Combinatorial Discrete Choice Problem

Notes on MATLAB Code

Yijiang Zhou [*]

In this note, I brief introduce the code that solves combinatorial discrete choice (CDC) problem based on the method in Arkolakis and Eckert (2017). The code files are written in MATLAB 2019b by myself and can be found at https://github.com/yijiangzhou/Study-Research/tree/master/CDC. A similar Python version of the code can be found on Professor Fabian Eckert's personal website.

The file `CDC_main.m` calls all the functions. I will walk through this file line by line, and introduce every self-written function contained in it.

```
1 % Calculating derivative of a Boolean-domain function
2 vec1 = [1 1 1];
3 vec2 = [1 1];
4 test1 = booldiff(@booldiff_test,2,vec1);
5 test2 = booldiff(@jia_test,2,vec1);
6 test3 = booldiff(@aeeg2_test,1,vec2);
7 clear vec1 vec2
```

The file begins by testing the function `booldiff()` that calculates the derivative of a Boolean vector. The derivative is defined as follows:

$$D_i\Pi(I) = \Pi(I^{i\to 1}) - \Pi(I^{i\to 0})$$

where $I^{i\to a}$ is the Boolean vector I with the ith coordinate set to a. What follows is the function `booldiff()` in `booldiff.m`, which sets the ith coordinate of the input vector to 1 and 0, respectively. It then calculates the value of $f(high)$ and $f(low)$ and takes difference of the two, where f is an input function.

```
1  function D_i = booldiff(f,i,vector)
2  % This function calculates the derivative of f on the Boolean domain with
3  % respect to the ith coordinate.
4  
5  vector(i) = 1;
6  high = vector;
7  vector(i) = 0;
8  low = vector;
9  
10 D_i = f(high) - f(low);
```

When testing whether `booldiff()` works properly, I arbitrarily constructed three input functions: `booldiff_test()`, `jia_test()` and `aeeg2_test()`. They are irrelevant to the CDC problem and thus omitted in the notes. Notice that in order to call another function in `booldiff()`, we must type "@" before the function name, like in line 4-6 of the first code block.

[*]Department of Economics, the Chinese University of Hong Kong. Email: yijiangzhou@link.cuhk.edu.hk

```matlab
% The AE iteration
Istar_te = AE(3,@jia_test,'super');
Istar_te2 = AE(2,@aeeg2_test,'super');
```

The next function we want to test is `AE()`, the core iteration algorithm to solve CDC problems. It corresponds to the following definition, borrowed directly from Arkolakis and Eckert (2017):

Definition 1. Consider a complete lattice $\mathbf{I}^0$, a payoff function $\Pi(I)$ and form:

$$\bar{\Omega}_1\left(\mathbf{I}^0\right) = \left\{i : D_i\left(\Pi\left(\sup \mathbf{I}^0\right)\right) < 0\right\} \quad \text{and} \quad \underline{\Omega}_1\left(\mathbf{I}^0\right) = \left\{i : D_i\left(\Pi\left(\inf \mathbf{I}^0\right)\right) \geq 0\right\}$$

Then, we define the mapping AE_1: $S\left(\mathbf{I}^0\right) \rightarrow S\left(\mathbf{I}^0\right)$ as

$$AE_1\left(\mathbf{I}^0\right) = \left\{I \in \mathbf{I}^0 : I_i = 0 \text{ and } I_j = 1, \forall i \in \bar{\Omega}_1\left(\mathbf{I}^0\right), \forall j \in \underline{\Omega}_1\left(\mathbf{I}^0\right)\right\}$$

Similarly, form the following:

$$\bar{\Omega}_2\left(\mathbf{I}^0\right) = \left\{i : D_i\left(\Pi\left(\sup \mathbf{I}^0\right)\right) > 0\right\} \quad \text{and} \quad \underline{\Omega}_2\left(\mathbf{I}^0\right) = \left\{i : D_i\left(\Pi\left(\inf \mathbf{I}^0\right)\right) \leq 0\right\}$$

Then, we define the mapping AE_2: $S\left(\mathbf{I}^0\right) \rightarrow S\left(\mathbf{I}^0\right)$ as

$$AE_2\left(\mathbf{I}^0\right) = \left\{I \in \mathbf{I}^0 : I_i = 1 \text{ and } I_j = 0, \forall i \in \bar{\Omega}_2\left(\mathbf{I}^0\right), \forall j \in \underline{\Omega}_2\left(\mathbf{I}^0\right)\right\}$$

The `AE.m` file contains the `AE()` function, presented below. It has three inputs: the dimensions of Boolean space that is to be evaluated, the payoff function, and an indicator on whether the function's modularity. If the user accidentally enters a N that is smaller than 2 or is not a integer, the function stops executing and reports an error. Otherwise, the function begins executing by creating a $1 \times N$ vector whose elements are all 0.5. This is a very useful trick borrowed from the Python version codes on Prof. Eckert's website. Instead of actually creating a N dimensional Boolean space as in Definition 1, we work with a $1 \times N$ vector whose "0.5"s indicate coordinates that are not yet determined by the `AE()` iteration. The two initially identical vectors, *output* and $output_{after}$, are then put into a `while` loop. By replacing the "0.5" coordinates with 1, we obtain $\sup \mathbf{I}^0$. Replacing the same coordinates with 0 produces $\inf \mathbf{I}^0$.

```matlab
function Istar = AE(N,func,modular)
% This function implements the AE1 and AE2 algorithm.

if N < 2 || floor(N) ~= N
    disp('Error! N must be an integer greater than 1.')
    Istar = NaN;
else
    output = zeros(1,N);
    output_after = zeros(1,N);
    output_after(output_after == 0) = 0.5;
    % output_after is the initial I, with all elements uncertain, i.e. all
    % elements equal to 0.5.
    iter = 0;

    while any((output ~= output_after))
        output = output_after;
        supr = output_after;
        infi = output_after;
        supr(supr == 0.5) = 1;
        infi(infi == 0.5) = 0;

        if strcmp(modular,'super') == 1 % The supermodular case
            for i = 1:N
                if booldiff(func,i,supr) < 0 && booldiff(func,i,infi) < 0
                    output_after(i) = 0;
                elseif booldiff(func,i,supr) >= 0 && booldiff(func,i,infi) >= 0
                    output_after(i) = 1;
                end
            end
        elseif strcmp(modular,'sub') == 1 % The submodular case
            for i = 1:N
                if booldiff(func,i,supr) > 0 && booldiff(func,i,infi) > 0
                    output_after(i) = 1;
                elseif booldiff(func,i,supr) <= 0 && booldiff(func,i,infi) <= 0
                    output_after(i) = 0;
                end
            end
        end

        iter = iter + 1;
    end

    if ~isempty(output_after(output_after == 0.5))
        Istar = output_after;
        disp('Iteration failed!')
        disp(['iter = ',num2str(iter-1)])
    else
        Istar = output_after;
        disp('Iteration succeeded!')
        disp(['iter = ',num2str(iter-1)])
    end

end
```

If the payoff function $\Pi(I)$ is super-modular, we would implement the mapping AE_1 as in Definition 1 on the vector $output_{after}$. The produced $AE_1(output_{after})$ are compared to *output* and the `while` loop would continue unless these two are equal. The next round of loop, if any, then sets the "0.5" elements of $AE_1(output_{after})$ to 1 or 0 to get $\sup AE_1(output_{after})$ and $\inf AE_1(output_{after})$, respectively. The AE_2 mapping (and the loop) is implemented in a similar fashion. Ultimately, the loop produces a $1 \times N$ vector that may or may not contain "0.5" elements and the iteration only succeeds when none of the element equals to 0.5.

As discussed in Arkolakis and Eckert (2017), the AE_1 or AE_2 mapping may not converge to a singleton. In the above algorithm, it means the output I^* might still contains elements equaling to 0.5. The paper therefore proposes the following AER algorithm whose outcome is surely a singleton.

Definition 2. Consider a function $\Pi : \mathcal{B}^n \longrightarrow \mathbb{R}$ on a non-singleton complete lattice $\mathbf{I}$ that exhibits single crossing differences from below property:

1. Iterate on AE_1 until convergence. If $|\mathbf{I}^\star| = 1, I^\star = \mathbf{I}^\star$, else continue.

2. Pick any sub-lattices $\mathbf{I}_1$, $\mathbf{I}_2$ of $\mathbf{I}$, s.t. $\mathbf{I} = \mathbf{I}_1 \cup \mathbf{I}_2$ and $\emptyset = \mathbf{I}_1 \cap \mathbf{I}_2$.

3. Iterate AE_1 on $\mathbf{I}_1, \mathbf{I}_2$ separately resulting in fixed points $\mathbf{I}_1^\star$ and $\mathbf{I}_2^\star$.

4. If $|\mathbf{I}_1^\star| = |\mathbf{I}_2^\star| = 1$ then $I^\star = \arg\max_{I \in \mathbf{I}_1^\star \cup \mathbf{I}_2^\star} \Pi(I)$.

5. Else pick sub-lattices $\mathbf{I}_{i,1}, \mathbf{I}_{i,2}$ of $\mathbf{I}_i^\star$ st. $\mathbf{I}_i = \mathbf{I}_{i,1} \cup \mathbf{I}_{i,2}$ and $\emptyset = \mathbf{I}_{i,1} \cap \mathbf{I}_{i,2}$ for $i = 1, 2$ and repeat.

AER is defined analogously for objectives that exhibit single crossing differences from above.

To achieve AER in MATLAB, we have to do a few modifications to `AE()` and this results in `AE_alter()`. The latter function performs the exact iteration steps as in `AE()` except its first input is now a specific vector instead of N. It does not show whether iteration is successful or not. The detailed execution steps have also been changed so that when a vector which does not contain any "0.5" element is put in, `AE_alter()` directly returns the same vector without going to the time-consuming iterations. The code of `AE_alter()` is omitted in this note.

```
% The alternative AE iteration
Istar_te3 = AE_alter([0.5 1],@aeeg2_test,'super');
Istar_te4 = AE_alter([1 0],@aeeg2_test,'super');
Istar_te5 = AE_alter([0.5 0.5 0.5],@jia_test,'super');

% % The AER iteration
[Istar_te6,pistar_te6] = AER(repmat(0.5,1,3),@jia_test,'super');
[Istar_te7,pistar_te7] = AER(repmat(0.5,1,2),@aeeg2_test,'super');
```

We are now ready to test the function `AE_alter()` and `AER()`. The file `AER.m` contains the `AER()` function and it is actually not complicated. Inputs include the initial $1 \times N$ vector, the payoff function and the modular indicator. The outputs are the optimal vector and the corresponding payoff value. After confirming that the input vector has at least one "0.5" element, the program puts the vector into `AE_alter()`. This could lead to one of the following two different results. In the first one, `AE_alter()` alone generates a definitive vector and returns I^* as that vector. In the second one, the output vector still contains some "0.5" elements and we model sub-lattice $\mathbf{I}_1$ by turning the first "0.5" element into 1. The sub-lattice $\mathbf{I}_0$ instead sets that same element to 0.

```
function [Istar,pistar] = AER(vector,func,modular)
% This function implements the AE1 and AE2 algorithm.

if isempty(vector(vector == 0.5))
    Istar = vector;
    pistar = func(vector);
else
    output = AE_alter(vector,func,modular);
    if isempty(output(output == 0.5)) % If the first AE attempt is
    successful...
        Istar = output;
        pistar = func(Istar);
    else
        sub_1 = output;
        sub_0 = output;
        for i = 1:length(output)
            if output(i) == 0.5
                sub_1(i) = 1;
                sub_0(i) = 0;
                break
            end
        end
        % Implement AER recursively
        [output_after_1,pi_after_1] = AER(sub_1,func,modular);
        [output_after_0,pi_after_0] = AER(sub_0,func,modular);
        if pi_after_1 >= pi_after_0
```

```
                Istar = output_after_1;
                pistar = pi_after_1;
            else
                Istar = output_after_0;
                pistar = pi_after_0;
            end
        end
end
```

The `AER()` function is a recursive one, in the sense that $\mathbf{I}_1$ and $\mathbf{I}_0$ are fed back to the function itself until they both converge to vectors whose coordinates are either 1 or 0. Of the two vectors, the one that delivers a higher payoff is the I^* we are looking for.

```
% The Jia (2008) payoff function
A1 = [1 1 0 1 0 1 1 0 0 1 0 1 0 0 0 1 1 1 0 1];
y1 = jia(A1);
```

The last function we want to test is the payoff function in Jia (2008). In the paper, a chain store chooses the optimal set of locations I^* to maximize the payoff. For example, $I^* = (1, 0, 1)$ indicates there are chain stores in location 1 and 3. The payoff function is as follows:

$$\Pi(I) = \sum_{i=1}^{n} I_i \times \left(X_i + \delta \times \sum_{i' \neq i}^{n} \frac{I_{i'}}{\tau_{ii'}} \right)$$

where i is a particular potential store location and X_i is the independent payoff at that location. I_i is therefore the ith element of I. The term $\delta \times \sum_{i' \neq i}^{n} \frac{I_{i'}}{\tau_{ii'}}$ controls for spillovers between different branches, with $\tau_{ii'}$ representing the distance between i and i'. A positive δ indicates positive spillovers as well as the super-modularity of $\Pi(I)$, and a negative δ means otherwise. The file `jia.m` contains this payoff function.

```
function y = jia(vector)
% A payoff function in Jia (2008)'s style.
% Note that the number 5 needs to be a factor of length(vector) in this
% example. This is due to how I calculate the "distance" between locations.

n = length(vector);
matrix = zeros(5,n/5);
for i = 1:n
    matrix(i) = vector(i);
end

sigma = 0.5; % sigma > (<) 0: jia() is supermodular (submodular).
seed = 1212;
rng(seed,'twister')
X = 10 * randn(5,n/5); % The normally distributed independent payoff
% The seed and rng() function ensures X is consistent, instead of being
% different every time jia() function is called in AE or AER loops.

dep_payoff = zeros(5,n/5); % The dependent payoff
for i = 1:n
    container = zeros(5,n/5);
    for j = 1:n
        if i == j
            container(j) = 0;
        else
            [rowj,colj] = ind2sub(size(container),j);
            [rowi,coli] = ind2sub(size(container),i);
```

```matlab
            tau_ij = sqrt((rowi - rowj)^2 + (coli - colj)^2);
            container(j) = matrix(j) / tau_ij;
        end
    end
    dep_payoff(i) = sigma * sum(container,'all');
end

container2 = zeros(1,n);
for i = 1:n
    container2(i) = matrix(i) * (X(i) + dep_payoff(i));
end

y = sum(container2,'all');
```

The function input is a $1 \times N$ vector. To simplify the distance measure, I place the vector coordinates in a $5 \times \frac{N}{5}$ matrix. An example would be the input vector I and the corresponding matrix $\mathbf{M}$:

$$I = (I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9, I_{10}, I_{11}, I_{12}, I_{13}, I_{14}, I_{15}) \Rightarrow \mathbf{M} = \begin{pmatrix} I_1 & I_6 & I_{11} \\ I_2 & I_7 & I_{12} \\ I_3 & I_8 & I_{13} \\ I_4 & I_9 & I_{14} \\ I_5 & I_{10} & I_{15} \end{pmatrix}$$

The distance between, say, I_8 and I_{12} would be $\sqrt{(3-2)^2 + (2-3)^2}$ since they are at the $(3, 2)$ and $(2, 3)$ location of the matrix, respectively. In the code, I assume the location-specific payoff X_i is drawn from normal distribution $N(0, 10^2)$ and setting a seed value (which can be arbitrary) would ensure that every time we put `jia()` function into AE or AER, the X_i values are consistent across iterations. I also assume $\delta = 0.5$ so that the payoff function is super-modular.

```matlab
1 % To test jia(), the input N (of AE and AER) must be equal to (5 * n),
2 % where n is a positive integer.
3 Istar1 = AE(20,@jia,'super');
4 [Istar2,pistar2] = AER(repmat(0.5,1,20),@jia,'super');
5 tic
6 [Istar3,pistar3] = AER(repmat(0.5,1,25),@jia,'super');
7 toc
```

We are now ready to implement the AE and AER algorithm on the `jia()` payoff function. When setting the dimension of Boolean space to be 20, the AE algorithm listed in line 3 of code block above would fail to converge to a singleton. That is where the AER algorithm steps in, which would converge to a specific vector and also return the maximum payoff. The AER algorithm is extremely efficient. Even if we set the dimension of Boolean space to be as high as 50, meaning that there would be 2^{50} Boolean vectors in it, AER would usually find the optimal vector within 2 seconds.

It would be fun to see how efficient AER is relative to brutal search. In what follows I model a 25-dimensional Boolean space and search the optimal vector with brutal force.

```matlab
% jia(), but with brutal force
tic
A = boolmatrix(25);
container = zeros(length(A),1);
for i = 1:length(A)
    container(i) = jia(A(i,:));
end
```

```
pistar_force = max(container);
Istar_force = A(container == pistar_force,:);
toc
```

The tricky part is to model the Boolean space. I set the Boolean vectors as row vectors and combine them into a $2^N \times N$ matrix, where N is the dimension of Boolean space. MATLAB has no built-in functions, as far as I know, that lists all vectors in a Boolean space and I have to write a function `boolmatrix()` to do it. I illustrate how `boolmatrix()` works in a simple example with $N = 3$:

$$B_1 = (0, 0, 0)$$

$$B_2 = (0, 0, 1)$$

$$B_3 = (0, 1, 0)$$

$$B_4 = (0, 1, 1)$$

$$B_5 = (1, 0, 0)$$

$$B_6 = (1, 0, 1)$$

$$B_7 = (1, 1, 0)$$

$$B_8 = (1, 1, 1)$$

If we order the 8 vectors in 3-dimensional Boolean space in a particular way as above, it is not hard to find some patterns. First, the initial two vectors are always $(..., 0, 0, 0, 0)$ and $(..., 0, 0, 0, 1)$ no matter how big N is. Second, B_3 and B_4 simply turns the next to last digit to 1 and borrows the last digit from the previous 2 vectors. Similarly, B_5 to B_8 turns the third from the last digit to 1 and borrows the last digit from the previous 4 vectors. Therefore, if we write down the first two vectors, the rest vectors of Boolean space can be easily generated: the B_{2^j+1} to $B_{2^{j+1}}$ vectors set the $(j+1)$th from the last digit to 1, and borrows the last digit from the previous 2^j vectors, $j = 1, 2, ..., N-1$. The file `boolmatrix.m` implements this process.

```
function bool = boolmatrix(N)
% This function creates Boolean vectors as rows of a matrix.

% e.g. Let N = 3 and the function creates a 2^3 * 3 matrix, whose rows are
% distinct Boolean vectors like [1 0 0] or [0 1 0].

matrix = zeros(2^N,N);
matrix(1,:) = zeros(1,N);
matrix(2,:) = [zeros(1,N-1) 1];

for i = 3:2^N
    for j = 1:N-1
        if 1 + 2^j <= i && i <= 2^(j+1)
            vector = zeros(1,N);
            vector(N-j) = 1;
            matrix(i,:) = matrix(i-2^j,:) + vector;
        end
    end
end

bool = matrix;
```

When setting $N = 25$, a $2^{25} \times 25$ matrix is generated and MATLAB searches the row that maximizes payoff function `jia()`. It took more than 3 hours to find that optimal row vector using brutal force search. In contrast, AER gets the same thing done in less than 1 second. When N is even bigger, MATLAB would get stuck and eventually crash when constructing Boolean space. This demonstrates the absolute necessity of relying on AER when solving CDC problems with a large N.

References

Arkolakis, C., & Eckert, F. (2017). *Combinatorial Discrete Choice.* doi: 10.2139/ssrn.3455353

Jia, P. (2008). What Happens When Wal-Mart Comes to Town: An Empirical Analysis of the Discount Retailing Industry. *Econometrica*, *76*(6), 1263–1316.