Computing Moral Hazard Programs With Lotteries Using Matlab

Alexander Karaivanov
Department of Economics
University of Chicago

a-karaivanov@uchicago.edu

This draft: June 25, 2001

Abstract

This paper provides a step-by-step hands-on introduction to the techniques used in setting up and solving moral hazard programs with lotteries using Matlab. It uses a linear programming approach due to its relative simplicity and the high reliability of the available optimization algorithms.

1 Introduction

This paper provides a hands-on, step-by-step introduction to setting up and numerically solving moral hazard programs with lotteries using Matlab. It is intended as an introduction to computational techniques which can be used in various mechanism design setups. The theoretical justification of the approach is based upon Prescott and Townsend's (1984a,b) seminal contributions introducing methods through which incentive-constrained economies can be analyzed in the space of probability measures of economic variables. By construction, such methods involve optimization problems characterized with high dimensionality, which makes analytical solutions infeasible. This disadvantage is, however, offset by the fact that the intrinsically non-linear moral hazard problems are relatively easily transformed into linear programs with respect to the introduced probability measures. Furthermore, the method is completely general as opposed to the alternative 'first order approach' (see Rogerson, 1985), since the latter can be used only under some restrictive assumptions. Townsend (1987, 1988) was among the first to actually use linear programming techniques for obtaining numerical solutions to static incentive constrained problems. These methods were then extended to dynamic economies in Phelan and Townsend (1991) demonstrating their broad applicability. More recent contributions to the literature include Lehnert (1998), Prescott and Townsend (2000a,b), and Doepke and Townsend (2001) who apply linear programming to various mechanism design problems.

The present paper is based on the assumption that the reader has a good knowledge of the theory on incentive constrained problems[1] and basic knowledge of programming in Matlab, although I will try to limit the latter requirement by providing detailed explanations of the included code and examples of its usage.

I concentrate on the solution of moral hazard programs with lotteries using linear programming techniques due to the relative simplicity of the method and the high reliability of the available optimization algorithms. As mentioned above, this does not mean that linear programming is the only method available for solving moral hazard problems. In many cases one can successfully implement the first order approach using non-linear constrained optimization routines which are much faster because of their low dimensionality, but often the restrictions on the functional forms that can be employed are very stringent and preclude the method's applicability.

The linear programming formulation of the classic moral hazard problem described below has the advantage of being easy to solve with the available Matlab optimization toolbox functions, but it also has the disadvantage of being heavily time and memory-intensive. Nevertheless, in most practical applications, and especially when the first order approach is not valid, the linear programming method, which reformulates the moral hazard problem as one with lotteries is the best feasible way to proceed. In addition, the continuing progress in increasing computational speed and some advances in the algorithms used (e.g. the Danzig-Wolfe decomposition algorithm described in Prescott, 1998) facilitate enormously the application of linear programming.

[1]For a brief but excellent overview see Prescott (1999).

2 Formulation of the Moral Hazard Problem With Lotteries

In this section I describe the linear programming formulation of a classic moral hazard problem in its simplest form. In the end of the paper I will consider an alternative formulation in the form of a social planner's problem.

Following Prescott (1999), let's state the moral hazard program with lotteries:

$$\max_{\pi} \sum_{c,q,z} \pi(c,q,z) w(q-c)$$

$$\text{s.t.} \qquad \sum_{c,q,z} \pi(c,q,z) u(c,z) \geqq U \tag{1}$$

$$\sum_{c.q} \pi(c,q,z) u(c,z) \geqq \sum_{c,q} \pi(c,q,z) \frac{p(q|\hat{z})}{p(q|z)} u(c,\hat{z}),\ \forall (z,\ \hat{z} \neq z) \in Z \times Z \tag{2}$$

$$\sum_{c} \pi(c,\bar{q},\bar{z}) = p(\bar{q}|\bar{z}) \sum_{c,q} \pi(c,q,\bar{z}),\ \forall\ \bar{q},\bar{z} \tag{3}$$

$$\sum_{c,q,z} \pi(c,q,z) = 1 \text{ and } \forall\ c,q,z, \quad \pi(c,q,z) \geqq 0 \tag{4}$$

The notation used is the standard one for the literature: c denotes agent's consumption, q is the level of output and z is the level of action (e.g. effort) taken by the agent. The function $p(q|z)$ denotes the probability of achieving an output level of q, given an effort level z supplied by the agent. In this way, p can be interpreted as a production function. The utility function of the principal is given by $w(.)$, whereas the one of the agent is $u(c,z)$, where u is strictly quasi-concave, increasing in c and decreasing in z. The main ingredients in the above program are the choice variables $\pi(c,q,z)$, representing the probabilities that the agent is assigned a particular combination of consumption, output and effort levels (c,q,z). Thus, the possible values of the $\pi's$, satisfying the constraints (1)-(4) actually characterize the space of possible contracts between the agent and the principal and the solution to the above linear program, $\pi^*(c,q,z)$ represents the optimal contract. Note that both the objective function and the constraints are linear in the probabilities π, despite the intrinsic non-linear structure of the moral hazard problem.

The program exhibited above states that the principal maximizes her expected utility subject to the following constraints:

1. *Participation Constraint* (1) : the agent must be willing to participate in the contract, i.e. must obtain expected utility higher than her reservation level of U.

2. *Incentive Compatibility Constraints* (2): the agent must optimally undertake the recommended action z, as opposed to any other possible action level $\hat{z}$ belonging to the set of feasible actions Z.

3. *Mother Nature Constraints* (3): the relationship between conditional and unconditional probabilities, $\pi(c,q,z)$ must satisfy the probability laws (Bayes rule).

4. *Adding-up and non-negativity* of the probabilities (4): the $\pi's$ are interpreted as probabilities, hence they are required to be non-negative and must add up to one.

In addition, in order to have a well-defined linear program, there must be a finite number of variables and finite number of constraints, i.e. there can be only a finite number of $\pi's$ in the above program. An easy way to ensure this is to assume that the three variables, c, q and z take values on discrete grids, $C = (c_1, c_2 ... c_l)$, $Q = (q_1, q_2, ...q_m)$ and $Z = (z_1, z_2, ...z_n)$. Clearly then the total number of variables to solve for is lmn, and there are 1 participation constraint, $n(n-1)$ incentive compatibility constraints (ICCs), nm mother nature (technology) constraints and 1 adding up constraint.

Example

To clarify the above, take for example the case of two effort levels, two consumption levels and two output levels ($l = m = n = 2$). Let π_{ijk} be a shorthand for $\pi(c_i, q_j, z_k)$. The objective function then takes the following form:

$$\begin{aligned}&\max_{\pi} \ (\pi_{111} + \pi_{112})w(q_1 - c_1) + (\pi_{121} + \pi_{122})w(q_2 - c_1) + \\ &+(\pi_{211} + \pi_{212})w(q_1 - c_2) + (\pi_{221} + \pi_{222})w(q_2 - c_2)\end{aligned}$$

The participation constraint involves summation over all possible recommendations of (c, z, q) and thus it is only one and can be written for our example as:

$$\begin{aligned}&(\pi_{111} + \pi_{121})u(c_1, z_1) + (\pi_{112} + \pi_{122})u(c_1, z_2) + \\ &+(\pi_{211} + \pi_{221})u(c_2, z_1) + (\pi_{212} + \pi_{222})u(c_2, z_2) \ \geq \ U \end{aligned} \tag{PC}$$

The incentive compatibility constraints are $n(n-1) = 2(1) = 2$, since for each effort level z, there is only one alternative level $\hat{z}$:

$$\begin{aligned}&(\pi_{111} + \pi_{121})u(c_1, z_1) + (\pi_{211} + \pi_{221})u(c_2, z_1) \geq \\ &\geq \frac{p(q_1|z_2)}{p(q_1|z_1)}[\pi_{111}u(c_1, z_2) + \pi_{211}u(c_2, z_2)] \quad + \frac{p(q_2|z_2)}{p(q_2|z_1)}[\pi_{121}u(c_1, z_2) + \pi_{221}u(c_2, z_2)]\end{aligned} \tag{ICC1}$$

and

$$\begin{aligned}&(\pi_{112} + \pi_{122})u(c_1, z_2) + (\pi_{212} + \pi_{222})u(c_2, z_2) \geq \\ &\geq \frac{p(q_1|z_1)}{p(q_1|z_2)}[\pi_{112}u(c_1, z_1) + \pi_{212}u(c_2, z_1)] \quad + \frac{p(q_2|z_1)}{p(q_2|z_2)}[\pi_{122}u(c_1, z_1) + \pi_{222}u(c_2, z_1)]\end{aligned} \tag{ICC2}$$

There is a mother nature constraint for each output-effort combination (z, q), i.e. their total number is $mn = 2(2) = 4$:

$$\begin{aligned}\pi_{111} + \pi_{211} &\geq p(q_1|z_1)[\pi_{111} + \pi_{121} + \pi_{211} + \pi_{221}] \\ \pi_{112} + \pi_{212} &\geq p(q_1|z_2)[\pi_{112} + \pi_{122} + \pi_{212} + \pi_{222}] \\ \pi_{121} + \pi_{221} &\geq p(q_2|z_1)[\pi_{111} + \pi_{121} + \pi_{211} + \pi_{221}] \\ \pi_{122} + \pi_{222} &\geq p(q_2|z_2)[\pi_{112} + \pi_{122} + \pi_{212} + \pi_{222}]\end{aligned} \tag{MNC}$$

Notice that on the left hand side the summation is only with respect to the possible values for c, holding those for z and q as given, whereas in the right hand side only the value for z is held fixed. The adding up constraint is trivial and thus omitted from the example. I will be coming back repeatedly to this simple example in the discussion below to facilitate the understanding of the programming algorithm. For clarity of the presentation the sections of the text in which this is done will be marked with [**Example**] in the beginning.

It is easy to see that the dimension of the problem increases very fast in the grid sizes. Certainly the discreteness of the grids represents an abstraction from reality, and invariably introduces some error if continuous variables are modeled this way, but for big enough number of points the results become less sensitive to further increases in the grid dimensions. The common trade-off in numerical methods between accuracy of the results and time consumption is present here as well.

3 Coding the Moral Hazard Problem in Matlab

Let's take another look at the moral hazard problem exhibited above. It seems pretty comprehensible to the human (economist's) eye but actually it requires quite a bit of work to be put in a form such that it can be solved by a computer. This section describes in detail the steps through which the problem is coded in Matlab. In fact, the main thing that needs to be done is to represent the above problem in a format, which can be given as an input to the Matlab linear programming solver *linprog* from the Optimization Toolbox of versions 5.3 and above[2]. Depending on the particular structure of the problem at hand, the function *linprog* implements a large scale or medium scale optimization methods. The large scale algorithm is a primal-dual interior-point active set method, a variant of Mehrotra's predictor-corrector algorithm (see Mehrotra, 1992 and Zhang, 1995). The medium scale algorithm uses a projection active set method.

Once we have transformed the above problem into a format suitable for *linprog*, we obtain as output the vector of probabilities $\pi^*(c, q, z)$ which solves the above optimization problem. Typically this vector consists of many zeros and only a few positive entries. In the end of the section I discuss how to extract the relevant non-zero probabilities, which then can be used to compute the consumption, output and action levels of the optimal contract.

Let's proceed by describing the steps of the coding process in detail. All Matlab code is given in italics. The reader can cut and paste the supplied code into a text file, save it with an extension .m (the extension for Matlab program files) and then run it in Matlab to see how it works.

3.1 Step 1: Parametrization and Initialization

In general computers work with numbers and not Latin, or even Greek letters, so in order to turn the above program into Matlab code, we need to choose values for the three grids and give specific functional forms to the utility functions, u and w, and the production function, $p(q|z)$. These functional forms will almost certainly contain some parameters, so values for them must be assigned as well.

Let's assume for example that the principal is risk neutral, i.e. $w(q - c) = q - c$ and the agent is risk averse: $u(c, z) = \dfrac{c^{1-\gamma}}{1-\gamma} + \kappa\dfrac{(1-z)^{1-\delta}}{1-\delta}$ with $\gamma, \delta > 0$. In addition, assume that there

[2]The previous versions of Matlab used the function *lp* for solving linear programming problems, which was much slower and much less reliable in terms of the accuracy of the results.

are only two possible output levels q_l and q_h, such that $q_h > q_l > 0$.

For simplicity, let's use linear (equally-spaced) grids for all three variables having the form $[x_{\min}, ..x_{\max}]$, for $x = c, q$ or z[3]. In this way we would need to choose 3 numbers for each grid - its first and last element and the total number of elements. These give us nine additional parameters to assign values to. Let us also assume that $p(q = q_h|z) = z^\alpha$, $0 < \alpha < 1$. In order for this function to generate valid probabilities, we need to set $Z \subseteq [0, 1]$.

Finally, we are ready to begin programming in Matlab. We have to start with defining all the variables mentioned above and assigning values to them. Below I exhibit the actual Matlab code which performs the described operation, together with comments. Commenting in Matlab is done by putting the percentage sign in front of the text one wishes to have as comment (e.g. *%This is a comment!*). If the reader finds the provided explanations insufficient (s)he is encouraged to refer to the built-in Matlab help feature by simply typing at the command prompt: *help fname*, where *fname* stands for the name of the function or operator in Matlab on which help is needed.

Since Matlab uses only Latin letters, notice that in the code below I use Latin letters corresponding to the Greek ones denoting the various parameters in the model. I have commented almost every line of the program explaining what it does.

```
%SAMPLE MATLAB CODE FOR MORAL HAZARD PROBLEMS (part I)
clear all           %clears the memory from all variables
%1. Assign values to the parameters
g=.5;          %gamma
k=1;          %kappa
d=.5;          %delta
a=.7;          %alpha
U=1;          %reservation utility

%2. Define the grids
%Consumption Grid
nc=20;          %number of points in the consumption grid
cmin=10^-8;       %lowest possible consumption level (can't be 0 for the chosen function)
cmax=3;          %highest possible consumption level
c=linspace(cmin,cmax,nc);      %creates the actual grid for consumption as 20
          %equally spaced values between cmin and cmax
%Action level
nz=10;               %number of grid points
zmin=10^-8;            %minimum effort level
zmax=1-10^-8;      %maximum effort level
z=linspace(zmin,zmax,nz);           %creating the grid
%Output
nq=2;          %number of grid points
ql=1;          %low output level
qh=3;          %high output level
```

[3]Often, especially in dynamic settings, it is better to use log-spaced grids in order to ensure that more grid points are available at regions with high curvature of the objective function and thus improve the accuracy of the results.

```
q=[ql,qh];      %two possible output values by assumption, in order to
                %simplify the computation of probabilities
```

Notice the use of the function *linspace(arg1,arg2,arg3)*, which creates a vector of *arg3* equally-spaced numbers on the interval *[arg1,arg2]*.

[Example]
In the context of our example we have: $c = (c_1, c_2)$, $z = (z_1, z_2)$ and $q = (q_l, q_h)$, where z_1 is $z\min$, z_2 is zmax and respectively for c. Notice that there are only 2^3 =8 total possible combinations of values, which (c, q, z) might take in this case, i.e. the optimal contract is a convex combination of these 8 cases with weights given by the corresponding probabilities $\pi(c, q, z)$.

With the above code, we have defined and initialized all the needed variables corresponding to the elements of the model. The utility and probability functions are defined in the next step.

3.2 Step 2: Constructing the Matrices of Constraints and the Objective Function

As I already mentioned before, the linear program above needs to be transformed into a structure, which can be given as input to the Matlab linear program solver routine. The actual function we will be using is *linprog*. The following is an excerpt of what would come on your screen after typing *help linprog*[4]:

```
LINPROG Linear programming.
X=LINPROG(f,A,b) solves the linear programming problem:
min f'*x subject to: A*x <= b
 x
X=LINPROG(f,A,b,Aeq,beq) solves the problem above while additionally
satisfying the equality constraints Aeq*x = beq.
X=LINPROG(f,A,b,Aeq,beq,LB,UB) defines a set of lower and upper
bounds on the design variables, X, so that the solution is in
the range LB<=X<=UB. Use empty matrices for LB and UB
if no bounds exist. Set LB(i) = -Inf if X(i) is unbounded below;
set UB(i) = Inf if X(i) is unbounded above.
```

Let's try to map the above notation into our specific framework. First, it is clear that the unknown variable, X, is the vector $\pi(c, q, z)$ in our problem, i.e. we will be looking for a vector of $\pi's$, which maximizes the objective function of the principal subject to the constraints exhibited above. Next, we'll need to define all the matrices (vectors) with which *linprog* works, i.e. the vector of objective function coefficients f, the matrix of coefficients on the $\pi's$ in the inequality constraints, A, the vector b of intercepts in the inequality constraints; the corresponding matrix *Aeq* and vector *beq* for the equality constraints, and the lower and upper bounds on the unknowns LB and UB. Notice also that the function performs minimization, i.e. we will need to invert the sign of our objective function later in order to be able to use *linprog*.

Looking back at the moral hazard program with lotteries, we see that there are two types of equality constraints: the adding up and the mother nature constraints. We will need to combine

[4]Notice that this is not part of the code for solving the moral hazard program.

the coefficients on the $\pi's$ present in all of these constraints into a single matrix Aeq. There are also two types of inequality constraints: the participation and the incentive compatibility constraints. They also have to be transformed into a single matrix which I call A.

We are now ready to begin the construction of the input matrices required by the function *linprog*. First, we need to say how we are going to keep track of the $nc * nz * nq$ probabilities $\pi(c, q, z)$ in order to know how to assign the corresponding coefficients from the constraints to them. Without loss of generality, let's order the $\pi's$ numerically, from low to high grid values, sorting first on z, then on q and, finally on c, i.e. if $X = (x_1, x_2, ...x_{lmn})$ is the vector of probabilities we will have: $x_1 = \pi(c_1, q_1, z_1)$, $x_2 = \pi(c_2, q_1, z_1)$,...$x_{l+1} = \pi(c_1, q_2, z_1)$, ...$x_{lm+1} = \pi(c_1, q_1, z_2)$, etc. Second, it turns out to be very helpful to construct an auxiliary vector, P consisting of the conditional probabilities $p(q|z)$ corresponding to all the triples (c, q, z) in the order described above. This is what is done next:

```
%SAMPLE MATLAB CODE FOR MORAL HAZARD PROBLEMS (part 2)
P(1:2:nz*nq-1)=1-z.^a;      %the conditional probabilities corresponding to q_l
P(2:2:nz*nq)=z.^a;          %the conditional probabilities corresponding to q_h
```

The above code creates a vector P, the odd elements of which (i.e. $P_1, P_3...P_{nz*nq-1}$) are set equal to $1 - z^{\alpha}$ for the corresponding effort level z, while the even elements $(P_2, P_4...P_{nz*nq})$ are set to z^{α}. Notice the use of the dot operator .^which performs exponentiation of the effort grid vector z element by element.

[**Example**]

In order to make the understanding of the supplied Matlab code as easy as possible, I continue to use the example with two values for each of the consumption, effort and output levels to illustrate how the matrices and vectors created by the code look like. Notice first that, in terms of the notation used above, the ordering of the 8 $\pi's$ is as follows:

$$\pi_{111}, \pi_{211}, \pi_{121}, \pi_{221}, \pi_{112}, \pi_{212}, \pi_{122}, \pi_{222} \tag{5}$$

In the context of the example the above code creates the following vector

$$P \equiv (p(q_l|z_1), p(q_h|z_1), p(q_l|z_2), p(q_h|z_2) = (1 - z_1^{\alpha}, z_1^{\alpha}, 1 - z_2^{\alpha}, z_2^{\alpha})$$

The first line of the code assigns its first and third element, and the second line - the second and fourth.

We can now start creating the input matrices required by *linprog*. Let's begin with the easy ones:

1. Lower and upper bounds

Since the $\pi's$ are probabilities we need to set 0 and 1 as the bounds between which *linprog* will search for a solution. In Matlab this is:

```
%SAMPLE MATLAB CODE FOR MORAL HAZARD PROBLEMS (part 3)
UB=ones(nz*nq*nc,1);       %the vector of upper bounds
LB=zeros(nz*nq*nc,1);      %lower bounds
```

The Matlab function *ones(arg1,arg2)* creates an arg1-by-arg2-dimensional matrix with all elements set to 1. Similarly the function *zeros(arg1,arg2)* creates an arg1-by-arg2-dimensional matrix of zeros. Thus the above code creates two vectors: the nz*nq*nc-by-1 upper bounds vector UB with all elements equal to 1 and the nz*nq*nc-by-1 lower bounds vector LB consisting of zeros. Note that *linprog* requires that the vector of unknowns X be a column vector and that is why we need column vectors for the bounds as well.

[**Example**]
In our example the vectors created above are given by: $UB = (1, 1, 1, 1, 1, 1, 1, 1)^T$ and $LB = (0, 0, 0, 0, 0, 0, 0, 0)^T$. Notice that this takes care of the non-negativity constraints imposed on the probabilities $\pi(c, q, z)$ in our problem.

2. Objective Function
Next, we need to construct the vector of coefficients of the objective function, f. There are many possible ways to do this, but the most efficient one in terms of computational speed is to use the Matlab function *kron*, which computes the Kronecker product of two matrices. Let's see how it works by typing *help kron*:[5]

KRON Kronecker tensor product.
KRON(X,Y) is the Kronecker tensor product of X and Y.
The result is a large matrix formed by taking all possible
products between the elements of X and those of Y. For
example, if X is 2 by 3, then KRON(X,Y) is
*[X(1,1)*Y X(1,2)*Y X(1,3)*Y*
*X(2,1)*Y X(2,2)*Y X(2,3)*Y]*
If either X or Y is sparse, only nonzero elements are multiplied
in the computation, and the result is sparse.

Perhaps it is not obvious why this function should help in the construction of our matrices but hopefully this will become more clear below. Since this is the main function I am going to use below, let me try to explain how it works by an example. Let's take an $n_1 \times m_1$ matrix X and an $n_2 \times m_2$ matrix Y, for example:

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{bmatrix} \text{ and } Y = \begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix}$$

Using the definition above their Kronecker product is the 4-by-6 matrix Z given by:

$$\begin{aligned} Z &= \begin{bmatrix} x_{11}\begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix} & x_{12}\begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix} & x_{13}\begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix} \\ x_{21}\begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix} & x_{22}\begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix} & x_{23}\begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix} \end{bmatrix} = \\ &= \begin{bmatrix} x_{11}y_{11} & x_{11}y_{12} & x_{12}y_{11} & x_{12}y_{12} & x_{13}y_{11} & x_{13}y_{12} \\ x_{11}y_{21} & x_{11}y_{22} & x_{12}y_{21} & x_{12}y_{22} & x_{12}y_{21} & x_{12}y_{22} \\ x_{21}y_{11} & x_{21}y_{21} & x_{22}y_{11} & x_{22}y_{12} & x_{23}y_{11} & x_{23}y_{12} \\ x_{21}y_{21} & x_{21}y_{22} & x_{22}y_{21} & x_{22}y_{22} & x_{23}y_{21} & x_{23}y_{22} \end{bmatrix} \end{aligned}$$

[5]See the previous footnote.

Thus what *kron* does is take each element of X and multiply it by Y and stack the resulting $n_1 m_1$ matrices into an $n_1 n_2 \times m_1 m_2$ dimensional matrix. Notice also that it is not a symmetric operator, i.e. $kron(X, Y) \neq kron(Y, X)$. I strongly encourage the reader to do some more experimenting with *kron*, trying different examples to get a feeling of how the function works as this is essential to what follows.

Let's return to the construction of the vector of objective function coefficients, f :

```
%SAMPLE MATLAB CODE FOR MORAL HAZARD PROBLEMS (part 4)
f = -kron(ones(1, nz), kron(q, ones(1, nc))) + kron(ones(1, nq*nz), c);
```

We can see how powerful *kron* is indeed: in only one line we constructed the $1 \times nz * nc * nq$ vector of the coefficients on the $\pi's$ in the objective function. The first part of the above expression gives the values of q corresponding to all possible (c, q, z) triples in the assumed order. The second does the same for the values of c. The use of *kron* together with *ones* provides a short, fast and easy way to construct matrices consisting of repeated grid values or any combinations thereof, such as the ones appearing in the coefficients of the objective and the constraints. Of course, in principle the same thing could be done by using loops but this would take considerably more time to compute, which is scarce especially for high-dimensional linear programs. Finally, f has the opposite sign compared to the original coefficients in the program because *linprog* performs minimization.

[**Example**]

Let's illustrate what is done above in our simple example setup. Start with the second expression in the sum, *kron(ones(1, nq*nz), c).* Following the same logic as above, this command creates the vector $F_1 = (c_1, c_2, c_1, c_2, c_1, c_2, c_1, c_2, c_1, c_2)$ by concatenating four times the c vector. Thus, F_1 is the vector of consumption values for the respective (c, q, z) tuples ordered as described above. Similarly, the command *kron(q, ones(1, nc))* creates the vector $F_2 = (q_1, q_1, q_2, q_2)$ and *kron(ones(1, nz), kron(q, ones(1, nc)))* - $F_3 = (F_2, F_2) = (q_1, q_1, q_2, q_2, q_1, q_1, q_2, q_2)$, i.e. the vector of output levels corresponding to the assumed ordering of the (c, q, z) tuples. Given this, $f = -F_3 + F_1$ gives the vector of values for the principal's utility levels $c - q$ for the 8 possible (c, q, z) combinations that the contract may prescribe.

3. Adding-up Constraint

The probabilities need to sum up to 1. This is one of the equality constraints in our problem and it is coded as follows:

```
%SAMPLE MATLAB CODE FOR MORAL HAZARD PROBLEMS (part 5)
Aeq1=ones(1, nz*nq*nc);       %the coefficients are ones on each π
beq1=1;                       %the sum of probabilities needs to be 1.
```

In the above code $Aeq1$ is the vector of coefficient on the $\pi's$ in the expression $\sum\limits_{c,q,z} \pi(c, q, z)$, which are clearly all ones, while $beq1$ is the intercept, which also equals to 1 in this case. This is perhaps the easiest constraint and the only thing to be careful about here is keeping track of the dimensions of the vectors involved.

[Example]

In the setup of our example, the code above creates the vector $Aeq1 = (1,1,1,1,1,1,1,1)$ and the scalar $beq1 = 1$, which are part of the matrices Aeq and beq governing the coefficients on the equality constraints in the linear program.

4. **Mother Nature Constraints**

Here we put to use the matrix P, which we constructed above:

```
%SAMPLE MATLAB CODE FOR MORAL HAZARD PROBLEMS (part 6)
Aeq2 = kron(eye(nz*nq), ones(1,nc)) - kron(kron(eye(nz), ones(nq,1)).*...
(P'*ones(1,nz)), ones(1,nq*nc));
beq2 = zeros(nz*nq, 1);
```

The first *kron* operator corresponds to the left hand side of (3), summing over the relevant values for consumption for given q and z. The second one is the right hand side. The Matlab function *eye*(*arg1*) creates an *arg1*×*arg1* identity matrix. Notice the ingenuous use of this function, together with *kron*, which helps create the matrix of coefficients of all nm mother nature (equality) constraints, $Aeq2$ in just one line of code. The vector of intercepts, $beq2$ contains zeros since we have put all the terms in (3) on the left hand side in order to satisfy the prescribed input requirements of *linprog*. Let's use our example framework to see how the above works. Compare also the results with the analytical expressions derived previously.

[Example]

Let's start with the first term, *kron(eye(nz*nq), ones(1,nc))*. Using the definition of the $kron$ operator and our previous example, we see that it creates the 4-by-8 matrix

$$M_1 = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

which is exactly the matrix of coefficients on the $\pi' s$ in the left hand side of (MNC) above. The second term creates the right hand side of (MNC). First it uses the command *kron(eye(nz), ones(nq,1))* to create the matrix

$$M_2 = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}$$

which is multiplied element by element (note the .* operator) by the matrix M_3 created by *P'*ones(1,nz),* which is given by (remember what P was equal to) to obtain:

$$M_4 \equiv M_2.*M_3 = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} .* \begin{bmatrix} 1-z_1^\alpha & 1-z_1^\alpha \\ z_1^\alpha & z_1^\alpha \\ 1-z_2^\alpha & 1-z_2^\alpha \\ z_2^\alpha & z_2^\alpha \end{bmatrix} = \begin{bmatrix} 1-z_1^\alpha & 0 \\ z_1^\alpha & 0 \\ 0 & 1-z_2^\alpha \\ 0 & z_2^\alpha \end{bmatrix}$$

where .* denotes element by element matrix multiplication. Second, M_4 is used to create the matrix M_5 of the RHS coefficients in (MNC) through *kron(kron(eye(nz), ones(nq,1)).*(P'*ones(1,nz)), ones(1,nq*nc)),* which is equivalent in our case to *kron*(M_4, *ones*(1, 4)) and results in

$$M_5 = \begin{bmatrix} 1\text{-}z_1^\alpha & 1\text{-}z_1^\alpha & 1\text{-}z_1^\alpha & 1\text{-}z_1^\alpha & 0 & 0 & 0 & 0 \\ z_1^\alpha & z_1^\alpha & z_1^\alpha & z_1^\alpha & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1\text{-}z_2^\alpha & 1\text{-}z_2^\alpha & 1\text{-}z_2^\alpha & 1\text{-}z_2^\alpha \\ 0 & 0 & 0 & 0 & z_2^\alpha & z_2^\alpha & z_2^\alpha & z_2^\alpha \end{bmatrix}$$

Finally, putting all the pieces together, we see that the code constructs the matrix $Aeq2 \equiv M_1 - M_5$ (not exhibited here), which corresponds to putting the right hand side terms in (MNC) with opposite signs in the left hand side. The intercept vector is given by $beq2 = (0, 0, 0, 0)^T$ and with this we are done with the equality constraints.

Notice that the coefficient matrices we need consists of repeated patterns of some numbers and the function *kron*, combined with *ones* and *eye* is very efficient in creating such patterns. Alternatively, one could use loops, which however leads to longer and much slower programs. The strength of the Matlab programming language is in performing matrix operations very quickly and we see how we can use this to our best advantage.

5. Participation Constraint

This is the first inequality constraint in our problem. Looking back at the help for *linprog* we see that it has to be put into the form $Ax \leq b$, i.e. we will need to rearrange the terms in (PC). The code is:

```
%SAMPLE MATLAB CODE FOR MORAL HAZARD PROBLEMS (part 7)
A1 = -(1-g)^-1*kron(ones(1, nq*nz),c).^(1-g) - k/(1-d)*...
     (kron(1-z, ones(1,nc*nq))).^(1-d);
b1 = -U;
```

Again, the first term in $A1$ takes care of the consumption part of the utility function and the second one of the action level. The different ordering of the terms within the *kron* operator is due to the fact that c and z are of different ranks in the ordering of the $\pi's$ described above. Also notice the use of the dot operator, since each of the elements of the vectors c and z needs to be raised to a certain power.

[Example]

The vector $A1$ consists of the utility levels achieved by the agent at each of the 8 possibilities for (c, q, z). It is constructed in two steps. First, *kron(1-z, ones(1,nc*nq))* creates the vector $C_1 \equiv (1 - z_1, 1 - z_1, 1 - z_1, 1 - z_1, 1 - z_2, 1 - z_1, 1 - z_1, 1 - z_1)$, of the effort levels in the 8 possible cases, each element of which is then raised to power $1 - \delta$ (notice the .^operator), and multiplied by $-\kappa/(1 - \delta)$. Second, *kron(ones(1, nq*nz),c)* creates the consumption levels vector $C_2 \equiv (c_1, c_2, c_1, c_2, c_1, c_2, c_1, c_2, c_1, c_2)$, familiar to us from the part where I discussed the objective function. Each element of the latter vector is raised to power $1 - \gamma$ and multiplied by $-1/(1 - \gamma)$. The vector $A1$ is just the sum of the above two expressions and takes the form $A_1 = (u(c_1, z_1), u(c_2, z_1), u(c_1, z_1), u(c_2, z_1), u(c_1, z_2), u(c_2, z_2), u(c_1, z_2), u(c_2, z_2))$. The scalar $b1$ equals $-U$, i.e. the reservation utility level taken with the appropriate sign.

6. Incentive Compatibility Constraints

This is perhaps the hardest part of the problem in terms of programming. The reason is that there are constraints not only for each possible action level, z but also for each alternative level, $\hat{z}$. In order to code this part we have to use loops on the action levels and construct the constraints one by one for each possible pair $(z, \hat{z})$. Details about each part of the actual program code are given as comments.

```
%SAMPLE MATLAB CODE FOR MORAL HAZARD PROBLEMS (part 8)
for iz=1: nz                  %loop on the recommended action level, z
zh = [1:iz-1 iz+1:nz];        %vector of all possible alternative action levels, ẑ
 for jz=1:nz-1               %loop on the alternative action level, ẑ

%Constructing the constraints one by one
A2((nz-1)*(iz-1)+jz, :) = kron([zeros(1, iz-1), 1, zeros(1, nz-iz)],...
     kron(ones(1,nq), A1(nc*nq*(iz-1)+1:nc*nq*(iz-1)+nc))) ...
     + kron([zeros(1, iz-1), 1, zeros(1,nz-iz)], ones(1, nq*nc)).*...
     (kron(ones(1, nz), kron([P(2*zh(jz)-1)/P(2*iz-1), P(2*zh(jz))/P(2*iz)] ...
      ,-A1(nc*nq*(zh(jz)-1)+1: nc*nq*(zh(jz)-1)+nc))));
end
end
b2=zeros(nz*(nz-1), 1);
```

Notice two things in the above code. First, we use the already constructed vector of agent's utilities at a point (c, q, z) (the vector $A1$), from where we take the values for $u(c, z)$ which appear in the ICCs. We see that choosing the right order in which one defines the constraint matrices really can help and thus one has to be careful about it. Second, note that the above code is written in a way optimized for the case $m = 2$. If one wants to consider more output levels, apart from obviously changing the value of nq, one needs to modify the part containing the likelihood ratio, *.[P(2*zh(jz)-1)/P(2*iz-1), P(2*zh(jz))/P(2*iz)]* using a loop on the output levels. Note that here there are two ratios, whereas in general they have to be m and the indexation within the P vector needs to be changed accordingly. The construction of the $A2$ matrix looks quite cumbersome on a first sight, but hopefully our usual example will be helpful in understanding how it works.

[Example]

We already know from above that for $nz = 2$, there will be two incentive compatibility constraints. The first value the outer loop counter takes is $iz = 1$, corresponding to (ICC1). Then we have zh=[1:0 2:2], which means that in this case $zh = 2$. It is a scalar since there is only one alternative effort level to be taken. The next line sets $jz = 1$. The expression *(nz-1)*(iz-1)+jz* gives the index of the current ICC in the $A2$ matrix, and equals 1 in this case. Now we move to the actual construction of the vector of coefficients on the $\pi' s$ in the current constraint. As before we start by the inner $kron$ operators first. We take *A1(nc*nq*(iz-1)+1:nc*nq*(iz-1)+nc),* which is A1(1:2) in this case, i.e. the 1st and 2nd elements of the utility vector, which are exactly the $u(c_1, z_1)$ and $u(c_2, z_1)$, which appear in the LHS of (ICC1). Next, the operator *kron(ones(1,nq), A1(nc*nq*(iz-1)+1:nc*nq*(iz-1)+nc))* creates the vector $I_1 = [u(c_1, z_1), u(c_2, z_1), u(c_1, z_1), u(c_2, z_1)]$. The other part of the code in the outer $kron$: *[zeros(1, iz-1), 1, zeros(1, nz-iz)]* is used to create an indicator vector containing one at the index of the current effort level z and zero(s) at all alternative action levels , i.e. in our case it equals $I_2 \equiv (1,0)$. Finally, the outer *kron* creates the vector

$$I_3 = [u(c_1,z_1), 0, u(c_2,z_1), 0, u(c_1,z_1), 0, u(c_2,z_1), 0],$$

which consists of the needed coefficients in the LHS of the first ICC (see (ICC1) and (5)). Creating the vector of coefficients for the RHS is very similar, with the exception that a multiplication by the relevant likelihood ratios is needed. Once again start by 'deciphering' *A1(nc*nq*(zh(jz)-1)+1: nc*nq*(zh(jz)-1)+nc)),* which gives just A1(5:6) in our example, i.e. the values $u(c_1,z_2)$ and $u(c_2,z_2)$ used in the RHS of ICC1. Next, *[P(2*zh(jz)-1)/P(2*iz-1), P(2*zh(jz))/P(2*iz)]* creates the two likelihood ratios (one per output level) given by $P(3)/P(1)$ and $P(4)/P(2)$, i.e. exactly the fractions $r_1 \equiv \frac{p(q_l|z_2)}{p(q_l|z_1)}$ and $r_2 \equiv \frac{p(q_h|z_2)}{p(q_h|z_1)}$ appearing in ICC1. The inner *kron* operator then creates the vector

$$I_4 \equiv [-r_1 u(c_1,z_2), -r_1 u(c_2,z_1), -r_2 u(c_1,z_2), -r_2 u(c_2,z_2)],$$

which using *kron(ones(1, nz),I*$_4$*)* is transformed into $I_5 \equiv [I_4, I_4]$, i.e. a horizontal concatenation of the I_4 vector. The negative signs appear because we have to put all terms on the left hand side of the inequality. Again, we use the indicator vector I_2 in *kron([zeros(1, iz-1), 1, zeros(1,nz-iz)], ones(1, nq*nc))* to create the vector $I_6 \equiv [1,1,1,1,0,0,0,0]$, which is, as a final step, multiplied element by element with I_5 to yield the vector of coefficients of the RHS of ICC1

$$I_7 \equiv [-r_1 u(c_1,z_2), -r_1 u(c_2,z_1), -r_2 u(c_1,z_2), -r_2 u(c_2,z_2), 0, 0, 0, 0]$$

Finally, by adding I_3 and I_7, the construction of the first row of A2 is completed. The second ICC is created in exactly the same way at the next loop iteration ($iz = 2$). The intercept vector is $b2 = [0,0]^T$ by the same logic as for the participation constraints.

With this final type of constraints we are finally done with creating the ingredients we need to start the actual optimization, which is what is done in the next step.

3.3 Step 3: Actual Linear Programming

In this step we first need to put together the pieces of the matrices of constraint coefficients and feed them into *linprog*. I also discuss shortly how one can customize the working of *linprog* using the command *optimset*. Let's start by constructing the matrices required by *linprog*:

```
%SAMPLE MATLAB CODE FOR MORAL HAZARD PROBLEMS (part 9)
Aeq=[Aeq1; Aeq2];      %matrix of coefficients on the equality constraints
beq=[beq1; beq2];      %intercepts
A=[A1; A2];                 %matrix of coefficients on the inequality constraints
b=[b1; b2];                 %intercepts
```

The above code just performs the vertical concatenation of the matrices created above in order to put them into form suitable for *linprog*. Now we are finally in position to obtain the solution to our moral hazard problem using *linprog*. The resulting unconditional probabilities will be stored in the vector X and the value of the objective function at the optimum is returned in *vobj* :

```
        %SAMPLE MATLAB CODE FOR MORAL HAZARD PROBLEMS (part 10a)
[X, vobj] = linprog(f, A, b, Aeq, beq, LB, UB);
```

It is quite simple now! Depending on the dimensions on the grids, one waits 5-10 seconds and the program returns the answer. The really hard part was constructing the input matrices.

It is also possible to customize the working of *linprog* by setting different error tolerances, display properties, or the type of optimization algorithm used. This is achieved by the Matlab command *optimset* (type *help optimset* on the command prompt for a complete description). For example one can write[6]:

```
        %SAMPLE MATLAB CODE FOR MORAL HAZARD PROBLEMS (part 10b)
options = optimset('Display','off','TolFun', 10^-9,'MaxIter',150,'TolX', 10^-8);
[X, vobj] = linprog(f, A, b, Aeq, beq, LB, UB,[],options);
```

which makes Matlab not to display the interim output of *linprog* on the screen, sets the error tolerance for the function value to 10^-9, the maximal number of iterations to 150 and the error tolerance of the variables vector to 10^-8. The *optimset* command should precede *linprog* in the code in order to have effect. Note also the need to put an empty matrix, [] in the *linprog* inputs, since we've skipped the initial guess input (see *help linprog*).

3.4 Step 4: Recovering the Optimal Contract

What *linprog* gives us as its output is the vector of probabilities, which constitute a lottery over all possible (c, q, z) combinations. Usually this vector contains a lot of zeros and also, in most cases, we are not interested in the probabilities per se, but instead in the optimal values for the consumption, action and output levels implied by them. The purpose of this section is to describe how one can obtain those using the output of *linprog*, x.

The easiest way to do it is to start by creating three vectors of length lmn, consisting respectively of the values for consumption, output and action level for all possible c, z, q. This is done as usual with the help of *kron*.

```
%SAMPLE MATLAB CODE FOR MORAL HAZARD PROBLEMS (part 11)
cc = kron(ones(1, nq*nz), c);
qq = kron(ones(1, nz), kron(q, ones(1, nc)));
zz = kron(z,ones(1, nc*nq));
```

[Example]
In our example, using the definition of $kron$, we will have:

$$\begin{aligned} cc &= [c_1, c_2, c_1, c_2, c_1, c_2, c_1, c_2] \\ qq &= [q_1, q_1, q_2, q_2, q_1, q_1, q_2, q_2] \\ zz &= [z_1, z_1, z_1, z_1, z_2, z_2, z_2, z_2] \end{aligned}$$

which are exactly the consumption, output and action levels corresponding to the ordered 8 cases.

[6]Note that this piece of code is an alternative to part 10a above and one should not use both in the same program.

Now, for each element of X (i.e. each (c, q, z)), we can recover directly the values for consumption, output and action level corresponding to it. Let's however save the computer some work by picking only the non-zero values from the vector of probabilities X.

```
%SAMPLE MATLAB CODE FOR MORAL HAZARD PROBLEMS (part 12)
xp=find(X >10^-4);      %gives the indices of all elements of X > 10^-4
```

Note that one must not write $X > 0$ above. The reason is that there is numerical error inherent in the computation and in practice all X will turn out to be positive, eventhough they are not at the true solution. One should always remember that the computer gives only approximate answers, not the true ones. The operator $find(\arg 1 > \arg 2)$ gives as an output a vector containing the indices of the elements of the vector arg 1, which are bigger than some number arg 2. For example if $\arg 1 = [3, 1, 2, 4, 5]$ then $find(\arg 1 > 2)$ yields the vector $[1, 4, 5]$.

Finally, we are ready to compute the optimal contract implied by the lottery putting probability on a given triple (c, q, z), i.e. the recommended action and the consumption the agent will get for a given output level. The following code exhibits the contract in a compact form, getting rid of all near zero probability events:

```
%SAMPLE MATLAB CODE FOR MORAL HAZARD PROBLEMS (part 13)
disp('        z            q            c            prob')
disp('——————————————————————————')
disp([zz(xp)', qq(xp)', cc(xp)', X(xp)]);
```

The operator *disp*(*string*) prints on the screen the contents of the text in *string*. Thus, the above code prints out four columns in which one can see respectively the recommended action level, output and consumption together with the probability which the lottery assigns to them. In some cases, it is possible to obtain positive probabilities on two consumption levels for the same values of z and q in a strictly concave problem, for which we know that a single c is optimal in theory. This is a so-called 'grid lottery' and has no economic interpretation in the case of continuous consumption and action, but reflects the fact that the optimal consumption level is just not present in the grid[7]. Clearly the error introduced by grid lotteries diminishes as one increases the grid dimensions by adding points near the optimal solution.

3.5 Solving for the First Best Contract

In the end, it is worth mentioning that the above program can also be used to compute the first best (full information) principal-agent contract. In order to do so all that needs to be changed is to replace A with $A1$ and b with $b1$ in the *linprog* command line, which is equivalent to ignoring the incentive compatibility constraints. I encourage the reader to run the code and check that s(he) obtains a different contract after making this replacement, and also that this contract is such that gives higher utility to the principal and (barring grid lotteries) gives constant consumption to the agent as predicted by the theory.

[7]For more on grid lotteries see the discussion in Prescott (1999).

4 An Alternative Specification of the Moral Hazard Problem With Lotteries

In this section I consider a slightly modified version of the 'standard' moral hazard problem described above. Instead of having a principal who maximizes her utility subject to incentive compatibility and participation by the agent, we now have a social planner who maximizes a welfare function subject to a resource (zero-profit) constraint. Using the same notation as in the previous section, the problem looks as follows:

$$\max_{\pi(c,q,z)} \sum_{c,q,z} \pi(c,q,z)u(c,z) \tag{6}$$

s.t.

$$\sum_{c} \pi(c,q,z) = p(q|z) \sum_{c,q} \pi(c,q,z) \quad \text{for all } q,z \tag{7}$$

$$\sum_{c,q,z} \pi(c,q,z)(c-q) = W \tag{8}$$

$$\sum_{c,q} \pi(c,q,z)u(c,z) \geqq \sum_{c,q} \pi(c,q,z)\frac{p(q|z')}{p(q|z)}u(c,z') \qquad \text{for all } z,z' \tag{9}$$

The above linear program can be interpreted as a social planner's problem of maximizing the expected utility of the agent. The planner recommends an action level, z and gives consumption c to the agent. We see that the only difference from before, except for the objective function, is the absence of the participation constraint, which is replaced by (7) - a zero-profit constraint, stating that the agent's wealth, W plus the expected produced output q must be enough to cover for the expected consumption granted, c.

It is clear that the method described above can be used to solve this problem. In fact, only two slight modifications to our previous code are needed. First, we need to change the expression for f to reflect the different objective function. Note, however, that this does not amount to doing anything new, since we already have the needed vector of coefficients in the matrix $A1$ from before. Thus we only need to write:

```
f = -(1-g)^-1*kron(ones(1,nq*nz), c).^(1-g) - k/(1-d)*...
     (kron(1-z, ones(1,nc*nq))).^(1-d);
```

Second, we also need to replace the participation constraint with the zero-profit one, i.e. a new $A1$ is needed. Looking at (7) however, we see that, once again, nothing new needs to be done - the vector of coefficients we need is nothing but the former f vector, i.e. in a sense the two problems are dual to each other. Also, now this is an equality constraint, so we'll call it $Aeq3$ instead of $A1$ and then put it together with the other equality constraints, leaving only $A2$ as the matrix of inequality constraints coefficients.

```
Aeq3 = -kron(ones(1, nz), kron(q, ones(1, nc))) + kron(ones(1, nq*nz),c);
beq3 = W;
```

Modifying accordingly the matrices A, b, Aeq and beq we have already written the code for solving this type of moral hazard problems as well.

5 Conclusive Remarks

Above I have described in detail how one can solve moral hazard problems with lotteries numerically using Matlab. The method I presented has been optimized for computational speed, and that is why some fragments of the supplied code might look quite cryptic at first sight. I recommend that readers study the provided examples carefully and also try some examples of their own within the framework of the above programs in order to understand how they really work. Re-writing all the code without using the 'mysterious' *kron* operator, but instead more standard programming techniques as loops and perhaps the function *repmat* would be a good exercise for readers who experience troubles understanding how the above works. The resulting loss in speed will not be crucial for low-dimensional problems, however the actual code will be much longer, increasing the probability of coding errors and, subsequently, the debugging time.

The linear programming method is only one of the several available for solving moral hazard problems, and in fact it is suitable only if we allow for lotteries. Another possible approach is to disregard lotteries and solve the general moral hazard non-linear optimization problem using the first-order approach (when it is valid) and Matlab minimization routines as *fminsearch* (using the penalty function method) or *fmincon* (using quadratic programming algorithms).

References

[1] Doepke, M. and R. Townsend, 2001, "Credit Guarantees, Moral Hazard, and the Optimality of Public Reserves", Mimeo.

[2] Lehnert, A., 1998, "Asset Pooling, Credit Rationing, And Growth", Working Paper, Board of Governors of the Federal Reserve System, Washington, DC.

[3] Mehrotra, S., 1992, "On the Implementation of a Primal-Dual Interior Point Method," *SIAM Journal on Optimization*, Vol. 2, pp. 575-601.

[4] Phelan, C. and R. Townsend, 1991, "Computing Multi-Period Information-Constrained Optima", *Review of Economic Studies*, 5, pp.853-881.

[5] Prescott, E.C. and R. Townsend, 1984a, "General Competitive Analysis in an Economy with Private Information", *International Economic Review*, 25, pp.1-20.

[6] Prescott, E.C. and R. Townsend, 1984b, "Pareto Optima and Competitive Equilibria with Adverse Selection and Moral Hazard", *Econometrica*, 52, pp.21-46.

[7] Prescott, E. S., 1998, "Computing Moral Hazard Problems Using the Dantzig-Wolfe Decomposition Algorithm", Federal Reserve Bank of Richmond Working Paper 98-6.

[8] Prescott, E. S., 1999, "A Primer on Moral Hazard Models", *Federal Reserve Bank of Richmond Quarterly Review*, vol. 85/1.

[9] Prescott, E.S. and R. Townsend, 2000a, "Inequality, Risk Sharing, and the Boundaries of Collective Organization", Mimeo.

[10] Prescott, E.S. and R. Townsend, 2000b, "Firms As Clubs in Walrasian Markets with Private Information", Federal Reserve Bank of Richmond Working Paper 00-8.

[11] Rogerson, W., 1985, "The First-Order Approach to Principal-Agent Problems", *Econometrica* 53(6), pp.1357-1368.

[12] Townsend, R., 1987, "Economic Organization with Limited Communication", *American Economic Review*, 77, pp.954-971.

[13] Townsend, R., 1988, "Information Constrained Insurance: The Revelation Principle Extended", *Journal of Monetary Economics*, 21, pp.411-450.

[14] Zhang, Y., 1995, "Solving Large-Scale Linear Programs by Interior-Point Methods Under the MATLAB Environment," Technical Report TR96-01, Department of Mathematics and Statistics, University of Maryland, Baltimore, MD.