competitive and sustainable equilibrium sets.
        """

        t = time.time()
        diff = tol + 1
        iters = 0

        print('### --------------- ###')
        print('Solving Chang Model Using Outer Hyperplane Approximation')
        print('### --------------- ### \n')

        print('Maximum difference when updating hyperplane levels:')

        while diff > tol and iters < max_iter:
            iters = iters + 1
            self.solve_worst_spe()
            self.solve_subgradient()
            diff = max(np.maximum(abs(self.c0_c - self.c1_c),
                       abs(self.c0_s - self.c1_s)))
            print(diff)

            # Update hyperplane levels
            self.c0_c, self.c0_s = np.copy(self.c1_c), np.copy(self.c1_s)

            # Update bounds for w and θ
            wmin_c, wmax_c = np.min(self.z1_c, axis=1)[0], \
                np.max(self.z1_c, axis=1)[0]
            pmin_c, pmax_c = np.min(self.z1_c, axis=1)[1], \
                np.max(self.z1_c, axis=1)[1]

            wmin_s, wmax_s = np.min(self.z1_s, axis=1)[0], \
                np.max(self.z1_s, axis=1)[0]
            pmin_S, pmax_S = np.min(self.z1_s, axis=1)[1], \
                np.max(self.z1_s, axis=1)[1]

            self.w_bnds_s, self.w_bnds_c = (wmin_s, wmax_s), (wmin_c, wmax_c)
            self.p_bnds_s, self.p_bnds_c = (pmin_S, pmax_S), (pmin_c, pmax_c)

            # Save iteration
            self.c_dic_c[iters], self.c_dic_s[iters] = np.copy(self.c1_c), \
                np.copy(self.c1_s)
            self.iters = iters
```

(continues on next page)

(continued from previous page)

```
        elapsed = time.time() - t
        print('Convergence achieved after {} iterations and {} \
            seconds'.format(iters, round(elapsed, 2)))

    def solve_bellman(self, θ_min, θ_max, order, disp=False, tol=1e-7, maxiters=100):
        """
        Continuous Method to solve the Bellman equation in section 25.3
        """
        mbar = self.mbar

        # Utility and production functions
        uc = lambda c: np.log(c)
        uc_p = lambda c: 1 / c
        v = lambda m: 1 / 500 * (mbar * m - 0.5 * m**2)**0.5
        v_p = lambda m: 0.5/500 * (mbar*m - 0.5 * m**2)**(-0.5) * (mbar - m)
        u = lambda h, m: uc(f(h, m)) + v(m)

        def f(h, m):
            x = m * (h - 1)
            f = 180 - (0.4 * x)**2
            return f

        def θ(h, m):
            x = m * (h - 1)
            θ = uc_p(f(h, m)) * (m + x)
            return θ

        # Bounds for Maximization
        lb1 = np.array([self.h_min, 0, θ_min])
        ub1 = np.array([self.h_max, self.mbar - 1e-5, θ_max])
        lb2 = np.array([self.h_min, θ_min])
        ub2 = np.array([self.h_max, θ_max])

        # Initialize Value Function coefficients
        # Calculate roots of Chebyshev polynomial
        k = np.linspace(order, 1, order)
        roots = np.cos((2 * k - 1) * np.pi / (2 * order))
        # Scale to approximation space
        s = θ_min + (roots - -1) / 2 * (θ_max - θ_min)
        # Create a basis matrix
        Φ = cheb.chebvander(roots, order - 1)
        c = np.zeros(Φ.shape[0])

        # Function to minimize and constraints
        def p_fun(x):
            scale = -1 + 2 * (x[2] - θ_min)/(θ_max - θ_min)
            p_fun = - (u(x[0], x[1]) \
                + self.β * np.dot(cheb.chebvander(scale, order - 1), c))
            return p_fun

        def p_fun2(x):
            scale = -1 + 2*(x[1] - θ_min)/(θ_max - θ_min)
            p_fun = - (u(x[0],mbar) \
                + self.β * np.dot(cheb.chebvander(scale, order - 1), c))
            return p_fun
```

(continues on next page)

(continued from previous page)

```
        cons1 = ({'type': 'eq',   'fun': lambda x: uc_p(f(x[0], x[1])) * x[1]
                    * (x[0] - 1) + v_p(x[1]) * x[1] + self.β * x[2] - θ},
                 {'type': 'eq',   'fun': lambda x: uc_p(f(x[0], x[1]))
                    * x[0] * x[1] - θ})
        cons2 = ({'type': 'ineq', 'fun': lambda x: uc_p(f(x[0], mbar)) * mbar
                    * (x[0] - 1) + v_p(mbar) * mbar + self.β * x[1] - θ},
                 {'type': 'eq',   'fun': lambda x: uc_p(f(x[0], mbar))
                    * x[0] * mbar - θ})

        bnds1 = np.concatenate([lb1.reshape(3, 1), ub1.reshape(3, 1)], axis=1)
        bnds2 = np.concatenate([lb2.reshape(2, 1), ub2.reshape(2, 1)], axis=1)

        # Bellman Iterations
        diff = 1
        iters = 1

        while diff > tol:
        # 1. Maximization, given value function guess
            p_iter1 = np.zeros(order)
            for i in range(order):
                θ = s[i]
                res = minimize(p_fun,
                               lb1 + (ub1-lb1) / 2,
                               method='SLSQP',
                               bounds=bnds1,
                               constraints=cons1,
                               tol=1e-10)
                if res.success == True:
                    p_iter1[i] = -p_fun(res.x)
                res = minimize(p_fun2,
                               lb2 + (ub2-lb2) / 2,
                               method='SLSQP',
                               bounds=bnds2,
                               constraints=cons2,
                               tol=1e-10)
                if -p_fun2(res.x) > p_iter1[i] and res.success == True:
                    p_iter1[i] = -p_fun2(res.x)

            # 2. Bellman updating of Value Function coefficients
            c1 = np.linalg.solve(Φ, p_iter1)
            # 3. Compute distance and update
            diff = np.linalg.norm(c - c1)
            if bool(disp == True):
                print(diff)
            c = np.copy(c1)
            iters = iters + 1
            if iters > maxiters:
                print('Convergence failed after {} iterations'.format(maxiters))
                break

        self.θ_grid = s
        self.p_iter = p_iter1
        self.Φ = Φ
        self.c = c
        print('Convergence achieved after {} iterations'.format(iters))
```

(continues on next page)

(continued from previous page)

```
        # Check residuals
        θ_grid_fine = np.linspace(θ_min, θ_max, 100)
        resid_grid = np.zeros(100)
        p_grid = np.zeros(100)
        θ_prime_grid = np.zeros(100)
        m_grid = np.zeros(100)
        h_grid = np.zeros(100)
        for i in range(100):
            θ = θ_grid_fine[i]
            res = minimize(p_fun,
                           lb1 + (ub1-lb1) / 2,
                           method='SLSQP',
                           bounds=bnds1,
                           constraints=cons1,
                           tol=1e-10)
            if res.success == True:
                p = -p_fun(res.x)
                p_grid[i] = p
                θ_prime_grid[i] = res.x[2]
                h_grid[i] = res.x[0]
                m_grid[i] = res.x[1]
            res = minimize(p_fun2,
                           lb2 + (ub2-lb2)/2,
                           method='SLSQP',
                           bounds=bnds2,
                           constraints=cons2,
                           tol=1e-10)
            if -p_fun2(res.x) > p and res.success == True:
                p = -p_fun2(res.x)
                p_grid[i] = p
                θ_prime_grid[i] = res.x[1]
                h_grid[i] = res.x[0]
                m_grid[i] = self.mbar
            scale = -1 + 2 * (θ - θ_min)/(θ_max - θ_min)
            resid_grid[i] = np.dot(cheb.chebvander(scale, order-1), c) - p

        self.resid_grid = resid_grid
        self.θ_grid_fine = θ_grid_fine
        self.θ_prime_grid = θ_prime_grid
        self.m_grid = m_grid
        self.h_grid = h_grid
        self.p_grid = p_grid
        self.x_grid = m_grid * (h_grid - 1)

        # Simulate
        θ_series = np.zeros(31)
        m_series = np.zeros(30)
        h_series = np.zeros(30)

        # Find initial θ
        def ValFun(x):
            scale = -1 + 2*(x - θ_min)/(θ_max - θ_min)
            p_fun = np.dot(cheb.chebvander(scale, order - 1), c)
            return -p_fun
```

(continues on next page)

(continued from previous page)

```
        res = minimize(ValFun,
                      (θ_min + θ_max)/2,
                      bounds=[(θ_min, θ_max)])
        θ_series[0] = res.x

        # Simulate
        for i in range(30):
            θ = θ_series[i]
            res = minimize(p_fun,
                           lb1 + (ub1-lb1)/2,
                           method='SLSQP',
                           bounds=bnds1,
                           constraints=cons1,
                           tol=1e-10)
            if res.success == True:
                p = -p_fun(res.x)
                h_series[i] = res.x[0]
                m_series[i] = res.x[1]
                θ_series[i+1] = res.x[2]
            res2 = minimize(p_fun2,
                            lb2 + (ub2-lb2)/2,
                            method='SLSQP',
                            bounds=bnds2,
                            constraints=cons2,
                            tol=1e-10)
            if -p_fun2(res2.x) > p and res2.success == True:
                h_series[i] = res2.x[0]
                m_series[i] = self.mbar
                θ_series[i+1] = res2.x[1]

        self.θ_series = θ_series
        self.m_series = m_series
        self.h_series = h_series
        self.x_series = m_series * (h_series - 1)
```

```
ch1 = ChangModel(β=0.3, mbar=30, h_min=0.9, h_max=2, n_h=8, n_m=35, N_g=10)
ch1.solve_sustainable()
```

```
### ---------------- ###
Solving Chang Model Using Outer Hyperplane Approximation
### ---------------- ###

Maximum difference when updating hyperplane levels:
```

```
[1.9168]
```

```
[0.66782]
```

```
[0.49235]
```

```
[0.32412]
```

```
[0.19022]
```

```
[0.10863]
```

```
[0.05817]
```

```
[0.0262]
```

```
[0.01836]
```

```
[0.01415]
```

```
[0.00297]
```

```
[0.00089]
```

```
[0.00027]
```

```
[0.00008]
```

```
[0.00002]
```

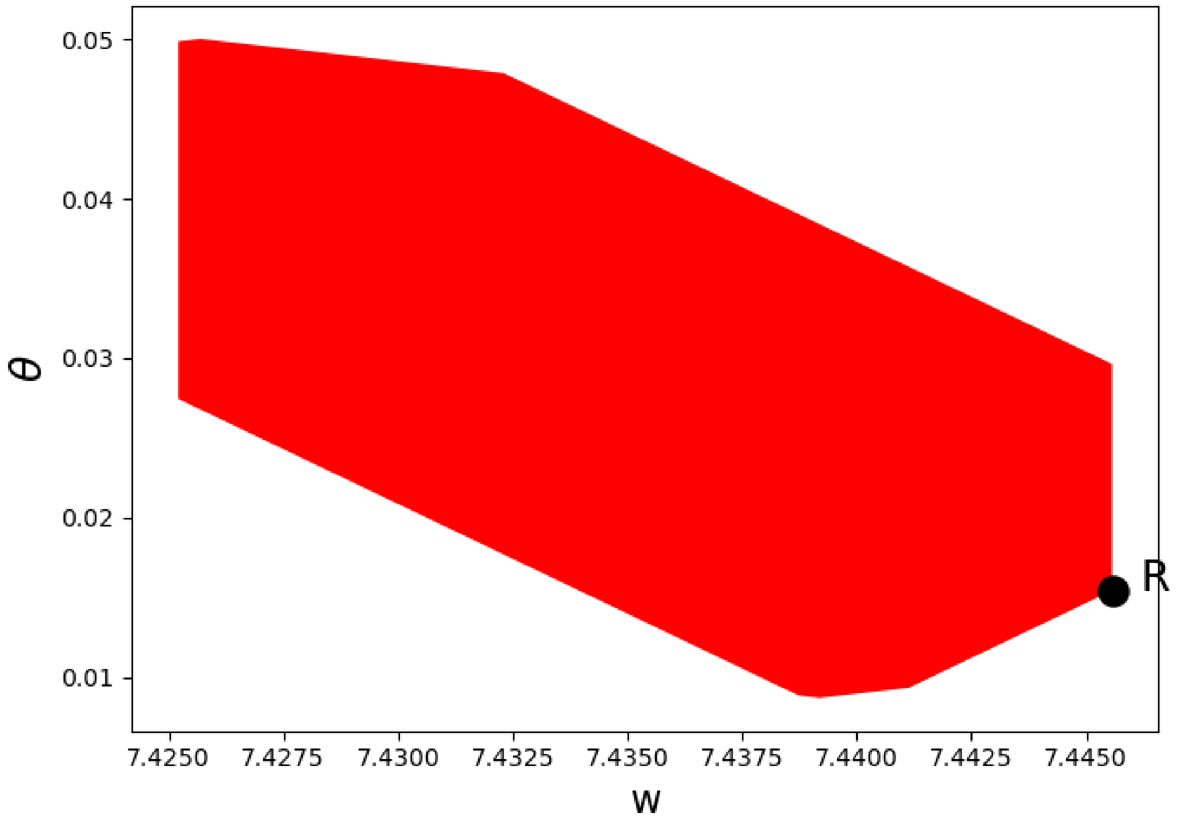
```
[0.00001]
Convergence achieved after 16 iterations and 42.36             seconds
```

```python
def plot_competitive(ChangModel):
    """
    Method that only plots competitive equilibrium set
    """
    poly_C = polytope.Polytope(ChangModel.H, ChangModel.c1_c)
    ext_C = polytope.extreme(poly_C)

    fig, ax = plt.subplots(figsize=(7, 5))

    ax.set_xlabel('w', fontsize=16)
    ax.set_ylabel(r"$\theta$", fontsize=18)

    ax.fill(ext_C[:,0], ext_C[:,1], 'r', zorder=0)
    ChangModel.min_theta = min(ext_C[:, 1])
    ChangModel.max_theta = max(ext_C[:, 1])

    # Add point showing Ramsey Plan
    idx_Ramsey = np.where(ext_C[:, 0] == max(ext_C[:, 0]))[0][0]
    R = ext_C[idx_Ramsey, :]
    ax.scatter(R[0], R[1], 150, 'black', 'o', zorder=1)
    w_min = min(ext_C[:, 0])

    # Label Ramsey Plan slightly to the right of the point
```

(continues on next page)

(continued from previous page)

```
    ax.annotate("R", xy=(R[0], R[1]), xytext=(R[0] + 0.03 * (R[0] - w_min),
                R[1]), fontsize=18)

    plt.tight_layout()
    plt.show()

plot_competitive(ch1)
```



```
ch2 = ChangModel(β=0.8, mbar=30, h_min=0.9, h_max=1/0.8,
                 n_h=8, n_m=35, N_g=10)
ch2.solve_sustainable()
```

```
### --------------- ###
Solving Chang Model Using Outer Hyperplane Approximation
### --------------- ###

Maximum difference when updating hyperplane levels:
```

```
[0.06369]
```

```
[0.02476]
```

```
[0.02153]
```

```
[0.01915]
```

```
[0.01795]
```

```
[0.01642]
```

```
[0.01507]
```

```
[0.01284]
```

```
[0.01106]
```

```
[0.00694]
```

```
[0.0085]
```

```
[0.00781]
```

```
[0.00433]
```

```
[0.00492]
```

```
[0.00303]
```

```
[0.00182]
```

```
[0.00638]
```

```
[0.00116]
```

```
[0.00093]
```

```
[0.00075]
```

```
[0.0006]
```

```
[0.00494]
```

```
[0.00038]
```

```
[0.00121]
```

```
[0.00024]
```

```
[0.0002]
```

```
[0.00016]
```

```
[0.00013]
```

```
[0.0001]
```

```
[0.00008]
```

```
[0.00006]
```

```
[0.00005]
```

```
[0.00004]
```

```
[0.00003]
```

```
[0.00003]
```

```
[0.00002]
```

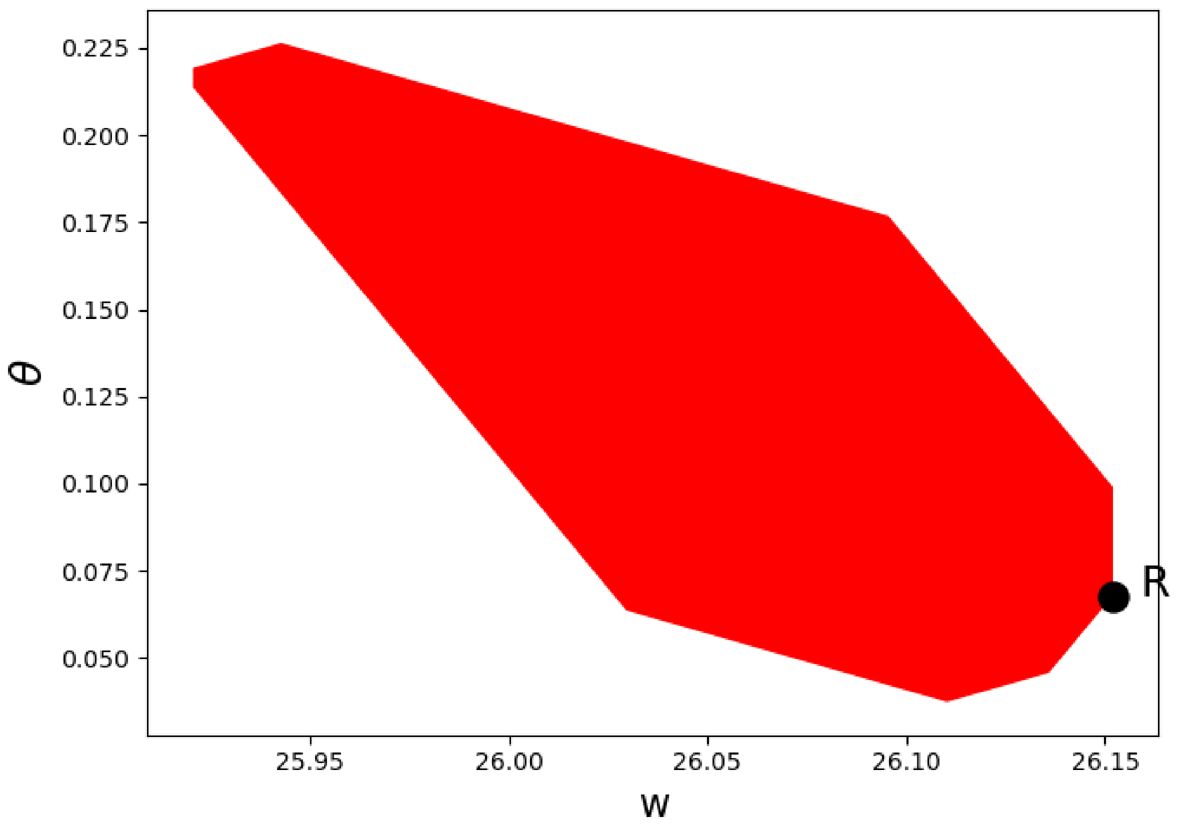
```
[0.00002]
```

```
[0.00001]
```

```
[0.00001]
```

```
[0.00001]
Convergence achieved after 40 iterations and 122.58             seconds
```

```
plot_competitive(ch2)
```



8.7 Solving a Continuation Ramsey Planner's Bellman Equation

In this section we solve the Bellman equation confronting a **continuation Ramsey planner**.

The construction of a Ramsey plan is decomposed into a two subproblems in *Ramsey plans, time inconsistency, sustainable plans* and *dynamic Stackelberg problems*.

- Subproblem 1 is faced by a sequence of continuation Ramsey planners at $t \geq 1$.
- Subproblem 2 is faced by a Ramsey planner at $t = 0$.

The problem is:

$$J(\theta) = \max_{m,x,h,\theta'} u(f(x)) + v(m) + \beta J(\theta')$$

subject to:

$$\theta \leq u'(f(x))x + v'(m)m + \beta\theta'$$

$$\theta = u'(f(x))(m + x)$$

$$x = m(h - 1)$$

$$(m, x, h) \in E$$

$$\theta' \in \Omega$$

To solve this Bellman equation, we must know the set Ω.

We have solved the Bellman equation for the two sets of parameter values for which we computed the equilibrium value sets above.

Hence for these parameter configurations, we know the bounds of Ω.

The two sets of parameters differ only in the level of β.

From the figures earlier in this lecture, we know that when $\beta = 0.3$, $\Omega = [0.0088, 0.0499]$, and when $\beta = 0.8$, $\Omega = [0.0395, 0.2193]$

```
ch1 = ChangModel(β=0.3, mbar=30, h_min=0.99, h_max=1/0.3,
                 n_h=8, n_m=35, N_g=50)
ch2 = ChangModel(β=0.8, mbar=30, h_min=0.1, h_max=1/0.8,
                 n_h=20, n_m=50, N_g=50)
```

```
/tmp/ipykernel_2455/1608401414.py:33: RuntimeWarning: invalid value encountered in␣
↪log
  uc = lambda c: np.log(c)
```

```
ch1.solve_bellman(θ_min=0.01, θ_max=0.0499, order=30, tol=1e-6)
ch2.solve_bellman(θ_min=0.045, θ_max=0.15, order=30, tol=1e-6)
```

```
/tmp/ipykernel_2455/1608401414.py:382: DeprecationWarning: Conversion of an array␣
↪with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you␣
↪extract a single element from your array before performing this operation.␣
↪(Deprecated NumPy 1.25.)
  p_iter1[i] = -p_fun(res.x)
/tmp/ipykernel_2455/1608401414.py:309: RuntimeWarning: invalid value encountered␣
↪in log
  uc = lambda c: np.log(c)
```

```
Convergence achieved after 15 iterations
```

```
/tmp/ipykernel_2455/1608401414.py:427: DeprecationWarning: Conversion of an array␣
↪with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you␣
↪extract a single element from your array before performing this operation.␣
↪(Deprecated NumPy 1.25.)
  p_grid[i] = p
/tmp/ipykernel_2455/1608401414.py:444: DeprecationWarning: Conversion of an array␣
↪with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you␣
↪extract a single element from your array before performing this operation.␣
↪(Deprecated NumPy 1.25.)
  resid_grid[i] = np.dot(cheb.chebvander(scale, order-1), c) - p
```

```
/tmp/ipykernel_2455/1608401414.py:468: DeprecationWarning: Conversion of an array␣
↪with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you␣
↪extract a single element from your array before performing this operation.␣
↪(Deprecated NumPy 1.25.)
  θ_series[0] = res.x
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.11/site-packages/scipy/optimize/
↪_optimize.py:404: RuntimeWarning: Values in x were outside bounds during a␣
↪minimize step, clipping to bounds
  warnings.warn("Values in x were outside bounds during a "
Convergence achieved after 72 iterations
```

First, a quick check that our approximations of the value functions are good.

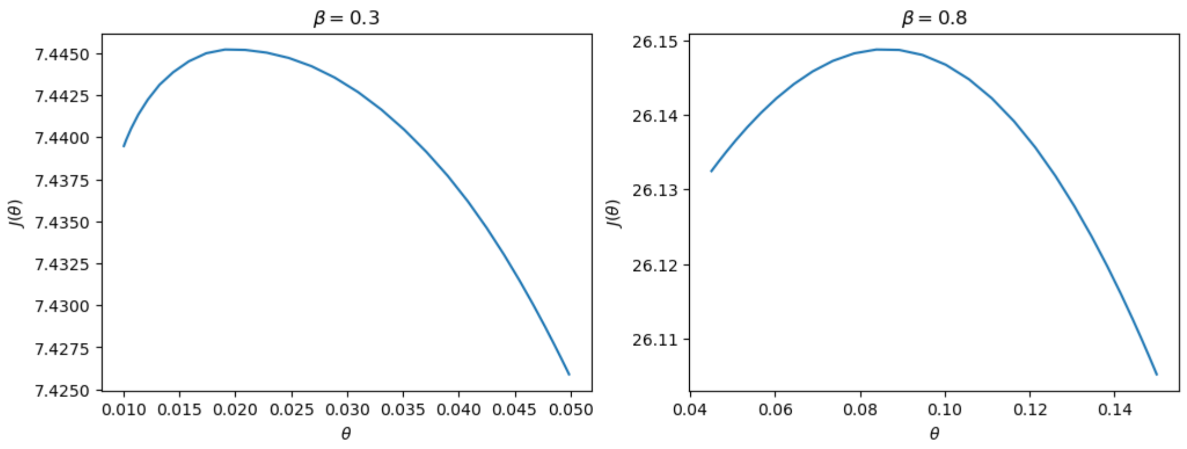
We do this by calculating the residuals between iterates on the value function on a fine grid:

```
max(abs(ch1.resid_grid)), max(abs(ch2.resid_grid))
```

```
(6.46313155971967e-06, 6.875358415925348e-07)
```

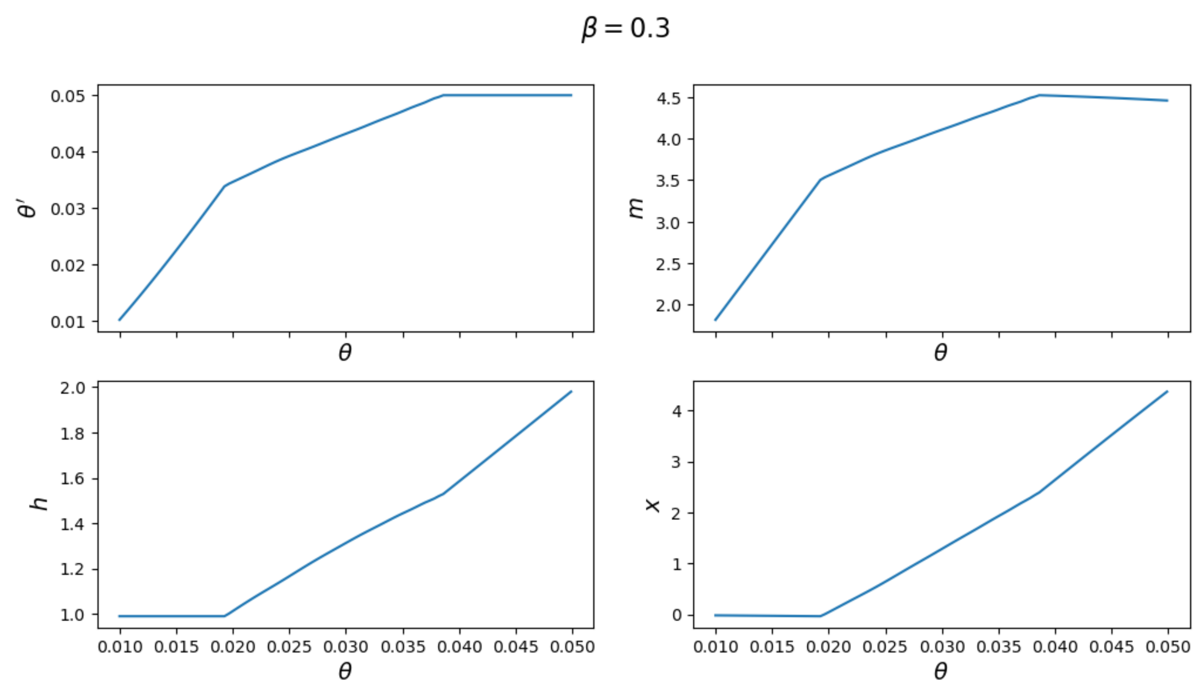
The value functions plotted below trace out the right edges of the sets of equilibrium values plotted above

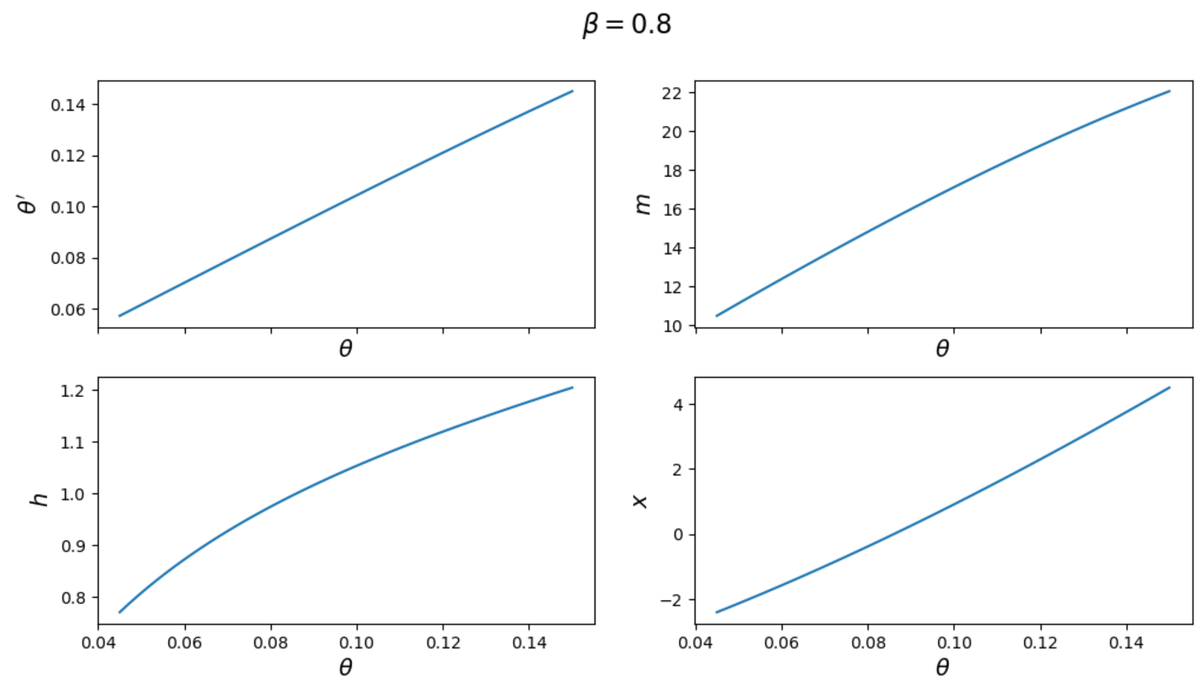
```
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

for ax, model in zip(axes, (ch1, ch2)):
    ax.plot(model.θ_grid, model.p_iter)
    ax.set(xlabel=r"$\theta$",
           ylabel=r"$J(\theta)$",
           title=rf"$\beta = {model.β}$")

plt.show()
```



The next figure plots the optimal policy functions; values of θ', m, x, h for each value of the state θ:

```
for model in (ch1, ch2):

    fig, axes = plt.subplots(2, 2, figsize=(12, 6), sharex=True)
    fig.suptitle(rf"$\beta = {model.β}$", fontsize=16)

    plots = [model.θ_prime_grid, model.m_grid,
             model.h_grid, model.x_grid]
    labels = [r"$\theta'$", "$m$", "$h$", "$x$"]

    for ax, plot, label in zip(axes.flatten(), plots, labels):
        ax.plot(model.θ_grid_fine, plot)
        ax.set_xlabel(r"$\theta$", fontsize=14)
        ax.set_ylabel(label, fontsize=14)

    plt.show()
```





With the first set of parameter values, the value of θ' chosen by the Ramsey planner quickly hits the upper limit of Ω.

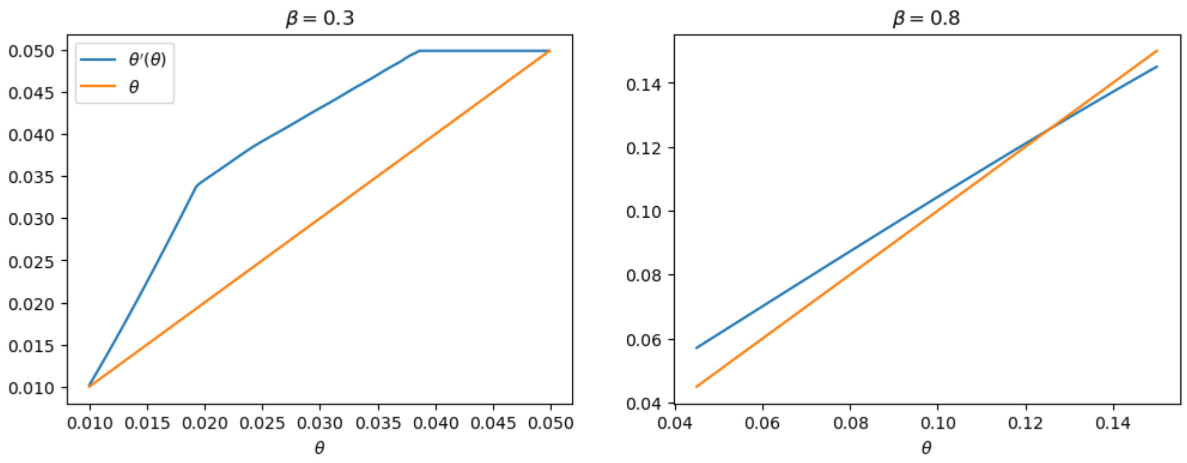
But with the second set of parameters it converges to a value in the interior of the set.

Consequently, the choice of $\bar{\theta}$ is clearly important with the first set of parameter values.

One way of seeing this is plotting $\theta'(\theta)$ for each set of parameters.

With the first set of parameter values, this function does not intersect the 45-degree line until $\bar{\theta}$, whereas in the second set of parameter values, it intersects in the interior.

```
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

for ax, model in zip(axes, (ch1, ch2)):
    ax.plot(model.θ_grid_fine, model.θ_prime_grid, label=r"$\theta'(\theta)$")
    ax.plot(model.θ_grid_fine, model.θ_grid_fine, label=r"$\theta$")
    ax.set(xlabel=r"$\theta$", title=rf"$\beta = {model.β}$")

axes[0].legend()
plt.show()
```

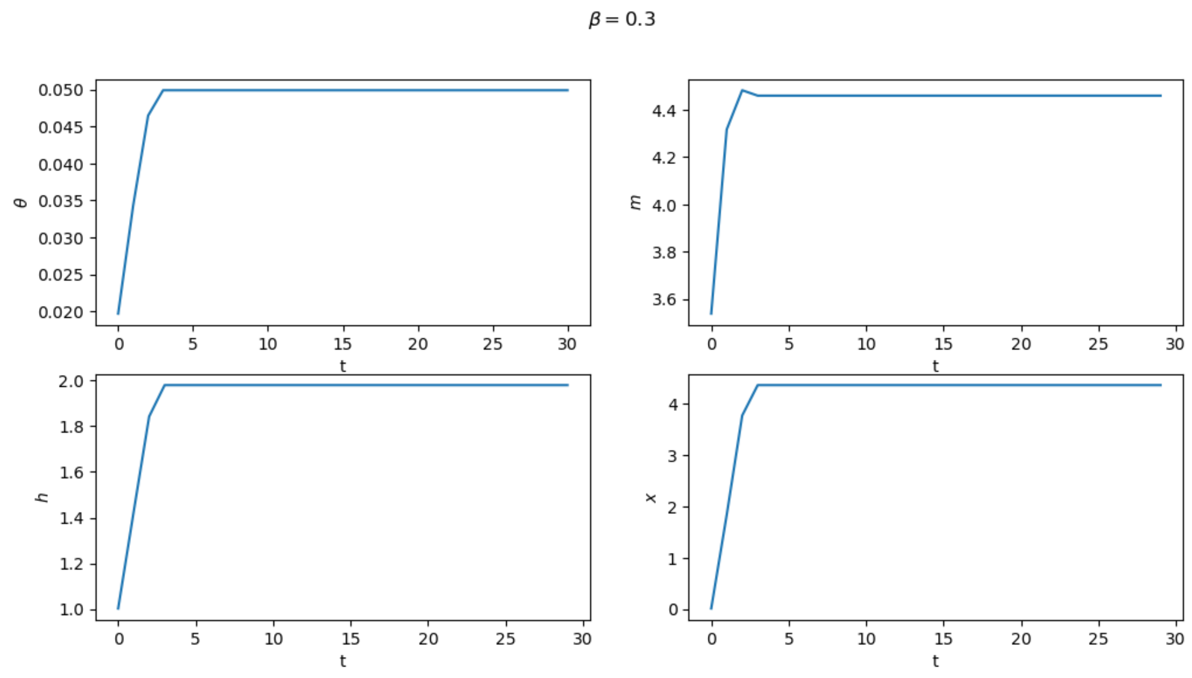


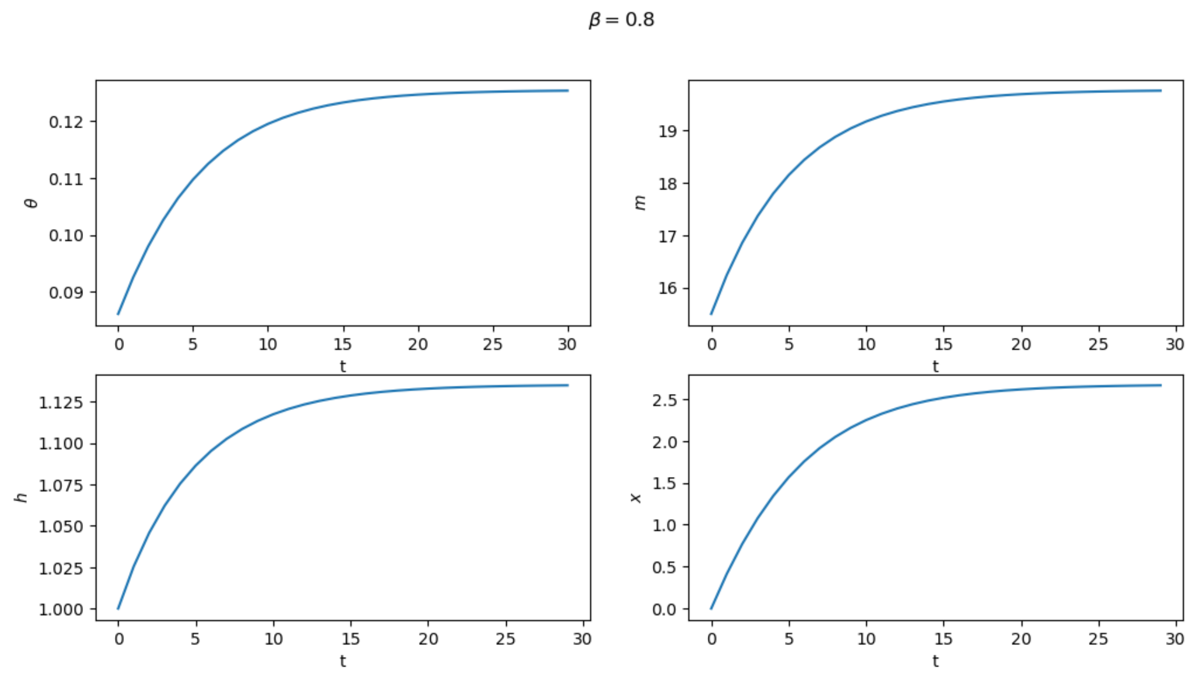
Subproblem 2 is equivalent to the planner choosing the initial value of θ (i.e. the value which maximizes the value function).

From this starting point, we can then trace out the paths for $\{\theta_t, m_t, h_t, x_t\}_{t=0}^{\infty}$ that support this equilibrium.

These are shown below for both sets of parameters

```
for model in (ch1, ch2):

    fig, axes = plt.subplots(2, 2, figsize=(12, 6))
    fig.suptitle(rf"$\beta = {model.β}$")

    plots = [model.θ_series, model.m_series, model.h_series, model.x_series]
    labels = [r"$\theta$", "$m$", "$h$", "$x$"]

    for ax, plot, label in zip(axes.flatten(), plots, labels):
        ax.plot(plot)
        ax.set(xlabel='t', ylabel=label)

    plt.show()
```


$\beta = 0.3$
θ
m
h
x
t


$\beta = 0.8$
θ
m
h
x
t

8.7.1 Next Steps

In *Credible Government Policies in Chang Model* we shall find a subset of competitive equilibria that are **sustainable** in the sense that a sequence of government administrations that chooses sequentially, rather than once and for all at time 0 will choose to implement them.

In the process of constructing them, we shall construct another, smaller set of competitive equilibria.

CHAPTER

NINE

CREDIBLE GOVERNMENT POLICIES IN A MODEL OF CHANG

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install polytope
```

9.1 Overview

Some of the material in this lecture and *competitive equilibria in the Chang model* can be viewed as more sophisticated and complete treatments of the topics discussed in *Ramsey plans, time inconsistency, sustainable plans*.

This lecture assumes almost the same economic environment analyzed in *competitive equilibria in the Chang model*.

The only change – and it is a substantial one – is the timing protocol for making government decisions.

In *competitive equilibria in the Chang model*, a *Ramsey planner* chose a comprehensive government policy once-and-for-all at time 0.

Now in this lecture, there is no time 0 Ramsey planner.

Instead there is a sequence of government decision-makers, one for each t.

The time t government decision-maker choose time t government actions after forecasting what future governments will do.

We use the notion of a *sustainable plan* proposed in [CK90], also referred to as a *credible public policy* in [Sto89].

Technically, this lecture starts where lecture *competitive equilibria in the Chang model* on Ramsey plans within the Chang [Cha98] model stopped.

That lecture presents recursive representations of *competitive equilibria* and a *Ramsey plan* for a version of a model of Calvo [Cal78] that Chang used to analyze and illustrate these concepts.

We used two operators to characterize competitive equilibria and a Ramsey plan, respectively.

In this lecture, we define a *credible public policy* or *sustainable plan*.

Starting from a large enough initial set Z_0, we use iterations on Chang's set-to-set operator $\tilde{D}(Z)$ to compute a set of values associated with sustainable plans.

Chang's operator $\tilde{D}(Z)$ is closely connected with the operator $D(Z)$ introduced in lecture *competitive equilibria in the Chang model*.

- $\tilde{D}(Z)$ incorporates all of the restrictions imposed in constructing the operator $D(Z)$, but ….
- It adds some additional restrictions
  - these additional restrictions incorporate the idea that a plan must be *sustainable*.
  - *sustainable* means that the government wants to implement it at all times after all histories.

Let's start with some standard imports:

```
import numpy as np
import polytope
import matplotlib.pyplot as plt
%matplotlib inline
```

```
`polytope` failed to import `cvxopt.glpk`.
```

```
will use `scipy.optimize.linprog`
```

9.2 The Setting

We begin by reviewing the set up deployed in *competitive equilibria in the Chang model*.

Chang's model, adopted from Calvo, is designed to focus on the intertemporal trade-offs between the welfare benefits of deflation and the welfare costs associated with the high tax collections required to retire money at a rate that delivers deflation.

A benevolent time 0 government can promote utility generating increases in real balances only by imposing an infinite sequence of sufficiently large distorting tax collections.

To promote the welfare increasing effects of high real balances, the government wants to induce *gradual deflation*.

We start by reviewing notation.

For a sequence of scalars $\vec{z} \equiv \{z_t\}_{t=0}^{\infty}$, let $\vec{z}^t = (z_0, \dots, z_t)$, $\vec{z}_t = (z_t, z_{t+1}, \dots)$.

An infinitely lived representative agent and an infinitely lived government exist at dates $t = 0, 1, \dots$.

The objects in play are

- an initial quantity M_{-1} of nominal money holdings
- a sequence of inverse money growth rates $\vec{h}$ and an associated sequence of nominal money holdings $\vec{M}$
- a sequence of values of money $\vec{q}$
- a sequence of real money holdings $\vec{m}$
- a sequence of total tax collections $\vec{x}$
- a sequence of per capita rates of consumption $\vec{c}$
- a sequence of per capita incomes $\vec{y}$

A benevolent government chooses sequences $(\vec{M}, \vec{h}, \vec{x})$ subject to a sequence of budget constraints and other constraints imposed by competitive equilibrium.

Given tax collection and price of money sequences, a representative household chooses sequences $(\vec{c}, \vec{m})$ of consumption and real balances.

In competitive equilibrium, the price of money sequence $\vec{q}$ clears markets, thereby reconciling decisions of the government and the representative household.

9.2.1 The Household's Problem

A representative household faces a nonnegative value of money sequence $\vec{q}$ and sequences $\vec{y}, \vec{x}$ of income and total tax collections, respectively.

The household chooses nonnegative sequences $\vec{c}, \vec{M}$ of consumption and nominal balances, respectively, to maximize

$$\sum_{t=0}^{\infty} \beta^t \left[u(c_t) + v(q_t M_t)\right] \tag{9.1}$$

subject to

$$q_t M_t \leq y_t + q_t M_{t-1} - c_t - x_t \tag{9.2}$$

and

$$q_t M_t \leq \bar{m} \tag{9.3}$$

Here q_t is the reciprocal of the price level at t, also known as the *value of money*.

Chang [Cha98] assumes that

- $u : \mathbb{R}_+ \rightarrow \mathbb{R}$ is twice continuously differentiable, strictly concave, and strictly increasing;
- $v : \mathbb{R}_+ \rightarrow \mathbb{R}$ is twice continuously differentiable and strictly concave;
- $u'(c)_{c \rightarrow 0} = \lim_{m \rightarrow 0} v'(m) = +\infty$;
- there is a finite level $m = m^f$ such that $v'(m^f) = 0$

Real balances carried out of a period equal $m_t = q_t M_t$.

Inequality (9.2) is the household's time t budget constraint.

It tells how real balances $q_t M_t$ carried out of period t depend on income, consumption, taxes, and real balances $q_t M_{t-1}$ carried into the period.

Equation (9.3) imposes an exogenous upper bound $\bar{m}$ on the choice of real balances, where $\bar{m} \geq m^f$.

9.2.2 Government

The government chooses a sequence of inverse money growth rates with time t component $h_t \equiv \frac{M_{t-1}}{M_t} \in \Pi \equiv [\underline{\pi}, \overline{\pi}]$, where $0 < \underline{\pi} < 1 < \frac{1}{\beta} \leq \overline{\pi}$.

The government faces a sequence of budget constraints with time t component

$$-x_t = q_t (M_t - M_{t-1})$$

which, by using the definitions of m_t and h_t, can also be expressed as

$$-x_t = m_t (1 - h_t) \tag{9.4}$$

The restrictions $m_t \in [0, \bar{m}]$ and $h_t \in \Pi$ evidently imply that $x_t \in X \equiv [(\underline{\pi} - 1)\bar{m}, (\overline{\pi} - 1)\bar{m}]$.

We define the set $E \equiv [0, \bar{m}] \times \Pi \times X$, so that we require that $(m, h, x) \in E$.

To represent the idea that taxes are distorting, Chang makes the following assumption about outcomes for per capita output:

$$y_t = f(x_t) \tag{9.5}$$

where $f : \mathbb{R} \to \mathbb{R}$ satisfies $f(x) > 0$, is twice continuously differentiable, $f''(x) < 0$, and $f(x) = f(-x)$ for all $x \in \mathbb{R}$, so that subsidies and taxes are equally distorting.

The purpose is not to model the causes of tax distortions in any detail but simply to summarize the *outcome* of those distortions via the function $f(x)$.

A key part of the specification is that tax distortions are increasing in the absolute value of tax revenues.

The government chooses a competitive equilibrium that maximizes (9.1).

9.2.3 Within-period Timing Protocol

For the results in this lecture, the *timing* of actions within a period is important because of the incentives that it activates.

Chang assumed the following within-period timing of decisions:

- first, the government chooses h_t and x_t;
- then given $\vec{q}$ and its expectations about future values of x and y's, the household chooses M_t and therefore m_t because $m_t = q_t M_t$;
- then output $y_t = f(x_t)$ is realized;
- finally $c_t = y_t$

This within-period timing confronts the government with choices framed by how the private sector wants to respond when the government takes time t actions that differ from what the private sector had expected.

This timing will shape the incentives confronting the government at each history that are to be incorporated in the construction of the $\tilde{D}$ operator below.

9.2.4 Household's Problem

Given M_{-1} and $\{q_t\}_{t=0}^\infty$, the household's problem is

$$\begin{aligned} \mathcal{L} = \max_{\vec{c}, \vec{M}} \min_{\vec{\lambda}, \vec{\mu}} \sum_{t=0}^\infty \beta^t \{ & u(c_t) + v(M_t q_t) + \lambda_t [y_t - c_t - x_t + q_t M_{t-1} - q_t M_t] \\ & + \mu_t [\bar{m} - q_t M_t] \} \end{aligned}$$

First-order conditions with respect to c_t and M_t, respectively, are

$$\begin{aligned} u'(c_t) &= \lambda_t \\ q_t [u'(c_t) - v'(M_t q_t)] &\leq \beta u'(c_{t+1}) q_{t+1}, \quad = \text{ if } M_t q_t < \bar{m} \end{aligned}$$

Using $h_t = \frac{M_{t-1}}{M_t}$ and $q_t = \frac{m_t}{M_t}$ in these first-order conditions and rearranging implies

$$m_t [u'(c_t) - v'(m_t)] \leq \beta u'(f(x_{t+1})) m_{t+1} h_{t+1}, \quad = \text{ if } m_t < \bar{m} \tag{9.6}$$

Define the following key variable

$$\theta_{t+1} \equiv u'(f(x_{t+1}))m_{t+1}h_{t+1} \tag{9.7}$$

This is real money balances at time $t+1$ measured in units of marginal utility, which Chang refers to as 'the marginal utility of real balances'.

From the standpoint of the household at time t, equation (9.7) shows that θ_{t+1} intermediates the influences of $(\vec{x}_{t+1}, \vec{m}_{t+1})$ on the household's choice of real balances m_t.

By "intermediates" we mean that the future paths $(\vec{x}_{t+1}, \vec{m}_{t+1})$ influence m_t entirely through their effects on the scalar θ_{t+1}.

The observation that the one dimensional promised marginal utility of real balances θ_{t+1} functions in this way is an important step in constructing a class of competitive equilibria that have a recursive representation.

A closely related observation pervaded the analysis of Stackelberg plans in *dynamic Stackelberg problems* and *the Calvo model*.

9.2.5 Competitive Equilibrium

Definition:

- A *government policy* is a pair of sequences $(\vec{h}, \vec{x})$ where $h_t \in \Pi \ \forall t \geq 0$.
- A *price system* is a non-negative value of money sequence $\vec{q}$.
- An *allocation* is a triple of non-negative sequences $(\vec{c}, \vec{m}, \vec{y})$.

It is required that time t components $(m_t, x_t, h_t) \in E$.

Definition:

Given M_{-1}, a government policy $(\vec{h}, \vec{x})$, price system $\vec{q}$, and allocation $(\vec{c}, \vec{m}, \vec{y})$ are said to be a *competitive equilibrium* if

- $m_t = q_t M_t$ and $y_t = f(x_t)$.
- The government budget constraint is satisfied.
- Given $\vec{q}, \vec{x}, \vec{y}$, $(\vec{c}, \vec{m})$ solves the household's problem.

9.2.6 A Credible Government Policy

Chang works with

A credible government policy with a recursive representation

- Here there is no time 0 Ramsey planner.
- Instead there is a sequence of governments, one for each t, that choose time t government actions after forecasting what future governments will do.
- Let $w = \sum_{t=0}^{\infty} \beta^t \left[u(c_t) + v(q_t M_t)\right]$ be a value associated with a particular competitive equilibrium.
- A recursive representation of a credible government policy is a pair of initial conditions (w_0, θ_0) and a five-tuple of functions

  $$h(w_t, \theta_t), m(h_t, w_t, \theta_t), x(h_t, w_t, \theta_t), \chi(h_t, w_t, \theta_t), \Psi(h_t, w_t, \theta_t)$$

  mapping w_t, θ_t and in some cases h_t into $\hat{h}_t, m_t, x_t, w_{t+1}$, and θ_{t+1}, respectively.

- Starting from an initial condition (w_0, θ_0), a credible government policy can be constructed by iterating on these functions in the following order that respects the within-period timing:

$$\begin{aligned} \hat{h}_t &= h(w_t, \theta_t) \\ m_t &= m(h_t, w_t, \theta_t) \\ x_t &= x(h_t, w_t, \theta_t) \\ w_{t+1} &= \chi(h_t, w_t, \theta_t) \\ \theta_{t+1} &= \Psi(h_t, w_t, \theta_t) \end{aligned} \tag{9.8}$$

- Here it is to be understood that $\hat{h}_t$ is the action that the government policy instructs the government to take, while h_t possibly not equal to $\hat{h}_t$ is some other action that the government is free to take at time t.

The plan is *credible* if it is in the time t government's interest to execute it.

Credibility requires that the plan be such that for all possible choices of h_t that are consistent with competitive equilibria,

$$\begin{aligned} &u(f(x(\hat{h}_t, w_t, \theta_t))) + v(m(\hat{h}_t, w_t, \theta_t)) + \beta\chi(\hat{h}_t, w_t, \theta_t) \\ &\geq u(f(x(h_t, w_t, \theta_t))) + v(m(h_t, w_t, \theta_t)) + \beta\chi(h_t, w_t, \theta_t) \end{aligned}$$

so that at each instance and circumstance of choice, a government attains a weakly higher lifetime utility with continuation value $w_{t+1} = \Psi(h_t, w_t, \theta_t)$ by adhering to the plan and confirming the associated time t action $\hat{h}_t$ that the public had expected earlier.

Please note the subtle change in arguments of the functions used to represent a competitive equilibrium and a Ramsey plan, on the one hand, and a credible government plan, on the other hand.

The extra arguments appearing in the functions used to represent a credible plan come from allowing the government to contemplate disappointing the private sector's expectation about its time t choice $\hat{h}_t$.

A credible plan induces the government to confirm the private sector's expectation.

The recursive representation of the plan uses the evolution of continuation values to deter the government from wanting to disappoint the private sector's expectations.

Technically, a Ramsey plan and a credible plan both incorporate history dependence.

For a Ramsey plan, this is encoded in the dynamics of the state variable θ_t, a promised marginal utility that the Ramsey plan delivers to the private sector.

For a credible government plan, we the two-dimensional state vector (w_t, θ_t) encodes history dependence.

9.2.7 Sustainable Plans

A government strategy σ and an allocation rule α are said to constitute a *sustainable plan* (SP) if.

1. σ is admissible.
2. Given σ, α is competitive.
3. After any history $\vec{h}^{t-1}$, the continuation of σ is optimal for the government; i.e., the sequence $\vec{h}_t$ induced by σ after $\vec{h}^{t-1}$ maximizes over CE_π given α.

Given any history $\vec{h}^{t-1}$, the continuation of a sustainable plan is a sustainable plan.

Let $\Theta = \{(\vec{m}, \vec{x}, \vec{h}) \in CE : \text{there is an SP whose outcome is}(\vec{m}, \vec{x}, \vec{h})\}$.

Sustainable outcomes are elements of Θ.

Now consider the space

$$S = \Big\{(w, \theta) : \text{there is a sustainable outcome } (\vec{m}, \vec{x}, \vec{h}) \in \Theta$$

with value

$$w = \sum_{t=0}^{\infty} \beta^t [u(f(x_t)) + v(m_t)] \text{ and such that } u'(f(x_0))(m_0 + x_0) = \theta \Big\}$$

The space S is a compact subset of $W \times \Omega$ where $W = [\underline{w}, \overline{w}]$ is the space of values associated with sustainable plans. Here $\underline{w}$ and $\overline{w}$ are finite bounds on the set of values.

Because there is at least one sustainable plan, S is nonempty.

Now recall the within-period timing protocol, which we can depict $(h, x) \to m = qM \to y = c$.

With this timing protocol in mind, the time 0 component of an SP has the following components:

1. A period 0 action $\hat{h} \in \Pi$ that the public expects the government to take, together with subsequent within-period consequences $m(\hat{h}), x(\hat{h})$ when the government acts as expected.
2. For any first-period action $h \neq \hat{h}$ with $h \in CE_\pi^0$, a pair of within-period consequences $m(h), x(h)$ when the government does not act as the public had expected.
3. For every $h \in \Pi$, a pair $(w'(h), \theta'(h)) \in S$ to carry into next period.

These components must be such that it is optimal for the government to choose $\hat{h}$ as expected; and for every possible $h \in \Pi$, the government budget constraint and the household's Euler equation must hold with continuation θ being $\theta'(h)$.

Given the timing protocol within the model, the representative household's response to a government deviation to $h \neq \hat{h}$ from a prescribed $\hat{h}$ consists of a first-period action $m(h)$ and associated subsequent actions, together with future equilibrium prices, captured by $(w'(h), \theta'(h))$.

At this point, Chang introduces an idea in the spirit of Abreu, Pearce, and Stacchetti [APS90].

Let Z be a nonempty subset of $W \times \Omega$.

Think of using pairs (w', θ') drawn from Z as candidate continuation value, promised marginal utility pairs.

Define the following operator:

$$\begin{aligned} \tilde{D}(Z) = \Big\{(w, \theta) : \text{there is } \hat{h} \in CE_\pi^0 \text{ and for each } h \in CE_\pi^0 \\ \text{a four-tuple } (m(h), x(h), w'(h), \theta'(h)) \in [0, \bar{m}] \times X \times Z \end{aligned} \tag{9.9}$$

such that

$$w = u(f(x(\hat{h}))) + v(m(\hat{h})) + \beta w'(\hat{h}) \tag{9.10}$$

$$\theta = u'(f(x(\hat{h})))(m(\hat{h}) + x(\hat{h})) \tag{9.11}$$

and for all $h \in CE_\pi^0$

$$w \geq u(f(x(h))) + v(m(h)) + \beta w'(h) \tag{9.12}$$

$$x(h) = m(h)(h - 1) \tag{9.13}$$

and

$$m(h)(u'(f(x(h))) - v'(m(h))) \leq \beta \theta'(h) \tag{9.14}$$

$$\text{with equality if } m(h) < \bar{m} \Big\}$$

This operator adds the key incentive constraint to the conditions that had defined the earlier $D(Z)$ operator defined in *competitive equilibria in the Chang model*.

Condition (9.12) requires that the plan deter the government from wanting to take one-shot deviations when candidate continuation values are drawn from Z.

Proposition:

1. If $Z \subset \tilde{D}(Z)$, then $\tilde{D}(Z) \subset S$ ('self-generation').
2. $S = \tilde{D}(S)$ ('factorization').

Proposition:.

1. Monotonicity of $\tilde{D}$: $Z \subset Z'$ implies $\tilde{D}(Z) \subset \tilde{D}(Z')$.
2. Z compact implies that $\tilde{D}(Z)$ is compact.

Chang establishes that S is compact and that therefore there exists a highest value SP and a lowest value SP.

Further, the preceding structure allows Chang to compute S by iterating to convergence on $\tilde{D}$ provided that one begins with a sufficiently large initial set Z_0.

This structure delivers the following recursive representation of a sustainable outcome:

1. choose an initial $(w_0, \theta_0) \in S$;
2. generate a sustainable outcome recursively by iterating on (9.8), which we repeat here for convenience:

$$\begin{aligned}\hat{h}_t &= h(w_t, \theta_t)\\ m_t &= m(h_t, w_t, \theta_t)\\ x_t &= x(h_t, w_t, \theta_t)\\ w_{t+1} &= \chi(h_t, w_t, \theta_t)\\ \theta_{t+1} &= \Psi(h_t, w_t, \theta_t)\end{aligned}$$

9.3 Calculating the Set of Sustainable Promise-Value Pairs

Above we defined the $\tilde{D}(Z)$ operator as (9.9).

Chang (1998) provides a method for dealing with the final three constraints.

These incentive constraints ensure that the government wants to choose $\hat{h}$ as the private sector had expected it to.

Chang's simplification starts from the idea that, when considering whether or not to confirm the private sector's expectation, the government only needs to consider the payoff of the *best* possible deviation.

Equally, to provide incentives to the government, we only need to consider the harshest possible punishment.

Let h denote some possible deviation. Chang defines:

$$P(h; Z) = \min u(f(x)) + v(m) + \beta w'$$

where the minimization is subject to

$$x = m(h-1)$$

$$m(h)(u'(f(x(h))) + v'(m(h))) \leq \beta\theta'(h) \text{ (with equality if } m(h) < \bar{m})\}$$

$$(m, x, w', \theta') \in [0, \bar{m}] \times X \times Z$$

For a given deviation h, this problem finds the worst possible sustainable value.

We then define:

$$BR(Z) = \max P(h; Z) \text{ subject to } h \in CE^0_\pi$$

$BR(Z)$ is the value of the government's most tempting deviation.

With this in hand, we can define a new operator $E(Z)$ that is equivalent to the $\tilde{D}(Z)$ operator but simpler to implement:

$$E(Z) = \Big\{(w, \theta) : \exists h \in CE^0_\pi \text{ and } (m(h), x(h), w'(h), \theta'(h)) \in [0, \bar{m}] \times X \times Z$$

such that

$$w = u(f(x(h))) + v(m(h)) + \beta w'(h)$$

$$\theta = u'(f(x(h)))(m(h) + x(h))$$

$$x(h) = m(h)(h - 1)$$

$$m(h)(u'(f(x(h))) - v'(m(h))) \leq \beta\theta'(h) \text{ (with equality if } m(h) < \bar{m})$$

and

$$w \geq BR(Z)\Big\}$$

Aside from the final incentive constraint, this is the same as the operator in *competitive equilibria in the Chang model*.

Consequently, to implement this operator we just need to add one step to our *outer hyperplane approximation algorithm* :

1. Initialize subgradients, H, and hyperplane levels, C_0.
2. Given a set of subgradients, H, and hyperplane levels, C_t, calculate $BR(S_t)$.
3. Given H, C_t, and $BR(S_t)$, for each subgradient $h_i \in H$:
   - Solve a linear program (described below) for each action in the action space.
   - Find the maximum and update the corresponding hyperplane level, $C_{i,t+1}$.
4. If $|C_{t+1} - C_t| > \epsilon$, return to 2.

Step 1 simply creates a large initial set S_0.

Given some set S_t, **Step 2** then constructs the value $BR(S_t)$.

To do this, we solve the following problem for each point in the action space (m_j, h_j):

$$\min_{[w', \theta']} u(f(x_j)) + v(m_j) + \beta w'$$

subject to

$$H \cdot (w', \theta') \leq C_t$$

$$x_j = m_j(h_j - 1)$$

$$m_j(u'(f(x_j)) - v'(m_j)) \leq \beta\theta' \quad (= \text{if } m_j < \bar{m})$$

This gives us a matrix of possible values, corresponding to each point in the action space.

To find $BR(Z)$, we minimize over the m dimension and maximize over the h dimension.

Step 3 then constructs the set $S_{t+1} = E(S_t)$. The linear program in Step 3 is designed to construct a set S_{t+1} that is as large as possible while satisfying the constraints of the $E(S)$ operator.

To do this, for each subgradient h_i, and for each point in the action space (m_j, h_j), we solve the following problem:

$$\max_{[w',\theta']} h_i \cdot (w,\theta)$$

subject to

$$H \cdot (w',\theta') \leq C_t$$

$$w = u(f(x_j)) + v(m_j) + \beta w'$$

$$\theta = u'(f(x_j))(m_j + x_j)$$

$$x_j = m_j(h_j - 1)$$

$$m_j(u'(f(x_j)) - v'(m_j)) \leq \beta\theta' \quad (= \text{if } m_j < \bar{m})$$

$$w \geq BR(Z)$$

This problem maximizes the hyperplane level for a given set of actions.

The second part of Step 3 then finds the maximum possible hyperplane level across the action space.

The algorithm constructs a sequence of progressively smaller sets $S_{t+1} \subset S_t \subset S_{t-1} \cdots \subset S_0$.

Step 4 ends the algorithm when the difference between these sets is small enough.

We have created a Python class that solves the model assuming the following functional forms:

$$u(c) = log(c)$$

$$v(m) = \frac{1}{500}(m\bar{m} - 0.5m^2)^{0.5}$$

$$f(x) = 180 - (0.4x)^2$$

The remaining parameters $\{\beta, \bar{m}, \underline{h}, \bar{h}\}$ are then variables to be specified for an instance of the Chang class.

Below we use the class to solve the model and plot the resulting equilibrium set, once with $\beta = 0.3$ and once with $\beta = 0.8$. We also plot the (larger) competitive equilibrium sets, which we described in *competitive equilibria in the Chang model*.

(We have set the number of subgradients to 10 in order to speed up the code for now. We can increase accuracy by increasing the number of subgradients)

The following code computes sustainable plans

```
"""
Provides a class called ChangModel to solve different
parameterizations of the Chang (1998) model.
"""

import numpy as np
import quantecon as qe
import time

from scipy.spatial import ConvexHull
from scipy.optimize import linprog, minimize, minimize_scalar
from scipy.interpolate import UnivariateSpline
import numpy.polynomial.chebyshev as cheb


class ChangModel:
```

(continues on next page)

(continued from previous page)

```python
    """
    Class to solve for the competitive and sustainable sets in the Chang (1998)
    model, for different parameterizations.
    """

    def __init__(self, β, mbar, h_min, h_max, n_h, n_m, N_g):
        # Record parameters
        self.β, self.mbar, self.h_min, self.h_max = β, mbar, h_min, h_max
        self.n_h, self.n_m, self.N_g = n_h, n_m, N_g

        # Create other parameters
        self.m_min = 1e-9
        self.m_max = self.mbar
        self.N_a = self.n_h*self.n_m

        # Utility and production functions
        uc = lambda c: np.log(c)
        uc_p = lambda c: 1/c
        v = lambda m: 1/500 * (mbar * m - 0.5 * m**2)**0.5
        v_p = lambda m: 0.5/500 * (mbar * m - 0.5 * m**2)**(-0.5) * (mbar - m)
        u = lambda h, m: uc(f(h, m)) + v(m)

        def f(h, m):
            x = m * (h - 1)
            f = 180 - (0.4 * x)**2
            return f

        def θ(h, m):
            x = m * (h - 1)
            θ = uc_p(f(h, m)) * (m + x)
            return θ

        # Create set of possible action combinations, A
        A1 = np.linspace(h_min, h_max, n_h).reshape(n_h, 1)
        A2 = np.linspace(self.m_min, self.m_max, n_m).reshape(n_m, 1)
        self.A = np.concatenate((np.kron(np.ones((n_m, 1)), A1),
                                 np.kron(A2, np.ones((n_h, 1)))), axis=1)

        # Pre-compute utility and output vectors
        self.euler_vec = -np.multiply(self.A[:, 1], \
            uc_p(f(self.A[:, 0], self.A[:, 1])) - v_p(self.A[:, 1]))
        self.u_vec = u(self.A[:, 0], self.A[:, 1])
        self.Θ_vec = θ(self.A[:, 0], self.A[:, 1])
        self.f_vec = f(self.A[:, 0], self.A[:, 1])
        self.bell_vec = np.multiply(uc_p(f(self.A[:, 0],
                                      self.A[:, 1])),
                                      np.multiply(self.A[:, 1],
                                      (self.A[:, 0] - 1))) \
                        + np.multiply(self.A[:, 1],
                                         v_p(self.A[:, 1]))

        # Find extrema of (w, θ) space for initial guess of equilibrium sets
        p_vec = np.zeros(self.N_a)
        w_vec = np.zeros(self.N_a)
        for i in range(self.N_a):
            p_vec[i] = self.Θ_vec[i]
```

(continues on next page)

(continued from previous page)

```
            w_vec[i] = self.u_vec[i]/(1 - β)

        w_space = np.array([min(w_vec[~np.isinf(w_vec)]),
                            max(w_vec[~np.isinf(w_vec)])])
        p_space = np.array([0, max(p_vec[~np.isinf(w_vec)])])
        self.p_space = p_space

        # Set up hyperplane levels and gradients for iterations
        def SG_H_V(N, w_space, p_space):
            """
            This function  initializes the subgradients, hyperplane levels,
            and extreme points of the value set by choosing an appropriate
            origin and radius. It is based on a similar function in QuantEcon's
            Games.jl
            """

            # First, create a unit circle. Want points placed on [0, 2π]
            inc = 2 * np.pi / N
            degrees = np.arange(0, 2 * np.pi, inc)

            # Points on circle
            H = np.zeros((N, 2))
            for i in range(N):
                x = degrees[i]
                H[i, 0] = np.cos(x)
                H[i, 1] = np.sin(x)

            # Then calculate origin and radius
            o = np.array([np.mean(w_space), np.mean(p_space)])
            r1 = max((max(w_space) - o[0])**2, (o[0] - min(w_space))**2)
            r2 = max((max(p_space) - o[1])**2, (o[1] - min(p_space))**2)
            r = np.sqrt(r1 + r2)

            # Now calculate vertices
            Z = np.zeros((2, N))
            for i in range(N):
                Z[0, i] = o[0] + r*H.T[0, i]
                Z[1, i] = o[1] + r*H.T[1, i]

            # Corresponding hyperplane levels
            C = np.zeros(N)
            for i in range(N):
                C[i] = np.dot(Z[:, i], H[i, :])

            return C, H, Z

        C, self.H, Z = SG_H_V(N_g, w_space, p_space)
        C = C.reshape(N_g, 1)
        self.c0_c, self.c0_s, self.c1_c, self.c1_s = np.copy(C), np.copy(C), \
            np.copy(C), np.copy(C)
        self.z0_s, self.z0_c, self.z1_s, self.z1_c = np.copy(Z), np.copy(Z), \
            np.copy(Z), np.copy(Z)

        self.w_bnds_s, self.w_bnds_c = (w_space[0], w_space[1]), \
            (w_space[0], w_space[1])
        self.p_bnds_s, self.p_bnds_c = (p_space[0], p_space[1]), \
```

(continues on next page)

(continued from previous page)

```
            (p_space[0], p_space[1])

        # Create dictionaries to save equilibrium set for each iteration
        self.c_dic_s, self.c_dic_c = {}, {}
        self.c_dic_s[0], self.c_dic_c[0] = self.c0_s, self.c0_c

    def solve_worst_spe(self):
        """
        Method to solve for BR(Z). See p.449 of Chang (1998)
        """

        p_vec = np.full(self.N_a, np.nan)
        c = [1, 0]

        # Pre-compute constraints
        aineq_mbar = np.vstack((self.H, np.array([0, -self.β])))
        bineq_mbar = np.vstack((self.c0_s, 0))

        aineq = self.H
        bineq = self.c0_s
        aeq = [[0, -self.β]]

        for j in range(self.N_a):
            # Only try if consumption is possible
            if self.f_vec[j] > 0:
                # If m = mbar, use inequality constraint
                if self.A[j, 1] == self.mbar:
                    bineq_mbar[-1] = self.euler_vec[j]
                    res = linprog(c, A_ub=aineq_mbar, b_ub=bineq_mbar,
                                  bounds=(self.w_bnds_s, self.p_bnds_s))
                else:
                    beq = self.euler_vec[j]
                    res = linprog(c, A_ub=aineq, b_ub=bineq, A_eq=aeq, b_eq=beq,
                                  bounds=(self.w_bnds_s, self.p_bnds_s))
                if res.status == 0:
                    p_vec[j] = self.u_vec[j] + self.β * res.x[0]

        # Max over h and min over other variables (see Chang (1998) p.449)
        self.br_z = np.nanmax(np.nanmin(p_vec.reshape(self.n_m, self.n_h), 0))

    def solve_subgradient(self):
        """
        Method to solve for E(Z). See p.449 of Chang (1998)
        """

        # Pre-compute constraints
        aineq_C_mbar = np.vstack((self.H, np.array([0, -self.β])))
        bineq_C_mbar = np.vstack((self.c0_c, 0))

        aineq_C = self.H
        bineq_C = self.c0_c
        aeq_C = [[0, -self.β]]

        aineq_S_mbar = np.vstack((np.vstack((self.H, np.array([0, -self.β]))),
                                  np.array([-self.β, 0])))
        bineq_S_mbar = np.vstack((self.c0_s, np.zeros((2, 1))))
```

(continues on next page)

(continued from previous page)

```
        aineq_S = np.vstack((self.H, np.array([-self.β, 0])))
        bineq_S = np.vstack((self.c0_s, 0))
        aeq_S = [[0, -self.β]]

        # Update maximal hyperplane level
        for i in range(self.N_g):
            c_a1a2_c, t_a1a2_c = np.full(self.N_a, -np.inf), \
                np.zeros((self.N_a, 2))
            c_a1a2_s, t_a1a2_s = np.full(self.N_a, -np.inf), \
                np.zeros((self.N_a, 2))

            c = [-self.H[i, 0], -self.H[i, 1]]

            for j in range(self.N_a):
                # Only try if consumption is possible
                if self.f_vec[j] > 0:

                    # COMPETITIVE EQUILIBRIA
                    # If m = mbar, use inequality constraint
                    if self.A[j, 1] == self.mbar:
                        bineq_C_mbar[-1] = self.euler_vec[j]
                        res = linprog(c, A_ub=aineq_C_mbar, b_ub=bineq_C_mbar,
                                      bounds=(self.w_bnds_c, self.p_bnds_c))
                    # If m < mbar, use equality constraint
                    else:
                        beq_C = self.euler_vec[j]
                        res = linprog(c, A_ub=aineq_C, b_ub=bineq_C, A_eq = aeq_C,
                                      b_eq = beq_C, bounds=(self.w_bnds_c, \
                                          self.p_bnds_c))
                    if res.status == 0:
                        c_a1a2_c[j] = self.H[i, 0] * (self.u_vec[j] \
                            + self.β * res.x[0]) + self.H[i, 1] * self.Θ_vec[j]
                        t_a1a2_c[j] = res.x

                    # SUSTAINABLE EQUILIBRIA
                    # If m = mbar, use inequality constraint
                    if self.A[j, 1] == self.mbar:
                        bineq_S_mbar[-2] = self.euler_vec[j]
                        bineq_S_mbar[-1] = self.u_vec[j] - self.br_z
                        res = linprog(c, A_ub=aineq_S_mbar, b_ub=bineq_S_mbar,
                                      bounds=(self.w_bnds_s, self.p_bnds_s))
                    # If m < mbar, use equality constraint
                    else:
                        bineq_S[-1] = self.u_vec[j] - self.br_z
                        beq_S = self.euler_vec[j]
                        res = linprog(c, A_ub=aineq_S, b_ub=bineq_S, A_eq = aeq_S,
                                      b_eq = beq_S, bounds=(self.w_bnds_s, \
                                          self.p_bnds_s))
                    if res.status == 0:
                        c_a1a2_s[j] = self.H[i, 0] * (self.u_vec[j] \
                            + self.β*res.x[0]) + self.H[i, 1] * self.Θ_vec[j]
                        t_a1a2_s[j] = res.x

            idx_c = np.where(c_a1a2_c == max(c_a1a2_c))[0][0]
            self.z1_c[:, i] = np.array([self.u_vec[idx_c]
```

(continues on next page)

(continued from previous page)

```
                                        + self.β * t_a1a2_c[idx_c, 0],
                                        self.Θ_vec[idx_c]])

            idx_s = np.where(c_a1a2_s == max(c_a1a2_s))[0][0]
            self.z1_s[:, i] = np.array([self.u_vec[idx_s]
                                        + self.β * t_a1a2_s[idx_s, 0],
                                        self.Θ_vec[idx_s]])

        for i in range(self.N_g):
            self.c1_c[i] = np.dot(self.z1_c[:, i], self.H[i, :])
            self.c1_s[i] = np.dot(self.z1_s[:, i], self.H[i, :])

    def solve_sustainable(self, tol=1e-5, max_iter=250):
        """
        Method to solve for the competitive and sustainable equilibrium sets.
        """

        t = time.time()
        diff = tol + 1
        iters = 0

        print('### --------------- ###')
        print('Solving Chang Model Using Outer Hyperplane Approximation')
        print('### --------------- ### \n')

        print('Maximum difference when updating hyperplane levels:')

        while diff > tol and iters < max_iter:
            iters = iters + 1
            self.solve_worst_spe()
            self.solve_subgradient()
            diff = max(np.maximum(abs(self.c0_c - self.c1_c),
                       abs(self.c0_s - self.c1_s)))
            print(diff)

            # Update hyperplane levels
            self.c0_c, self.c0_s = np.copy(self.c1_c), np.copy(self.c1_s)

            # Update bounds for w and θ
            wmin_c, wmax_c = np.min(self.z1_c, axis=1)[0], \
                np.max(self.z1_c, axis=1)[0]
            pmin_c, pmax_c = np.min(self.z1_c, axis=1)[1], \
                np.max(self.z1_c, axis=1)[1]

            wmin_s, wmax_s = np.min(self.z1_s, axis=1)[0], \
                np.max(self.z1_s, axis=1)[0]
            pmin_S, pmax_S = np.min(self.z1_s, axis=1)[1], \
                np.max(self.z1_s, axis=1)[1]

            self.w_bnds_s, self.w_bnds_c = (wmin_s, wmax_s), (wmin_c, wmax_c)
            self.p_bnds_s, self.p_bnds_c = (pmin_S, pmax_S), (pmin_c, pmax_c)

            # Save iteration
            self.c_dic_c[iters], self.c_dic_s[iters] = np.copy(self.c1_c), \
                np.copy(self.c1_s)
            self.iters = iters
```

(continues on next page)

(continued from previous page)

```
        elapsed = time.time() - t
        print('Convergence achieved after {} iterations and {} \
            seconds'.format(iters, round(elapsed, 2)))

    def solve_bellman(self, θ_min, θ_max, order, disp=False, tol=1e-7, maxiters=100):
        """
        Continuous Method to solve the Bellman equation in section 25.3
        """
        mbar = self.mbar

        # Utility and production functions
        uc = lambda c: np.log(c)
        uc_p = lambda c: 1 / c
        v = lambda m: 1 / 500 * (mbar * m - 0.5 * m**2)**0.5
        v_p = lambda m: 0.5/500 * (mbar*m - 0.5 * m**2)**(-0.5) * (mbar - m)
        u = lambda h, m: uc(f(h, m)) + v(m)

        def f(h, m):
            x = m * (h - 1)
            f = 180 - (0.4 * x)**2
            return f

        def θ(h, m):
            x = m * (h - 1)
            θ = uc_p(f(h, m)) * (m + x)
            return θ

        # Bounds for Maximization
        lb1 = np.array([self.h_min, 0, θ_min])
        ub1 = np.array([self.h_max, self.mbar - 1e-5, θ_max])
        lb2 = np.array([self.h_min, θ_min])
        ub2 = np.array([self.h_max, θ_max])

        # Initialize Value Function coefficients
        # Calculate roots of Chebyshev polynomial
        k = np.linspace(order, 1, order)
        roots = np.cos((2 * k - 1) * np.pi / (2 * order))
        # Scale to approximation space
        s = θ_min + (roots - -1) / 2 * (θ_max - θ_min)
        # Create a basis matrix
        Φ = cheb.chebvander(roots, order - 1)
        c = np.zeros(Φ.shape[0])

        # Function to minimize and constraints
        def p_fun(x):
            scale = -1 + 2 * (x[2] - θ_min)/(θ_max - θ_min)
            p_fun = - (u(x[0], x[1]) \
                + self.β * np.dot(cheb.chebvander(scale, order - 1), c))
            return p_fun

        def p_fun2(x):
            scale = -1 + 2*(x[1] - θ_min)/(θ_max - θ_min)
            p_fun = - (u(x[0],mbar) \
                + self.β * np.dot(cheb.chebvander(scale, order - 1), c))
            return p_fun
```

(continues on next page)

(continued from previous page)

```
        cons1 = ({'type': 'eq',   'fun': lambda x: uc_p(f(x[0], x[1])) * x[1]
                    * (x[0] - 1) + v_p(x[1]) * x[1] + self.β * x[2] - θ},
                 {'type': 'eq',   'fun': lambda x: uc_p(f(x[0], x[1]))
                    * x[0] * x[1] - θ})
        cons2 = ({'type': 'ineq', 'fun': lambda x: uc_p(f(x[0], mbar)) * mbar
                    * (x[0] - 1) + v_p(mbar) * mbar + self.β * x[1] - θ},
                 {'type': 'eq',   'fun': lambda x: uc_p(f(x[0], mbar))
                    * x[0] * mbar - θ})

        bnds1 = np.concatenate([lb1.reshape(3, 1), ub1.reshape(3, 1)], axis=1)
        bnds2 = np.concatenate([lb2.reshape(2, 1), ub2.reshape(2, 1)], axis=1)

        # Bellman Iterations
        diff = 1
        iters = 1

        while diff > tol:
        # 1. Maximization, given value function guess
            p_iter1 = np.zeros(order)
            for i in range(order):
                θ = s[i]
                res = minimize(p_fun,
                               lb1 + (ub1-lb1) / 2,
                               method='SLSQP',
                               bounds=bnds1,
                               constraints=cons1,
                               tol=1e-10)
                if res.success == True:
                    p_iter1[i] = -p_fun(res.x)
                res = minimize(p_fun2,
                               lb2 + (ub2-lb2) / 2,
                               method='SLSQP',
                               bounds=bnds2,
                               constraints=cons2,
                               tol=1e-10)
                if -p_fun2(res.x) > p_iter1[i] and res.success == True:
                    p_iter1[i] = -p_fun2(res.x)

            # 2. Bellman updating of Value Function coefficients
            c1 = np.linalg.solve(Φ, p_iter1)
            # 3. Compute distance and update
            diff = np.linalg.norm(c - c1)
            if bool(disp == True):
                print(diff)
            c = np.copy(c1)
            iters = iters + 1
            if iters > maxiters:
                print('Convergence failed after {} iterations'.format(maxiters))
                break

        self.θ_grid = s
        self.p_iter = p_iter1
        self.Φ = Φ
        self.c = c
        print('Convergence achieved after {} iterations'.format(iters))
```

(continues on next page)

(continued from previous page)

```
        # Check residuals
        θ_grid_fine = np.linspace(θ_min, θ_max, 100)
        resid_grid = np.zeros(100)
        p_grid = np.zeros(100)
        θ_prime_grid = np.zeros(100)
        m_grid = np.zeros(100)
        h_grid = np.zeros(100)
        for i in range(100):
            θ = θ_grid_fine[i]
            res = minimize(p_fun,
                           lb1 + (ub1-lb1) / 2,
                           method='SLSQP',
                           bounds=bnds1,
                           constraints=cons1,
                           tol=1e-10)
            if res.success == True:
                p = -p_fun(res.x)
                p_grid[i] = p
                θ_prime_grid[i] = res.x[2]
                h_grid[i] = res.x[0]
                m_grid[i] = res.x[1]
            res = minimize(p_fun2,
                           lb2 + (ub2-lb2)/2,
                           method='SLSQP',
                           bounds=bnds2,
                           constraints=cons2,
                           tol=1e-10)
            if -p_fun2(res.x) > p and res.success == True:
                p = -p_fun2(res.x)
                p_grid[i] = p
                θ_prime_grid[i] = res.x[1]
                h_grid[i] = res.x[0]
                m_grid[i] = self.mbar
            scale = -1 + 2 * (θ - θ_min)/(θ_max - θ_min)
            resid_grid[i] = np.dot(cheb.chebvander(scale, order-1), c) - p

        self.resid_grid = resid_grid
        self.θ_grid_fine = θ_grid_fine
        self.θ_prime_grid = θ_prime_grid
        self.m_grid = m_grid
        self.h_grid = h_grid
        self.p_grid = p_grid
        self.x_grid = m_grid * (h_grid - 1)

        # Simulate
        θ_series = np.zeros(31)
        m_series = np.zeros(30)
        h_series = np.zeros(30)

        # Find initial θ
        def ValFun(x):
            scale = -1 + 2*(x - θ_min)/(θ_max - θ_min)
            p_fun = np.dot(cheb.chebvander(scale, order - 1), c)
            return -p_fun
```

(continues on next page)

(continued from previous page)

```
        res = minimize(ValFun,
                       (θ_min + θ_max)/2,
                       bounds=[(θ_min, θ_max)])
        θ_series[0] = res.x

        # Simulate
        for i in range(30):
            θ = θ_series[i]
            res = minimize(p_fun,
                           lb1 + (ub1-lb1)/2,
                           method='SLSQP',
                           bounds=bnds1,
                           constraints=cons1,
                           tol=1e-10)
            if res.success == True:
                p = -p_fun(res.x)
                h_series[i] = res.x[0]
                m_series[i] = res.x[1]
                θ_series[i+1] = res.x[2]
            res2 = minimize(p_fun2,
                            lb2 + (ub2-lb2)/2,
                            method='SLSQP',
                            bounds=bnds2,
                            constraints=cons2,
                            tol=1e-10)
            if -p_fun2(res2.x) > p and res2.success == True:
                h_series[i] = res2.x[0]
                m_series[i] = self.mbar
                θ_series[i+1] = res2.x[1]

        self.θ_series = θ_series
        self.m_series = m_series
        self.h_series = h_series
        self.x_series = m_series * (h_series - 1)
```

9.3.1 Comparison of Sets

The set of (w, θ) associated with sustainable plans is smaller than the set of (w, θ) pairs associated with competitive equilibria, since the additional constraints associated with sustainability must also be satisfied.

Let's compute two examples, one with a low β, another with a higher β

```
ch1 = ChangModel(β=0.3, mbar=30, h_min=0.9, h_max=2, n_h=8, n_m=35, N_g=10)
```

```
ch1.solve_sustainable()
```

```
### --------------- ###
Solving Chang Model Using Outer Hyperplane Approximation
### --------------- ###

Maximum difference when updating hyperplane levels:
```

```
[1.9168]
```

```
[0.66782]
```

```
[0.49235]
```

```
[0.32412]
```

```
[0.19022]
```

```
[0.10863]
```

```
[0.05817]
```

```
[0.0262]
```

```
[0.01836]
```

```
[0.01415]
```

```
[0.00297]
```

```
[0.00089]
```

```
[0.00027]
```

```
[0.00008]
```

```
[0.00002]
```

```
[0.00001]
Convergence achieved after 16 iterations and 42.62             seconds
```

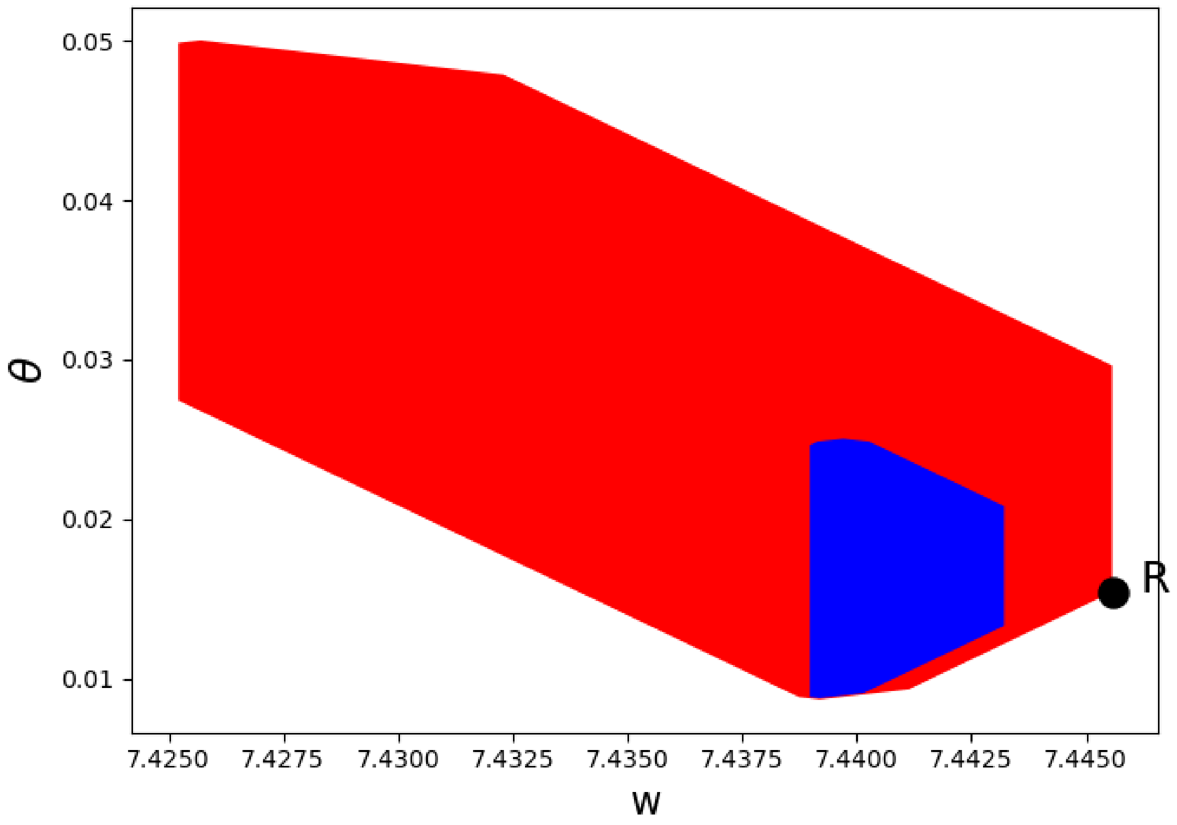
The following plot shows both the set of w, θ pairs associated with competitive equilibria (in red) and the smaller set of w, θ pairs associated with sustainable plans (in blue).

```python
def plot_equilibria(ChangModel):
    """
    Method to plot both equilibrium sets
    """
    fig, ax = plt.subplots(figsize=(7, 5))

    ax.set_xlabel('w', fontsize=16)
    ax.set_ylabel(r"$\theta$", fontsize=18)

```

(continues on next page)

(continued from previous page)

```
    poly_S = polytope.Polytope(ChangModel.H, ChangModel.c1_s)
    poly_C = polytope.Polytope(ChangModel.H, ChangModel.c1_c)
    ext_C = polytope.extreme(poly_C)
    ext_S = polytope.extreme(poly_S)

    ax.fill(ext_C[:, 0], ext_C[:, 1], 'r', zorder=-1)
    ax.fill(ext_S[:, 0], ext_S[:, 1], 'b', zorder=0)

    # Add point showing Ramsey Plan
    idx_Ramsey = np.where(ext_C[:, 0] == max(ext_C[:, 0]))[0][0]
    R = ext_C[idx_Ramsey, :]
    ax.scatter(R[0], R[1], 150, 'black', 'o', zorder=1)
    w_min = min(ext_C[:, 0])

    # Label Ramsey Plan slightly to the right of the point
    ax.annotate("R", xy=(R[0], R[1]),
                xytext=(R[0] + 0.03 * (R[0] - w_min),
                R[1]), fontsize=18)

    plt.tight_layout()
    plt.show()

plot_equilibria(ch1)
```



Evidently, the Ramsey plan, denoted by the R, is not sustainable.

Let's raise the discount factor and recompute the sets

```
ch2 = ChangModel(β=0.8, mbar=30, h_min=0.9, h_max=1/0.8,
    n_h=8, n_m=35, N_g=10)
```

```
ch2.solve_sustainable()
```

```
### --------------- ###
Solving Chang Model Using Outer Hyperplane Approximation
### --------------- ###

Maximum difference when updating hyperplane levels:
```

```
[0.06369]
```

```
[0.02476]
```

```
[0.02153]
```

```
[0.01915]
```

```
[0.01795]
```

```
[0.01642]
```

```
[0.01507]
```

```
[0.01284]
```

```
[0.01106]
```

```
[0.00694]
```

```
[0.0085]
```

```
[0.00781]
```

```
[0.00433]
```

```
[0.00492]
```

```
[0.00303]
```

```
[0.00182]
```

```
[0.00638]
```

```
[0.00116]
```

```
[0.00093]
```

```
[0.00075]
```

```
[0.0006]
```

```
[0.00494]
```

```
[0.00038]
```

```
[0.00121]
```

```
[0.00024]
```

```
[0.0002]
```

```
[0.00016]
```

```
[0.00013]
```

```
[0.0001]
```

```
[0.00008]
```

```
[0.00006]
```

```
[0.00005]
```

```
[0.00004]
```

```
[0.00003]
```

```
[0.00003]
```

```
[0.00002]
```

```
[0.00002]
```

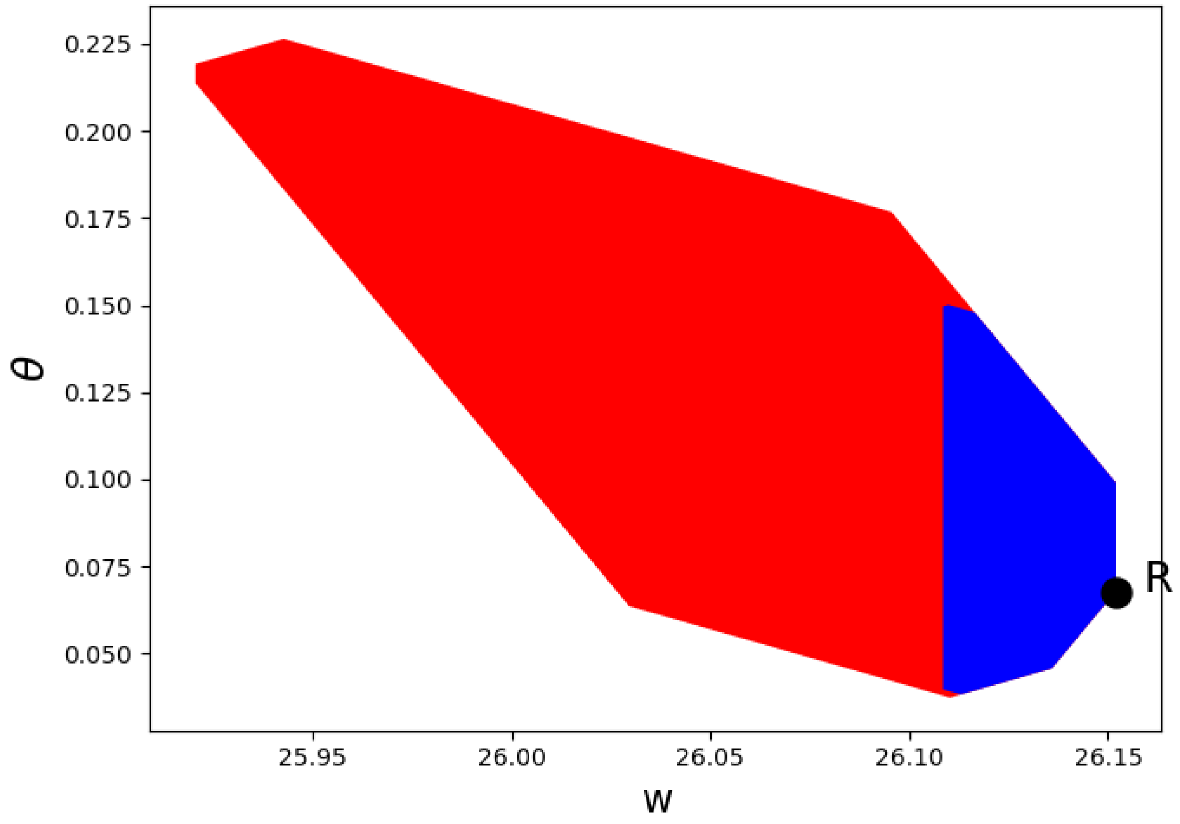
```
[0.00001]
```

```
[0.00001]
```

```
[0.00001]
Convergence achieved after 40 iterations and 123.73             seconds
```

Let's plot both sets

```
plot_equilibria(ch2)
```



Evidently, the Ramsey plan is now sustainable.

Part II

Other

CHAPTER TEN

TROUBLESHOOTING

Contents

This page is for readers experiencing errors when running the code from the lectures.

10.1 Fixing Your Local Environment

The basic assumption of the lectures is that code in a lecture should execute whenever

1. it is executed in a Jupyter notebook and
2. the notebook is running on a machine with the latest version of Anaconda Python.

You have installed Anaconda, haven't you, following the instructions in this lecture?

Assuming that you have, the most common source of problems for our readers is that their Anaconda distribution is not up to date.

Here's a useful article on how to update Anaconda.

Another option is to simply remove Anaconda and reinstall.

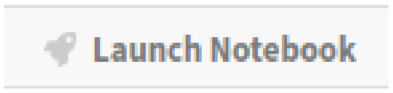
You also need to keep the external code libraries, such as QuantEcon.py up to date.

For this task you can either

- use conda install -y quantecon on the command line, or
- execute !conda install -y quantecon within a Jupyter notebook.

If your local environment is still not working you can do two things.

First, you can use a remote machine instead, by clicking on the Launch Notebook icon available for each lecture



Second, you can report an issue, so we can try to fix your local set up.

We like getting feedback on the lectures so please don't hesitate to get in touch.

10.2 Reporting an Issue

One way to give feedback is to raise an issue through our issue tracker.

Please be as specific as possible. Tell us where the problem is and as much detail about your local set up as you can provide.

Another feedback option is to use our discourse forum.

Finally, you can provide direct feedback to contact@quantecon.org

CHAPTER

ELEVEN

REFERENCES

CHAPTER

TWELVE

EXECUTION STATISTICS

This table contains the latest execution statistics.

Document	Modified	Method	Run Time (s)	Status
amss	2024-05-01 01:41	cache	193.55	✅
amss2	2024-05-01 01:42	cache	54.87	✅
amss3	2024-05-01 01:46	cache	244.51	✅
calvo	2024-05-01 01:46	cache	6.01	✅
chang_credible	2024-05-01 01:49	cache	172.4	✅
chang_ramsey	2024-05-01 01:55	cache	337.37	✅
dyn_stack	2024-05-01 01:55	cache	7.28	✅
intro	2024-05-01 01:55	cache	4.02	✅
opt_tax_recur	2024-05-01 01:56	cache	72.12	✅
status	2024-05-01 01:56	cache	4.87	✅
troubleshooting	2024-05-01 01:55	cache	4.02	✅
un_insure	2024-05-01 01:57	cache	11.48	✅
zreferences	2024-05-01 01:55	cache	4.02	✅

These lectures are built on `linux` instances through `github actions`.

These lectures are using the following python version

```
!python --version
```

```
Python 3.11.7
```

and the following package versions

```
!conda list
```

BIBLIOGRAPHY

[Abr88] Dilip Abreu. On the theory of infinitely repeated games with discounting. *Econometrica*, 56:383–396, 1988.

[APS90] Dilip Abreu, David Pearce, and Ennio Stacchetti. Toward a theory of discounted repeated games with imperfect monitoring. *Econometrica*, 58(5):1041–1063, September 1990.

[AMSSeppala02] S Rao Aiyagari, Albert Marcet, Thomas J Sargent, and Juha Seppälä. Optimal taxation without state-contingent debt. *Journal of Political Economy*, 110(6):1220–1254, 2002.

[Bar79] Robert J Barro. On the Determination of the Public Debt. *Journal of Political Economy*, 87(5):940–971, 1979.

[BEGS17] Anmol Bhandari, David Evans, Mikhail Golosov, and Thomas J. Sargent. Fiscal Policy and Debt Management with Incomplete Markets. *The Quarterly Journal of Economics*, 132(2):617–663, 2017.

[Cag56] Philip Cagan. The monetary dynamics of hyperinflation. In Milton Friedman, editor, *Studies in the Quantity Theory of Money*, pages 25–117. University of Chicago Press, Chicago, 1956.

[Cal78] Guillermo A. Calvo. On the time consistency of optimal policy in a monetary economy. *Econometrica*, 46(6):1411–1428, 1978.

[Cha98] Roberto Chang. Credible monetary policy in an infinite horizon model: recursive approaches. *Journal of Economic Theory*, 81(2):431–461, 1998.

[CK90] Varadarajan V Chari and Patrick J Kehoe. Sustainable plans. *Journal of Political Economy*, pages 783–802, 1990.

[HN97] Hugo A Hopenhayn and Juan Pablo Nicolini. Optimal Unemployment Insurance. *Journal of Political Economy*, 105(2):412–438, April 1997. URL: https://ideas.repec.org/a/ucp/jpolec/v105y1997i2p412-38.html, doi:10.1086/262078.

[Jud98] K L Judd. *Numerical Methods in Economics*. Scientific and Engineering. MIT Press, 1998.

[JYC03] Kenneth L. Judd, Sevin Yeltekin, and James Conklin. Computing Supergame Equilibria. *Econometrica*, 71(4):1239–1254, 07 2003. URL: https://ideas.repec.org/a/ecm/emetrp/v71y2003i4p1239-1254.html, doi:.

[KP80] Finn E Kydland and Edward C Prescott. Dynamic optimal taxation, rational expectations and optimal control. *Journal of Economic Dynamics and Control*, 2:79–91, 1980.

[LS18] L Ljungqvist and T J Sargent. *Recursive Macroeconomic Theory*. MIT Press, 4 edition, 2018.

[LS83] Robert E Lucas, Jr. and Nancy L Stokey. Optimal Fiscal and Monetary Policy in an Economy without Capital. *Journal of monetary Economics*, 12(3):55–93, 1983.

[Sar77] Thomas J Sargent. The Demand for Money During Hyperinflations under Rational Expectations: I. *International Economic Review*, 18(1):59–82, February 1977.

[Sar87] Thomas J Sargent. *Macroeconomic Theory*. Academic Press, New York, 2nd edition, 1987.

[SW79] Steven Shavell and Laurence Weiss. The optimal payment of unemployment insurance benefits over time. *Journal of political Economy*, 87(6):1347–1362, 1979.

[Sto89] Nancy L Stokey. Reputation and time consistency. *The American Economic Review*, pages 134–139, 1989.

[Sto91] Nancy L. Stokey. Credible public policy. *Journal of Economic Dynamics and Control*, 15(4):627–656, October 1991.

INDEX