# Optimal Taxation without State-Contingent Debt

Thomas J. Sargent and John Stachurski

November 15, 2020

## 1 Contents



**Software Requirement:**

This lecture requires the use of some older software versions to run. If you would like to execute this lecture please download the following amss_environment.yml file. This specifies the software required and an environment can be created using conda:

Open a terminal:

```
conda env create --file amss_environment.yml
conda activate amss
```

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

```
In [1]: !pip install --upgrade quantecon
```

## 2 Overview

Let's start with following imports:

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
        from scipy.optimize import root, fmin_slsqp
        from scipy.interpolate import UnivariateSpline
        from quantecon import MarkovChain
```

In an earlier lecture, we described a model of optimal taxation with state-contingent debt due to Robert E. Lucas, Jr., and Nancy Stokey [3].

Aiyagari, Marcet, Sargent, and Seppälä [1] (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.

## 3 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 $s^t = [s_t, s_{t-1}, \ldots, s_0]$.

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 $\ell_t$ and labor $n_t$:

$$n_t(s^t) + \ell_t(s^t) = 1 \tag{1}$$

Output equals $n_t(s^t)$ and can be divided between consumption $c_t(s^t)$ and $g(s_t)$

$$c_t(s^t) + g(s_t) = n_t(s^t) \tag{2}$$

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

$$\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) u[c_t(s^t), \ell_t(s^t)] \tag{3}$$

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 [1] 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 [2] than they do in [3].

### 3.1 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 $t$ 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 Section **??**

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

$$\begin{aligned} b_t(s^{t-1}) &= \tau_t^n(s^t) n_t(s^t) - g_t(s_t) - T_t(s^t) + \frac{b_{t+1}(s^t)}{R_t(s^t)} \\ &\equiv z(s^t) + \frac{b_{t+1}(s^t)}{R_t(s^t)}, \end{aligned} \tag{4}$$

where $z(s^t)$ 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 (4) yields:

$$b_t(s^{t-1}) = z(s^t) + \beta \sum_{s^{t+1}|s^t} \pi_{t+1}(s^{t+1}|s^t) \frac{u_c(s^{t+1})}{u_c(s^t)} \, b_{t+1}(s^t) \tag{5}$$

Components of $z(s^t)$ on the right side depend on $s^t$, but the left side is required to depend on $s^{t-1}$ only.

**This is what it means for one-period government debt to be risk-free**.

Therefore, the sum on the right side of equation (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) by the right side of next period's budget constraint (associated with a particular realization $s_t$) we get

$$b_t(s^{t-1}) = z(s^t) + \sum_{s^{t+1}|s^t} \beta \pi_{t+1}(s^{t+1}|s^t) \frac{u_c(s^{t+1})}{u_c(s^t)} \left[ z(s^{t+1}) + \frac{b_{t+2}(s^{t+1})}{R_{t+1}(s^{t+1})} \right]$$

After making similar repeated substitutions for all future occurrences of government indebtedness, and by invoking the natural debt limit, we arrive at:

$$b_t(s^{t-1}) = \sum_{j=0}^{\infty} \sum_{s^{t+j}|s^t} \beta^j \pi_{t+j}(s^{t+j}|s^t) \frac{u_c(s^{t+j})}{u_c(s^t)} \, z(s^{t+j}) \tag{6}$$

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 $z(s^t)$ as

$$z(s^t) = \left[1 - \frac{u_\ell(s^t)}{u_c(s^t)}\right] [c_t(s^t) + g_t(s_t)] - g_t(s_t) - T_t(s^t) \, . \tag{7}$$

If we substitute the appropriate versions of the right side of (7) for $z(s^{t+j})$ into equation (6), we obtain a sequence of *implementability constraints* on a Ramsey allocation in an AMSS economy.

Expression (6) at time $t = 0$ and initial state $s^0$ was also an *implementability constraint* on a Ramsey allocation in a Lucas-Stokey economy:

$$b_0(s^{-1}) = \mathbb{E}_0 \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^j)}{u_c(s^0)} \, z(s^j) \tag{8}$$

Indeed, it was the *only* implementability constraint there.

But now we also have a large number of additional implementability constraints

$$b_t(s^{t-1}) = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} \, z(s^{t+j}) \tag{9}$$

Equation (9) must hold for each $s^t$ for each $t \geq 1$.

### 3.2 Comparison with Lucas-Stokey Economy

The expression on the right side of (9) in the Lucas-Stokey (1983) economy would equal the present value of a continuation stream of government 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 surpluses in the Lucas-Stokey model must belong to the **marketable subspace** of the AMSS model.

### 3.3 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

$$\max_{\{c_t(s^t),b_{t+1}(s^t)\}} \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t u\left(c_t(s^t), 1 - c_t(s^t) - g_t(s_t)\right)$$

where the maximization is subject to

$$\mathbb{E}_0 \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^j)}{u_c(s^0)} \, z(s^j) \geq b_0(s^{-1}) \tag{10}$$

and

$$\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} \, z(s^{t+j}) = b_t(s^{t-1}) \quad \forall \, s^t \tag{11}$$

given $b_0(s^{-1})$.

#### 3.3.1 Lagrangian Formulation

Let $\gamma_0(s^0)$ be a non-negative Lagrange multiplier on constraint (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 stochastic processes $\{\gamma_t(s^t)\}_{t=1}^{\infty}$ of Lagrange multipliers to the implementability constraints (11).

Depending on how the constraints bind, these multipliers can be positive or negative:

$$\begin{aligned} \gamma_t(s^t) \;&\geq\; (\leq)\; 0 \quad \text{if the constraint binds in this direction} \\ \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} \, z(s^{t+j}) \;&\geq\; (\leq)\; b_t(s^{t-1}) \end{aligned}$$

A negative multiplier $\gamma_t(s^t) < 0$ means that if we could relax constraint (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 Section **??**.

These features flow from the fact that the government cannot use state-contingent debt and therefore cannot allocate its indebtedness efficiently across future states.

### 3.4 Some Calculations

It is helpful to apply two transformations to the Lagrangian.

Multiply constraint (10) by $u_c(s^0)$ and the constraints (11) by $\beta^t u_c(s^t)$.

Then a Lagrangian for the Ramsey problem can be represented as

$$\begin{aligned} J &= \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \Big\{ u\left(c_t(s^t), 1 - c_t(s^t) - g_t(s_t)\right) \\ &\quad + \gamma_t(s^t) \Big[ \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j u_c(s^{t+j})\, z(s^{t+j}) - u_c(s^t)\, b_t(s^{t-1}) \Big] \Big\} \\ &= \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \Big\{ u\left(c_t(s^t), 1 - c_t(s^t) - g_t(s_t)\right) \\ &\quad + \Psi_t(s^t)\, u_c(s^t)\, z(s^t) - \gamma_t(s^t)\, u_c(s^t)\, b_t(s^{t-1}) \Big\} \end{aligned} \tag{12}$$

where

$$\Psi_t(s^t) = \Psi_{t-1}(s^{t-1}) + \gamma_t(s^t) \quad \text{and} \quad \Psi_{-1}(s^{-1}) = 0 \tag{13}$$

In (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

$$\begin{aligned} u_c(s^t) - u_\ell(s^t) + \Psi_t(s^t) \left\{ \left[ u_{cc}(s^t) - u_{c\ell}(s^t) \right] z(s^t) + u_c(s^t)\, z_c(s^t) \right\} \\ - \gamma_t(s^t) \left[ u_{cc}(s^t) - u_{c\ell}(s^t) \right] b_t(s^{t-1}) = 0 \end{aligned} \tag{14}$$

and with respect to $b_t(s^t)$ as

$$\mathbb{E}_t \left[ \gamma_{t+1}(s^{t+1})\, u_c(s^{t+1}) \right] = 0 \tag{15}$$

If we substitute $z(s^t)$ from (7) and its derivative $z_c(s^t)$ into the first-order condition (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 (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.

1. The Lagrange multiplier $\Psi_t(s^t)$ in the first-order condition (14) may change over time in response to realizations of the state, while the multiplier $\Phi$ in the Lucas-Stokey economy is time-invariant.

We need some code from our an earlier lecture on optimal taxation with state-contingent debt sequential allocation implementation:

In [3]:
```
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, 0.5 * np.ones(2 * S))

        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

    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)

        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 np.array([cHist, nHist, Bhist, THist, sHist, μHist, 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.

## 4 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 (8) from the Lucas-Stokey economy, but
- adds measurability constraints (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.

### 4.1 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 makes a possibly important difference from AMSS [1], who restricted transfers to be non-negative).

In this case, the household faces a sequence of budget constraints

$$b_t(s^{t-1}) + (1-\tau_t(s^t))n_t(s^t) = c_t(s^t) + b_{t+1}(s^t)/R_t(s^t) \tag{16}$$

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 $\tau_t$ from budget constraint (16) gives

$$b_t(s^{t-1}) + \frac{u_{l,t}(s^t)}{u_{c,t}(s^t)}n_t(s^t) = c_t(s^t) + \frac{\beta(\mathbb{E}_t u_{c,t+1})b_{t+1}(s^t)}{u_{c,t}(s^t)} \tag{17}$$

or

$$u_{c,t}(s^t)b_t(s^{t-1}) + u_{l,t}(s^t)n_t(s^t) = u_{c,t}(s^t)c_t(s^t) + \beta(\mathbb{E}_t u_{c,t+1})b_{t+1}(s^t) \tag{18}$$

Now define

$$x_t \equiv \beta b_{t+1}(s^t)\mathbb{E}_t u_{c,t+1} = u_{c,t}(s^t)\frac{b_{t+1}(s^t)}{R_t(s^t)} \tag{19}$$

and represent the household's budget constraint at time $t$, history $s^t$ as

$$\frac{u_{c,t}x_{t-1}}{\beta \mathbb{E}_{t-1}u_{c,t}} = u_{c,t}c_t - u_{l,t}n_t + x_t \tag{20}$$

for $t \geq 1$.

### 4.2 Measurability Constraints

Write equation (18) as

$$b_t(s^{t-1}) = c_t(s^t) - \frac{u_{l,t}(s^t)}{u_{c,t}(s^t)} n_t(s^t) + \frac{\beta(\mathbb{E}_t u_{c,t+1}) b_{t+1}(s^t)}{u_{c,t}} \tag{21}$$

The right side of equation (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 (21) must equal $b_t(s^{t-1})$.

That implies that it has to be *measurable* with respect to $s^{t-1}$.

Equations (21) are the *measurability constraints* that the AMSS model adds to the single time 0 implementation constraint imposed in the Lucas and Stokey model.

### 4.3 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

$$V(x_-, s_-) = \max_{\{n(s), x(s)\}} \sum_s \Pi(s|s_-) \left[ u(n(s) - g(s), 1 - n(s)) + \beta V(x(s), s) \right] \tag{22}$$

subject to the following collection of implementability constraints, one for each $s \in S$:

$$\frac{u_c(s) x_-}{\beta \sum_{\tilde{s}} \Pi(\tilde{s}|s_-) u_c(\tilde{s})} = u_c(s)(n(s) - g(s)) - u_l(s) n(s) + x(s) \tag{23}$$

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

$$W(b_0, s_0) = \max_{n_0, x_0} u(n_0 - g_0, 1 - n_0) + \beta V(x_0, s_0) \tag{24}$$

where maximization is subject to

$$u_{c,0} b_0 = u_{c,0}(n_0 - g_0) - u_{l,0} n_0 + x_0 \tag{25}$$

### 4.4 Martingale Supercedes State-Variable Degeneracy

Let $\mu(s|s_-)\Pi(s|s_-)$ be a Lagrange multiplier on the constraint (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

$$\beta V_x(x(s), s) = \mu(s|s_-) \tag{26}$$

Applying the envelope theorem to Bellman equation (22) gives

$$V_x(x_-, s_-) = \sum_s \Pi(s|s_-)\mu(s|s_-)\frac{u_c(s)}{\beta \sum_{\tilde{s}} \Pi(\tilde{s}|s_-)u_c(\tilde{s})} \tag{27}$$

Equations (26) and (27) imply that

$$V_x(x_-, s_-) = \sum_s \left( \Pi(s|s_-)\frac{u_c(s)}{\sum_{\tilde{s}} \Pi(\tilde{s}|s_-)u_c(\tilde{s})} \right) V_x(x(s), s) \tag{28}$$

Equation (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**: 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.

### 4.5 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 (i.e., $x_t$ is an exact 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.

### 4.6 Digression on Non-negative Transfers

Throughout this lecture, we have imposed that transfers $T_t = 0$.

AMSS [1] 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 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.

### 4.7 Code

The recursive formulation is implemented as follows

In [4]:
```
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-4, tol=1e-4):

        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

            # 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):
        '''
        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 np.array([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, 0.5 * np.ones(2 * S))
        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):
                    Vprime[s] = Vf[s](xprime[s])

                return -π[s_] @ (U(c, n) + β * Vprime)

            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,
                                                  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])
```

## 5 Examples

We now turn to some examples.

We will first build some useful functions for solving the model

```
In [5]: 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):
        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
```

## 5.1 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 consider the same government expenditure process studied in the 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.

In [6]:
```
import numpy as np


class CRRAutility:

    def __init__(self,
                 β=0.9,
                 σ=2,
                 γ=2,
                 π=0.5*np.ones((2, 2)),
                 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)
```

The following figure plots the Ramsey plan under both complete and incomplete markets for both possible realizations of the state at time $t = 3$.

Optimal policies when the government has access to state contingent debt are represented by black lines, while the optimal policies when there is only a risk-free bond are in red.

Paths with circles are histories in which there is peace, while those with triangle denote war.

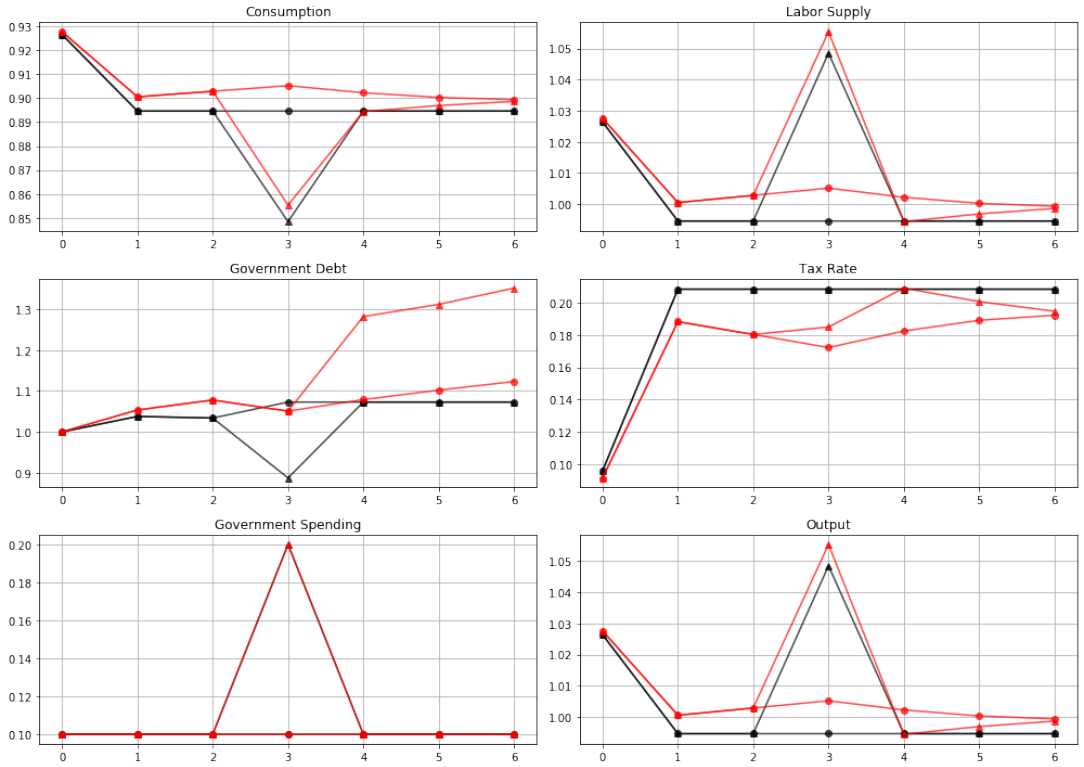
In [7]:
```
# Initialize μgrid for value function iteration
μ_grid = np.linspace(-0.7, 0.01, 200)

time_example = CRRAutility()

time_example.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]])
```

```
    time_example.G = np.array([0.1, 0.1, 0.1, 0.2, 0.1, 0.1])
    time_example.Θ = np.ones(6)  # Θ can in principle be random

    time_example.transfers = True              # Government can use transfers
    # Solve sequential problem
    time_sequential = SequentialAllocation(time_example)
    # Solve recursive problem
    time_bellman = RecursiveAllocationAMSS(time_example, μ_grid)

    sHist_h = np.array([0, 1, 2, 3, 5, 5, 5])
    sHist_l = np.array([0, 1, 2, 4, 5, 5, 5])

    sim_seq_h = time_sequential.simulate(1, 0, 7, sHist_h)
    sim_bel_h = time_bellman.simulate(1, 0, 7, sHist_h)
    sim_seq_l = time_sequential.simulate(1, 0, 7, sHist_l)
    sim_bel_l = time_bellman.simulate(1, 0, 7, sHist_l)

    # Government spending paths
    sim_seq_l[4] = time_example.G[sHist_l]
    sim_seq_h[4] = time_example.G[sHist_h]
    sim_bel_l[4] = time_example.G[sHist_l]
    sim_bel_h[4] = time_example.G[sHist_h]

    # Output paths
    sim_seq_l[5] = time_example.Θ[sHist_l] * sim_seq_l[1]
    sim_seq_h[5] = time_example.Θ[sHist_h] * sim_seq_h[1]
    sim_bel_l[5] = time_example.Θ[sHist_l] * sim_bel_l[1]
    sim_bel_h[5] = time_example.Θ[sHist_h] * sim_bel_h[1]


    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, bel_l, bel_h in zip(axes.flatten(), titles,
                                                     sim_seq_l, sim_seq_h,
                                                     sim_bel_l, sim_bel_h):
        ax.plot(sim_l, '-ok', sim_h, '-^k', bel_l, '-or', bel_h, '-^r',
↪alpha=0.7)
        ax.set(title=title)
        ax.grid()

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

```
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:24:
RuntimeWarning: divide by zero encountered in reciprocal
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:29:
RuntimeWarning: divide by zero encountered in power
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:235:
RuntimeWarning: invalid value encountered in true_divide
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:228:
RuntimeWarning: invalid value encountered in matmul
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:233:
RuntimeWarning: invalid value encountered in matmul
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:235:
RuntimeWarning: invalid value encountered in multiply
```

0.6029333236643732
0.11899588245816164
0.09881553222482795
0.083541067123041
0.07149555165781138
0.061730368562270536
0.05366019923600977
0.04689112010309679
0.041151783361805694
0.036240012996977156
0.03200623802767769
0.028368464298785867
0.025192688510675774
0.02240584283749033
0.019947746596861597
0.017777612457184643
0.015863113233478984
0.014157555815666256
0.012655688108653255
0.011323561335112373
0.01013434235262587
0.009067133148757954
0.00813336288612958
0.007289176649859819
0.0065414142692250665
0.005872916325737843
0.005262679947231285
0.004730774790763247
0.00425304535076248
0.0038185016441054827
0.0034264404729681375
0.0030793645662841626
0.00276826663824067
0.0024904277711437395
0.0022405919962000076
0.0020186945277340827
0.0018171340595572291
0.001636401890175593
0.0014731338929050088
0.0013228186521306716
0.0011905280522741955
0.0010699233146931647
0.0009619063701918545
0.00086610648912093
0.0007801797603634053
0.000704404030125401
0.0006357998611731389
0.0005726395027541625
0.0005148786009604965
0.0004655396622616272
0.00041934364600035334
0.0003774055070147406 7
0.000339364461753 4906
0.00030505083973 74688
0.00027489392163708453
0.00024661007756555605
0.00022217617613994842
0.0002001738277695678
0.00018111718881366342
0.0001635894087493217
0.00014736944417742374
0.00013236625373264476
0.0001185375904583787

```
0.00010958655893643692
9.59415239737652e-05
```




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 $g_3 = g_l$
- 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

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 $t = 3$.

It responds to this constraint by smoothing tax rates across time.

To finance a war it raises taxes and issues more debt.

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.

### 5.1.1 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.

In [8]:
```
import numpy as np

class LogUtility:

    def __init__(self,
                 β=0.9,
                 ψ=0.69,
                 π=0.5*np.ones((2, 2)),
                 G=np.array([0.1, 0.2]),
                 Θ=np.ones(2),
                 transfers=False):

        self.β, self.ψ, self.π = β, ψ, π
        self.G, self.Θ, self.transfers = G, Θ, transfers

    # Utility function
    def U(self, c, n):
        return np.log(c) + self.ψ * np.log(1 - n)

    # Derivatives of utility function
    def Uc(self, c, n):
        return 1 / c

    def Ucc(self, c, n):
        return -c**(-2)

    def Un(self, c, n):
        return -self.ψ / (1 - n)

    def Unn(self, c, n):
        return -self.ψ / (1 - n)**2
```

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).

In [9]:
```
log_example = LogUtility()
log_example.transfers = True                    # Government can use transfers
log_sequential = SequentialAllocation(log_example)  # Solve sequential
↪problem
log_bellman = RecursiveAllocationAMSS(log_example, μ_grid)

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])

# Simulate
sim_seq = log_sequential.simulate(0.5, 0, T, sHist)
sim_bel = log_bellman.simulate(0.5, 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]
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/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:18:
RuntimeWarning: invalid value encountered in log
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:18:
RuntimeWarning: divide by zero encountered in log
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:22:
RuntimeWarning: divide by zero encountered in true_divide
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:235:
RuntimeWarning: invalid value encountered in true_divide
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:235:
RuntimeWarning: invalid value encountered in multiply
```

```
        ---------------------------------------------------------------
--------------

        ValueError                                Traceback (most
↪recent call last)

        <ipython-input-9-764875dee216> in <module>
```

```
        2 log_example.transfers = True                    #
↪Government can use transfers
        3 log_sequential = SequentialAllocation(log_example)  #
↪Solve sequential problem
  ----> 4 log_bellman = RecursiveAllocationAMSS(log_example, μ_grid)
        5
        6 T = 20


      <ipython-input-4-cc6b33fcda51> in __init__(self, model,
↪μgrid, tol_diff, tol)
       15
       16         # Find the first best allocation
  ---> 17         self.solve_time1_bellman()
       18         self.T.time_0 = True  # Bellman equation now
↪solves time 0 problem
       19


      <ipython-input-4-cc6b33fcda51> in solve_time1_bellman(self)
       62             PF = T(Vf)
       63
  ---> 64             Vfnew, policies = self.fit_policy_function(PF)
       65             diff = np.abs((Vf(xgrid) - Vfnew(xgrid)) /
↪Vf(xgrid)).max()
       66


      <ipython-input-4-cc6b33fcda51> in fit_policy_function(self,
↪PF)
       81         cf, nf, xprimef, Tf, Vf = [], [], [], [], []
       82         for s_ in range(S):
  ---> 83             PFvec = np.vstack([PF(x, s_) for x in self.
↪xgrid]).T
       84             Vf.append(interp(xgrid, PFvec[0, :]))
       85             cf.append(interp(xgrid, PFvec[1:1 + S]))


      <ipython-input-4-cc6b33fcda51> in <listcomp>(.0)
       81         cf, nf, xprimef, Tf, Vf = [], [], [], [], []
       82         for s_ in range(S):
  ---> 83             PFvec = np.vstack([PF(x, s_) for x in self.
↪xgrid]).T
       84             Vf.append(interp(xgrid, PFvec[0, :]))
       85             cf.append(interp(xgrid, PFvec[1:1 + S]))


      <ipython-input-4-cc6b33fcda51> in PF(x, s)
      207         '''
      208         if not self.time_0:
```

```
  --> 209             def PF(x, s): return self.
↪get_policies_time1(x, s, Vf)
      210         else:
      211             def PF(B_, s0): return self.
↪get_policies_time0(B_, s0, Vf)


      <ipython-input-4-cc6b33fcda51> in get_policies_time1(self,
↪x, s_, Vf)
      245                                                 
↪f_eqcons=cons, bounds=bounds,
      246                                                 
↪full_output=True, iprint=0,
  --> 247                                                  acc=self.
↪tol, iter=self.maxiter)
      248
      249         if imode > 0:


      ~/anaconda3/lib/python3.7/site-packages/scipy/optimize/slsqp.
↪py in fmin_slsqp(func, x0, eqcons, f_eqcons, ieqcons, f_ieqcons,
↪bounds, fprime, fprime_eqcons, fprime_ieqcons, args, iter, acc,
↪iprint, disp, full_output, epsilon, callback)
      204
      205     res = _minimize_slsqp(func, x0, args, jac=fprime,
↪bounds=bounds,
  --> 206                           constraints=cons, **opts)
      207     if full_output:
      208         return res['x'], res['fun'], res['nit'],
↪res['status'], res['message']


      ~/anaconda3/lib/python3.7/site-packages/scipy/optimize/slsqp.
↪py in _minimize_slsqp(func, x0, args, jac, bounds, constraints,
↪maxiter, ftol, iprint, disp, eps, callback, finite_diff_rel_step,
↪**unknown_options)
      424
      425         if mode == -1:  # gradient evaluation required
  --> 426             g = append(sf.grad(x), 0.0)
      427            a = _eval_con_normals(x, cons, la, n, m, meq,
↪mieq)
      428


      ~/anaconda3/lib/python3.7/site-packages/scipy/optimize/
↪_differentiable_functions.py in grad(self, x)
      186         if not np.array_equal(x, self.x):
      187             self._update_x_impl(x)
  --> 188         self._update_grad()
      189         return self.g
```

```
      190


      ~/anaconda3/lib/python3.7/site-packages/scipy/optimize/
↪_differentiable_functions.py in _update_grad(self)
      169     def _update_grad(self):
      170         if not self.g_updated:
  --> 171             self._update_grad_impl()
      172             self.g_updated = True
      173


      ~/anaconda3/lib/python3.7/site-packages/scipy/optimize/
↪_differentiable_functions.py in update_grad()
       90                 self.ngev += 1
       91                 self.g = approx_derivative(fun_wrapped,
↪self.x, f0=self.f,
  ---> 92                                            
↪**finite_diff_options)
       93
       94         self._update_grad_impl = update_grad


      ~/anaconda3/lib/python3.7/site-packages/scipy/optimize/
↪_numdiff.py in approx_derivative(fun, x0, method, rel_step,
↪abs_step, f0, bounds, sparsity, as_linear_operator, args, kwargs)
      389
      390     if np.any((x0 < lb) | (x0 > ub)):
  --> 391         raise ValueError("`x0` violates bound
↪constraints.")
      392
      393     if as_linear_operator:


      ValueError: `x0` violates bound constraints.
```

When the government experiences a prolonged period of peace, it is able to reduce government debt and set permanently 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.

```
In [10]: T = 200  # 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 = ['Consumption', 'Labor Supply', 'Government Debt',
          'Tax Rate', 'Government Spending', 'Output']

# Government spending paths
sim_seq_long[4] = log_example.G[sHist_long]
sim_bel_long[4] = log_example.G[sHist_long]

# Output paths
sim_seq_long[5] = log_example.Θ[sHist_long] * sim_seq_long[1]
sim_bel_long[5] = log_example.Θ[sHist_long] * sim_bel_long[1]

fig, axes = plt.subplots(3, 2, figsize=(14, 10))

for ax, title, seq, bel in zip(axes.flatten(), titles, sim_seq_long, \
        sim_bel_long):
    ax.plot(seq, '-k', bel, '-.b', alpha=0.5)
    ax.set(title=title)
    ax.grid()

axes[0, 0].legend(('Complete Markets','Incomplete Markets'))
plt.tight_layout()
plt.show()
```

```
        ---------------------------------------------------------------
--------------

        NameError                                 Traceback (most
 ↪recent call last)

        <ipython-input-10-2a879ce8f36c> in <module>
          2 sim_seq_long = log_sequential.simulate(0.5, 0, T)
          3 sHist_long = sim_seq_long[-3]
    ----> 4 sim_bel_long = log_bellman.simulate(0.5, 0, T, sHist_long)
          5
          6 titles = ['Consumption', 'Labor Supply', 'Government
 ↪Debt',

        NameError: name 'log_bellman' is not defined
```

**Footnotes**

[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.

[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 (11) takes a positive value, so $\gamma_t(\tilde{s}^t) > 0$.

## References


[1] 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.

[2] Robert J Barro. On the Determination of the Public Debt. *Journal of Political Economy*, 87(5):940–971, 1979.

[3] 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.