KU LEUVEN

Computing score functions numerically using Mata

2021 UK Stata Conference

Álvaro A. Gutiérrez-Vargas (@alvarogutyerrez, ,)

Research Centre for Operations Research and Statistics (ORStat)
Faculty of Economics and Business
KU Leuven, Belgium

1 Outline

1 Introduction

- **In short**:

1 Introduction

- **In short**: We will see a workaround that allows us to compute robust variance-covariance matrices when the `ml` (see [R] **ml** and Gould et al. (2010)) command fails to provide them.

1 Introduction

- **In short**: We will see a workaround that allows us to compute robust variance-covariance matrices when the `ml` (see [R] **ml** and Gould et al. (2010)) command fails to provide them.
- **When will I need this?**:

1 Introduction

- **In short**: We will see a workaround that allows us to compute robust variance-covariance matrices when the `ml` (see [R] **ml** and Gould et al. (2010)) command fails to provide them.
- **When will I need this?**: Only when working with models that do not meet the linear-form restrictions. Otherwise, `ml` does it automatically.

1 Introduction

- **In short**: We will see a workaround that allows us to compute robust variance-covariance matrices when the `ml` (see [R] **ml** and Gould et al. (2010)) command fails to provide them.
- **When will I need this?**: Only when working with models that do not meet the linear-form restrictions. Otherwise, `ml` does it automatically.
- **Why is this relevant?**:

1 Introduction

- **In short**: We will see a workaround that allows us to compute robust variance-covariance matrices when the `ml` (see [R] **ml** and Gould et al. (2010)) command fails to provide them.
- **When will I need this?**: Only when working with models that do not meet the linear-form restrictions. Otherwise, `ml` does it automatically.
- **Why is this relevant?**: Because **we cannot** longer only type "**r**obust" to implement robust/clustered corrected variance-covariance matrices in our programs.

1 Introduction

- **In short**: We will see a workaround that allows us to compute robust variance-covariance matrices when the `ml` (see [R] **ml** and Gould et al. (2010)) command fails to provide them.
- **When will I need this?**: Only when working with models that do not meet the linear-form restrictions. Otherwise, `ml` does it automatically.
- **Why is this relevant?**: Because **we cannot** longer only type "**r**obust" to implement robust/clustered corrected variance-covariance matrices in our programs.
- **How can we solve such problem?**:

1 Introduction

- **In short**: We will see a workaround that allows us to compute robust variance-covariance matrices when the `ml` (see [R] **ml** and Gould et al. (2010)) command fails to provide them.
- **When will I need this?**: Only when working with models that do not meet the linear-form restrictions. Otherwise, `ml` does it automatically.
- **Why is this relevant?**: Because **we cannot** longer only type "**r**obust" to implement robust/clustered corrected variance-covariance matrices in our programs.
- **How can we solve such problem?**: We will numerically approximate the score functions using Mata's `deriv()` function (see [R] **deriv** and Gould (2018)) *squeezing* our log-likelihood function and using them to compute sandwich variance estimators.

1 Introduction

- **In short**: We will see a workaround that allows us to compute robust variance-covariance matrices when the `ml` (see [R] **ml** and Gould et al. (2010)) command fails to provide them.
- **When will I need this?**: Only when working with models that do not meet the linear-form restrictions. Otherwise, `ml` does it automatically.
- **Why is this relevant?**: Because **we cannot** longer only type "**r**obust" to implement robust/clustered corrected variance-covariance matrices in our programs.
- **How can we solve such problem?**: We will numerically approximate the score functions using Mata's `deriv()` function (see [R] **deriv** and Gould (2018)) *squeezing* our log-likelihood function and using them to compute sandwich variance estimators.
- **The talk seems off from my interests. Should I grab a coffee instead?**:

1 Introduction

- **In short**: We will see a workaround that allows us to compute robust variance-covariance matrices when the `ml` (see [R] **ml** and Gould et al. (2010)) command fails to provide them.
- **When will I need this?**: Only when working with models that do not meet the linear-form restrictions. Otherwise, `ml` does it automatically.
- **Why is this relevant?**: Because **we cannot** longer only type "robust" to implement robust/clustered corrected variance-covariance matrices in our programs.
- **How can we solve such problem?**: We will numerically approximate the score functions using Mata's `deriv()` function (see [R] **deriv** and Gould (2018)) *squeezing* our log-likelihood function and using them to compute sandwich variance estimators.
- **The talk seems off from my interests. Should I grab a coffee instead?**: Well... maybe **(?)**

1 Introduction

- **In short**: We will see a workaround that allows us to compute robust variance-covariance matrices when the `ml` (see [R] **ml** and Gould et al. (2010)) command fails to provide them.

- **When will I need this?**: Only when working with models that do not meet the linear-form restrictions. Otherwise, `ml` does it automatically.

- **Why is this relevant?**: Because **we cannot** longer only type "robust" to implement robust/clustered corrected variance-covariance matrices in our programs.

- **How can we solve such problem?**: We will numerically approximate the score functions using Mata's `deriv()` function (see [R] **deriv** and Gould (2018)) *squeezing* our log-likelihood function and using them to compute sandwich variance estimators.

- **The talk seems off from my interests. Should I grab a coffee instead?**: Well... maybe **(?)**, but you will lose some "very" interesting tricks about **numerical derivatives** using Mata that might be useful someday!

2 Outline

1. Introduction
2. The `ml` command
3. Linear-form Restriction
4. The Problem
5. Robust Variance Covariance Matrix: A very brief review
6. The Solution
7. Conclusions

2 The `ml` command

- The **`ml`** command allows us to fit models using Maximum Likelihood.

2 The `ml` command

- The **`ml`** command allows us to fit models using Maximum Likelihood.
- The command has different types of evaluators (e.g., `lf`-family, `gf`-family, and `d`-family) which vary in terms of what kind of models they can be fit.

2 The `ml` command

- The **`ml`** command allows us to fit models using Maximum Likelihood.
- The command has different types of evaluators (e.g., `lf`-family, `gf`-family, and `d`-family) which vary in terms of what kind of models they can be fit.
- In particular: we will focus on models where the log-likelihood function does not meet the linear-form restrictions, which can be fitted using the `d`-family of evaluators.

2 The `ml` command

- The `ml` command allows us to fit models using Maximum Likelihood.
- The command has different types of evaluators (e.g., `lf`-family, `gf`-family, and `d`-family) which vary in terms of what kind of models they can be fit.
- In particular: we will focus on models where the log-likelihood function does not meet the linear-form restrictions, which can be fitted using the `d`-family of evaluators.
- The minimum requirement to implement a model using the `ml` command is to write its log-likelihood function (i.e., `d0` evaluator).

2 The `ml` command

- The `ml` command allows us to fit models using Maximum Likelihood.
- The command has different types of evaluators (e.g., `lf`-family, `gf`-family, and `d`-family) which vary in terms of what kind of models they can be fit.
- In particular: we will focus on models where the log-likelihood function does not meet the linear-form restrictions, which can be fitted using the `d`-family of evaluators.
- The minimum requirement to implement a model using the `ml` command is to write its log-likelihood function (i.e., `d0` evaluator).
- Faster methods can be implemented depending on what we provide the `ml` command with:
  - `d0` evaluator = **Log-likelihood**
  - `d1` evaluator = Log-likelihood + Gradient
  - `d2` evaluator = Log-likelihood + Gradient + Hessian

3 Outline

3 Linear-form Restriction? [1]

- We say that a likelihood function meets the linear-form restrictions when:

3 Linear-form Restriction? [1]

- We say that a likelihood function meets the linear-form restrictions when:
  - The log-likelihood contribution can be calculated separately for each observation.

3 Linear-form Restriction? [1]

- We say that a likelihood function meets the linear-form restrictions when:
  - The log-likelihood contribution can be calculated separately for each observation.
  - The sum of the individual contributions equals the overall log-likelihood.

3 Linear-form Restriction? [1]

- We say that a likelihood function meets the linear-form restrictions when:
  - The log-likelihood contribution can be calculated separately for each observation.
  - The sum of the individual contributions equals the overall log-likelihood.
- Take, for example, the normal linear regression model:

KU LEUVEN

3 Linear-form Restriction? [1]

- We say that a likelihood function meets the linear-form restrictions when:
  - The log-likelihood contribution can be calculated separately for each observation.
  - The sum of the individual contributions equals the overall log-likelihood.
- Take, for example, the normal linear regression model:

$$\ln L = \sum_{i=1}^{N} \left[\ln\left\{\phi\left(y_i - \boldsymbol{x}_i\beta\right)/\sigma\right\} - \ln\sigma\right]$$

- This model does meet the Linear-form Restriction!

3 Linear-form Restriction? [1]

- We say that a likelihood function meets the linear-form restrictions when:
  - The log-likelihood contribution can be calculated separately for each observation.
  - The sum of the individual contributions equals the overall log-likelihood.
- Take, for example, the normal linear regression model:

$$\ln L = \sum_{i=1}^{N} \left[\ln \left\{ \phi \left(y_i - \boldsymbol{x}_i \beta \right) / \sigma \right\} - \ln \sigma \right]$$

- This model does meet the Linear-form Restriction!

```
. list in 1/3, sep(1)
```

	y	x1	x2
1.	-1.09811	-.3591099	.3387246
2.	-1.742268	.1902105	-1.498368
3.	1.273768	-1.602709	1.034604

3 Linear-form Restriction? [2]

- On the other hand, a conditional logistic regression (see [R] **clogit**) does **NOT** meet the Linear-form Restriction!

3 Linear-form Restriction? [2]

- On the other hand, a conditional logistic regression (see [R] **clogit**) does **NOT** meet the Linear-form Restriction!

$$\ln L = \sum_{n=1}^{N} \sum_{i=1}^{J} y_{in} \ln\left(P_{in}\right) = \sum_{n=1}^{N} \sum_{i=1}^{J} y_{in} \ln\left(\frac{\exp\left(\boldsymbol{\beta}' x_{in}\right)}{\sum_{j=1}^{J} \exp\left(\boldsymbol{\beta}' x_{in}\right)}\right)$$

3 Linear-form Restriction? [2]

- On the other hand, a conditional logistic regression (see [R] **clogit**) does **NOT** meet the Linear-form Restriction!

$$\ln L = \sum_{n=1}^{N} \sum_{i=1}^{J} y_{in} \ln\left(P_{in}\right) = \sum_{n=1}^{N} \sum_{i=1}^{J} y_{in} \ln\left(\frac{\exp\left(\boldsymbol{\beta}' x_{in}\right)}{\sum_{j=1}^{J} \exp\left(\boldsymbol{\beta}' x_{in}\right)}\right)$$

- Where:
  - y_{in} response variable: 1 if the alternative i is selected and 0 otherwise.

3 Linear-form Restriction? [2]

- On the other hand, a conditional logistic regression (see [R] **clogit**) does **NOT** meet the Linear-form Restriction!

$$\ln L = \sum_{n=1}^{N}\sum_{i=1}^{J} y_{in} \ln\left(P_{in}\right) = \sum_{n=1}^{N}\sum_{i=1}^{J} y_{in} \ln\left(\frac{\exp\left(\boldsymbol{\beta}' x_{in}\right)}{\sum_{j=1}^{J}\exp\left(\boldsymbol{\beta}' x_{in}\right)}\right)$$

- Where:
  - y_{in} response variable: 1 if the alternative i is selected and 0 otherwise.
  - x_{in} is the attribute level of alternative i for individual n.

3 Linear-form Restriction? [2]

- On the other hand, a conditional logistic regression (see [R] **clogit**) does **NOT** meet the Linear-form Restriction!

$$\ln L = \sum_{n=1}^{N} \sum_{i=1}^{J} y_{in} \ln\left(P_{in}\right) = \sum_{n=1}^{N} \sum_{i=1}^{J} y_{in} \ln\left(\frac{\exp\left(\boldsymbol{\beta}' x_{in}\right)}{\sum_{j=1}^{J} \exp\left(\boldsymbol{\beta}' x_{in}\right)}\right)$$

- Where:
  - y_{in} response variable: 1 if the alternative i is selected and 0 otherwise.
  - x_{in} is the attribute level of alternative i for individual n.
  - $\boldsymbol{\beta}$ is the vector of alternative-specific regression coefficients.

3 Linear-form Restriction? [2]

- On the other hand, a conditional logistic regression (see [R] **clogit**) does **NOT** meet the Linear-form Restriction!

$$\ln L = \sum_{n=1}^{N} \sum_{i=1}^{J} y_{in} \ln \left(P_{in} \right) = \sum_{n=1}^{N} \sum_{i=1}^{J} y_{in} \ln \left(\frac{\exp \left(\boldsymbol{\beta}' x_{in} \right)}{\sum_{j=1}^{J} \exp \left(\boldsymbol{\beta}' x_{in} \right)} \right)$$

- Where:
  - y_{in} response variable: 1 if the alternative i is selected and 0 otherwise.
  - x_{in} is the attribute level of alternative i for individual n.
  - $\boldsymbol{\beta}$ is the vector of alternative-specific regression coefficients.

```
. list in 1/6 , sep(3)
```

	id	altern~e	x1	x2	choice
1.	1	1	-1.666827	-1.969941	0
2.	1	2	.5580259	-.2189879	0
3.	1	3	1.054737	1.894969	1
4.	2	1	-1.913301	-.1506114	0
5.	2	2	-.1818884	-.2132395	1
6.	2	3	1.19467	-.6775483	0

3 Linear-form Restriction? [3]

- Other examples of models that do not meet said restriction are:

3 Linear-form Restriction? [3]

- Other examples of models that do not meet said restriction are:
  - The Cox regression (see [R] **stcox**)

3 Linear-form Restriction? [3]

- Other examples of models that do not meet said restriction are:
  - The Cox regression (see [R] **stcox**)
  - Panel Data (see [XT] **xtreg**)

3 Linear-form Restriction? [3]

- Other examples of models that do not meet said restriction are:
  - The Cox regression (see [R] **stcox**)
  - Panel Data (see [XT] **xtreg**)
  - Conditional Logistic regression (see [R] **clogit**)

3 Linear-form Restriction? [3]

- Other examples of models that do not meet said restriction are:
  - The Cox regression (see [R] **stcox**)
  - Panel Data (see [XT] **xtreg**)
  - Conditional Logistic regression (see [R] **clogit**)
- In other words, if the model uses data in `long` format, it probably does not meet the restriction.

4 Outline

4 The Problem [1]

- To illustrate the problem, say we write our own conditional logistic regression (`MyClogit`) using the `ml` command. (Program available on slide 32).

4 The Problem [1]

- To illustrate the problem, say we write our own conditional logistic regression (`MyClogit`) using the `ml` command. (Program available on slide 32).

```
. qui clogit choice x1 x2 , gr(id) nolog
. matrix b_clogit = e(b)
. MyClogit choice x1 x2 , gr(id) nolog
MyClogit                                        Number of obs     =        300
                                                Wald chi2(2)      =      38.72
Log likelihood =  -53.10466                     Prob > chi2       =     0.0000

------------------------------------------------------------------------------
      choice |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
          x1 |   .5233348   .1771384     2.95   0.003     .1761499    .8705197
          x2 |   1.922775   .3146272     6.11   0.000     1.306117    2.539433
------------------------------------------------------------------------------

. matrix b_MyClogit = e(b)
. di mreldif(b_MyClogit, b_clogit)
2.308e-08
```

  - We also check that the estimates from our program are numerically equivalent to Stata's `clogit` command.

4 The Problem [2]

- So far... So good, right?

4 The Problem [2]

- So far... So good, right?
- We managed to replicate Stata's `clogit` command results.

4 The Problem [2]

- So far... So good, right?
- We managed to replicate Stata's `clogit` command results.
- Now, say, we would like to compute **robust standard errors**.

4 The Problem [2]

- So far... So good, right?
- We managed to replicate Stata's `clogit` command results.
- Now, say, we would like to compute **robust standard errors**.
- As usually, we would type **r**obust.

4 The Problem [2]

- So far... So good, right?
- We managed to replicate Stata's `clogit` command results.
- Now, say, we would like to compute **robust standard errors**.
- As usually, we would type **r**obust.
- **However**...

4 The Problem [2]

- So far... So good, right?
- We managed to replicate Stata's `clogit` command results.
- Now, say, we would like to compute **robust standard errors**.
- As usually, we would type robust.
- **However**...

```
. MyClogit choice x1 x2 , gr(id) nolog robust
option vce(robust) is not allowed with evaltype d0
r(198);
```

4 The Problem [2]

- So far... So good, right?
- We managed to replicate Stata's `clogit` command results.
- Now, say, we would like to compute **robust standard errors**.
- As usually, we would type robust.
- **However**...

```
. MyClogit choice x1 x2 , gr(id) nolog robust
option vce(robust) is not allowed with evaltype d0
r(198);
```

- **Hence, we are in ⚠ trouble ⚠ !**

5 Outline

KU LEUVEN

5 Robust Variance Covariance Matrix: A very brief review

- We can write every maximum likelihood estimator as

$$G(\boldsymbol{\beta}) = \sum_{n=1}^{N} \boldsymbol{S}(\boldsymbol{\beta}; y_n, \boldsymbol{x}_n) = \mathbf{0} \text{ where } \underbrace{\boldsymbol{S}(\boldsymbol{\beta}; y_n, \boldsymbol{x}_n)}_{\text{Score functions}} = \partial \ln L_n / \partial \boldsymbol{\beta}$$

5 Robust Variance Covariance Matrix: A very brief review

- We can write every maximum likelihood estimator as

$$G(\boldsymbol{\beta}) = \sum_{n=1}^{N} \boldsymbol{S}(\boldsymbol{\beta}; y_n, \boldsymbol{x}_n) = \mathbf{0} \text{ where } \underbrace{\boldsymbol{S}(\boldsymbol{\beta}; y_n, \boldsymbol{x}_n)}_{\text{Score functions}} = \partial \ln L_n / \partial \boldsymbol{\beta}$$

- Then, we can compute the robust variance-estimator of β as:

5 Robust Variance Covariance Matrix: A very brief review

- We can write every maximum likelihood estimator as

$$G(\boldsymbol{\beta}) = \sum_{n=1}^{N} \boldsymbol{S}(\boldsymbol{\beta}; y_n, \boldsymbol{x}_n) = \mathbf{0} \text{ where } \underbrace{\boldsymbol{S}(\boldsymbol{\beta}; y_n, \boldsymbol{x}_n)}_{\text{Score functions}} = \partial \ln L_n / \partial \boldsymbol{\beta}$$

- Then, we can compute the robust variance-estimator of β as:

$$\widehat{V}(\widehat{\boldsymbol{\beta}}) = \boldsymbol{W} \left(\frac{N}{N-1} \sum_{n=1}^{N} \boldsymbol{u}_n' \boldsymbol{u}_n \right) \boldsymbol{W} \tag{1}$$

5 Robust Variance Covariance Matrix: A very brief review

- We can write every maximum likelihood estimator as

$$G(\boldsymbol{\beta}) = \sum_{n=1}^{N} \boldsymbol{S}(\boldsymbol{\beta}; y_n, \boldsymbol{x}_n) = \boldsymbol{0} \text{ where } \underbrace{\boldsymbol{S}(\boldsymbol{\beta}; y_n, \boldsymbol{x}_n)}_{\text{Score functions}} = \partial \ln L_n / \partial \boldsymbol{\beta}$$

- Then, we can compute the robust variance-estimator of β as:

$$\widehat{V}(\widehat{\boldsymbol{\beta}}) = \boldsymbol{W} \left(\frac{N}{N-1} \sum_{n=1}^{N} \boldsymbol{u}_n' \boldsymbol{u}_n \right) \boldsymbol{W} \qquad (1)$$

- $\boldsymbol{W} = -H^{-1}$ is the negative of the inverse of the hessian.

5 Robust Variance Covariance Matrix: A very brief review

- We can write every maximum likelihood estimator as

$$G(\boldsymbol{\beta}) = \sum_{n=1}^{N} \boldsymbol{S}(\boldsymbol{\beta}; y_n, \boldsymbol{x}_n) = \boldsymbol{0} \quad \text{where} \quad \underbrace{\boldsymbol{S}(\boldsymbol{\beta}; y_n, \boldsymbol{x}_n)}_{\text{Score functions}} = \partial \ln L_n / \partial \boldsymbol{\beta}$$

- Then, we can compute the robust variance-estimator of β as:

$$\widehat{V}(\widehat{\boldsymbol{\beta}}) = \boldsymbol{W} \left(\frac{N}{N-1} \sum_{n=1}^{N} \boldsymbol{u}_n' \boldsymbol{u}_n \right) \boldsymbol{W} \tag{1}$$

- $\boldsymbol{W} = -H^{-1}$ is the negative of the inverse of the hessian.
  - We already have this *"for free"*: (e.g., `e(V)` matrix).

5 Robust Variance Covariance Matrix: A very brief review

- We can write every maximum likelihood estimator as

$$G(\boldsymbol{\beta}) = \sum_{n=1}^{N} \boldsymbol{S}(\boldsymbol{\beta}; y_n, \boldsymbol{x}_n) = \mathbf{0} \text{ where } \underbrace{\boldsymbol{S}(\boldsymbol{\beta}; y_n, \boldsymbol{x}_n)}_{\text{Score functions}} = \partial \ln L_n / \partial \boldsymbol{\beta}$$

- Then, we can compute the robust variance-estimator of $\boldsymbol{\beta}$ as:

$$\widehat{V}(\widehat{\boldsymbol{\beta}}) = \boldsymbol{W} \left(\frac{N}{N-1} \sum_{n=1}^{N} \boldsymbol{u}_n' \boldsymbol{u}_n \right) \boldsymbol{W} \tag{1}$$

- $\boldsymbol{W} = -H^{-1}$ is the negative of the inverse of the hessian.
  - We already have this *"for free"*: (e.g., `e(V)` matrix).
- $\boldsymbol{u}_n = \boldsymbol{S}(\widehat{\boldsymbol{\beta}}; y_n, \boldsymbol{x}_n)$ are row vectors that contains the score functions evaluated at $\widehat{\boldsymbol{\beta}}$.

5 Robust Variance Covariance Matrix: A very brief review

- We can write every maximum likelihood estimator as

$$G(\boldsymbol{\beta}) = \sum_{n=1}^{N} \boldsymbol{S}(\boldsymbol{\beta}; y_n, \boldsymbol{x}_n) = \mathbf{0} \text{ where } \underbrace{\boldsymbol{S}(\boldsymbol{\beta}; y_n, \boldsymbol{x}_n)}_{\text{Score functions}} = \partial \ln L_n / \partial \boldsymbol{\beta}$$

- Then, we can compute the robust variance-estimator of $\boldsymbol{\beta}$ as:

$$\widehat{V}(\widehat{\boldsymbol{\beta}}) = \boldsymbol{W} \left(\frac{N}{N-1} \sum_{n=1}^{N} \boldsymbol{u}_n' \boldsymbol{u}_n \right) \boldsymbol{W} \tag{1}$$

- $\boldsymbol{W} = -H^{-1}$ is the negative of the inverse of the hessian.
  - We already have this *"for free"*: (e.g., `e(V)` matrix).
- $\boldsymbol{u}_n = \boldsymbol{S}(\widehat{\boldsymbol{\beta}}; y_n, \boldsymbol{x}_n)$ are row vectors that contains the score functions evaluated at $\widehat{\boldsymbol{\beta}}$.
- Hence, $\boldsymbol{u}_n$ is the only object that is missing in order to compute $\widehat{V}(\widehat{\boldsymbol{\beta}})$.

6 Outline

6 The Solution [1]: Two possible ways to proceed

1. **One possible solution**: Write a **separate program** that computes the score functions **analytically**. This involve two additional steps.

6 The Solution [1]: Two possible ways to proceed

1. **One possible solution**: Write a **separate program** that computes the score functions **analytically**. This involve two additional steps.
   - First (and the most obvious one), the developer needs to derive the score functions by hand (using pencil and paper + calculus).

6 The Solution [1]: Two possible ways to proceed

1. **One possible solution**: Write a **separate program** that computes the score functions **analytically**. This involve two additional steps.

    - First (and the most obvious one), the developer needs to derive the score functions by hand (using pencil and paper + calculus).
    - Second, after knowing the algebraic expression, it has to be coded on Stata or Mata.

6 The Solution [1]: Two possible ways to proceed

1. **One possible solution**: Write a **separate program** that computes the score functions **analytically**. This involve two additional steps.
   - First (and the most obvious one), the developer needs to derive the score functions by hand (using pencil and paper + calculus).
   - Second, after knowing the algebraic expression, it has to be coded on Stata or Mata.
2. **Another possible solution**: **Numerically** approximate the score functions, using what we already have coded: **the log-likelihood function**.

6 The Solution [1]: Two possible ways to proceed

1. **One possible solution**: Write a **separate program** that computes the score functions **analytically**. This involve two additional steps.
   - First (and the most obvious one), the developer needs to derive the score functions by hand (using pencil and paper + calculus).
   - Second, after knowing the algebraic expression, it has to be coded on Stata or Mata.
2. **Another possible solution**: **Numerically** approximate the score functions, using what we already have coded: **the log-likelihood function**.

- ⚠**SPOILER ALERT**⚠:

6 The Solution [1]: Two possible ways to proceed

1. **One possible solution**: Write a **separate program** that computes the score functions **analytically**. This involve two additional steps.
   - First (and the most obvious one), the developer needs to derive the score functions by hand (using pencil and paper + calculus).
   - Second, after knowing the algebraic expression, it has to be coded on Stata or Mata.
2. **Another possible solution**: **Numerically** approximate the score functions, using what we already have coded: **the log-likelihood function**.

- ⚠**SPOILER ALERT**⚠:
  - Our solution will consist in:

6 The Solution [1]: Two possible ways to proceed

1. **One possible solution**: Write a **separate program** that computes the score functions **analytically**. This involve two additional steps.
   - First (and the most obvious one), the developer needs to derive the score functions by hand (using pencil and paper + calculus).
   - Second, after knowing the algebraic expression, it has to be coded on Stata or Mata.
2. **Another possible solution**: **Numerically** approximate the score functions, using what we already have coded: **the log-likelihood function**.

- ⚠**SPOILER ALERT**⚠:
  - Our solution will consist in:
    1. (Numerically) approximate the vector $\boldsymbol{u}_n$.

6 The Solution [1]: Two possible ways to proceed

1. **One possible solution**: Write a **separate program** that computes the score functions **analytically**. This involve two additional steps.
   - First (and the most obvious one), the developer needs to derive the score functions by hand (using pencil and paper + calculus).
   - Second, after knowing the algebraic expression, it has to be coded on Stata or Mata.
2. **Another possible solution**: **Numerically** approximate the score functions, using what we already have coded: **the log-likelihood function**.

- ⚠**SPOILER ALERT**⚠:
  - Our solution will consist in:
    1. (Numerically) approximate the vector $\boldsymbol{u}_n$.
    2. Compute $\widehat{V}(\widehat{\boldsymbol{\beta}})$ using it.

6 The Solution [2]: Collecting everything we need

- First, we provide Mata with everything we need to compute the log-likelihood contribution of each individual.

```
. // We create relevant matrices on Stata to push them to Mata afterwards.
. matrix b = e(b)                // Maximum Likelihood estimates
. matrix W = e(V)                // Non-robust variance-covariance matrix
. // We initialize Mata
. mata:
------------------------------------------------- mata (type end to exit) ----------------
: // Invoking Stata matrices
: betas = st_matrix("b")          // Calls from Stata the matrix "b"
: W     = st_matrix("W")          // Calls from Stata the matrix "W"
:
: // Invoking Stata Variables
: st_view(X = ., ., "x1 x2")      // View of all regressors x1 and x2
: st_view(Y = ., ., "choice")     // View of response variable "choice"
: XY = (Y,X)                      // Generates XY matrix for future usage.
:
: // Extracting information about the id of individuals.
: st_view(panvar = ., ., "id")    // View of individuals id
: paninfo = panelsetup(panvar, 1) // Sets up panel processing
: N = panelstats(paninfo)[1]      // Number of Individuals
: end
------------------------------------------------------------------------------------------
```

6 The Solution [3]: Writing our log-likelihood function

- Second, we will create a void function, `LL_d()`, that resembles our log-likelihood function.

6 The Solution [3]: Writing our log-likelihood function

- Second, we will create a void function, `LL_d()`, that resembles our log-likelihood function.
- We will invoke it later when using Mata's `deriv()` function.

6 The Solution [3]: Writing our log-likelihood function

- Second, we will create a void function, `LL_d()`, that resembles our log-likelihood function.
- We will invoke it later when using Mata's `deriv()` function.

```
. mata:
------------------------------------------------- mata (type end to exit) -----------------------
: // Creating the function we will invoke using Mata´s deriv().
: void LL_d(real rowvector b ,   // 1ST ARGUMENT: Maximum likelihood estimates
>           real matrix    XY ,  // 2ND ARGUMENT: Convariates + dependent variable
>           real scalar lnf)     // Output:       Log-likelihood contribution
> {
>   Y = XY[.,1]                 // Extract variable Y
>   X = XY[., (2::cols(XY))]    // Extract the regressors (x1 and x2)
>   U = rowsum(b:*X)            // Observed Utility
>   P = exp(U):/colsum(exp(U )) // Multinomial Probability
>   lnf = colsum(Y:*ln(P))      // Individual contribution to the log-likelihood
> }
: end
-------------------------------------------------------------------------------------------------
```

6 The Solution [3]: Writing our log-likelihood function

- Second, we will create a void function, `LL_d()`, that resembles our log-likelihood function.
- We will invoke it later when using Mata's `deriv()` function.

```
. mata:
------------------------------------------------- mata (type end to exit) ----------------------
: // Creating the function we will invoke using Mata´s deriv().
: void LL_d(real rowvector b ,   // 1ST ARGUMENT: Maximum likelihood estimates
>           real matrix    XY ,  // 2ND ARGUMENT: Convariates + dependent variable
>           real scalar lnf)     // Output:       Log-likelihood contribution
> {
>   Y = XY[.,1]                 // Extract variable Y
>   X = XY[., (2::cols(XY))]    // Extract the regressors (x1 and x2)
>   U = rowsum(b:*X)            // Observed Utility
>   P = exp(U):/colsum(exp(U )) // Multinomial Probability
>   lnf = colsum(Y:*ln(P))      // Individual contribution to the log-likelihood
> }
: end
------------------------------------------------------------------------------------------------
```

- As you can see, this resembles exactly our log-likelihood.

$$\ln L = \sum_{n=1}^{N} \sum_{i=1}^{J} y_{in} \ln \left(P_{in} \right) = \sum_{n=1}^{N} \sum_{i=1}^{J} y_{in} \ln \left(\frac{\exp\left(\beta' x_{in}\right)}{\sum_{j=1}^{J} \exp\left(\beta' x_{in}\right)} \right)$$

6 The Solution [4]: Score function of the first individual

- Third, to begin with, we will illustrate how to compute the score function of the first individual using `deriv()`:

```
. mata:
------------------------------------------------- mata (type end to exit) ----------------------
: D =deriv_init()                  // Init deriv() ´struct´ and call it "D"
: deriv_init_evaluator(D, &LL_d()) // We provide the ´struct´ D with function LL_d()
: deriv_init_evaluatortype(D,"d")  // Set that deriv() must returns a scalar
: deriv_init_params(D, betas)      // Provide D with beta estimates (deriv at)
: xy_n = panelsubmatrix(XY, 1, paninfo) // Extract first individual´s X and Y
: xy_n
                 1              2              3
    +----------------------------------------------+
  1 |            0   -1.666826963   -1.969941497  |
  2 |            0    .5580258965    -.218987897  |
  3 |            1    1.054736972    1.894969106  |
    +----------------------------------------------+

: deriv_init_argument(D, 1, xy_n)  // provide D with X and Y of the first individual
: score_fn= deriv(D, 1)            // <--- Perform the numerical derivation!
: score_fn                         // Display it
                 1              2
    +-------------------------------+
  1 |  .006871893   .0281603095  |
    +-------------------------------+

: end
------------------------------------------------------------------------------------------------
```

6 The Solution [4]: Score function of the first individual

- Third, to begin with, we will illustrate how to compute the score function of the first individual using `deriv()`:

```
. mata:
------------------------------------------------ mata (type end to exit) --------------------
: D = deriv_init()                 // Init deriv() ´struct´ and call it "D"
: deriv_init_evaluator(D, &LL_d()) // We provide the ´struct´ D with function LL_d()
: deriv_init_evaluatortype(D,"d")  // Set that deriv() must returns a scalar
: deriv_init_params(D, betas)      // Provide D with beta estimates (deriv at)
: xy_n = panelsubmatrix(XY, 1, paninfo) // Extract first individual´s X and Y
: xy_n
                 1              2              3
    +----------------------------------------------+
  1 |            0   -1.666826963   -1.969941497   |
  2 |            0    .5580258965    -.218987897   |
  3 |            1    1.054736972    1.894969106   |
    +----------------------------------------------+

: deriv_init_argument(D, 1, xy_n)  // provide D with X and Y of the first individual
: score_fn= deriv(D, 1)            // <--- Perform the numerical derivation!
: score_fn                         // Display it
                 1              2
    +-----------------------------+
  1 |   .006871893   .0281603095  |
    +-----------------------------+

: end
---------------------------------------------------------------------------------------------
```

6 The Solution [4]: Score function of the first individual

- Third, to begin with, we will illustrate how to compute the score function of the first individual using `deriv()`:

```
. mata:
------------------------------------------------- mata (type end to exit) ------------------
: D = deriv_init()                   // Init deriv() ´struct´ and call it "D"
: deriv_init_evaluator(D, &LL_d())  // We provide the ´struct´ D with function LL_d()
: deriv_init_evaluatortype(D,"d")   // Set that deriv() must returns a scalar
: deriv_init_params(D, betas)       // Provide D with beta estimates (deriv at)
: xy_n = panelsubmatrix(XY, 1, paninfo) // Extract first individual´s X and Y
: xy_n
                  1              2              3
    +----------------------------------------------+
  1 |             0   -1.666826963   -1.969941497  |
  2 |             0    .5580258965    -.218987897  |
  3 |             1    1.054736972    1.894969106  |
    +----------------------------------------------+

: deriv_init_argument(D, 1, xy_n)   // provide D with X and Y of the first individual
: score_fn= deriv(D, 1)             // <--- Perform the numerical derivation!
: score_fn                          // Display it
                 1              2
    +-------------------------------+
  1 |   .006871893    .0281603095   |
    +-------------------------------+

: end
---------------------------------------------------------------------------------------------
```

6 The Solution [4]: Score function of the first individual

- Third, to begin with, we will illustrate how to compute the score function of the first individual using `deriv()`:

```
. mata:
------------------------------------------------- mata (type end to exit) -------------------------
: D = deriv_init()                   // Init deriv() ´struct´ and call it "D"
: deriv_init_evaluator(D, &LL_d()) // We provide the ´struct´ D with function LL_d()
: deriv_init_evaluatortype(D,"d")  // Set that deriv() must returns a scalar
: deriv_init_params(D, betas)      // Provide D with beta estimates (deriv at)
: xy_n = panelsubmatrix(XY, 1, paninfo) // Extract first individual´s X and Y
: xy_n
                  1              2              3
    +----------------------------------------------+
  1 |             0   -1.666826963   -1.969941497  |
  2 |             0    .5580258965    -.218987897  |
  3 |             1    1.054736972    1.894969106  |
    +----------------------------------------------+

: deriv_init_argument(D, 1, xy_n)  // provide D with X and Y of the first individual
: score_fn= deriv(D, 1)            // <--- Perform the numerical derivation!
: score_fn                         // Display it
                 1             2
    +-----------------------------+
  1 |   .006871893   .0281603095  |
    +-----------------------------+

: end
---------------------------------------------------------------------------------------------------
```

6 The Solution [4]: Score function of the first individual

- Third, to begin with, we will illustrate how to compute the score function of the first individual using `deriv()`:

```
. mata:
------------------------------------------------- mata (type end to exit) ----------------------
: D = deriv_init()                   // Init deriv() ´struct´ and call it "D"
: deriv_init_evaluator(D, &LL_d())  // We provide the ´struct´ D with function LL_d()
: deriv_init_evaluatortype(D,"d")   // Set that deriv() must returns a scalar
: deriv_init_params(D, betas)       // Provide D with beta estimates (deriv at)
: xy_n = panelsubmatrix(XY, 1, paninfo) // Extract first individual´s X and Y
: xy_n
                 1              2              3
    +----------------------------------------------+
  1 |            0   -1.666826963   -1.969941497  |
  2 |            0    .5580258965    -.218987897  |
  3 |            1    1.054736972    1.894969106  |
    +----------------------------------------------+

: deriv_init_argument(D, 1, xy_n)  // provide D with X and Y of the first individual
: score_fn= deriv(D, 1)             // <--- Perform the numerical derivation!
: score_fn                          // Display it
                1              2
    +-------------------------------+
  1 |  .006871893   .0281603095    |
    +-------------------------------+

: end
------------------------------------------------------------------------------------------------
```

6 The Solution [4]: Score function of the first individual

- Third, to begin with, we will illustrate how to compute the score function of the first individual using `deriv()`:

```
. mata:
------------------------------------------------- mata (type end to exit) -----------------------
: D = deriv_init()                  // Init deriv() ´struct´ and call it "D"
: deriv_init_evaluator(D, &LL_d()) // We provide the ´struct´ D with function LL_d()
: deriv_init_evaluatortype(D,"d")  // Set that deriv() must returns a scalar
: deriv_init_params(D, betas)      // Provide D with beta estimates (deriv at)
: xy_n = panelsubmatrix(XY, 1, paninfo) // Extract first individual´s X and Y
: xy_n
                 1              2              3
    +----------------------------------------------+
  1 |            0   -1.666826963   -1.969941497  |
  2 |            0    .5580258965    -.218987897  |
  3 |            1    1.054736972    1.894969106  |
    +----------------------------------------------+

: deriv_init_argument(D, 1, xy_n)  // provide D with X and Y of the first individual
: score_fn= deriv(D, 1) // <--- Perform the numerical derivation!
: score_fn                          // Display it
                 1              2
    +-------------------------------+
  1 |  .006871893    .0281603095  |
    +-------------------------------+

: end
-------------------------------------------------------------------------------------------------
```

6 The Solution [5]: Score functions of the entire sample

- Now that we know how to perform the derivative of a function we can apply it to the whole sample (e.g., to all the individuals in the sample):

```
. mata:
------------------------------------------------- mata (type end to exit) -----------------------------
: D = deriv_init()                    // Init deriv() ´struct´
: deriv_init_evaluator(D, &LL_d()) // ´struct´ D is prodived with the pointer LL_d()
: deriv_init_evaluatortype(D,"d")  // Set that deriv() must returns a scalar
: score_fn = J(0, cols(betas),.)   // Vector length 0xcols(betas)
: for(n=1; n <= N; ++n) {           // Looping over n individuals
>           xy_n = panelsubmatrix(XY, n, paninfo)  // Extract submatrix of individual n
>           deriv_init_params(D, betas)            // provide D with beta estimates
>           deriv_init_argument(D, 1, xy_n)        // provide D with attributes values
>           score_fn = score_fn \  deriv(D, 1)     // Collect score functions from each individual
>
> }
: score_fn[1..4,]  // display the score functions of the first 4 individuals
                  1              2
    +-------------------------------+
  1 |    .006871893    .0281603095  |
  2 |  -.1607972318    .1576732297  |
  3 |  -.0730075944    .1282049291  |
  4 |   .0035216089    .0050014822  |
    +-------------------------------+

: // Finally, we save the score functions  as S just for a handy matrix multiplication afterwards.
: S = score_fn
: end
--------------------------------------------------------------------------------------------------------
```

6 The Solution [6]: Obtaining the robust correction

- All we have to do now is just perform the matrix multiplication described below to find the robust variance-covariance matrix.

6 The Solution [6]: Obtaining the robust correction

- All we have to do now is just perform the matrix multiplication described below to find the robust variance-covariance matrix.

$$\widehat{V}(\widehat{\boldsymbol{\beta}}) = \boldsymbol{W}\left(\frac{N}{N-1}\sum_{n=1}^{N}\boldsymbol{u}_n'\boldsymbol{u}_n\right)\boldsymbol{W} \tag{2}$$

6 The Solution [6]: Obtaining the robust correction

- All we have to do now is just perform the matrix multiplication described below to find the robust variance-covariance matrix.

$$\widehat{V}(\widehat{\boldsymbol{\beta}}) = \boldsymbol{W} \left(\frac{N}{N-1} \sum_{n=1}^{N} \boldsymbol{u}_n' \boldsymbol{u}_n \right) \boldsymbol{W} \tag{2}$$

- $\boldsymbol{W} = -H^{-1}$ is the negative of the inverse of the hessian (Object `W`).

6 The Solution [6]: Obtaining the robust correction

- All we have to do now is just perform the matrix multiplication described below to find the robust variance-covariance matrix.

$$\widehat{V}(\widehat{\boldsymbol{\beta}}) = \boldsymbol{W}\left(\frac{N}{N-1}\sum_{n=1}^{N}\boldsymbol{u}_n'\boldsymbol{u}_n\right)\boldsymbol{W} \tag{2}$$

- $\boldsymbol{W} = -H^{-1}$ is the negative of the inverse of the hessian (Object `W`).
- $\boldsymbol{u}_n = \boldsymbol{S}(\widehat{\boldsymbol{\beta}}; y_n, \boldsymbol{x}_n)$ are row vectors that contains the score functions evaluated at $\widehat{\beta}$ (Object `S`).

6 The Solution [6]: Obtaining the robust correction

- All we have to do now is just perform the matrix multiplication described below to find the robust variance-covariance matrix.

$$\widehat{V}(\widehat{\boldsymbol{\beta}}) = \boldsymbol{W}\left(\frac{N}{N-1}\sum_{n=1}^{N}\boldsymbol{u}_n'\boldsymbol{u}_n\right)\boldsymbol{W} \tag{2}$$

- $\boldsymbol{W} = -H^{-1}$ is the negative of the inverse of the hessian (Object `W`).

- $\boldsymbol{u}_n = \boldsymbol{S}(\widehat{\boldsymbol{\beta}}; y_n, \boldsymbol{x}_n)$ are row vectors that contains the score functions evaluated at $\widehat{\beta}$ (Object `S`).

- Accordingly, it is as simple as:

6 The Solution [6]: Obtaining the robust correction

- All we have to do now is just perform the matrix multiplication described below to find the robust variance-covariance matrix.

$$\widehat{V}(\widehat{\boldsymbol{\beta}}) = \boldsymbol{W} \left(\frac{N}{N-1} \sum_{n=1}^{N} \boldsymbol{u}_n' \boldsymbol{u}_n \right) \boldsymbol{W} \tag{2}$$

- $\boldsymbol{W} = -H^{-1}$ is the negative of the inverse of the hessian (Object `W`).
- $\boldsymbol{u}_n = \boldsymbol{S}(\widehat{\boldsymbol{\beta}}; y_n, \boldsymbol{x}_n)$ are row vectors that contains the score functions evaluated at $\widehat{\boldsymbol{\beta}}$ (Object `S`).
- Accordingly, it is as simple as:

```
. mata:
------------------------------------------------- mata (type end to exit) ---------------------------
: meat =  (N/(N-1)) * S´ * S                     // Some people call this part the "meat".
: V_robust_approx= W * meat * W                  // Approximated robust variance-covariance matrix.
: st_matrix("V_robust_approx", V_robust_approx) // Save robust matrix into a Stata Matrix.
: end
------------------------------------------------------------------------------------------------------
```

6 The Solution [7]: Checking our approximation

- Using `V_robust_approx` we can check how far are our numerically approximated robust covariance matrices compared with Stata's `clogit`.

```
. clogit choice x* ,gr(id) robust nolog
Conditional (fixed-effects) logistic regression
                                                Number of obs     =        300
                                                Wald chi2(2)      =      42.15
                                                Prob > chi2       =     0.0000
Log pseudolikelihood =  -53.10466               Pseudo R2         =     0.5166
                                   (Std. Err. adjusted for clustering on id)
------------------------------------------------------------------------------
             |               Robust
      choice |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
          x1 |   .5233348   .1587735     3.30   0.001     .2121444    .8345252
          x2 |   1.922775   .3334521     5.77   0.000      1.26922    2.576329
------------------------------------------------------------------------------

. mat V_robust_clogit = e(V)
. mat li V_robust_approx
symmetric V_robust_approx[2,2]
           c1          c2
r1  .02520903
r2  .00291664   .11119032
. mat li V_robust_clogit
symmetric V_robust_clogit[2,2]
               choice:     choice:
                   x1          x2
choice:x1   .02520903
choice:x2   .00291664   .11119031
. display mreldif(V_robust_approx, V_robust_clogit)
7.734e-09
```

7 Outline

7 Conclusions

- We have seen a workaround for those ***rare*** cases when the `ml` command fails to produce robust standard errors.

7 Conclusions

- We have seen a workaround for those ***rare*** cases when the `ml` command fails to produce robust standard errors.
- The illustrated solution is not meant to replace the algebraic computation of the score functions, but a complement and a way to check our results.

7 Conclusions

- We have seen a workaround for those ***rare*** cases when the `ml` command fails to produce robust standard errors.
- The illustrated solution is not meant to replace the algebraic computation of the score functions, but a complement and a way to check our results.
- All the source code of this talk is available at this GitHub repository.

7 Conclusions

- We have seen a workaround for those ***rare*** cases when the `ml` command fails to produce robust standard errors.
- The illustrated solution is not meant to replace the algebraic computation of the score functions, but a complement and a way to check our results.
- All the source code of this talk is available at this GitHub repository.
- Bonus track: Further readings about things we used but not I did not necessarily explained in much detail.

7 Conclusions

- We have seen a workaround for those ***rare*** cases when the `ml` command fails to produce robust standard errors.

- The illustrated solution is not meant to replace the algebraic computation of the score functions, but a complement and a way to check our results.

- All the source code of this talk is available at this GitHub repository.

- Bonus track: Further readings about things we used but not I did not necessarily explained in much detail.
  - Gould (2007) for a better understanding about **`-struct-`** type objects.

7 Conclusions

- We have seen a workaround for those ***rare*** cases when the `ml` command fails to produce robust standard errors.
- The illustrated solution is not meant to replace the algebraic computation of the score functions, but a complement and a way to check our results.
- All the source code of this talk is available at this GitHub repository.
- Bonus track: Further readings about things we used but not I did not necessarily explained in much detail.
  - Gould (2007) for a better understanding about **`-struct-`** type objects.
  - Gould (2018) Section 3.4.4 for a detailed explanation about **`-&pointers-`**.

7 Conclusions

- We have seen a workaround for those ***rare*** cases when the `ml` command fails to produce robust standard errors.
- The illustrated solution is not meant to replace the algebraic computation of the score functions, but a complement and a way to check our results.
- All the source code of this talk is available at this GitHub repository.
- Bonus track: Further readings about things we used but not I did not necessarily explained in much detail.
  - Gould (2007) for a better understanding about **`-struct-`** type objects.
  - Gould (2018) Section 3.4.4 for a detailed explanation about **`-&pointers-`**.
  - Gould (2001) for more insights about *Statistical Software Certification*.

8 Outline

1. Introduction
2. The `ml` command
3. Linear-form Restriction
4. The Problem
5. Robust Variance Covariance Matrix: A very brief review
6. The Solution
7. Conclusions

9 Bibliography

Gould, W. (2001). Statistical software certification. The Stata Journal, 1(1):29–50.

Gould, W. (2007). Mata matters: Structures. The Stata Journal, 7(4):556–570.

Gould, W., Pitblado, J., and Poi, B. (2010). Maximum Likelihood Estimation with Stata. StataCorp LP, 4th edition.

Gould, W. W. (2018). The Mata Book: A Book for Serious Programmers and Those who Want to be. Stata Press.

10 Outline

10 MyClogit.ado

```
program MyClogit
    version 12
    if replay() {
    if ("`e(cmd)´" != "MyClogit") error 301
    Replay `0´
    }
    else Estimate `0´
end

program Estimate, eclass sortpreserve
    syntax varlist(fv) [if] [in] , GRoup(varname) ///
        [TECHnique(passthru) noLOg ROBUST ]
    local mlopts  `technique´
    if ("`technique´" == "technique(bhhh)") {
    di in red "technique(bhhh) is not allowed."
    exit 498
    }
    gettoken lhs rhs : varlist
    marksample touse
    markout `touse´ `group´
    global MY_panel = "`group´"
    ml model d0 MyLikelihood_LL()      ///
        (MyClogit: `lhs´ = `rhs´, nocons)  ///
        if `touse´, missing  first   `log´ ///
        title("MyClogit") `robust´ maximize
        // Show model
        ereturn local cmd MyClogit
        Replay , level(`level´)
        ereturn local  cmdline `"`0´"´
end

 program Replay
    syntax [, Level(cilevel) ]
    ml display , level(`level´)
 end

// include mata functions from MyLikelihood_LL.mata
findfile "MyLikelihood_LL.mata"
do "`r(fn)´"
```

11 Outline

9. MyClogit

10. MyLikelihood_LL.mata

11 MyLikelihood_LL.mata

```
mata:
    void MyLikelihood_LL(transmorphic scalar M, real scalar todo,
    real rowvector b, real scalar lnf,
    real rowvector g, real matrix H)
{
  // variables declaration
  real matrix panvar
  real matrix paninfo
  real scalar npanels
  real scalar n
  real matrix Y
  real matrix X
  real matrix x_n
  real matrix y_n
  Y = moptimize_util_depvar(M, 1)               // Response Variable
  X = moptimize_init_eq_indepvars(M,1)          // Attributes
  id_beta_eq=moptimize_util_eq_indices(M,1)     // id parameters
  betas= b[|id_beta_eq|]                        // parameters
  st_view(panvar = ., ., st_global("MY_panel"))
  paninfo = panelsetup(panvar, 1)
  npanels = panelstats(paninfo)[1]
  lnfj = J(npanels, 1, 0)                       // object to store loglikelihood
  for(n=1; n <= npanels; ++n) {
        x_n = panelsubmatrix(X, n, paninfo)
        y_n = panelsubmatrix(Y, n, paninfo)
        U_n =exp(rowsum(betas :* x_n))        // Linear utility
        p_i = colsum(U_n:* y_n) / colsum(U_n) // Probability of each alternative
        lnfj[n] = ln(p_i)                     // Add contribution to the likelihood
}
  lnf = moptimize_util_sum(M, lnfj)
}
end
```