

Hayashi

A Language for Applied Econometrics

Flávio de Vasconcellos Corrêa^{*}

Version 0.2.7-dev

July 3, 2026

Book and documentation licensed under CC BY-SA 4.0

^{*}Master in Applied Economics, Graduate Program in Organizations and Markets (PPGOM), Federal University of Pelotas (UFPel).

Preface

HAYASHI was born out of a concrete frustration: existing tools for applied econometrics force the researcher to choose between expressiveness and performance, between a syntax close to statistical notation and the ability to automate complex analyses.

Stata has the right syntax, but it is proprietary, closed, and not a general-purpose language. R has the right ecosystem, but its syntax is inconsistent and its data model is surprising for those coming from Stata. Python has the right flexibility, but its statistical packages treat econometrics as a special case of machine learning.

HAYASHI is an interpreted, general-purpose language built specifically for econometric analysis. The syntax pays homage to Stata where it matters and adopts modern idioms where Stata shows its age. The engine is written in Rust — no virtual machine, no garbage collector, no perceptible startup time.

This book documents both the language and the econometric core: from the type structure and control flow to time series estimators, causal inference, and advanced methods. It is a snapshot of the current state of the project — a living document that evolves with the software.

A project by one developer — and by the entire community

HAYASHI is, to date, the work of a single developer. This has practical consequences the reader should know about.

On the positive side, there is design coherence: every syntax decision, every estimator interface, every error message was made with the same set of values in mind. There are no committees, no bikeshedding, no compatibility layers accumulated over decades. The code is lean because nobody added anything “just to keep it”.

On the limitation side, one pair of eyes — however careful — does not see everything. The project relies on three layers of verification: unit tests covering the language primitives; integration scripts that exercise HAYASHI as a black box; and DGP (data-generating process) scripts for each estimator, where the true parameter is known and recovery can be verified numerically. Even so, edge cases exist, and some have certainly escaped detection.

License. HAYASHI is free software, distributed under the terms of the **GNU General Public License version 3** (GPL v3). This means you may use, study, modify, and redistribute the program — including for commercial purposes — provided that any derivative work is equally free. The source code is public and belongs, in the most literal sense, to whoever uses it.

The choice of GPL v3 is not accidental. Proprietary econometric tools create institutional dependency: universities and research labs become hostage to licenses, pricing policies, and corporate decisions about what the software can or cannot do. Free software breaks that cycle. HAYASHI should be accessible to any researcher anywhere in the world, regardless of funding.

Contributors and testers are welcome. If you found an unexpected result, an estimator that fails to converge when it should, an obscure error message, or simply a syntax that could be clearer — that is valuable information. Open an issue in the repository. If you want to go further, a well-tested fix or a new documentation chapter is as legitimate a contribution as any line of Rust code.

The project especially needs people willing to test HAYASHI on real data: time series with seasonality, unbalanced panels, models with multicollinearity, data with missing values. Those are the cases that DGP scripts do not cover, because synthetic data is, by construction, well-behaved.

HAYASHI only becomes robust through use. And honest use — which includes reporting what breaks — is the most direct way to contribute.

June 2026.

Contents

Preface	i
I The Language	1
1 Introduction	3
1.1 What is Hayashi	3
1.2 Design philosophy	3
1.3 Hayashi and Stata	4
1.4 Hayashi and R / Python	5
1.5 Empirical validation programme	5
1.6 Structure of this book	5
2 Installation	7
2.1 Requirements	7
2.2 Via Cargo (recommended)	7
2.3 Pre-compiled binaries	7
2.4 Building from source	8
2.5 Verifying the installation	8
3 The REPL	9
3.1 What is the REPL	9
3.2 Exiting the REPL	9
3.3 Navigation and history	9
3.4 Expressions in the REPL	10
3.5 Running scripts	10
3.6 Error format	10
4 Types and Values	11
4.1 Primitive types	11

4.1.1	Int	11
4.1.2	Float	11
4.1.3	Bool	12
4.1.4	Nil	12
4.2	Strings	12
4.3	Lists	13
4.4	Dictionaries	13
4.5	DataFrame	13
4.6	Conversions	13
5	Variables and Scope	15
5.1	Declaration with <code>let</code>	15
5.2	Constants with <code>const</code>	15
5.3	Assignment	16
5.4	Scopes	16
5.5	Special variables	17
6	Operators	19
6.1	Arithmetic	19
6.2	Comparison	19
6.3	Logical	20
6.4	The <code>in</code> operator	20
6.5	The pipe operator <code> ></code>	20
6.6	Precedence	21
7	Control Flow	23
7.1	<code>if / else</code>	23
7.2	<code>match</code>	23
7.3	Expression block	25
8	Loops	27
8.1	<code>for</code>	27
8.2	<code>while</code>	28
8.3	<code>break</code> and <code>continue</code>	28
8.4	Loops with DataFrames	28
9	Functions	31
9.1	Declaration	31
9.2	Functions as values	32

9.3	Closures	32
9.4	Recursion	32
9.5	Optional parameters and defaults	33
9.6	Function scope	33
9.7	<code>const</code> inside functions	33
10	Error Handling	35
10.1	Error model	35
10.2	<code>try</code> / <code>catch</code>	35
10.3	Common errors and their messages	36
10.4	Automatic suggestions	36
10.5	Re-throwing errors	37
11	Strings and F-Strings	39
11.1	String literals	39
11.2	String operations	39
11.3	F-Strings	40
11.4	Multiline strings	41
12	Lists and Dictionaries	43
12.1	Lists	43
12.1.1	Creation	43
12.1.2	Index access	43
12.1.3	Length	43
12.1.4	Mutation	44
12.1.5	Slicing	44
12.1.6	Iteration	44
12.1.7	Functional operations	44
12.2	Ranges	44
12.2.1	<code>chain</code>	45
12.2.2	Composition with closures	45
12.2.3	Comparison with other languages	46
12.2.4	Performance note	46
12.3	Dictionaries	46
12.3.1	Creation	46
12.3.2	Access	46
12.3.3	Insertion and update	47
12.3.4	Checking for a key	47

12.3.5	Dictionaries as function return values	47
12.3.6	Iterating over dictionaries	47
13	DataFrames	49
13.1	What is a DataFrame	49
13.2	Copy semantics (COW)	49
13.3	Creating DataFrames	50
13.4	Saving DataFrames	50
13.5	Inspecting DataFrames	51
13.6	Special variables	51
13.7	Time series operators	52
14	Data Manipulation	53
14.1	<code>generate</code> and <code>mutate</code>	53
14.2	<code>filter</code>	53
14.3	<code>select</code> and <code>drop</code>	54
14.4	<code>rename</code>	54
14.5	<code>sort</code>	54
14.6	<code>dropna</code>	54
14.7	<code>append</code>	55
14.8	<code>collapse</code> and <code>group_by</code>	55
14.9	<code>pivot_longer</code> and <code>pivot_wider</code>	55
14.10	<code>replace</code>	56
14.11	<code>tabulate</code>	56
15	The Pipe Operator	57
15.1	Basic syntax	57
15.2	Semantics with DataFrames	57
15.3	COW and pipes	58
15.4	Long chains	59
15.5	Using placeholders	59
15.6	Pipe with closures	59
16	Modules, Packages, and Plugins	61
16.1	The Hayashi Module System	61
16.1.1	The <code>source</code> command	61
16.1.2	The <code>import</code> command	61
16.2	Package Manager and GitHub Publishing	62
16.2.1	CLI Commands	62

16.2.2	Standard Package Structure	62
16.3	Developing Advanced Plugins	63
16.3.1	1. Script Plugins (.hay)	63
16.3.2	2. Native Plugins (Exclusive to Rust)	63
16.3.3	3. WebAssembly Plugins (WASM - For other languages)	65
II	Applied Econometrics	67
17	Descriptive Statistics and Moments	69
17.1	Expected value	69
17.2	Variance and standard deviation	69
17.3	Conditional moments	70
17.4	Covariance	71
17.5	Correlation	71
17.5.1	Scalar between two variables	71
17.5.2	Correlation matrix	72
17.6	Median and quantiles	72
17.7	Detailed summary	73
17.8	Quick reference	74
17.9	Complete example	74
18	Ordinary Least Squares	77
18.1	The model	77
18.2	The estimator	77
18.3	Properties	78
18.4	DGP Verification	78
18.5	Standard errors	81
18.6	Inference	81
18.7	Syntax	82
18.8	Formula	82
18.9	Standard error options	83
18.10	Conditional sample	83
18.11	Capturing the result	83
18.12	Post-estimation	83
18.12.1	<code>predict</code>	83
18.12.2	<code>vif</code>	84
18.12.3	<code>test</code>	84

18.12.4 <code>lincom</code>	84
18.12.5 <code>margins</code>	84
18.13 Results table with <code>esttab</code>	85
18.14 Bootstrap	85
19 Instrumental Variables	87
19.1 The endogeneity problem	87
19.2 Identification conditions	87
19.3 The 2SLS estimator	88
19.4 Asymptotic variance	88
19.5 Weak instruments	89
19.6 DGP Verification	89
19.7 Syntax	91
19.8 Standard error options	91
19.9 Capturing the result	92
19.10 Weak instrument diagnostics	92
19.11 Fitted values	92
19.12 Comparison table	92
20 Panel Data	95
20.1 The model	95
20.2 Fixed Effects	95
20.3 Random Effects	96
20.4 Hausman Test	96
20.5 DGP Verification	96
20.6 Syntax	99
20.7 Standard error options	99
20.8 Capturing results	99
21 Difference-in-Differences	101
21.1 The model	101
21.2 Regression form	101
21.3 Parallel trends assumption	101
21.4 DGP Verification	102
21.5 Syntax	104
21.6 Capturing the result	104

22 Logit and Probit	105
22.1 The latent variable model	105
22.2 Interpretation of coefficients	105
22.3 DGP Verification	106
22.4 Syntax	107
22.5 Predicted probabilities	108
23 Quantile Regression	109
23.1 The model	109
23.2 The estimator	109
23.3 Advantages over OLS	110
23.4 DGP Verification	110
23.5 Syntax	111
23.6 Capturing the result	112
24 Regression Discontinuity Design	113
24.1 The model	113
24.2 Identification	113
24.3 The estimator	114
24.4 DGP Verification	114
24.5 Syntax	116
24.6 Capturing the result	116
24.7 Empirical validation	117
25 Generalized Method of Moments	119
25.1 The framework	119
25.2 The two-step estimator	120
25.3 J-test of overidentification	120
25.4 DGP Verification	120
25.5 Syntax	122
25.6 Capturing the result	122
III Time Series	123
26 AR, MA and ARMA	125
26.1 Stochastic processes and stationarity	125
26.2 AR(p) process	125
26.3 MA(q) process	126

26.4	ARMA(p, q) process	126
26.5	DGP Verification	126
26.6	Order selection	128
26.7	Syntax	129
26.8	Empirical validation	130
27	ARIMA and Unit Roots	131
27.1	Integrated processes	131
27.2	Augmented Dickey-Fuller Test	131
27.3	The ARIMA model	132
27.4	DGP Verification	132
27.5	Workflow	134
27.6	Syntax	135
27.7	Empirical validation	135
28	VAR and Granger Causality	137
28.1	The VAR model	137
28.2	Granger Causality	137
28.3	Impulse Response Function	138
28.4	DGP Verification	138
28.5	Lag selection	140
28.6	Syntax	141
28.7	Empirical validation	141
29	Cointegration and VECM	143
29.1	Long-run Relationship Between I(1) Series	143
29.2	Engle-Granger Test	143
29.3	Error Correction Model (VECM)	144
29.4	DGP Verification	144
29.5	Syntax	146
29.6	Empirical validation	147
30	GARCH and Conditional Volatility	149
30.1	Conditional Heteroskedasticity	149
30.2	ARCH-LM Test	150
30.3	DGP Verification	150
30.4	Variants: EGARCH and GJR-GARCH	151
30.5	Syntax	152
30.6	Empirical validation	153

IV Causal Inference	155
31 Propensity Score Matching	157
31.1 Selection on Observables	157
31.2 DGP Verification	158
31.3 Syntax	159
31.4 Empirical validation	160
32 Synthetic Control	161
32.1 The Single-Unit Problem	161
32.2 DGP Verification	162
32.3 Syntax	165
32.4 Empirical validation	166
V Qualitative Response and Limited Dependence	167
33 Count Models	169
33.1 Count Data and the Inadequacy of OLS	169
33.2 DGP Verification	170
33.3 Syntax	171
33.4 Empirical validation	172
34 Limited Dependent Variable Models	173
34.1 Censoring and Truncation	173
34.2 DGP Verification	174
34.3 Syntax	175
34.4 Empirical validation	176
35 Multiple Choice Models	177
35.1 Ordinal and Nominal Response	177
35.2 DGP Verification	178
35.3 Syntax	179
35.4 Empirical validation	179
36 Survival Analysis	181
36.1 Failure Times and Censoring	181
36.2 DGP Verification	182
36.3 Syntax	183

VI	Advanced Methods	185
37	Advanced Panel I — GMM, PCSE and GEE	187
37.1	Limitations of Static Estimators	187
37.2	DGP Verification	188
37.3	Syntax	189
38	Seemingly Unrelated Regressions	191
38.1	The Zellner Estimator	191
38.2	DGP Verification	191
38.3	Syntax	193
38.4	Empirical validation	193
39	Regularization and Variable Selection	195
39.1	The Dimensionality Problem	195
39.2	DGP Verification	196
39.3	Syntax	197
39.4	Empirical validation	198
40	Bootstrap	199
40.1	Inference by Resampling	199
40.2	DGP Verification	199
40.3	Syntax	200
41	Multivariate Analysis	203
41.1	Dimensionality reduction and latent structure	203
41.2	DGP Verification	204
41.3	Syntax	206
42	Generalized Linear Models	207
42.1	The exponential family	207
42.2	DGP Verification	208
42.3	Syntax	209
43	Regime-Switching Models	211
43.1	Structural heterogeneity over time	211
43.2	DGP Verification	211
43.3	Syntax	212

44 Time Series Filters and Decomposition	215
44.1 Separation of components	215
44.2 DGP Verification	216
44.3 Syntax	217
45 Structural VAR	219
45.1 From reduced form to structure	219
45.2 DGP Verification	219
45.3 Syntax	221
VII Complementary Methods	223
46 Classical Inference	225
46.1 Parametric and non-parametric hypothesis tests	225
46.2 DGP Verification	226
46.3 Syntax	227
47 Advanced Panel II	229
47.1 Beyond basic FE and RE	229
47.2 DGP Verification	230
47.3 Syntax	231
48 Unit Roots II	233
48.1 Beyond the ADF: PP and Zivot-Andrews	233
48.2 DGP Verification	234
48.3 Syntax	234
49 Series Decomposition II	237
49.1 STL and Christiano-Fitzgerald	237
49.2 DGP Verification	238
49.3 Syntax	239
50 SVAR with Long-Run Restrictions	241
50.1 Blanchard-Quah Identification	241
50.2 DGP Verification	241
50.3 Syntax	243
51 Local and Weighted Estimation	245
51.1 Weights, windows and smoothing	245

51.2 DGP Verification	246
51.3 Syntax	247
52 AutoReg and ARDL	249
52.1 Models with lags in y and x	249
52.2 DGP Verification	250
52.3 Syntax	254
53 Kalman Filter	255
53.1 Hidden state and noisy observation	255
53.2 DGP Verification	256
53.3 Syntax	258
Advanced Installation	261
53.4 Compilation with ODBC support	261
53.5 Environment variables	261
53.6 Modules and optional features	261
53.7 Editor integration	262
53.8 Development builds	262
Quick Reference	265
53.9 Types	265
53.10 Operators	265
53.11 Time series operators	265
53.12 Keywords	266
53.13 Descriptive statistics	266
53.14 DataFrame manipulation	267
53.15 Random generators	267
53.16 Linear regression and extensions	268
53.17 Panel data	268
53.18 Causal inference	269
53.19 Special dependent variables	269
53.20 Survival analysis	270
53.21 Time series	270
53.22 Filters and decomposition	270
53.23 Multivariate analysis	271
53.24 Hypothesis tests	271
53.25 Presenting results	272
53.26 Building a panel (formulas)	272

53.27REPL shortcuts	272
Equivalent Code by Platform	273
53.28OLS — DGP without noise and comparison with omitted variable . .	273
53.29IV — Endogeneity with valid instrument	275
53.30Panel — FE, RE and Hausman	276
53.31Logit and Probit	277
53.32Count models	278
53.33Tobit and Heckman	278
53.34Ordered and multinomial models	279
53.35Regularization (LASSO, Ridge, Elastic Net)	280
53.36Survival analysis	281
53.37SVAR — Cholesky and Blanchard-Quah	281
53.38Decomposition filters	282
53.39PSM and Synthetic Control	283
53.40Comparative summary	284
53.41Validation matrix	287
Index	289

Part I

The Language

Chapter 1

Introduction

1.1 What is Hayashi

HAYASHI is an interpreted programming language designed for applied econometric analysis. The name honours Fumio Hayashi, whose book *Econometrics* (2000) is a canonical reference in the field.

The language has three central goals:

1. **Familiar syntax** for Stata users — commands such as `generate`, `summarize`, and `regress` work as expected.
2. **General-purpose programming** — functions, closures, control structures, modules. It is not merely a collection of statistical commands.
3. **Performance without configuration** — the interpreter is written in Rust; no JVM, no GC, no runtime dependencies.

1.2 Design philosophy

Clear semantics before concise syntax

When an operation mutates an object, it should look visibly imperative. When it returns a new value, it should look functional. HAYASHI distinguishes these two forms consistently:

```
1 // imperative: modifies df in place (Stata-style)
2 generate df z = x^2
3
4 // functional: returns a new DataFrame, df unchanged
```

```
5 let df2 = generate(df, z = x^2)
```

Listing 1.1: Imperative form vs. functional form

Copy-on-Write for DataFrames

DataFrames use copy-on-write (COW) semantics: assigning a DataFrame to another variable creates a cheap alias. The actual copy only happens at the moment of mutation. This allows efficient sharing without surprises:

```
1 load "data.csv" as df
2 let baseline = df          // alias, no copy
3
4 let transformed = df |> generate(z = x^2) |> filter(z > 10)
5 // baseline still has no column z
```

Listing 1.2: COW in action

Error messages as documentation

Errors in HAYASHI include the code snippet that caused them, the line number, and, when possible, a suggested fix (*did you mean?*). A well-formatted error message is more useful than a manual.

1.3 Hayashi and Stata

HAYASHI is not a Stata clone. The data command syntax is inspired by Stata because that syntax is good — it is close to the language economists use naturally. But HAYASHI is a full language: it has first-class functions, closures, recursion, and structured error handling.

The table below summarises the key differences:

Aspect	Stata	Hayashi
License	Proprietary	GPL-3.0
Functions	Limited	First-class
Closures	No	Yes
Types	Implicit	Explicit in functions
DataFrames	One at a time	Multiple, COW
Modularisation	do	source
Performance	C	Rust

1.4 Hayashi and R / Python

R and Python have vast ecosystems. HAYASHI does not compete with them on the volume of available packages — it competes on the clarity of syntax for the specific task of applied econometrics. A panel regression script in HAYASHI has less visual noise than its equivalent in any other language.

1.5 Empirical validation programme

HAYASHI includes a reproducible validation programme in the `validation/` directory. Each case runs a HAYASHI script and compares it, within declared tolerances, against reference implementations in R and Python (statsmodels). The cases use both public real datasets and simulated DGPs taken from the chapters of this book.

To run the full validation suite:

```
1 python -m venv validation/.venv
2 validation/.venv/bin/pip install -r validation/requirements.
   txt
3 hay validate
```

Listing 1.3: Run the validation programme

The current status of every case is recorded in `validation/MATRIX.md`.

1.6 Structure of this book

Chapters 2–3 Installation and use of the REPL.

Chapters 4–12 Type system, variables, operators, control flow, functions, errors, strings, lists, and dictionaries.

Chapters 13–15 DataFrames, data manipulation, and the pipe operator.

Chapter 16 Modularisation with [source](#).

Parts II through VII cover descriptive statistics, OLS, IV, panel, time series, limited dependent variable models, causal inference, and advanced methods.

Chapter 2

Installation

2.1 Requirements

HAYASHI is distributed as a static binary for Linux, macOS, and Windows. There are no runtime dependencies: the `hay` executable is self-contained.

- Linux x86-64 or ARM64
- macOS 12+ (Intel or Apple Silicon)
- Windows 10+ (x86-64)

2.2 Via Cargo (recommended)

If Rust is installed:

```
cargo install hayashi-lang
```

The `hay` binary will be installed in `$HOME/.cargo/bin`. Make sure that directory is on your `PATH`.

```
hay --version
```

2.3 Pre-compiled binaries

Available on the repository releases page: <https://github.com/sheep-farm/hayashi/releases>

Download the archive for your platform, extract it, and place the `hay` binary in any directory on your `PATH`.

2.4 Building from source

```
git clone https://github.com/sheep-farm/hayashi
cd hayashi
cargo build --release
# binary at target/release/hay
```

2.5 Verifying the installation

```
hay --version
hay --help
```

To run a script:

```
hay my_script.hay
```

To enter the interactive REPL:

```
hay
```

Note

On Arch Linux with Omarchy/Hyprland, `$HOME/.cargo/bin` is usually on the `PATH` if the shell was configured with `rustup`. Otherwise, add to `.bashrc` or `.zshrc`: `export PATH="$HOME/.cargo/bin:$PATH"`

Chapter 3

The REPL

3.1 What is the REPL

The REPL (*Read-Eval-Print Loop*) is the interactive mode of HAYASHI. Each line typed is evaluated immediately and the result is displayed. It is the ideal environment for exploring data, testing expressions, and prototyping analyses.

```
hay

Hayashi 0.2.7-dev - Applied Econometrics Language
In honor of Fumio Hayashi. Type 'exit' or Ctrl-D to quit.

hay>
```

3.2 Exiting the REPL

Command	Effect
exit	Exit the REPL
Ctrl-D	Exit the REPL (EOF)
Ctrl-C	Cancel the current line

3.3 Navigation and history

The REPL uses `rustyline` and supports:

- Up/down arrow: command history
- Ctrl-R: history search

- Ctrl-A / Ctrl-E: beginning/end of line
- Ctrl-C: cancel the current line
- Ctrl-D: exit the REPL (equivalent to `exit`)

History is persisted across sessions in `$HOME/.hay_history`.

3.4 Expressions in the REPL

Any valid expression can be typed directly:

```
1 hay> 2 + 2
2 4
3
4 hay> let x = 10
5 hay> x * 3
6 30
7
8 hay> let s = "hello, world"
9 hay> s
10 "hello, world"
```

Listing 3.1: Basic interactive session

3.5 Running scripts

`.hay` scripts are text files with one statement per line. To run one:

```
hay analysis.hay
```

Output goes to `stdout`. Errors go to `stderr`.

3.6 Error format

HAYASHI displays errors with visual context:

```
error: line 3: undefined variable 'y' - did you mean 'x'?
  3 | display y + 1
    | ~~~~~
```

The code snippet, line number, and suggestion (when available) make correction straightforward without having to count lines manually.

Chapter 4

Types and Values

HAYASHI is dynamically typed: variables have no declared type, but each value carries its type at runtime. The type system is consistent — operations between incompatible types produce clear errors rather than silent coercions.

4.1 Primitive types

4.1.1 Int

64-bit signed integer. Integer literals are written without a decimal point:

```
1 let n      = 42
2 let negative = -7
3 let large   = 1_000_000    // _ as visual separator
```

4.1.2 Float

64-bit floating point (IEEE 754). Requires a decimal point or scientific notation:

```
1 let pi      = 3.14159
2 let e       = 2.71828
3 let small   = 1.5e-10
```

Arithmetic between Int and Float promotes to Float:

```
1 let x = 3 / 2      // 1      (integer division)
2 let y = 3 / 2.0    // 1.5
3 let z = 3.0 / 2    // 1.5
```

Dividing a Float by zero returns `inf` or `-inf`, never an error — following IEEE 754:

```
1 display 1.0 / 0.0    // inf
2 display -1.0 / 0.0   // -inf
3 display 0.0 / 0.0    // nan
```

Note

Dividing an `Int` by zero produces a runtime error. Dividing a `Float` by zero follows IEEE 754 and returns `inf`.

4.1.3 Bool

True or false. Literals: `true` and `false`.

```
1 let active = true
2 let closed = false
```

Logical operators: `and`, `or`, `not`.

4.1.4 Nil

Absence of value. Some functions return `nil` when there is no relevant result (for example, `push`, which mutates a list in place).

```
1 let nothing = nil
2 display nothing    // nil
```

4.2 Strings

Strings are Unicode character sequences delimited by double quotes:

```
1 let name = "Hayashi"
2 let path = "/home/user/data.csv"
```

Escape sequences: `\n` (newline), `\t` (tab), `\"` (quote), `\\` (backslash).

F-strings allow expression interpolation:

```
1 let year = 2026
2 display f"Published in {year}"    // Published in 2026
```

4.3 Lists

Ordered collections of values of any type, delimited by brackets:

```
1 let numbers = [1, 2, 3, 4, 5]
2 let mixed   = [1, "two", true, nil]
3 let empty   = []
```

4.4 Dictionaries

Key-value mappings. Keys and values may be of any type:

```
1 let config = {"host": "localhost", "port": 5432}
2 let empty  = {}
```

Access by key:

```
1 config["host"]    // "localhost"
```

4.5 DataFrame

The central type of HAYASHI. A DataFrame is a two-dimensional table with named columns. Each column is a vector of `Float` (missing values represented by `NaN`).

```
1 input x y
2   1  2
3   3  4
4   5  6
5 end
```

Listing 4.1: Creating a DataFrame inline

DataFrames are discussed in depth in [Chapter 13](#).

4.6 Conversions

HAYASHI does not perform implicit coercions between types. Conversions must be explicit:

```
1 let s = str(42)           // "42"
2 let n = int("7")          // 7
3 let f = float("3.14")     // 3.14
```

Attempting to add an `Int` to a `String` produces a type error:

```
1 42 + "3"  
2 // error: type mismatch - expected Int or Float, got String
```

Chapter 5

Variables and Scope

5.1 Declaration with `let`

Every variable is declared with `let`. The declaration binds a name to a value:

```
1 let x      = 10
2 let name = "Hayashi"
3 let active = true
```

`let` without an initial value does not exist in HAYASHI — every variable is initialised at the point of declaration.

Redeclaration

A variable may be redeclared with `let` in the same scope. The new value replaces the previous one without affecting other variables that point to the old value:

```
1 let x = 5
2 let y = x      // y points to 5
3 let x = 99     // x is now 99; y remains 5
4 display y     // 5
```

5.2 Constants with `const`

`const` declares a name that cannot be redeclared or reassigned in the same scope. Useful for configuration values or fixed parameters:

```
1 const ALPHA    = 0.05
2 const PROJECT = "my_lab"
```

Attempting to assign to a constant produces an error:

```
1 const N = 100
2 N = 200      // error: cannot assign to constant 'N'
```

Note

const in HAYASHI is evaluated at runtime, not at compile time. The distinction from **let** is purely one of binding mutability, not performance.

5.3 Assignment

After declaration with **let**, a variable can be reassigned with **=**:

```
1 let counter = 0
2 counter = counter + 1
3 counter = counter + 1
4 display counter    // 2
```

Compound assignment operators:

```
1 let n = 10
2 n += 5      // n = 15
3 n -= 3      // n = 12
4 n *= 2      // n = 24
5 n /= 4      // n = 6
```

5.4 Scopes

Scopes are delimited by **{ }** blocks. Variables declared inside a block exist only until the end of that block:

```
1 let x = 1
2
3 {
4   let y = 2
5   display x    // 1 - x is visible here
6   display y    // 2
7 }
8
9 display x      // 1
10 display y     // error: undefined variable 'y'
```

Listing 5.1: Variables local to a block

Scopes in functions

Each function call creates a new scope. Parameters and local variables are destroyed when the function returns:

```
1 fn sum(a, b) {  
2   let result = a + b  
3   return result  
4 }  
5  
6 display sum(3, 4)    // 7  
7 display result      // error: undefined variable 'result'
```

Outer scope capture (closures)

Anonymous functions (closures) capture variables from the scope where they were defined:

```
1 let base = 100  
2  
3 let add = fn(x) {  
4   return x + base    // captures 'base' from outer scope  
5 }  
6  
7 display add(20)      // 120
```

5.5 Special variables

`_n` and `_N` in data contexts

Inside `generate` and `mutate`, two implicit variables are available:

- `_n`: index of the current row (starting at 1)
- `_N`: total number of rows in the DataFrame

```
1 input x
2   10  20  30
3 end
4
5 generate df id      = _n      // [1, 2, 3]
6 generate df weight = 1/_N     // [0.333, 0.333, 0.333]
```

Listing 5.2: Using `_n` and `_N`

`_` (discard)

The identifier `_` discards a value without creating a binding:

```
1 for _ in 1..5 {
2   display "iteration"
3 }
```

Chapter 6

Operators

6.1 Arithmetic

Operator	Operation	Example
+	Addition	$3 + 2 \rightarrow 5$
-	Subtraction	$5 - 3 \rightarrow 2$
*	Multiplication	$4 * 3 \rightarrow 12$
/	Division	$7 / 2 \rightarrow 3$ (integers)
^	Power	$2^{10} \rightarrow 1024$
%	Modulo	$7 \% 3 \rightarrow 1$

Division between two `Int` values is integer division (truncated toward zero). For real division, at least one operand must be `Float`:

```
1 7 / 2      // 3
2 7 / 2.0    // 3.5
3 7.0 / 2    // 3.5
```

6.2 Comparison

Operator	Meaning	Example
==	Equal to	$3 == 3 \rightarrow \text{true}$
!=	Not equal to	$3 != 4 \rightarrow \text{true}$
>	Greater than	$5 > 3 \rightarrow \text{true}$
<	Less than	$3 < 5 \rightarrow \text{true}$
>=	Greater than or equal	$3 >= 3 \rightarrow \text{true}$
<=	Less than or equal	$2 <= 3 \rightarrow \text{true}$

6.3 Logical

```
1 true and false    // false
2 true or false     // true
3 not true          // false
```

`and` and `or` use short-circuit evaluation: the second operand is not evaluated if the result is already determined by the first.

6.4 The `in` operator

Tests membership in a list:

```
1 let regions = ["north", "south", "center"]
2 "south" in regions    // true
3 "east" in regions     // false
```

Also works with ranges:

```
1 5 in 1..10    // true
2 15 in 1..10   // false
```

6.5 The pipe operator `|>`

The `|>` operator passes the value on the left as the first argument to the function on the right:

```
1 // without pipe
2 display filter(df, x > 5)
3
4 // with pipe - more readable
5 df |> filter(x > 5) |> display
```

The pipe has special semantics with DataFrames: when used without an explicit assignment (`let`), the result is assigned back to the source variable:

```
1 let df = load("data.csv")
2
3 // updates df in place
4 df |> filter(x > 0) |> sort(x)
5
6 // creates df2, df remains unchanged
```

```
7 let df2 = df |> filter(x > 0)
```

This behaviour is discussed in detail in Chapter 15.

6.6 Precedence

From lowest to highest priority:

1. or
2. and
3. not
4. == != > < >= <=
5. + -
6. * / %
7. ^ (right-associative)
8. Unary -
9. Function call, indexing, field access

Use parentheses for clarity when precedence is not obvious:

```
1 2 + 3 * 4      // 14  (multiplication first)
2 (2 + 3) * 4    // 20
```


Chapter 7

Control Flow

7.1 if / else

```
1 let grade = 7.5
2
3 if grade >= 7 {
4   display "passed"
5 } else if grade >= 5 {
6   display "make-up exam"
7 } else {
8   display "failed"
9 }
```

Listing 7.1: Basic if/else structure

if as an expression

`if` can be used as an expression — it returns the value of the chosen branch:

```
1 let status = if grade >= 7 { "passed" } else { "failed" }
2 display status
```

When used as an expression, both branches must have the same type (or both `nil`).

7.2 match

`match` compares a value against multiple patterns. The first matching pattern is executed:

```
1 let code = 2
2
3 match code {
4   1 => display "beginner"
5   2 => display "intermediate"
6   3 => display "advanced"
7   _ => display "unknown"
8 }
```

Listing 7.2: match with literals

The `_` pattern is the wildcard — it matches any value.

match with strings

```
1 let state = "TX"
2
3 match state {
4   "TX" => display "Texas"
5   "CA" => display "California"
6   "NY" => display "New York"
7   _    => display f"Unknown state: {state}"
8 }
```

match as an expression

```
1 let description = match code {
2   1 => "beginner"
3   2 => "intermediate"
4   _ => "advanced"
5 }
```

Multiple values per arm

```
1 match code {
2   1 | 2    => display "basic"
3   3 | 4    => display "advanced"
4   _       => display "other"
5 }
```

7.3 Expression block

A `{ }` block is an expression that evaluates to the last value produced inside it:

```
1 let result = {  
2   let a = 10  
3   let b = 20  
4   a + b      // this is the block's value  
5 }  
6 display result    // 30
```


Chapter 8

Loops

8.1 for

Iterates over a sequence. The iteration variable is visible inside the block and *persists* after the loop with the last value it held:

```
1 for i in 1..4 {  
2     display i    // 1, 2, 3  (4 not included)  
3 }  
4 display i    // 3 - variable persists after the loop
```

Listing 8.1: for over a range

Exclusive and inclusive ranges

HAYASHI follows the same convention as Rust:

```
1 for i in 1..3 { }    // 1, 2    (exclusive: end not  
    included)  
2 for i in 1..=3 { }   // 1, 2, 3  (inclusive: end included)
```

Ranges are also expressions that evaluate to lists — see Section [12.2](#).

Iterating over lists

```
1 let fruits = ["apple", "banana", "grape"]  
2  
3 for fruit in fruits {  
4     display fruit  
5 }
```

Discarding the iteration variable

When the index is not needed:

```
1 for _ in 1..3 {  
2   display "repetition"  
3 }
```

8.2 while

Executes while the condition is true:

```
1 let n = 1  
2  
3 while n <= 5 {  
4   display n  
5   n += 1  
6 }
```

Listing 8.2: Basic while

Warning

HAYASHI does not detect infinite loops. A `while true { }` without `break` will run indefinitely. Use `Ctrl-C` to interrupt.

8.3 break and continue

`break` exits the loop immediately. `continue` advances to the next iteration:

```
1 for i in 1..10 {  
2   if i == 3 { continue }    // skip 3  
3   if i == 7 { break }      // stop at 7  
4   display i  
5 }  
6 // prints: 1, 2, 4, 5, 6
```

Listing 8.3: break and continue

8.4 Loops with DataFrames

`for` can iterate over lists produced by data functions:

```
1 load "sales.csv" as df
2
3 for state in ["TX", "CA", "NY"] {
4   let sub = filter(df, region == state)
5   display f"{state}: {count(sub)} obs"
6 }
```

Listing 8.4: Iterating over groups

Chapter 9

Functions

9.1 Declaration

Functions are declared with `fn`:

```
1 fn square(x) {  
2   return x^2  
3 }  
4  
5 display square(5)    // 25
```

Listing 9.1: Basic function

Implicit return

The last expression in a function is returned automatically:

```
1 fn square(x) {  
2   x^2    // implicit return  
3 }
```

Multiple parameters

```
1 fn mean(a, b) {  
2   (a + b) / 2.0  
3 }  
4  
5 display mean(10, 20)    // 15.0
```

9.2 Functions as values

Functions are first-class values — they can be assigned to variables, passed as arguments, and returned by other functions:

```
1 fn apply(f, x) {  
2   f(x)  
3 }  
4  
5 fn double(n) { n * 2 }  
6  
7 display apply(double, 7)    // 14
```

Listing 9.2: Function as argument

9.3 Closures

Closures are anonymous functions that capture the environment where they were defined:

```
1 let multiplier = fn(factor) {  
2   fn(x) { x * factor }    // captures 'factor'  
3 }  
4  
5 let triple = multiplier(3)  
6 display triple(10)    // 30
```

Listing 9.3: Basic closure

Closures as arguments

```
1 fn apply_two(f, a, b) {  
2   f(a) + f(b)  
3 }  
4  
5 let sum_of_squares = apply_two(fn(x) { x^2 }, 3, 4)  
6 display sum_of_squares    // 9 + 16 = 25
```

9.4 Recursion

Functions can call themselves:

```

1 fn fact(n) {
2   if n <= 1 { return 1 }
3   n * fact(n - 1)
4 }
5
6 display fact(10)    // 3628800

```

Listing 9.4: Recursive factorial

Note

HAYASHI does not optimise tail calls. For long sequences, prefer loops.

9.5 Optional parameters and defaults

HAYASHI has no native optional parameters, but the idiomatic pattern is to use `nil` as a sentinel and check explicitly:

```

1 fn format_val(value, digits) {
2   let d = if digits == nil { 2 } else { digits }
3   // ...
4 }

```

9.6 Function scope

Functions declared with `fn` at module level are available throughout the file. Functions declared inside blocks are local to that block:

```

1 {
2   fn local(x) { x + 1 }
3   display local(5)    // 6
4 }
5 display local(5)    // error: undefined function 'local'

```

9.7 const inside functions

`const` can be used inside functions to fix values that should not change during execution:

```
1 fn bmi(weight, height) {  
2   const FORMULA = "weight / height^2"  
3   weight / height^2  
4 }
```

Chapter 10

Error Handling

10.1 Error model

HAYASHI distinguishes two types of error:

Parse errors Detected before execution, while analysing the code. Include line number and code snippet.

Runtime errors Occur during execution — undefined variable, wrong type, division by zero, missing key.

Both types display the code snippet and line number, making the source easy to locate:

```
error: line 5: undefined variable 'income' - did you mean '
    gross_income'?
5 | display income * 12
  | ~~~~~
```

10.2 try / catch

Runtime errors can be caught with `try/catch`:

```
1 try {
2   let result = 10 / 0
3   display result
4 } catch e {
5   display f"Caught error: {e}"
6 }
```

Listing 10.1: Basic try/catch

The variable `e` inside `catch` contains the error message as a string.

Catching file errors

```
1 try {  
2   load "missing_file.csv" as df  
3 } catch e {  
4   display f"File not found: {e}"  
5 }
```

try as an expression

`try/catch` can return a value:

```
1 let value = try {  
2   int("not a number")  
3 } catch _ {  
4   -1  
5 }  
6 display value    // -1
```

10.3 Common errors and their messages

Situation	Typical message
Undeclared variable	undefined variable 'x' - did you mean 'y'?
Unknown function	undefined function 'foo' - did you mean 'for'?
Wrong type	type mismatch - expected Int, got String
Index out of bounds	index 5 out of bounds (len=3)
Missing dict key	key 'name' not found in dict
Integer division by zero	division by zero
File not found	file not found: 'path.csv'

10.4 Automatic suggestions

When the name of a variable or function resembles something that exists, HAYASHI suggests a correction:

```
error: undefined function 'regres' - did you mean 'regress'?
```

Suggestions use edit distance (Levenshtein) over all names defined in scope and in built-in functions.

10.5 Re-throwing errors

To propagate a caught error:

```
1 try {  
2   risky_operation()  
3 } catch e {  
4   display f"log: {e}"  
5   error(e)    // re-throws the error  
6 }
```

The function `error(msg)` throws a runtime error with the given message.

Chapter 11

Strings and F-Strings

11.1 String literals

Strings are delimited by double quotes:

```
1 let s      = "Hello, world"
2 let path = "/home/user/data.csv"
3 let empty = ""
```

Escape sequences

Sequence	Meaning
<code>\n</code>	Newline
<code>\t</code>	Tab
<code>\r</code>	Carriage return
<code>\"</code>	Literal double quote
<code>\\</code>	Literal backslash

11.2 String operations

Concatenation

```
1 let name      = "Hayashi"
2 let version = "0.2.7-dev"
3 let title     = name + " " + version    // "Hayashi 0.2.7-dev"
```

Length

```
1 let s = "econometrics"
2 display len(s)      // 12
```

Conversions

```
1 let n = str(42)      // "42"
2 let f = str(3.14)    // "3.14"
3 let b = str(true)    // "true"
```

Comparison

Strings are compared lexicographically with `==`, `!=`, `<`, `>`, `<=`, `>=`.

11.3 F-Strings

F-strings allow arbitrary expressions to be interpolated inside a string. The prefix `f` indicates interpolation; expressions between `{ }` are evaluated and converted to string:

```
1 let city = "Austin"
2 let pop  = 978_908
3
4 display f"{city} has {pop} inhabitants"
5 // Austin has 978908 inhabitants
6
7 display f"Mean: {(10 + 20 + 30) / 3.0:.2f}"
8 // Mean: 20.00
```

Listing 11.1: F-strings

Expressions in f-strings

Any expression can be interpolated — variables, arithmetic, function calls:

```
1 let n = 5
2 display f"{n}^2 = {n^2}"
3 // 5^2 = 25
4
```

```
5 display f"sqrt({n}) = {sqrt(n):.4f}"  
6 // sqrt(5) = 2.2361
```

F-strings in display

`display` accepts `end=` and `sep=` options to control the terminator and separator between multiple values:

```
1 display "a", "b", "c" sep=", " // a, b, c  
2 display "line" end="" // line (no newline)
```

11.4 Multiline strings

HAYASHI has no multiline string literal. Use explicit concatenation:

```
1 let sql = "SELECT * FROM data " +  
2         "WHERE year = 2026 " +  
3         "ORDER BY state"
```


Chapter 12

Lists and Dictionaries

12.1 Lists

12.1.1 Creation

```
1 let nums    = [1, 2, 3, 4, 5]
2 let states = ["TX", "CA", "NY"]
3 let mixed   = [1, "two", true]
4 let empty   = []
```

12.1.2 Index access

Indices start at 0. Negative indices count from the end:

```
1 let xs = [10, 20, 30, 40]
2 display xs[0]      // 10
3 display xs[3]      // 40
4 display xs[-1]     // 40 (last)
5 display xs[-2]     // 30
```

Out-of-bounds access produces a runtime error:

```
1 xs[10]
2 // error: index 10 out of bounds (len=4)
```

12.1.3 Length

```
1 len([1, 2, 3])    // 3
```

12.1.4 Mutation

`push` appends to the end of a list (mutates in place, returns `nil`):

```
1 let list = [1, 2, 3]
2 push(list, 4)
3 display list      // [1, 2, 3, 4]
```

Warning

`push` is a mutating function — it modifies `list` directly and returns `nil`. Do not write `let new = push(list, 4)`.

`pop` removes and returns the last element:

```
1 let last = pop(list)
```

12.1.5 Slicing

```
1 let xs = [0, 1, 2, 3, 4, 5]
2 display slice(xs, 1, 4)      // [1, 2, 3] (indices 1..3)
```

12.1.6 Iteration

```
1 for x in [10, 20, 30] {
2   display x * 2
3 }
```

12.1.7 Functional operations

```
1 let xs = [1, 2, 3, 4, 5]
2
3 let doubled = map(xs, fn(x) { x * 2 })
4 let evens   = filter(xs, fn(x) { x % 2 == 0 })
5 let total   = reduce(xs, 0, fn(acc, x) { acc + x })
```

12.2 Ranges

Ranges are expressions that produce lists of integers. The syntax follows Rust:

```

1 let a = 1..5           // [1, 2, 3, 4]    (exclusive)
2 let b = 1..=5          // [1, 2, 3, 4, 5] (inclusive)

```

Listing 12.1: Ranges as lists

Since they are ordinary lists, ranges work with any function that accepts a list:

```

1 map(1..=5, fn(x) { x^2 })           // [1, 4, 9, 16, 25]
2 filter(1..10, fn(x) { x % 2 == 0 }) // [2, 4, 6, 8]
3 len(1..100)                         // 99

```

12.2.1 chain

`chain` concatenates multiple lists (or ranges) into one:

```

1 chain(1..3, 9..=11)           // [1, 2, 9, 10, 11]
2 chain(1..=3, [10, 20], 5..7) // [1, 2, 3, 10, 20, 5, 6]

```

Listing 12.2: chain with ranges and lists

Typical use: iterating over non-contiguous sequences:

```

1 for lag in chain(1..=3, 6..=8) {
2   display f"lag {lag}"
3 }

```

12.2.2 Composition with closures

Ranges, `chain`, and closures compose naturally in a single expression. The syntax `|x| expr` defines a closure without the ceremony of `fn`:

```

1 let r = map(chain(1..=10, 5..9, 56..70), |n| n * 3)

```

Listing 12.3: Composed expression — single line

The pipe also works in `for` headers, composing transformations before entering the loop body:

```

1 for i in chain(1..=10, 20..=25) |> filter(|n| (n % 2) == 0) {
2   display i
3 }

```

Listing 12.4: Pipeline in the for header

12.2.3 Comparison with other languages

Language	Form
Python	<code>filter(lambda n: n%2==0, chain(range(1,11), range(5,9), range(56,70)))</code>
Rust	<code>(1..=10).chain(5..9).chain(56..70).filter(n n%2==0)</code>
Haskell	<code>filter even \$ [1..10] ++ [5..8] ++ [56..69]</code>
Julia	<code>Iterators.filter(n->n%2==0, Iterators.flatten([1:10,5:8,56:69]))</code>
R	<code>Filter(function(n) n%%2==0, c(1:10, 5:8, 56:69))</code>
Hayashi	<code>chain(1..=10,5..9,56..70) > filter(n (n%2)==0)</code>

HAYASHI is the only language in which the pipeline runs strictly left-to-right, with no imports, no namespaces, and a variadic `chain` that treats all arguments symmetrically.

12.2.4 Performance note

`chain` materialises ranges into a list before iterating. `for i in 1..N`, by contrast, iterates directly without allocating. `chain` is for small non-contiguous sequences — lags, years, variable groups. Any contiguous sequence is simply `1..N` or `1..=N`.

12.3 Dictionaries

12.3.1 Creation

```

1 let config = {
2     "host": "localhost",
3     "port": 5432,
4     "ssl": true
5 }
6
7 let empty = {}

```

Keys can be strings or integers. Values can be of any type.

12.3.2 Access

```

1 config["host"]    // "localhost"
2 config["port"]    // 5432

```

A missing key produces an error:

```
1 config["user"]
2 // error: key 'user' not found in dict
```

Use `try/catch` for safe access:

```
1 let user = try { config["user"] } catch _ { "anonymous" }
```

12.3.3 Insertion and update

```
1 let d = {"a": 1}
2 let d2 = dict_set(d, "b", 2) // {"a":1, "b":2}
```

`dict_set` is functional — it returns a new dictionary without mutating the original.

12.3.4 Checking for a key

```
1 "host" in config // true
2 "password" in config // false
```

12.3.5 Dictionaries as function return values

Analysis functions often return dictionaries with multiple results:

```
1 load "data.csv" as df
2 let stats = summarize(df, income)
3
4 display stats["mean"] // mean
5 display stats["sd"] // standard deviation
6 display stats["N"] // number of observations
```

Listing 12.5: `summarize` returns a dictionary

12.3.6 Iterating over dictionaries

```
1 let d = {"a": 1, "b": 2, "c": 3}
2
3 for key in keys(d) {
4   display f"{key} = {d[key]}"
5 }
```


Chapter 13

DataFrames

13.1 What is a DataFrame

A DataFrame is a two-dimensional table with named columns. In HAYASHI, each column is a 64-bit floating-point vector. Missing values are represented by `NaN` (*Not a Number*).

The DataFrame is the central type of the language — virtually every econometric analysis starts with a DataFrame loaded from some source.

13.2 Copy semantics (COW)

DataFrames use copy-on-write (COW) semantics: assigning a DataFrame to another variable creates an alias that shares the same internal data. The actual copy occurs only when one of them is modified.

```
1 load "panel.csv" as df
2
3 let baseline = df           // cheap alias, no copy
4
5 let df2 = df |> generate(z = x^2)    // df2 has 'z', df does
   not
6 let df3 = df |> filter(year > 2000) // df3 filtered, df
   intact
7
8 // baseline == original df at any point
```

Listing 13.1: COW: baseline preserved

This semantics allows creating analysis variants with no memory cost until an actual modification is needed.

13.3 Creating DataFrames

input block

For small inline datasets — ideal for examples and tests:

```
1 input x y z
2     1  2 10
3     3  4 20
4     5  6 30
5 end
```

Listing 13.2: input block

The `input` block creates a DataFrame named `df` by default. The number of columns is defined by the header; each data row must have the same number of values.

Note

`input` accepts only numeric values. Use `.` to represent missing values. For data with text columns, use `load`.

load — external files

```
1 load "data.csv" as sales
2 load "panel.dta" as panel      // Stata format
3 load "results.parquet" as df
4 load "sheet.xlsx" as df       // first sheet
5 load "sheet.xlsx" sheet="Data" as df
```

Listing 13.3: Loading files

Supported formats: CSV, Parquet, Stata (`.dta`), Excel (`.xlsx`), SQLite.

13.4 Saving DataFrames

```
1 save df "output.csv"
2 save df "output.parquet"
```

```

3 save df "output.dta"
4 save df "output.xlsx"

```

13.5 Inspecting DataFrames

display

Displays the first rows of the DataFrame:

```

1 display df
2 display df, 20    // display up to 20 rows

```

describe

Displays the structure: number of observations, variables, and types:

```

1 describe df

```

Typical output:

```

DataFrame: 1250 obs, 8 vars
  year      Float   (0 missing)
  state     Float   (0 missing)
  income    Float   (12 missing)
  gini      Float   (3 missing)

```

count and nrow

Returns the number of rows:

```

1 let n  = count(df)           // total observations
2 let n2 = nrow(df)            // alias for count(df)
3 let n3 = count(df, income > 1000) // conditional count

```

13.6 Special variables

Inside row-by-row commands (**generate**, **mutate**, **filter**), two implicit variables are available:

Current row number (starts at 1).

Total rows in the DataFrame.

```
1 generate df id      = _n      // 1, 2, 3, ..., N
2 generate df weight = 1.0/_N    // uniform weight
```

13.7 Time series operators

Inside `generate` expressions (imperative form), special time-series operators are available:

Operator	Meaning	Example
<code>L.x</code>	Lag (previous value)	<code>L.gdp</code> $\rightarrow$ gdp_{t-1}
<code>-N_nF.x</code>	Lead (next value)	<code>F.rate</code> $\rightarrow$ rate_{t+1}
<code>D.x</code>	First difference	<code>D.gdp</code> $\rightarrow$ $\text{gdp}_t - \text{gdp}_{t-1}$
<code>L2.x</code>	Lag of order 2	<code>L2.gdp</code> $\rightarrow$ gdp_{t-2}

```
1 generate df dgdp      = D.gdp      // GDP growth
2 generate df gdp_lag   = L.gdp      // lagged GDP
```

Listing 13.4: Time series operators

Chapter 14

Data Manipulation

14.1 `generate` and `mutate`

Create or replace columns. The two forms are equivalent in result; they differ in mutation semantics:

Form	Semantics	Style
<code>generate df col = expr</code>	Imperative (mutates <code>df</code>)	Stata
<code>let df2 = generate(df, col = expr)</code>	Functional (df unchanged)	Functional
<code>df > generate(col = expr)</code>	Pipe assign-back (mutates <code>df</code>)	Chaining

```
1 load "data.csv" as df
2
3 generate df lincome = log(income)
4 generate df income2 = income^2
5 generate df dummy    = if(income > 1000, 1, 0)
```

Listing 14.1: `generate`: imperative form

```
1 let df2 = mutate(df, lincome = log(income), income2 = income
  ^2)
2 // df does not have lincome or income2
3 // df2 has them
```

Listing 14.2: `mutate`: functional form

14.2 `filter`

Selects rows that satisfy a condition:

```
1 let recent = filter(df, year >= 2010)
2 let high   = filter(df, income > 5000 and state == "TX")
```

The imperative form uses `keep`:

```
1 keep df if year >= 2010
```

14.3 select and drop

```
1 // keep only these columns
2 let slim   = select(df, year, state, income)
3 let no_id  = drop(df, id, code)    // remove these columns
```

14.4 rename

```
1 rename df old_name new_name
2 // rename family_income to income
3 rename df family_income income
```

14.5 sort

```
1 sort df year           // ascending
2 sort df income desc    // descending
3 sort df state year desc // multiple columns
```

14.6 dropna

Removes rows with missing values:

```
1 dropna df           // remove rows with any NaN
2 dropna df income gini // remove rows with NaN in income
   or gini
```

14.7 append

Stacks two DataFrames vertically (equivalent to `rbind`):

```
1 let df_total = append(df_2024, df_2025)
```

Columns must be compatible. Columns missing from one DataFrame are filled with NaN.

14.8 collapse and group_by

`group_by` aggregates by groups:

```
1 let state_means = group_by(df, state,
2   mean_income = mean(income),
3   n           = count()
4 )
```

Listing 14.3: Aggregation by group

`collapse` is the Stata-style imperative form:

```
1 collapse df (mean) income gini, by(state year)
```

14.9 pivot_longer and pivot_wider

```
1 let long = pivot_longer(df,
2   cols    = ["q1", "q2", "q3"],
3   names   = "quarter",
4   values  = "value"
5 )
```

Listing 14.4: Wide to long

```
1 let wide = pivot_wider(df,
2   id      = "county",
3   names   = "year",
4   values  = "gdp"
5 )
```

Listing 14.5: Long to wide

14.10 `replace`

Replaces values conditionally:

```
1 // zero out negative values
2 replace df income = 0 if income < 0
3 replace df gini = . if gini > 1    // mark as missing
```

14.11 `tabulate`

Frequency table:

```
1 tabulate df state
2 tabulate df state year    // cross-tabulation
```

Chapter 15

The Pipe Operator

15.1 Basic syntax

The `|>` operator passes the value on its left as the first argument to the function on its right:

```
1 expr |> f(args...)
2 // equivalent to: f(expr, args...)
```

Without pipe:

```
1 display sort(filter(df, income > 0), income desc)
```

With pipe:

```
1 df |> filter(income > 0) |> sort(income desc) |> display
```

The pipe allows reading transformations in the order they occur, left to right (or top to bottom), without nested parentheses.

15.2 Semantics with DataFrames

The pipe's behaviour with DataFrames follows a single rule:

Form	Effect
<code>df > f(...)</code>	Assign-back: df receives the result
<code>let df2 = df > f(...)</code>	Functional: df unchanged, df2 is new

Assign-back

When the pipe starts with a variable (`df |> ...`) and there is no `let`, the result is assigned back to the source variable:

```
1 load "data.csv" as df
2
3 df |> filter(year >= 2010)
4     |> dropna
5     |> sort(income desc)
6
7 // df is now the result of all transformations
```

Listing 15.1: Assign-back: df is modified

Functional form with let

```
1 load "data.csv" as df
2
3 let df_clean = df
4     |> filter(year >= 2010)
5     |> dropna
6
7 // df is intact; df_clean is the filtered version
```

Listing 15.2: let preserves the original

15.3 COW and pipes

COW semantics ensures that assign-back is efficient: while `df` and `df_clean` share the same data (before any mutation), no memory copy occurs. The actual copy only happens when the data diverges.

```
1 load "panel.csv" as df
2 let baseline = df           // cheap alias
3
4 df |> filter(treated == 1)   // modifies df, not baseline
5
6 display count(baseline)     // all original obs
7 display count(df)           // only the treatment group
```

Listing 15.3: Preserving baseline with let

15.4 Long chains

The pipe is especially useful in data cleaning and transformation pipelines:

```

1 load "survey.csv" as survey
2
3 let sample = survey
4   |> filter(age >= 18 and age <= 65)
5   |> filter(hours_worked > 0)
6   |> dropna
7   |> generate(lincome = log(income + 1))
8   |> generate(exp = age - years_educ - 6)
9   |> generate(exp2 = exp^2)
10  |> sort(state year)
11
12 display count(sample)

```

Listing 15.4: Complete preparation pipeline

15.5 Using placeholders

If you need to pass the left-hand side value to a specific argument position other than the first one, you can use the underscore (`_`) as a placeholder:

```

1 expr |> f(a, _)
2 // equivalent to: f(a, expr)

```

This is particularly useful when piping DataFrames into estimators (like `ols` or `iv`), where the DataFrame is not the first argument:

```

1 let model = df |> ols(y ~ x1 + x2, _)

```

15.6 Pipe with closures

The `|>` operator also works with closures:

```

1 let result = 5 |> fn(x) { x^2 + 1 }
2 display result // 26

```


Chapter 16

Modules, Packages, and Plugins

16.1 The Hayashi Module System

In HAYASHI, code sharing and organization are handled via two main commands: `source` and `import`.

16.1.1 The `source` command

The `source` command executes a `.hay` file directly within the current context. All definitions (functions and variables) inside the file become visible in the global scope of the caller, without namespaces:

```
1 source "lib/utils.hay"
2 let result = my_utility_function(10)
```

Listing 16.1: Basic usage of `source`

16.1.2 The `import` command

The `import` command loads a module and isolates its functions under a `**namespace**` using the `::` delimiter. This prevents naming conflicts and keeps your code organized:

```
1 // Import and use the filename as the default namespace
2 import("lib/math.hay")
3 let r1 = math::compute_mean([10, 20, 30])
4
5 // Import using a custom alias
6 import("lib/math.hay", as=m)
7 let r2 = m::compute_mean([10, 20, 30])
8
```

```
9 // Import specific functions directly into the local scope
10 import("lib/math.hay", only=["compute_mean"])
11 let r3 = compute_mean([10, 20, 30])
```

Listing 16.2: Importing with namespaces

16.2 Package Manager and GitHub Publishing

HAYASHI uses **GitHub directly as its package registry**. This eliminates the need for a proprietary central registry and promotes decentralized open-source package distribution.

16.2.1 CLI Commands

To manage third-party packages from your operating system terminal, use:

```
hay install usuario/repositorio -- Install a package from
                                GitHub (-y to skip overwrite prompt)
hay remove  usuario/repositorio -- Uninstall a package (
                                deletes script dirs, native files, and metadata)
hay list                                         -- List installed packages
                                and plugins
hay check-plugin [user/repo]                  -- Check integrity/version
                                with remote repository
hay update [user/repo] [-y]                   -- Update package(s) to
                                latest version (-y to skip prompt)
```

16.2.2 Standard Package Structure

Hayashi packages are convention-based and require zero configuration:

- **Script Packages:** Simply push your `.hay` script(s) to the root of a GitHub repository (e.g. `github.com/john/finance/interest.hay`).
- **Native Plugins:** Set up a GitHub Actions workflow to build and upload compiled binaries (`.so`, `.dll`, `.dylib`, or `.wasm`) as assets to a GitHub Release.

Once your package is pushed to GitHub, any user in the world can install and use it in Hayashi:

```
1 // In the OS terminal:
2 // hay install john/finance
3
4 // In your Hayashi script:
5 import("john/finance/interest")
6 let rate = finance::compute_rate(1000, 12)
```

Listing 16.3: Installing and importing a package

16.3 Developing Advanced Plugins

For needs demanding high performance or integration with other languages, Hayashi supports three plugin types under the exact same `import` syntax:

16.3.1 1. Script Plugins (.hay)

Written in pure Hayashi. Ideal for macro-like econometric routines and fast utilities.

16.3.2 2. Native Plugins (Exclusive to Rust)

Native plugins are developed in Rust to guarantee maximum performance and memory safety. They communicate with Hayashi through a stable, decoupled C-string JSON ABI.

To make development easier, you should use the official `hayashi-plugin-sdk`. The SDK provides proc-macros to automatically generate all unsafe FFI boilerplate and handle JSON serialization/deserialization:

```
1 use hayashi_plugin_sdk::{hayashi_fn, hayashi_plugin};
2
3 // The #[hayashi_fn] macro automatically handles JSON (de)
4   serialization.
5 // You can use standard Rust types (Vec, HashMap, String,
6   primitives, etc.).
7 #[hayashi_fn]
8 pub fn distance_to_point(lats: Vec<f64>, lons: Vec<f64>,
9   ref_lat: f64, ref_lon: f64) -> Vec<f64> {
10     lats.iter()
11       .zip(lons.iter())
```

```
9      .map(|(&lat, &lon)| {
10          // haversine distance calculation
11          let r = 6371.0;
12          let d_lat = (ref_lat - lat).to_radians();
13          let d_lon = (ref_lon - lon).to_radians();
14          let a = (d_lat / 2.0).sin().powi(2)
15                + lat.to_radians().cos() * ref_lat.to_radians
16                  ().cos() * (d_lon / 2.0).sin().powi(2);
17          let c = 2.0 * a.sqrt().atan2((1.0 - a).sqrt());
18          r * c
19      })
20      .collect()
21 }
22 // Generates the required FFI memory cleanup functions
23 hayashi_plugin!();
```

Listing 16.4: Developing a native plugin using the SDK

Compiled as a `cdylib`, you can publish your plugin by creating a tag (e.g. `v0.1.0`) on GitHub. The GitHub Actions release workflow compiles the plugin for Linux (`.so`), macOS (`.dylib`), and Windows (`.dll`), uploading them as release assets.

Any user can install it using the CLI:

```
hay install john/spatial
```

The package manager automatically downloads the binary that matches the current operating system and architecture.

You can then load it and name its namespace in Hayashi:

```
1 import("john/spatial", as=geo)
2 let dist = geo::distance_to_point(df["lat"], df["lon"],
3                                   -30.03, -51.22)
3 generate df dist_poa = dist
```

Security and Memory Considerations: JSON ABI vs. Zero-Copy FFI

While the default C-string JSON ABI offers extreme safety and decoupling (as any schema evolution in JSON doesn't break memory layouts), it can become a bottleneck when sharing large volumes of data (Big Data) due to serialization overhead.

To bridge this gap, Hayashi supports transparent zero-copy memory sharing using the Apache Arrow FFI format for complex array/dataframe exchanges. This

is implemented automatically by passing pointers to FFI structs (`FFI_Array` and `FFI_ArrowSchema`) in a zero-copy manner. However, developers must find the right balance:

- **JSON C String:** Offers maximum decoupling, safety, and binary version compatibility across different compiler versions. Best for control routines, small datasets, and general extensibility.
- **Apache Arrow Zero-Copy:** Offers maximum speed and zero memory allocation overhead. Best for data-intensive calculations, deep learning inference, or large econometric estimations. It requires strict memory lifecycle coordination (manual prevention of double-free bugs on the FFI boundary).

This duality represents an intentional design decision of the Hayashi language. Restricting zero-copy support strictly to DataFrames and homogeneous columns protects the ecosystem from binary compatibility (ABI) breakage in complex control structures and metadata, which are cheap to serialize via JSON C String. This guarantees long-term stability and ease of maintenance of the language without sacrificing performance on large data volume exchanges.

16.3.3 3. WebAssembly Plugins (WASM - For other languages)

For developers using other languages (such as C, Go, Zig, etc.), plugins must be compiled to WebAssembly (`.wasm`). They run inside a fully sandboxed environment via Hayashi's built-in WASM engine, ensuring absolute safety (any crash inside the plugin will not crash Hayashi itself) and instant cross-platform portability.

The WASM plugin must export `alloc(size: i32) -> i32` and `dealloc(ptr: i32, size: i32)` functions to manage passing JSON strings into WASM linear memory, and the plugin function should return an `i64` representing the pointer (in the upper 32 bits) and length (in the lower 32 bits) of the returned JSON string.

Part II

Applied Econometrics

Chapter 17

Descriptive Statistics and Moments

This chapter presents HAYASHI's descriptive statistics tools as expressions of probability theory. Each function maps directly to a theoretical concept, making the code readable both for those coming from mathematics and for those coming from Stata or R.

17.1 Expected value

The expected value $E(X)$ of a discrete random variable is the weighted average of its values. For observed data, the natural estimator is the sample mean:

$$\hat{E}(X) = \bar{X} = \frac{1}{n} \sum_{i=1}^n x_i$$

In HAYASHI, `mean` accepts a list or a DataFrame column:

```
1 // over a list
2 let xs = [2.0, 4.0, 4.0, 4.0, 5.0, 5.0, 7.0, 9.0]
3 display mean(xs)    // 5.0
4
5 // over a DataFrame column
6 load "data.csv" as df
7 display mean(df, income)
```

Listing 17.1: $E(X)$ over a list and over a column

17.2 Variance and standard deviation

The sample variance and sample standard deviation are:

$$\widehat{\text{Var}}(X) = s^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{X})^2 \quad s = \sqrt{s^2}$$

The divisor $n - 1$ (Bessel's correction) yields an unbiased estimator of σ^2 . All HAYASHI functions use this convention.

```
1 display variance(df, income)    // s2
2 display sd(df, income)         // s
```

Listing 17.2: Variance and standard deviation

Note

`sd` and `std` are aliases — both compute the sample standard deviation with divisor $n - 1$.

To obtain all moments at once, `summarize` returns a dictionary:

```
1 let stats = summarize(df, income)
2
3 display stats["mean"]           // mean
4 display stats["sd"]            // standard deviation
5 display stats["variance"]      // variance
6 display stats["min"]          // minimum
7 display stats["max"]          // maximum
8 display stats["N"]            // number of observations
```

17.3 Conditional moments

The conditional expected value $E(X \mid C)$ restricts the calculation to a subset of the sample without creating an auxiliary DataFrame. The `if =` option is available in all scalar aggregation functions:

```
1 // E(income | sector == "public")
2 display mean(df, income, if = sector == "public")
3
4 // Var(income | sector == "public")
5 display variance(df, income, if = sector == "public")
6
7 // SD(income | sector == "public")
8 display sd(df, income, if = sector == "public")
```

Listing 17.3: Conditional moments with `if =`

The condition is evaluated row by row on the DataFrame — any Boolean expression over columns is valid:

```
1 display mean(df, wage, if = educ >= 12 and sex == 1)
2 display sd(df, income, if = state == "TX" or state == "CA")
```

Listing 17.4: Compound conditions

Tip

`if` = does not modify the DataFrame. Use `filter` to create a permanent subsample; use `if` = for a one-off calculation.

17.4 Covariance

The sample covariance between X and Y measures their joint variation:

$$\widehat{\text{Cov}}(X, Y) = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{X})(y_i - \bar{Y})$$

```
1 display cov(df, educ, wage)
2
3 // conditional: formal workers only
4 display cov(df, educ, wage, if = formal == 1)
```

Listing 17.5: Covariance between two columns

Note

$\text{Cov}(X, X) = \text{Var}(X)$. To verify: `cov(df, x, x)` returns the same value as `variance(df, x)`.

17.5 Correlation

17.5.1 Scalar between two variables

Pearson's correlation coefficient normalises the covariance by the product of standard deviations:

$$\rho(X, Y) = \frac{\text{Cov}(X, Y)}{\text{SD}(X) \text{SD}(Y)} \in [-1, 1]$$

`corr_pair` returns this scalar directly:

```
1 display corr_pair(df, educ, wage)
2
3 // conditional: private sector
4 display corr_pair(df, educ, wage, if = sector == "private")
```

Listing 17.6: Scalar correlation

17.5.2 Correlation matrix

To explore all relationships among multiple variables, use `correlate`:

```
1 // all numeric variables
2 correlate(df)
3
4 // explicit subset
5 correlate(df, educ, exp, wage)
6
7 // with p-values (* p<0.1 ** p<0.05 *** p<0.01)
8 pwcorr(df, educ, exp, wage)
```

Listing 17.7: Full correlation matrix

Note

`correlate` prints the matrix but does not return a value — use `corr_pair` when you need the scalar in a calculation.

17.6 Median and quantiles

The median is the central value of the ordered distribution, less sensitive to extreme values than the mean.

```
1 display median(df, income)
2
3 // conditional
4 display median(df, income, if = region == "South")
```

Listing 17.8: Median

Quantiles generalise the median to any point $p \in [0, 1]$ of the distribution:

```
1 display quantile(df, income, 0.25) // 1st quartile
2 display quantile(df, income, 0.50) // median
```

```

3 display quantile(df, income, 0.75)    // 3rd quartile
4 display quantile(df, income, 0.90)    // 90th percentile

```

Listing 17.9: Quantiles

quantile accepts **if** =:

```

1 // 90th percentile of income among college-educated workers
2 display quantile(df, income, 0.90, if = educ >= 16)

```

Both functions accept lists directly:

```

1 let xs = [3.0, 1.0, 4.0, 1.0, 5.0, 9.0, 2.0, 6.0]
2 display median(xs)           // 3.5
3 display quantile(xs, 0.75)    // 5.25

```

17.7 Detailed summary

summarize with **detail=true** expands the return dictionary to include skewness, kurtosis, and the full percentile grid:

```

1 let s = summarize(df, income, detail=true)
2
3 display s["mean"]           // mean
4 display s["sd"]            // standard deviation
5 display s["p25"]           // 1st quartile
6 display s["p50"]           // median
7 display s["p75"]           // 3rd quartile
8 display s["p90"]           // 90th percentile
9 display s["p99"]           // 99th percentile
10 display s["skew"]          // skewness
11 display s["kurt"]          // kurtosis

```

Listing 17.10: summarize with detail

17.8 Quick reference

Notation	Hayashi	Forms
$E(X)$	<code>mean(df, x)</code>	<code>list / df / if =</code>
$\text{Var}(X)$	<code>variance(df, x)</code>	<code>list / df / if =</code>
$\text{SD}(X)$	<code>sd(df, x)</code>	<code>list / df / if =</code>
$\text{Cov}(X, Y)$	<code>cov(df, x, y)</code>	<code>df / if =</code>
$\rho(X, Y)$	<code>corr_pair(df, x, y)</code>	<code>df / if =</code>
$\text{Med}(X)$	<code>median(df, x)</code>	<code>list / df / if =</code>
$Q_p(X)$	<code>quantile(df, x, p)</code>	<code>list / df / if =</code>
<code>correlate(df, x, y, ...)</code>		correlation matrix (display)
<code>summarize(df, x)</code>		full dictionary of moments

17.9 Complete example

The snippet below illustrates the joint use of these functions in a typical exploratory analysis before estimating a model:

```

1 load "survey.csv" as df
2
3 // overview
4 summarize(df, income)
5 summarize(df, educ)
6 correlate(df, income, educ, exp)
7
8 // moments by group
9 display mean(df, income, if = sex == 1)      // men
10 display mean(df, income, if = sex == 0)      // women
11
12 display sd(df, income, if = sex == 1)
13 display sd(df, income, if = sex == 0)
14
15 // gap between median and mean: sign of skewness
16 display mean(df, income)
17 display median(df, income)
18
19 // linear association
20 display cov(df, educ, income)
21 display corr_pair(df, educ, income)

```

```
22  
23 // concentration at the top  
24 display quantile(df, income, 0.90)  
25 display quantile(df, income, 0.99)
```

Listing 17.11: Exploratory analysis — income and education

Chapter 18

Ordinary Least Squares

18.1 The model

Consider the linear regression model:

$$y = X\beta + \varepsilon$$

where y is an $n \times 1$ vector of observations of the dependent variable, X is an $n \times k$ matrix of regressors (the first column is identically 1, corresponding to the intercept), β is a $k \times 1$ vector of unknown parameters, and ε is an $n \times 1$ vector of errors.

Gauss-Markov Assumptions

1. **Linearity:** $y = X\beta + \varepsilon$.
2. **Full rank:** $\text{rank}(X) = k$ (no perfect multicollinearity).
3. **Strict exogeneity:** $E[\varepsilon \mid X] = 0$.
4. **Homoscedasticity:** $\text{Var}[\varepsilon \mid X] = \sigma^2 I_n$.
5. **No autocorrelation:** $\text{Cov}[\varepsilon_i, \varepsilon_j \mid X] = 0$ for $i \neq j$.

Under A1–A3 OLS is unbiased. Under A1–A5 it is BLUE (Gauss-Markov Theorem).

18.2 The estimator

The OLS estimator minimizes the sum of squared residuals:

$$\hat{\beta} = \arg \min_{\beta} (y - X\beta)'(y - X\beta)$$

The first-order condition $X'(y - X\beta) = 0$ — the *normal equations* — has an analytical solution:

$$\boxed{\hat{\beta} = (X'X)^{-1}X'y}$$

The residuals are $\hat{e} = y - X\hat{\beta} = M_X y$, where $M_X = I - X(X'X)^{-1}X'$ is the residual maker matrix (symmetric and idempotent).

18.3 Properties

Under A1–A3:

$$E[\hat{\beta}] = \beta$$

Under A1–A5, the conditional variance of the estimator is:

$$\text{Var}[\hat{\beta} | X] = \sigma^2(X'X)^{-1}$$

The unbiased estimator of the error variance is:

$$s^2 = \frac{\hat{e}'\hat{e}}{n - k}$$

and therefore $\widehat{\text{Var}}[\hat{\beta}] = s^2(X'X)^{-1}$.

18.4 DGP Verification

The most direct proof that the estimator is correct is to define a DGP (*data-generating process*) with known parameters, generate data from it and verify that OLS recovers them.

Choose the true parameters:

$$\beta_0 = 2,5 \quad \beta_1 = 1,8 \quad \beta_2 = -0,7$$

and generate $n = 1000$ observations *without* an error term ($\varepsilon = 0$):

$$y_i = 2,5 + 1,8 x_{1i} - 0,7 x_{2i}$$

With $\varepsilon = 0$ the Gauss-Markov assumptions are satisfied exactly, and OLS must

recover $(\beta_0, \beta_1, \beta_2)$ to machine precision.

```

1 seed(42)
2 input df
3 id
4 1
5 end
6 for i in 1..=10 { df = append(df, df) }
7 df |> filter(_n <= 1000)
8 generate df x1 = rnormal()
9 generate df x2 = rnormal()
10 generate df y = 2.5 + 1.8*x1 - 0.7*x2
11 ols(y ~ x1 + x2, df)

```

Listing 18.1: Noise-free DGP — numerical verification

Actual interpreter output:

```

===== OLS Regression Results =====
Dep. Variable:          y || R-squared:          1.0000
Model:                  OLS || Adj. R-squared:    1.0000
Covariance Type:        Non-Robust || F-statistic:  0.0000
No. Observations:       1000 || Prob (F-statistic): 1.0000e0
Df Residuals:           997 || Log-Likelihood:    31513.9419
AIC:                    -63021.8837 || BIC:         -63007.1605

-----
Variable |      coef |    std err |      t |    P>|t| | [0.025  0.975]
-----
const   |    2.5000 |    0.0000 | 6.17e+15 | 0.000 |    2.5000   2.5000
x1       |    1.8000 |    0.0000 | 3.36e+15 | 0.000 |    1.8000   1.8000
x2       |   -0.7000 |    0.0000 |-1.28e+15 | 0.000 |   -0.7000  -0.7000
=====

```

OLS recovers exactly $(\hat{\beta}_0, \hat{\beta}_1, \hat{\beta}_2) = (2.5; 1.8; -0.7)$. The zero standard error and $R^2 = 1$ are expected: without noise, the data lie exactly on the hyperplane defined by β , and the estimator is numerically exact to floating-point precision.

Note

The F -statistic appears as 0.0 and the p-value as 1.0 in this case — a numerical artifact of $0/0$ when $SSR = 0$. The t -statistics are on the order of 10^{15} for the same reason: finite numerator divided by a machine-precision denominator. In the presence of noise ($\varepsilon \neq 0$) all statistics take finite and interpretable values.

Misspecification effect

The same DGP allows us to verify what happens when x_2 is omitted. Two models are estimated and compared with `esttab`:

```
1 let m1 = ols(y ~ x1 + x2, df)
2 let m2 = ols(y ~ x1, df)
3 esttab(m1, m2)
```

Listing 18.2: Correct model vs. omitted x_2

	(1)	(2)
	OLS	OLS
x1	1.8000*** (0.0000)	1.7422*** (0.0224)
x2	-0.7000*** (0.0000)	
Constant	2.5000*** (0.0000)	2.1810*** (0.0129)
N	1000	1000
R ²	1.0000	0.8585
Adj. R ²	1.0000	0.8583
* p<0.10 ** p<0.05 *** p<0.01		

The coefficient on x_1 barely changes ($1.8000 \rightarrow 1.7422$) because x_1 and x_2 are generated independently — the sample correlation is close to zero. The constant, however, is distorted from 2.5000 to 2.1810 because it absorbs $\hat{\beta}_2 \bar{x}_2$, and the sample mean of x_2 is not exactly zero even with $n = 1000$.

Note

Omitted variable bias (OVB) *requires correlation* between the included and the omitted regressor. Without correlation, omission only inflates standard errors and reduces R^2 (loss of efficiency, not of consistency). For the formal treatment of endogeneity and instruments, see Chapter 19.

18.5 Standard errors

Classical

Assumes homoscedasticity. The estimated covariance matrix is $s^2(X'X)^{-1}$ and the standard error of coefficient j is $\text{se}_j = s \sqrt{[(X'X)^{-1}]_{jj}}$.

Heteroscedasticity-robust

When $\text{Var}[\varepsilon_i | x_i] = \sigma_i^2$ (heteroscedasticity), the sandwich estimator provides valid inference.

HC1 (White with degrees-of-freedom correction):

$$\hat{V}_{\text{HC1}} = \frac{n}{n-k} (X'X)^{-1} \left(\sum_{i=1}^n \hat{\varepsilon}_i^2 x_i x_i' \right) (X'X)^{-1}$$

HC3 (more conservative, recommended in small samples):

$$\hat{V}_{\text{HC3}} = (X'X)^{-1} \left(\sum_{i=1}^n \frac{\hat{\varepsilon}_i^2}{(1-h_{ii})^2} x_i x_i' \right) (X'X)^{-1}$$

where $h_{ii} = x_i'(X'X)^{-1}x_i$ is the *leverage* of observation i .

Clustered

When errors are correlated within groups $g = 1, \dots, G$:

$$\hat{V}_{\text{cl}} = \frac{G(n-1)}{(G-1)(n-k)} (X'X)^{-1} \left(\sum_{g=1}^G X_g' \hat{\varepsilon}_g \hat{\varepsilon}_g' X_g \right) (X'X)^{-1}$$

18.6 Inference

t -test on a single coefficient

Under $H_0: \beta_j = \beta_j^0$:

$$t_j = \frac{\hat{\beta}_j - \beta_j^0}{\text{se}(\hat{\beta}_j)} \xrightarrow{d} t_{n-k}$$

F-test on linear restrictions

For $R\beta = r$ with q restrictions:

$$F = \frac{(R\hat{\beta} - r)' [R(X'X)^{-1}R']^{-1} (R\hat{\beta} - r)}{q s^2} \xrightarrow{d} F_{q, n-k}$$

Note

With robust or clustered errors, $\hat{V}$ replaces $s^2(X'X)^{-1}$ in the computation of the F -test.

18.7 Syntax

```
1 ols(formula, df)
2 ols(formula, df, cov=type)
3 ols(formula, df, cov=cluster, cluster=column)
4 ols(formula, df, if = condition)
```

Alias: `reg()` is equivalent to `ols()`.

18.8 Formula

The formula follows the standard notation $y \sim x_1 + x_2 + \dots$:

```
1 input df
2 Y X
3 10 2
4 12 3
5 8 1
6 15 5
7 11 2
8 14 4
9 end
10
11 ols(Y ~ X, df)
```

Listing 18.3: Basic OLS

The intercept is included automatically. For multiple variables:

```
1 ols(Y ~ X1 + X2 + X3, df)
```

The formula can be passed as a string:

```
1 let formula = "Y ~ X1 + X2"
2 ols(formula, df)
```

18.9 Standard error options

Option	Formula
(default)	$s^2(X'X)^{-1}$
cov=robust	HC1 (White sandwich)
cov=HC1	same
cov=HC3	HC3 (leverage-adjusted)
cov=cluster, cluster=col	clustered by col

```
1 ols(Y ~ X, df) // classical
2 ols(Y ~ X, df, cov=robust) // HC1
3 ols(Y ~ X, df, cov=HC3) // HC3
4 ols(Y ~ X, df, cov=cluster, cluster=id) // clustered
```

Listing 18.4: Standard error variants

18.10 Conditional sample

The `if` = option restricts estimation to a subset without modifying the DataFrame:

```
1 ols(Y ~ X, df, if = group == 1)
```

18.11 Capturing the result

```
1 let m = ols(Y ~ X, df)
2 display m
```

When assigned, `ols()` does not print the table — it only stores the model.

18.12 Post-estimation

18.12.1 predict

```
1 let m = ols(Y ~ X, df)
2
3 //  $\hat{y} = X\hat{\beta}$ 
4 let yhat = predict(m, "xb")
5 //  $\hat{e} = y - X\hat{\beta}$ 
6 let ehat = predict(m, "residuals")
```

Accepted aliases: "fitted" for fitted values; "resid" or "e" for residuals.

18.12.2 vif

The VIF of regressor j is $VIF_j = 1/(1 - R_j^2)$, where R_j^2 is the R^2 of the regression of x_j on the remaining regressors. Values above 10 indicate severe multicollinearity.

```
1 let m = ols(Y ~ X1 + X2 + X3, df)
2 vif(m)
```

18.12.3 test

F -test of linear restrictions $H_0 : R\beta = r$:

```
1 let m = ols(Y ~ X1 + X2 + X3, df)
2 test(m, X2 = 0, X3 = 0) //  $H_0: X2 = X3 = 0$ 
```

18.12.4 lincom

Linear combination $c'\hat{\beta}$ with standard error $\sqrt{c'\hat{V}c}$:

```
1 let m = ols(Y ~ X1 + X2, df)
2 lincom(m, X1 + X2)
```

18.12.5 margins

Average marginal effects. In linear OLS $\partial E[y]/\partial x_j = \beta_j$, equivalent to the coefficients.

```
1 let m = ols(Y ~ X, df)
2 margins(m)
```

18.13 Results table with `esttab`

```
1 let m1 = ols(Y ~ X, df)
2 let m2 = ols(Y ~ X, df, cov=robust)
3 let m3 = ols(Y ~ X1 + X2, df)
4
5 esttab(m1, m2, m3)
```

Listing 18.5: Comparing specifications

```
1 eststo(ols(Y ~ X, df))
2 eststo(ols(Y ~ X1 + X2, df))
3 esttab()
```

18.14 Bootstrap

Bootstrap standard errors — a nonparametric alternative:

```
1 bootstrap(ols, Y ~ X, df, n=1000)
2 bootse(m, n=500)
```


Chapter 19

Instrumental Variables

19.1 The endogeneity problem

Consider the structural model:

$$y = X_1\beta_1 + X_2\beta_2 + \varepsilon$$

where X_1 are exogenous regressors ($n \times k_1$) and X_2 are potentially endogenous regressors ($n \times k_2$). OLS is biased and inconsistent when:

$$E[\varepsilon \mid X_2] \neq 0 \iff \text{plim } \hat{\beta}_{\text{OLS}} \neq \beta$$

The most common causes are:

- **Omitted variable** correlated with X_2 : if $y = X\beta + W\gamma + u$ and W is omitted, then $\varepsilon = W\gamma + u$ and $\text{Cov}(X_2, \varepsilon) \neq 0$.
- **Simultaneity**: y and X_2 are determined jointly.
- **Classical measurement error**: $X_2 = X_2^* + \nu$, with X_2^* the true value; the attenuation factor is $\text{plim } \hat{\beta} = \beta \cdot \sigma_{X^*}^2 / \sigma_X^2 < \beta$.

19.2 Identification conditions

Let Z be the $n \times L$ matrix of instruments (including X_1 as included instruments). A valid instrument satisfies two conditions:

1. **Relevance**: $\text{rank}(E[z_i x_i']) = k$, where $k = k_1 + k_2$. In practice: X_2 is correlated with Z in the first stage.

2. **Exogeneity (exclusion restriction):** $E[z_i \varepsilon_i] = 0$. The excluded instruments affect y *only* through X_2 .

The model is identified when $L \geq k$ (order condition). For $L = k$ (exact identification) the IV estimator is unique; for $L > k$ (overidentification) 2SLS is efficient in the class of IV estimators.

19.3 The 2SLS estimator

Define the projection matrix onto the column space of Z :

$$P_Z = Z(Z'Z)^{-1}Z'$$

The Two-Stage Least Squares (2SLS) estimator is:

$$\hat{\beta}_{2SLS} = (X'P_ZX)^{-1}X'P_Zy$$

Two-stage interpretation

First stage — projects each column of X_2 onto Z :

$$X_2 = Z\hat{\Pi} + \hat{V}, \quad \hat{\Pi} = (Z'Z)^{-1}Z'X_2$$

Second stage — regresses y on $\hat{X} = P_ZX = [X_1, \hat{X}_2]$:

$$\hat{\beta}_{2SLS} = (\hat{X}'X)^{-1}\hat{X}'y$$

Warning

Never run the two stages manually as sequential OLS. The standard errors from the second stage will be wrong because $\hat{X}_2$ is estimated. Always use `iv()`, which corrects the variance automatically.

19.4 Asymptotic variance

Under exogeneity and relevance, $\hat{\beta}_{2SLS}$ is consistent and asymptotically normal:

$$\sqrt{n}(\hat{\beta}_{2SLS} - \beta) \xrightarrow{d} \mathcal{N}(0, \sigma^2 Q_{XZ}^{-1} Q_{ZZ} Q_{XZ}^{-1'})$$

where $Q_{XZ} = E[x_i z_i']$ and $Q_{ZZ} = E[z_i z_i']$.

The variance estimator is:

$$\widehat{\text{Var}}[\hat{\beta}_{2\text{SLS}}] = s_{\text{IV}}^2 (X' P_Z X)^{-1}, \quad s_{\text{IV}}^2 = \frac{\hat{e}_{2\text{SLS}}' \hat{e}_{2\text{SLS}}}{n - k}$$

The same sandwich corrections from OLS (HC1, HC3, clustered) apply by replacing X with $\hat{X} = P_Z X$.

19.5 Weak instruments

An instrument is weak when its correlation with X_2 is low. The bias of 2SLS relative to OLS is approximately:

$$\frac{\text{Bias}_{2\text{SLS}}}{\text{Bias}_{\text{OLS}}} \approx \frac{1}{F + 1}$$

where F is the F -statistic from the first stage. For $F < 10$ (the rule of thumb from Staiger & Stock, 1997) the instrument is considered weak.

The **Cragg-Donald statistic** generalizes to multiple endogenous regressors:

$$CD = \frac{n - L}{L - k_1} \cdot \lambda_{\min}$$

where $\lambda_{\min}$ is the smallest eigenvalue of the concentration matrix. Stock & Yogo (2005) tabulate critical values for maximum relative bias of 5%, 10%, 20%, and 30% relative to OLS.

19.6 DGP Verification

The same method as Chapter 18 applies to IV: a DGP with known endogeneity is defined, OLS and IV are estimated, and it is verified that IV corrects the bias.

DGP structure

Let z and v be independent random variables $\sim \mathcal{N}(0, 1)$. Define ε as the *residual* from projecting v onto $[1, z]$:

$$\varepsilon = v - \delta_0 - \delta_1 z, \quad \delta_1 = \frac{\text{Cov}_n(z, v)}{\text{Var}_n(z)}, \quad \delta_0 = \bar{v} - \delta_1 \bar{z}$$

By construction $\sum \varepsilon_i = 0$ and $\sum z_i \varepsilon_i = 0$, i.e., $Z' \varepsilon = 0$ *exactly* in sample. Then:

$$x = 0,8 z + \varepsilon \quad y = 1,5 + 2,0 x + 0,7 \varepsilon$$

The structural error $0,7\varepsilon$ creates endogeneity ($\text{Cov}(x, 0,7\varepsilon) \neq 0$), but since $Z'\varepsilon = 0$ exactly, the IV moment conditions are satisfied without sampling noise. Result: IV recovers (β_0, β_1) to machine precision — the same as OLS did without noise in Chapter 18.

The OLS bias is predictable:

$$\text{plim } \hat{\beta}_{\text{OLS}} = \beta_1 + \frac{\text{Cov}(x, \varepsilon)}{\text{Var}(x)} \approx 2,0 + \frac{0,7}{0,64 + 1} \approx 2,43$$

Script

```
1 seed(42)
2 input df
3 id
4 1
5 end
6 for i in 1..=10 { df = append(df, df) }
7 df |> filter(_n <= 1000)
8 generate df z = rnormal()
9 generate df v = rnormal()
10 let d1 = cov(df, z, v) / variance(df, z)
11 let d0 = mean(df, v) - d1 * mean(df, z)
12 generate df eps = v - d0 - d1*z
13 generate df x = 0.8*z + eps
14 generate df y = 1.5 + 2.0*x + 0.7*eps
15 let m_ols = ols(y ~ x, df)
16 let m_iv = iv(y ~ x, ~ z, df)
17 esttab(m_ols, m_iv)
```

Listing 19.1: DGP with endogeneity — OLS vs. IV

Actual interpreter output:

	(1)	(2)
	OLS	IV/2SLS
x	2.4303*** (0.0108)	2.0000*** (0.0280)
const		1.5000*** (0.0129)
Constant	1.3273*** (0.0058)	

N	1000	1000
R ²	0.9807	
Adj. R ²	0.9807	

* p<0.10 ** p<0.05 *** p<0.01

IV recovers $(\hat{\beta}_0, \hat{\beta}_1) = (1,5000, 2,0000)$ to machine precision. OLS overestimates β_1 by +0,43 (2.4303 vs. 2.0000), consistent with the predicted asymptotic bias of +0,43. R^2 is not reported for IV since it has no standard interpretation under endogeneity.

19.7 Syntax

```
1 iv(structural_formula, instrument_formula, df)
2 iv(structural_formula, instrument_formula, df, cov=type)
```

The *structural formula* specifies the equation of interest. The *instrument formula* lists the excluded instruments after ~:

```
1 // structural equation: y ~ x (x is endogenous)
2 // excluded instrument: z
3 iv(y ~ x, ~ z, df)
```

Listing 19.2: Basic IV — one instrument

For multiple endogenous regressors and instruments:

```
1 iv(y ~ x1 + x2, ~ z1 + z2, df)
```

Note

Included exogenous regressors (X_1) do not need to be listed in the instrument formula — they are added automatically as their own instruments.

19.8 Standard error options

The same `cov=` options available in OLS apply to IV:

```
1 iv(y ~ x, ~ z, df) // classical
2 iv(y ~ x, ~ z, df, cov=robust) // HC1
3 iv(y ~ x, ~ z, df, cov=HC3) // HC3
4 iv(y ~ x, ~ z, df, cov=cluster, cluster=id) // clustered
```

19.9 Capturing the result

```
1 let m_iv = iv(y ~ x, ~ z, df)
2 display m_iv
```

19.10 Weak instrument diagnostics

```
1 weak_iv(y ~ x, ~ z, df)
```

Listing 19.3: Weak instrument test

The function reports:

- First-stage F -statistic (per endogenous variable)
- Cragg-Donald statistic CD
- Stock & Yogo (2005) critical values for 5%–30% relative bias
- Reference: $F > 10$ (Staiger & Stock, 1997)

19.11 Fitted values

```
1 let m_iv = iv(y ~ x, ~ z, df)
2 //  $\hat{y} = X\hat{\beta}_{\mathrm{2SLS}}$ 
3 let yhat = predict(m_iv, "xb")
```

19.12 Comparison table

The typical workflow quantifies the endogeneity bias by comparing OLS and IV:

```
1 load "data.csv" as df
2
3 let m_ols = ols(wage ~ educ, df)
4 let m_iv = iv(wage ~ educ, ~ distance, df, cov=robust)
5
6 // first-stage diagnostics
7 weak_iv(wage ~ educ, ~ distance, df)
8
9 // comparison table
```

```
10 esttab(m_ols, m_iv)
```

Listing 19.4: Complete IV analysis workflow

If $\hat{\beta}^{\text{IV}} > \hat{\beta}^{\text{OLS}}$, the OLS bias is negative — which is consistent with attenuation from measurement error or with an omitted variable that depresses both X_2 and y .

Chapter 20

Panel Data

20.1 The model

A panel observes n individuals over T periods. The general model is:

$$y_{it} = \alpha_i + X_{it}\beta + \varepsilon_{it}, \quad i = 1, \dots, n; \quad t = 1, \dots, T$$

where α_i is the **unobserved individual effect** — time-fixed characteristics that differ across individuals (ability, culture, location). Cross-sectional or pooled OLS ignores α_i , treating it as part of the error:

$$\text{plim } \hat{\beta}_{\text{OLS}} = \beta + \frac{\text{Cov}(x_{it}, \alpha_i)}{\text{Var}(x_{it})}$$

The bias disappears only if $\text{Cov}(x_{it}, \alpha_i) = 0$ — a strong assumption in most applications.

20.2 Fixed Effects

The **within** estimator eliminates α_i by subtracting each individual's time mean:

$$\tilde{y}_{it} = y_{it} - \bar{y}_i, \quad \tilde{X}_{it} = X_{it} - \bar{X}_i$$

The Fixed Effects estimator is then OLS on the demeaned variables:

$$\boxed{\hat{\beta}_{\text{FE}} = (\tilde{X}'\tilde{X})^{-1}\tilde{X}'\tilde{y}}$$

Because the within transformation annihilates α_i for any value of $\text{Cov}(x_{it}, \alpha_i)$, the estimator is consistent even under endogeneity of the individual effect. The

variance is:

$$\widehat{\text{Var}}[\hat{\beta}_{\text{FE}}] = s_{\text{FE}}^2(\tilde{X}'\tilde{X})^{-1}, \quad s_{\text{FE}}^2 = \frac{\hat{\varepsilon}'\hat{\varepsilon}}{nT - n - k}$$

Warning

Variables that do not vary over time (sex, race, region of birth) are eliminated by the within transformation. FE cannot estimate their effect. Use RE or hybrid estimators in those cases.

20.3 Random Effects

If $\alpha_i \sim \text{IID}(0, \sigma_\alpha^2)$ and $\text{Cov}(x_{it}, \alpha_i) = 0$, then α_i is part of the composite error $v_{it} = \alpha_i + \varepsilon_{it}$. The error structure implies GLS with quasi-demeaning:

$$y_{it}^* = y_{it} - \theta \bar{y}_i, \quad \theta = 1 - \frac{\sigma_\varepsilon}{\sqrt{\sigma_\varepsilon^2 + T\sigma_\alpha^2}} \in [0, 1)$$

The RE estimator is OLS on y_{it}^* and X_{it}^* . It is more efficient than FE under $H_0 : \text{Cov}(x_{it}, \alpha_i) = 0$, but *inconsistent* when that assumption fails.

20.4 Hausman Test

The Hausman test compares FE (always consistent) with RE (consistent under H_0 , efficient):

$$H_0 : \text{Cov}(x_{it}, \alpha_i) = 0 \implies \text{plim}(\hat{\beta}_{\text{FE}} - \hat{\beta}_{\text{RE}}) = 0$$

The test statistic is:

$$H = (\hat{\beta}_{\text{FE}} - \hat{\beta}_{\text{RE}})' [\widehat{\text{Var}}(\hat{\beta}_{\text{FE}}) - \widehat{\text{Var}}(\hat{\beta}_{\text{RE}})]^{-1} (\hat{\beta}_{\text{FE}} - \hat{\beta}_{\text{RE}}) \xrightarrow{d} \chi^2(k)$$

Rejecting H_0 indicates that RE is inconsistent — use FE.

20.5 DGP Verification

Exact recovery (no noise)

The DGP uses 100 individuals $\times$ 10 periods ($n = 1000$). The fixed effect $\alpha_i = 0,05 \cdot i$ is deterministic and correlated with x :

$$x_{it} = \alpha_i + u_{it}, \quad u \sim \mathcal{N}(0, 1) \quad y_{it} = 1,0 + 2,0 x_{it} + \alpha_i \quad (\varepsilon = 0)$$

Without idiosyncratic noise the within transformation eliminates α_i exactly, and FE recovers β to machine precision:

```

1 seed(42)
2 input df
3 id
4 1
5 end
6 for i in 1..=10 { df = append(df, df) }
7 df |> filter(_n <= 1000)
8 generate df id      = (_n - 1) % 100 + 1
9 generate df alpha = id * 0.05
10 generate df u      = rnormal()
11 generate df x      = alpha + u
12 generate df y      = 1.0 + 2.0*x + alpha
13 let m_ols = ols(y ~ x, df)
14 let m_fe  = fe(y ~ x, df, id=id)
15 esttab(m_ols, m_fe)

```

Listing 20.1: FE with correlated fixed effect — no noise

	(1)	(2)
	OLS	FE
x	2.9663*** (0.0060)	2.0000*** (0.0000)
Constant	0.6002*** (0.0202)	
N	1000	1000
R ²	0.9959	
Adj. R ²	0.9959	
* p<0.10 ** p<0.05 *** p<0.01		

OLS overestimates β by +0.97 because $\text{Cov}(x, \alpha) = \text{Var}(\alpha) > 0$. FE recovers 2,0000 exactly.

Hausman with noise

With idiosyncratic noise $\varepsilon \sim \mathcal{N}(0, 1)$, FE remains consistent and RE is inconsistent (since α_i is still correlated with x). Hausman should reject H_0 :

```

1 // same DGP; adds idiosyncratic noise
2 generate df e = rnormal()
3 generate df y2 = 1.0 + 2.0*x + alpha + e
4 let m_fe2 = fe(y2 ~ x, df, id=id)
5 let m_re2 = re(y2 ~ x, df, id=id)
6 esttab(m_fe2, m_re2)
7 hausman(m_fe2, m_re2)

```

Listing 20.2: Hausman — FE vs RE with correlated effect

	(1)	(2)
	FE	RE
x	2.0977*** (0.0338)	2.9347*** (0.0118)
const		1.1930*** (0.0399)
N	1000	0
* p<0.10 ** p<0.05 *** p<0.01		
Hausman Test: FE vs RE		
H: individual effects uncorrelated with regressors		
Common Coefficients		
Variable	_FE	_RE Δ
x	2.0977	2.9347 -0.8369
Statistic		
$\chi^2(1) = 698.1711$ p = 0.0000 ***		
Conclusion		
Rejects H → use FIXED EFFECTS (RE may be inconsistent)		

RE absorbs the effect of α_i into the coefficient on x (2.9347), replicating the OLS bias. Hausman rejects H_0 with $p < 0.001$ — the endogeneity of the individual effect is detected.¹

¹FE recovers 2.0977, not 2.0000, because the within transformation eliminates α_i but not ε_{it} . With idiosyncratic noise present, the estimator is consistent but not exact — the same reason OLS and IV deviate from their true parameters when $\varepsilon \neq 0$.

20.6 Syntax

```

1 fe(formula, df, id=col)
2 re(formula, df, id=col)
3 hausman(fe_model, re_model)

```

The `id` column identifies the individual. FE automatically removes the intercept (absorbed by the individual effects).

```

1 load "panel.csv" as df
2
3 let m_fe = fe(wage ~ educ + exp, df, id=id)
4 let m_re = re(wage ~ educ + exp, df, id=id)
5
6 esttab(m_fe, m_re)
7 hausman(m_fe, m_re)

```

Listing 20.3: Complete workflow

20.7 Standard error options

```

1 fe(y ~ x, df, id=id) // classical
2 fe(y ~ x, df, id=id, cov=robust) // HC1
3 // cluster by individual
4 fe(y ~ x, df, id=id, cov=cluster, cluster=id)

```

Note

In panels, errors clustered by individual (`cluster=id`) are the recommended default: they correct heteroscedasticity and serial correlation within the individual simultaneously.

20.8 Capturing results

```

1 let m = fe(y ~ x, df, id=id)
2 display m
3 let yhat = predict(m, "xb")

```


Chapter 21

Difference-in-Differences

21.1 The model

Difference-in-Differences (DiD) estimates the causal effect of a treatment by comparing the time variation of the treated group with that of the control group:

$$\hat{\delta}_{\text{DiD}} = \underbrace{(\bar{Y}_{1,\text{post}} - \bar{Y}_{1,\text{pre}})}_{\text{treated variation}} - \underbrace{(\bar{Y}_{0,\text{post}} - \bar{Y}_{0,\text{pre}})}_{\text{control variation}}$$

The 2×2 table summarizes the four groups:

	Pre	Post
Control ($D = 0$)	μ_{00}	μ_{01}
Treated ($D = 1$)	μ_{10}	μ_{11}

$$\hat{\delta} = (\mu_{11} - \mu_{10}) - (\mu_{01} - \mu_{00})$$

21.2 Regression form

The DiD estimator is equivalent to the coefficient on the interaction term in:

$$y_{it} = \beta_0 + \beta_1 \text{Post}_t + \beta_2 \text{Treated}_i + \delta (\text{Treated}_i \times \text{Post}_t) + \varepsilon_{it}$$

where $\hat{\delta} = \hat{\beta}_3$ is the Average Treatment effect on the Treated (ATT).

21.3 Parallel trends assumption

Identification of the ATT rests on:

$$E[Y_{it}^{(0)} - Y_{is}^{(0)} \mid D_i = 1] = E[Y_{it}^{(0)} - Y_{is}^{(0)} \mid D_i = 0] \quad \forall t \neq s$$

In the absence of treatment, treated and control groups would have the *same* time trend. The assumption is not testable in the post-treatment period, but can be checked in pre-treatment periods when more than one period is available.

Warning

The parallel trends assumption is stronger than it appears. Groups with structurally different growth trends (e.g. expanding regions vs. stagnant regions) violate the assumption even if their pre-treatment levels are similar.

21.4 DGP Verification

Exact recovery (no noise)

The DGP uses 500 individuals $\times$ 2 periods ($n = 1000$). The first 250 are treated, the rest are control. The true effect is $\delta = 3,0$:

$$y_{it} = 1,0 + 0,5 \cdot \text{Post}_t + 3,0 \cdot (D_i \times \text{Post}_t)$$

The four cell means are $\mu_{00} = 1,0$, $\mu_{01} = 1,5$, $\mu_{10} = 1,0$ and $\mu_{11} = 4,5$, so that $\hat{\delta} = (4,5 - 1,0) - (1,5 - 1,0) = 3,0$ exactly.

```

1 seed(42)
2 input df
3 id
4 1
5 end
6 for i in 1..=10 { df = append(df, df) }
7 df |> filter(_n <= 1000)
8 generate df id      = (_n - 1) % 500 + 1
9 generate df post    = _n > 500
10 generate df treated = id <= 250
11 generate df att     = treated * post
12 generate df y       = 1.0 + 0.5*post + 3.0*att
13 did(y ~ treated + post, df)

```

Listing 21.1: Noise-free DiD DGP — exact recovery

===== Difference-in-Differences (2x2 Canonical) =====

```

ATT (Effect):                3.0000 || R-squared:
1.0000
Std. Error:                  0.0000 || P-value:
0.0000
t-statistic:                  inf || Observations:
1000

----- Group Means -----
Control Group (Pre):         1.0000 | Control Group (Post):         1.5000
Treated Group (Pre):         1.0000 | Treated Group (Post):         4.5000
Parallel Trend Diff:         0.0000 (If > 0, Control grew more than Treated Pre
=====

```

With idiosyncratic noise

Adding $\varepsilon \sim \mathcal{N}(0, 1)$, DiD remains consistent:

```

1 generate df e = rnormal()
2 generate df y = 1.0 + 0.5*post + 3.0*att + e
3 did(y ~ treated + post, df)

```

Listing 21.2: DiD DGP with noise

```

===== Difference-in-Differences (2x2 Canonical) =====
ATT (Effect):                3.0114 || R-squared:
0.9645
Std. Error:                  0.0358 || P-value:
0.0000
t-statistic:                  84.0942 || Observations:
1000

----- Group Means -----
Control Group (Pre):         1.4842 | Control Group (Post):         2.0099
Treated Group (Pre):         1.4880 | Treated Group (Post):         5.0251
Parallel Trend Diff:         0.0000 (If > 0, Control grew more than Treated Pre
=====

```

The ATT of 3,0114 is close to the true 3,0; the deviation is sampling variance. The *Parallel Trend Diff* near zero confirms that the parallel trends assumption is satisfied by construction.

21.5 Syntax

```
1 did(outcome ~ treated + post, df)
2 did(outcome ~ treated + post, df, cov=robust)
```

The formula requires exactly two variables on the right-hand side: the group indicator (`treated`) and the period indicator (`post`). `HAYASHI` computes the interaction internally.

```
1 load "data.csv" as df
2
3 // create indicators
4 generate df post      = year >= 2020
5 generate df treated = state == "RS"
6
7 let m = did(wage ~ treated + post, df, cov=robust)
8 display m
```

Listing 21.3: Typical DiD workflow

21.6 Capturing the result

```
1 let m = did(y ~ treated + post, df)
2 display m
```

Note

For event studies with multiple pre- and post-treatment periods, the canonical 2×2 DiD may be insufficient. In those cases, estimate the regression model with period-by-period interactions and verify the absence of pre-treatment effects as a test of parallel trends.

Chapter 22

Logit and Probit

22.1 The latent variable model

When the dependent variable is binary ($y \in \{0, 1\}$), the linear model produces predictions outside the $[0, 1]$ interval and heteroscedastic errors by construction. The solution is to model the probability of $y = 1$ via a cumulative distribution function:

$$P(y_i = 1 \mid x_i) = F(x_i' \beta)$$

The motivation comes from the latent variable model:

$$y_i^* = x_i' \beta + \varepsilon_i, \quad y_i = \mathbf{1}[y_i^* > 0]$$

The distribution of ε determines the model:

Model	Distribution of ε	$F(\cdot)$
Logit	Logistic($0, \pi^2/3$)	$\Lambda(z) = \frac{1}{1 + e^{-z}}$
Probit	Normal($0, 1$)	$\Phi(z) = \int_{-\infty}^z \phi(t) dt$

Estimation is by Maximum Likelihood (MLE). The log-likelihood is:

$$\ell(\beta) = \sum_{i=1}^n \left[y_i \ln F(x_i' \beta) + (1 - y_i) \ln(1 - F(x_i' \beta)) \right]$$

22.2 Interpretation of coefficients

Logit and Probit coefficients **are not marginal effects** — they are parameters of the latent variable, without a directly interpretable unit. For continuous x_j , the marginal effect on the probability is:

$$\frac{\partial P(y = 1 \mid x)}{\partial x_j} = f(x' \beta) \beta_j$$

where $f = F'$ is the density of the chosen distribution. Since $f(x' \beta)$ varies across observations, the **Average Marginal Effect** (AME) is reported:

$$\text{AME}_j = \frac{1}{n} \sum_{i=1}^n f(x'_i \beta) \beta_j$$

Note

The relationship between Logit and Probit coefficients is approximately:

$$\hat{\beta}_{\text{Logit}} \approx \frac{\pi}{\sqrt{3}} \hat{\beta}_{\text{Probit}} \approx 1.81 \hat{\beta}_{\text{Probit}}$$

The AMEs of the two models are typically very close, even with coefficients on different scales.

22.3 DGP Verification

MLE consistency

Unlike the linear models in previous chapters, Logit and Probit *do not admit exact recovery* of parameters. The binary variable y is generated by Bernoulli sampling — there is irreducible randomness even without adding extra noise. What is verified here is **consistency**: with $n = 1000$ the MLE should converge close to the true parameters.

The DGP uses the logistic parametrization with $\beta_0 = -1,0$ and $\beta_1 = 2,0$:

$$x_i \sim \mathcal{N}(0, 1), \quad p_i = \Lambda(-1,0 + 2,0 x_i), \quad y_i \sim \text{Bernoulli}(p_i)$$

Individual Bernoulli is generated by $y_i = \mathbf{1}[u_i < p_i]$, with $u_i \sim \text{Uniform}(0, 1)$:

```
1 seed(42)
2 input df
3 id
4 1
5 end
6 for i in 1..=10 { df = append(df, df) }
7 df |> filter(_n <= 1000)
8 generate df x = rnormal()
9 generate df lp = -1.0 + 2.0*x
```

```

10 generate df p = 1 / (1 + exp(-lp))
11 generate df u = runiform()
12 generate df y = u < p
13 let m_l = logit(y ~ x, df)
14 let m_p = probit(y ~ x, df)
15 esttab(m_l, m_p)

```

Listing 22.1: Logistic DGP — Logit and Probit simultaneously

The true parameters are $(\beta_0, \beta_1) = (-1, 0; 2, 0)$ for the Logit DGP with $x_i \sim \mathcal{N}(0, 1)$. The Probit estimates are on the normal latent-index scale, so they should not be numerically identical to the Logit coefficients even when both models describe the same qualitative relationship.

Average marginal effects

Coefficients are only parameters of the latent variable. For economic interpretation, use `margins`:

```

1 margins(m_l, df)

```

Listing 22.2: Logit AME

The marginal effect is positive and varies with the fitted probability; it is reported by `margins(m_l, df)` on the probability scale.

22.4 Syntax

```

1 logit(y ~ x1 + x2, df)
2 probit(y ~ x1 + x2, df)
3 margins(model, df)

```

```

1 load "data.csv" as df
2
3 let m_l = logit(employed ~ educ + exp + sex, df)
4 let m_p = probit(employed ~ educ + exp + sex, df)
5
6 esttab(m_l, m_p)
7 margins(m_l, df)

```

Listing 22.3: Typical workflow — discrete choice

22.5 Predicted probabilities

```
1 let m = logit(y ~ x, df)
2 let phat = predict(m, "pr")      // P(y=1|x) for each obs
3 let yhat = predict(m, "class")  // predicted class (0 or 1)
```

Note

Use `predict(m, "class")` with default threshold of 0.5. In imbalanced data (low prevalence), adjust the threshold with `predict(m, "class", threshold=0.3)`.

Chapter 23

Quantile Regression

23.1 The model

OLS estimates $E[y \mid x]$ — the conditional mean. Quantile Regression (QR) estimates $Q_\tau[y \mid x]$ — the τ -th conditional quantile for any $\tau \in (0, 1)$:

$$Q_\tau(y \mid x) = x' \beta(\tau)$$

The vector $\beta(\tau)$ varies with τ : each quantile has its own set of coefficients. A single model can describe the entire conditional distribution of y .

23.2 The estimator

The estimator minimizes the **asymmetric loss function** (check function):

$$\hat{\beta}(\tau) = \arg \min_{\beta} \sum_{i=1}^n \rho_\tau(y_i - x_i' \beta)$$

where

$$\rho_\tau(u) = \begin{cases} \tau u & \text{if } u \geq 0 \\ (\tau - 1) u & \text{if } u < 0 \end{cases}$$

For $\tau = 0,5$: $\rho_{0,5}(u) = |u|/2$ — minimization of the sum of absolute deviations (LAD/median). For $\tau \neq 0,5$, positive and negative residuals receive asymmetric weights, shifting the estimate toward the τ -th quantile.

Note

Unlike OLS, QR has no closed-form solution — the problem is a linear program solved by the simplex method or interior-point methods. Therefore, standard inference uses bootstrap (`boot=`).

23.3 Advantages over OLS

- **Heteroscedasticity:** when $\text{Var}[y \mid x]$ varies with x , the slopes $\beta_j(\tau)$ differ across quantiles. OLS captures only the implicit weighted average.
- **Robustness:** the check function is linear in the residuals, not quadratic. Outliers in y influence $\hat{\beta}(\tau)$ less than $\hat{\beta}_{\text{OLS}}$.
- **Skewed distribution:** wages, income and asset prices have heavy right tails. QR describes effects in the tails without assumptions about the shape of the distribution.

23.4 DGP Verification

The DGP below introduces multiplicative heteroscedasticity: the dispersion of y grows with x , so that the effect of x is larger in upper quantiles than in lower ones.

$$y_i = 2,0 + 1,5 x_i + (1,0 + 0,8 x_i) \varepsilon_i, \quad \varepsilon_i \sim \mathcal{N}(0, 1)$$

The conditional quantile $Q_\tau(y \mid x)$ is increasing in x , and the slope $\partial Q_\tau / \partial x$ increases with τ : individuals in upper quantiles are more sensitive to x than those in the lower quantile.

```

1 seed(42)
2 input df
3 id
4 1
5 end
6 for i in 1..=10 { df = append(df, df) }
7 df |> filter(_n <= 1000)
8 generate df x = rnormal()
9 generate df e = rnormal()
10 generate df y = 2.0 + 1.5*x + (1.0 + 0.8*x)*e
11 let m_ols = ols(y ~ x, df)
12 let m_q25 = qreg(y ~ x, df, tau=0.25)

```

```

13 let m_q50 = qreg(y ~ x, df, tau=0.50)
14 let m_q75 = qreg(y ~ x, df, tau=0.75)
15 esttab(m_ols, m_q25, m_q50, m_q75)

```

Listing 23.1: Heteroscedastic DGP — OLS vs. four quantiles

	(1)	(2)	(3)
(4)			
	OLS	QReg (=0.25)	QReg (=0.50)
QReg (=0.75)			
x	1.9947***	1.8739***	2.0149***
	2.2462***		
	(0.0454)	(0.0687)	(0.0833)
	(0.0579)		
const		2.1659***	2.4525***
	2.6637***		
		(0.0302)	(0.0433)
	(0.0369)		
Constant	2.4538***		
	(0.0261)		
N	1000	0	0
0			
R ²	0.6594		
Adj. R ²	0.6591		
* p<0.10 ** p<0.05 *** p<0.01			

The slope of x grows monotonically: 1,87 at Q25, 2,01 at Q50 and 2,25 at Q75. OLS (1,99) is an implicit weighted average of these heterogeneous effects — it summarizes in a single number what QR decomposes across the entire conditional distribution.

The intercept also increases with τ (2,17 $\rightarrow$ 2,45 $\rightarrow$ 2,66), reflecting that upper quantiles of y are above the mean for any value of x .

23.5 Syntax

```

1 qreg(y ~ x1 + x2, df, tau=0.5)
2 qreg(y ~ x1 + x2, df, tau=0.5, boot=500)

```

The `tau` parameter defines the desired quantile ($\tau \in (0, 1)$, default 0.5). The `boot` parameter activates bootstrap for standard errors (default 200 replications).

```
1 load "pnad.csv" as df
2
3 let q10 = qreg(wage ~ educ + exp, df, tau=0.10, boot=500)
4 let q50 = qreg(wage ~ educ + exp, df, tau=0.50, boot=500)
5 let q90 = qreg(wage ~ educ + exp, df, tau=0.90, boot=500)
6 let ols = ols(wage ~ educ + exp, df)
7
8 esttab(ols, q10, q50, q90)
```

Listing 23.2: Typical workflow — wages at multiple quantiles

23.6 Capturing the result

```
1 let m = qreg(y ~ x, df, tau=0.75)
2 display m
3 let yhat = predict(m, "xb")
```

Chapter 24

Regression Discontinuity Design

24.1 The model

Regression Discontinuity Design (RDD) identifies causal effects when treatment is assigned mechanically based on a threshold of a continuous observable variable — the **running variable** x :

$$D_i = \mathbf{1}[x_i \geq c]$$

The estimator exploits the fact that individuals immediately below and above the threshold c are comparable in all respects except the treatment received. The identified effect is the **LATE at the cutoff** (*Local Average Treatment Effect at the cutoff*):

$$\tau_{\text{RDD}} = \lim_{x \downarrow c} E[y \mid x] - \lim_{x \uparrow c} E[y \mid x]$$

Note

The *Sharp* RDD (above) assumes that D changes deterministically at c . The *Fuzzy* RDD generalizes to when D increases *discontinuously* at c , without a jump from 0 to 1 — it requires a first stage and is equivalent to a local IV.

24.2 Identification

The central assumption is **continuity of potential outcomes** at c :

$$E[Y^{(0)} \mid x] \quad \text{and} \quad E[Y^{(1)} \mid x] \quad \text{are continuous at } x = c$$

This guarantees that any discontinuity in $E[y \mid x]$ at point c is caused by the

jump in D , and not by another factor that also changes at c .

The assumption is violated if agents *manipulate* x to place themselves above or below the threshold — students who revise answers to cross the passing cutoff, for example. The McCrary density test (continuity test for the density of x at c) checks for suspicious bunching of observations on one side of the threshold.

24.3 The estimator

The standard estimator is **local linear regression** fit separately on each side of the threshold, within a *bandwidth* h :

$$\hat{\tau} = \hat{\mu}_+ - \hat{\mu}_-$$

where $\hat{\mu}_+$ and $\hat{\mu}_-$ are the intercepts of the local linear regressions on the right side ($x \in [c, c + h)$) and left side ($x \in (c - h, c)$), respectively, weighted by the triangular kernel:

$$K\left(\frac{x - c}{h}\right) = \left(1 - \frac{|x - c|}{h}\right) \mathbf{1}[|x - c| \leq h]$$

The *bandwidth* h controls the bias-variance trade-off: wider windows reduce variance but increase bias (local linearity may be a poor approximation); narrower windows are more local but have fewer observations. Hayashi selects h automatically by the MSE-optimal criterion (Imbens-Kalyanaraman) when `bw` is not specified.

24.4 DGP Verification

Exact recovery (no noise)

The DGP uses a running variable $x \sim \text{Uniform}(-1, 1)$, threshold at $c = 0$ and true effect $\tau = 2,0$. The outcome function is linear in x on both sides of the threshold, without noise, so RDD should recover τ exactly.

$$y_i = 1,0 + 0,5 x_i + 2,0 D_i, \quad D_i = \mathbf{1}[x_i \geq 0]$$

```
1 seed(42)
2 input df
3 id
4 1
5 end
```

```

6 for i in 1..=10 { df = append(df, df) }
7 df |> filter(_n <= 1000)
8 generate df x = runiform() * 2 - 1
9 generate df d = x >= 0
10 generate df y = 1.0 + 0.5*x + 2.0*d
11 rd(y ~ x, 0.0, df)

```

Listing 24.1: Noise-free RDD DGP — exact recovery

```

Regression Discontinuity -   Sharp -   Local Linear (p=1)

Outcome: y                      Running var: x
Cutoff: 0.0000   Bandwidth: 0.0571   Kernel: Triangular
Obs total: 72   N left: 37   N right: 35

Treatment Effect ^():
      2.0000   SE      0.0000   z 2.443e15   P>|z|   0.0000   ***
95% CI: [2.0000, 2.0000]

*** p<0.01   ** p<0.05   * p<0.10

```

The estimator recovers $\hat{\tau} = 2,0000$ exactly. The bandwidth of 0,0571 was selected automatically — of the 1000 observations, 72 lie within the window around the cutoff.

With idiosyncratic noise

Adding idiosyncratic ε , the LATE remains identified:

```

1 generate df e = rnormal()
2 generate df y = 1.0 + 0.5*x + 2.0*d + e
3 rd(y ~ x, 0.0, df)

```

Listing 24.2: RDD DGP with noise

```

Regression Discontinuity -   Sharp -   Local Linear (p=1)

Outcome: y                      Running var: x
Cutoff: 0.0000   Bandwidth: 0.0741   Kernel: Triangular

```

```
Obs total: 94    N left: 43    N right: 51

Treatment Effect ^():
      2.0116    SE      0.1191    z    16.893    P>|z|    0.0000    ***
95% CI: [1.7782, 2.2449]

*** p<0.01    ** p<0.05    * p<0.10
```

With noise, $\hat{\tau} = 2.0116$ — the deviation from 2,0 is sampling variance. The 95% CI [1,78; 2,24] covers the true $\tau = 2,0$. The bandwidth was 0,0741: with more variance, the MSE-optimal criterion prefers a narrower window to control bias.

24.5 Syntax

```
1 rd(y ~ x, cutoff, df)
2 rd(y ~ x, cutoff, df, bw=0.3)
3 rd(y ~ x, cutoff, df, bw=0.3, poly=2)
```

Parameter	Default	Description
cutoff	required	threshold value in x
bw	auto (IK)	window width h
poly	1	degree of local polynomial
kernel	triangular	uniform, epanechnikov

```
1 load "school.csv" as df
2
3 // score centered at threshold 60
4 generate df xc = score - 60
5
6 let m = rd(approved ~ xc, 0.0, df)
7 display m
```

Listing 24.3: Typical workflow — test score cutoff and scholarship

24.6 Capturing the result

```
1 let m = rd(y ~ x, 0.0, df)
2 display m
```

Note

In real applications, validate the RDD with falsification tests: (1) estimate the model using *pre-treatment covariates* as the outcome — there should be no discontinuity at c ; (2) test discontinuities at *placebo thresholds* ($c \pm \delta$) — they should not be significant; (3) check the density of x at c (McCrary) to detect manipulation of the running variable.

24.7 Empirical validation

The DGP in this chapter with a uniform running variable and cutoff at 0 is used in the

`rddbookvalidationcaseinvalidation/cases/`. The case estimates the treatment effect by RDD in HA

Chapter 25

Generalized Method of Moments

25.1 The framework

The Generalized Method of Moments (GMM) unifies the estimators from previous chapters under a single principle: identification via population **moment conditions**.

A GMM model specifies L moment conditions:

$$E[g_i(\beta)] = 0, \quad g_i(\beta) \in \mathbb{R}^L$$

OLS and IV are special cases:

Estimator	Moment condition	Situation
OLS	$E[x_i \varepsilon_i] = 0$	$L = K$, identified
IV/2SLS	$E[z_i \varepsilon_i] = 0$	$L = K$, identified
GMM	$E[z_i \varepsilon_i] = 0$	$L > K$, overidentified

With $L = K$ moments, $g(\hat{\beta}) = 0$ is solved exactly --- equivalent to just-identified IV. With $L > K$, the system is overdetermined and no β zeroes all moments simultaneously; GMM minimizes a weighted quadratic form:

$$\hat{\beta}_{\text{GMM}} = \arg \min_{\beta} n \bar{g}(\beta)' \hat{W}^{-1} \bar{g}(\beta)$$

where $\bar{g}(\beta) = \frac{1}{n} \sum_i g_i(\beta)$ and $\hat{W}$ is the consistent estimate of the asymptotic variance of $\sqrt{n} \bar{g}(\beta)$.

25.2 The two-step estimator

1. **First step:** estimate $\hat{\beta}_1$ with $W = I$ (uniform weight).
2. **Efficient weight matrix:** compute $\hat{W} = \frac{1}{n} \sum_i z_i z_i' \hat{\varepsilon}_i^2$ using first-step residuals.
3. **Second step:** re-estimate $\hat{\beta}_2$ with $\hat{W}^{-1}$ --- the efficient GMM estimator.

The two-step estimator is asymptotically efficient in the class of GMM estimators with the same moment conditions.

25.3 J-test of overidentification

When $L > K$, it is possible to test whether the excess moments are valid. Hansen's statistic (*J-test*):

$$J = n \bar{g}(\hat{\beta})' \hat{W}^{-1} \bar{g}(\hat{\beta}) \xrightarrow{d} \chi^2(L - K) \quad \text{under } H_0 : E[g_i(\beta_0)] = 0$$

H_0 is the hypothesis that *all* L instruments are exogenous. Rejecting H_0 indicates that at least one instrument is correlated with ε --- the model is misspecified or an instrument is invalid.

Warning

The J-test is necessary but not sufficient: failing to reject does not prove that all instruments are valid, only that the overidentifying restrictions are consistent with the data.

25.4 DGP Verification

OLS, IV, and GMM side by side

The DGP repeats the structure from Chapter 19: x is endogenous ($\text{Cov}(x, \varepsilon) \neq 0$), with two valid instruments z_1 and z_2 . ε is orthogonalized against z_1 in-sample --- just-identified IV recovers exactly; GMM with two instruments is overidentified ($L = 2, K = 1$).

```
1 seed(42)
2 input df
3 id
4 1
```

```

5 end
6 for i in 1..=10 { df = append(df, df) }
7 df |> filter(_n <= 1000)
8 generate df z1 = rnormal()
9 generate df z2 = rnormal()
10 generate df v = rnormal()
11 let d1 = cov(df, z1, v) / variance(df, z1)
12 let d0 = mean(df, v) - d1 * mean(df, z1)
13 generate df eps = v - d0 - d1*z1
14 generate df x = 0.8*z1 + 0.4*z2 + eps
15 generate df y = 1.5 + 2.0*x + 0.7*eps
16 let m_ols = ols(y ~ x, df)
17 let m_iv = iv(y ~ x, ~ z1, df)
18 let m_gmm = gmm(y ~ x, ~ z1 + z2, df)
19 display m_ols
20 display m_iv
21 display m_gmm

```

Listing 25.1: GMM DGP — biased OLS, exact IV, efficient GMM

```

Dep. Variable: y   R²=0.9798   N=1000   F=48418.53
const    1.2757***   (0.0077)    x    2.3736***   (0.0108)

Dep. Variable: y   Estimator: 2SLS   N=1000
const    1.5000***   (0.0170)    x    2.0000***   (0.0264)

Dep. Variable: y   GMM Two-Step   N=1000   J=2.5012   Prob(J)=0.1138   DF=1
x0    1.4875***   (0.0145)    x1    2.0206***   (0.0226)

```

OLS is biased (endogeneity): $\hat{\beta}_1 = 2,37 \neq 2,0$. IV with z_1 recovers exactly 2,0000 (in-sample orthogonalization). GMM with z_1+z_2 converges to 2,0206 --- consistent, not exact, because it uses two moments to estimate one parameter. The J-test ($J = 2,50$, $p = 0,11$) does not reject H_0 : both instruments are valid.

Note

Hayashi displays the coefficients as x0 (intercept) and x1 (slope) in the GMM output. This is an internal display convention --- the coefficients correspond exactly to the order of variables in the formula.

J-test rejects — invalid instrument

If z_2 is contaminated with ε , the J-test detects the violation:

```
1 // z2b correlated with eps: invalid instrument
2 generate df z2b = z2 + 0.5*eps
3 let m_bad = gmm(y ~ x, ~ z1 + z2b, df)
4 display m_bad
```

Listing 25.2: Invalid instrument — J-test rejects

```
Dep. Variable: y    GMM Two-Step    N=1000    J=152.9417    Prob(J)=0.0000    DF=1
x0    1.3738***    (0.0089)    x1    2.2125***    (0.0124)
```

$J = 152,94$ ($p = 0,000$) rejects H_0 with high significance --- and the coefficients are biased ($\hat{\beta}_1 = 2,21$). The J-test worked as an alarm: before trusting the GMM estimates, check that J does not reject.

25.5 Syntax

```
1 gmm(y ~ x1 + x2, ~ z1 + z2 + z3, df)
2 gmm(y ~ x1 + x2, ~ z1 + z2 + z3, df, cov=robust)
```

The first right-hand side ($\sim x1 + x2$) lists the regressors; the second ($\sim z1 + z2 + z3$) lists the instruments. Exogenous variables in x must also appear as their own instruments.

```
1 load "data.csv" as df
2
3 // educ is exogenous (instrument for itself)
4 // exp is endogenous; z1 and z2 are instruments for exp
5 let m = gmm(wage ~ educ + exp, ~ educ + z1 + z2, df)
6 display m
```

Listing 25.3: Typical workflow with included exogenous variable

25.6 Capturing the result

```
1 let m = gmm(y ~ x, ~ z1 + z2, df)
2 display m
3 let yhat = predict(m, "xb")
```

Part III

Time Series

Chapter 26

AR, MA and ARMA

26.1 Stochastic processes and stationarity

A **stochastic process** $\{y_t\}$ is (covariance) stationary when:

$$E[y_t] = \mu, \quad \text{Var}[y_t] = \sigma^2, \quad \text{Cov}(y_t, y_{t-k}) = \gamma_k$$

for all t --- mean, variance and autocovariances depend only on the lag k , not on time t .

The simplest case is **white noise** $\varepsilon_t \sim \text{WN}(0, \sigma^2)$: $E[\varepsilon_t] = 0$, $\text{Var}[\varepsilon_t] = \sigma^2$ and $\text{Cov}(\varepsilon_t, \varepsilon_s) = 0$ for $t \neq s$. White noise is the building block of all models in this section.

26.2 AR(p) process

The autoregressive model of order p (**AR**(p)) expresses y_t as a linear combination of its own past values plus a shock:

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \cdots + \phi_p y_{t-p} + \varepsilon_t$$

For the AR(1), the stationarity condition is $|\phi_1| < 1$. When satisfied, the unconditional mean is $\mu = c/(1 - \phi_1)$ and the autocorrelation decays geometrically:

$$\rho_k = \phi_1^k, \quad k = 0, 1, 2, \dots$$

The **Partial Autocorrelation Function** (PACF) cuts off to zero after lag p --- a diagnostic signature of AR.

26.3 MA(q) process

The moving average model of order q (MA(q)) expresses y_t as a linear combination of current and past shocks:

$$y_t = \mu + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \cdots + \theta_q \varepsilon_{t-q}$$

The MA(q) is always stationary. The **Autocorrelation Function** (ACF) cuts off to zero after lag q --- in contrast with AR, whose ACF decays geometrically. The **invertibility** condition ($|\theta_1| < 1$ for MA(1)) guarantees a unique AR(∞) representation.

Note

Standard diagnostic: ACF decays slowly and PACF cuts off $\rightarrow$ AR(p);
ACF cuts off and PACF decays slowly $\rightarrow$ MA(q); both decay $\rightarrow$ ARMA.

26.4 ARMA(p, q) process

The ARMA(p, q) combines both mechanisms:

$$y_t = c + \sum_{i=1}^p \phi_i y_{t-i} + \varepsilon_t + \sum_{j=1}^q \theta_j \varepsilon_{t-j}$$

It is stationary when the roots of the AR polynomial lie outside the unit circle and invertible when the roots of the MA polynomial do as well.

26.5 DGP Verification

Since there is no lag generator in `generate`, the processes are constructed with a `while` loop that fills y_t row by row, using `mean(..., if = t == k)` as an accessor for row k .

AR(1) with $\phi = 0,7$

```
1 seed(42)
2 input df
3 id
4 1
5 end
```

```

6 for i in 1..=9 { df = append(df, df) }
7 df |> filter(_n <= 500)
8 generate df e = rnormal()
9 generate df y = 0.0
10 generate df t = _n
11 let n = nrow(df)
12 let i = 2
13 while i <= n {
14     let lag_y = mean(df, y, if = t == i - 1)
15     let et     = mean(df, e, if = t == i)
16     replace df y = 0.7 * lag_y + et if t == i
17     i = i + 1
18 }
19 let m = arima(df, y, p=1, d=0, q=0)
20 display m

```

Listing 26.1: AR(1) DGP and estimation

```

===== ARIMA(1,0,0) via Hannan-Rissanen =====
Observations:                494
AIC:                157.7072    BIC:                166.1123
Parameters:
    intercept    0.5454    std=0.0557    z=9.786    p=0.000    [0.4362, 0.6547]
    ar.L1         0.6638    std=0.0335    z=19.814    p=0.000    [0.5981, 0.7295]
=====

```

$\hat{\phi}_1 = 0,6638$ with true $\phi_1 = 0,7$; the 95% CI $[0,60; 0,73]$ covers the true value.

MA(1) with $\theta = 0,5$

```

1 generate df y = e
2 let i = 2
3 while i <= n {
4     let lag_e = mean(df, e, if = t == i - 1)
5     replace df y = y + 0.5 * lag_e if t == i
6     i = i + 1
7 }
8 let m = arima(df, y, p=0, d=0, q=1)
9 display m

```

Listing 26.2: MA(1) DGP and estimation

```

===== ARIMA(0,0,1) via Hannan-Rissanen =====
Observations:                493
AIC:                        162.5430    BIC:                        170.9441
Parameters:
  intercept    0.7297    std=0.0128    z=57.014    p=0.000    [0.7046, 0.7548]
  ma.L1        0.5087    std=0.0456    z=11.163    p=0.000    [0.4194, 0.5980]
=====

```

$\hat{\theta}_1 = 0,5087 \approx 0,5$; 95% CI $[0,42; 0,60]$ covers $\theta_1 = 0,5$.

ARMA(1,1) with $\phi = 0,6$ and $\theta = 0,4$

```

1 generate df y = e
2 let i = 2
3 while i <= n {
4   let lag_y = mean(df, y, if = t == i - 1)
5   let lag_e = mean(df, e, if = t == i - 1)
6   let et    = mean(df, e, if = t == i)
7   replace df y = 0.6*lag_y + et + 0.4*lag_e if t == i
8   i = i + 1
9 }
10 let m = arima(df, y, p=1, d=0, q=1)
11 display m

```

Listing 26.3: ARMA(1,1) DGP and estimation

```

===== ARIMA(1,0,1) via Hannan-Rissanen =====
Observations:                493
AIC:                        159.4217    BIC:                        172.0233
Parameters:
  intercept    0.8073    std=0.0660    z=12.227    p=0.000    [0.6779, 0.9367]
  ar.L1        0.5259    std=0.0380    z=13.831    p=0.000    [0.4514, 0.6004]
  ma.L1        0.4829    std=0.0591    z=8.167     p=0.000    [0.3670, 0.5987]
=====

```

Both parameters are recovered with good precision: $\hat{\phi} = 0,53$ (true 0,6) and $\hat{\theta} = 0,48$ (true 0,4).

26.6 Order selection

HAYASHI reports AIC and BIC automatically. The practical criterion:

1. Estimate models with orders (p, q) on a small grid (e.g. $0 \leq p, q \leq 3$).
2. Select the model with the lowest AIC (preference for fit) or BIC (preference for parsimony).
3. Verify that the residuals are white noise.

```

1 let m00 = arima(df, y, p=0, d=0, q=0)
2 let m10 = arima(df, y, p=1, d=0, q=0)
3 let m01 = arima(df, y, p=0, d=0, q=1)
4 let m11 = arima(df, y, p=1, d=0, q=1)
5 esttab(m00, m10, m01, m11)

```

Listing 26.4: Order selection by AIC

26.7 Syntax

```

1 arima(df, y, p=1, d=0, q=0)
2 arima(df, y, p=0, d=0, q=1)
3 arima(df, y, p=1, d=1, q=1)

```

Parameter	Default	Description
p	0	autoregressive order
d	0	degree of differencing
q	0	moving average order

```

1 load "cpi.csv" as df
2
3 let m = arima(df, inflation, p=1, d=0, q=1)
4 display m
5 let yhat = predict(m, "xb")

```

Listing 26.5: Typical workflow with real data

Note

The `while` loop in the DGPs above is $O(n^2)$ and is only used to illustrate the structure of the process. In real use, data are loaded from a file with `load` and the loop is not needed.

26.8 Empirical validation

The DGPs in this chapter are used as automated validation cases in `validation/cases/ar_book`,

Chapter 27

ARIMA and Unit Roots

27.1 Integrated processes

A process $\{y_t\}$ is **integrated of order d** --- denoted $I(d)$ --- when it needs to be differenced d times to become stationary. The most common case in economics is $I(1)$: the first difference is stationary, but the level is not.

The simplest example of an $I(1)$ process is the **random walk**:

$$y_t = y_{t-1} + \varepsilon_t, \quad \varepsilon_t \sim \text{WN}(0, \sigma^2)$$

The variance grows with t ($\text{Var}[y_t] = t\sigma^2$), so the process is not stationary. The first difference $\Delta y_t = y_t - y_{t-1} = \varepsilon_t$ is white noise --- therefore $I(0)$.

The random walk *with drift* adds a constant μ :

$$y_t = \mu + y_{t-1} + \varepsilon_t$$

with deterministic trend $E[y_t] = y_0 + \mu t$ --- the basic model for asset prices, exchange rates and many macroeconomic series.

27.2 Augmented Dickey-Fuller Test

The ADF (*Augmented Dickey-Fuller*) test formally tests for the presence of a unit root. The auxiliary model estimated is:

$$\Delta y_t = \alpha + \delta y_{t-1} + \sum_{j=1}^k \gamma_j \Delta y_{t-j} + \varepsilon_t$$

with hypotheses:

$$H_0 : \delta = 0 \quad (\text{unit root}) \quad H_1 : \delta < 0 \quad (\text{stationary})$$

The t -statistic of $\hat{\delta}$ does not follow the standard t -distribution under H_0 --- the Dickey-Fuller distribution with tabulated critical values is used.

Warning

ADF critical values are **negative**: H_0 is rejected when the statistic is sufficiently *negative*. A positive value (or near zero) does not reject H_0 --- evidence of a unit root.

27.3 The ARIMA model

ARIMA(p, d, q) generalizes ARMA for nonstationary series, applying d differences before estimation:

$$\underbrace{\phi(L)(1-L)^d}_{\text{AR in differences}} y_t = c + \underbrace{\theta(L)}_{\text{MA}} \varepsilon_t$$

where L is the lag operator ($Ly_t = y_{t-1}$) and $(1-L)^d$ is the difference operator of order d .

Model	Notation	Process
Random walk	ARIMA(0,1,0)	$\Delta y_t = c + \varepsilon_t$
AR(1) in differences	ARIMA(1,1,0)	$\Delta y_t = c + \phi \Delta y_{t-1} + \varepsilon_t$
MA(1) in differences	ARIMA(0,1,1)	$\Delta y_t = c + \varepsilon_t + \theta \varepsilon_{t-1}$

27.4 DGP Verification

Random walk: ADF does not reject; AR(1): ADF rejects

The DGP contrasts the two cases: random walk (I(1)) generated by cumsum (e) versus stationary AR(1) ($\phi = 0.7$) generated by a loop.

```
1 seed(42)
2 input df
3 id
4 1
```

```

5 end
6 for i in 1..=9 { df = append(df, df) }
7 df |> filter(_n <= 500)
8 generate df e = rnormal()
9 generate df rw = cumsum(e)
10 generate df y = 0.0
11 generate df t = _n
12 let n = nrow(df)
13 let i = 2
14 while i <= n {
15     let ly = mean(df, y, if = t == i - 1)
16     let et = mean(df, e, if = t == i)
17     replace df y = 0.7 * ly + et if t == i
18     i = i + 1
19 }
20 adf(df, rw)
21 adf(df, y)

```

Listing 27.1: ADF on random walk vs. AR(1)

```

===== Augmented Dickey-Fuller Test =====
Variable: rw    Lags used: 7
H: series has a unit root (non-stationary)
Test statistic:    -0.0980
Critical values:  1%=-3.430  5%=-2.860  10%=-2.570
Conclusion: Does not reject H - unit root present

===== Augmented Dickey-Fuller Test =====
Variable: y     Lags used: 7
H: series has a unit root (non-stationary)
Test statistic:    -6.4616
Critical values:  1%=-3.430  5%=-2.860  10%=-2.570
Conclusion: REJECTS H - stationary

```

The random walk has statistic -0.10 --- above the negative critical values, does not reject H_0 . The AR(1) with $\phi = 0.7$ has statistic -6.46 --- far below the 1% critical value (-3.43), rejects H_0 with high confidence.

ARIMA(0,1,0) estimation and order selection

With the unit root identified, estimate with $d = 1$. Three competing orders for the true random walk:

```
1 let m010 = arima(df, rw, p=0, d=1, q=0)
2 let m110 = arima(df, rw, p=1, d=1, q=0)
3 let m011 = arima(df, rw, p=0, d=1, q=1)
4 display m010
5 display m110
6 display m011
```

Listing 27.2: Order selection for I(1) series

```
-- ARIMA(0,1,0): AIC=156.77  BIC=160.97
  intercept  0.4863  std=0.0128  z=38.14  p=0.000

-- ARIMA(1,1,0): AIC=158.70  BIC=167.10
  intercept  0.4805  std=0.0253  z=18.96  p=0.000
  ar.L1      0.0120  std=0.0450  z=0.266  p=0.791  [insig.]

-- ARIMA(0,1,1): AIC=157.80  BIC=166.19
  intercept  0.4855  std=0.0128  z=38.07  p=0.000
  ma.L1      0.0058  std=0.0453  z=0.128  p=0.898  [insig.]
```

The true model ARIMA(0,1,0) has the lowest AIC and BIC. The additional AR and MA terms are insignificant ($p > 0.7$) --- BIC, which penalizes complexity more strongly, confirms the parsimony of the simple model.

27.5 Workflow

The standard protocol for univariate time series:

1. **Inspect** the level of the series --- visible trend suggests nonstationarity.
2. **Test** with ADF --- failing to reject H_0 indicates $d \geq 1$.
3. **Difference** once and re-test with ADF on the differenced series --- confirms $d = 1$.
4. **Select** (p, q) by AIC/BIC grid over the differenced series.
5. **Validate** residuals: they should be white noise.

27.6 Syntax

```
1 adf(df, y)
2 adf(df, y, lags=4)
3 arima(df, y, p=0, d=1, q=0)
4 arima(df, y, p=1, d=1, q=1)

1 load "exchange.csv" as df
2
3 adf(df, rate)
4
5 // if H0 is not rejected: estimate with d=1
6 let m = arima(df, rate, p=1, d=1, q=0)
7 display m
8 let yhat = predict(m, "xb")
```

Listing 27.3: Typical workflow with real data

27.7 Empirical validation

The random walk DGP in this chapter is used in the `arima_bookvalidationcaseinvalidat`

Chapter 28

VAR and Granger Causality

28.1 The VAR model

The Vector Autoregressive model (VAR(p)) generalizes AR to multivariate systems. With K variables and p lags:

$$\mathbf{y}_t = \mathbf{c} + \mathbf{A}_1\mathbf{y}_{t-1} + \cdots + \mathbf{A}_p\mathbf{y}_{t-p} + \boldsymbol{\varepsilon}_t$$

where $\mathbf{y}_t = (y_{1t}, \dots, y_{Kt})'$ is the vector of variables, $\mathbf{A}_j$ are $K \times K$ coefficient matrices and $\boldsymbol{\varepsilon}_t \sim (0, \Sigma_u)$ is the vector of shocks that are uncorrelated over time but potentially correlated with each other.

The VAR is estimated equation by equation via OLS --- each equation is an AR(p) in the K lagged variables. The stationarity condition requires that all eigenvalues of $\mathbf{A}_1 + \cdots + \mathbf{A}_p$ lie inside the unit circle.

28.2 Granger Causality

x Granger-causes y if lags of x have predictive power for y beyond what lags of y alone provide. Formally, the hypothesis tested is:

$$H_0 : A_{12}(1) = A_{12}(2) = \cdots = A_{12}(p) = 0$$

in the equation for y_1 , where $A_{12}(j)$ is the coefficient of $y_{2,t-j}$.

The test is a standard F-test in the auxiliary equation:

$$F = \frac{(RSS_r - RSS_u)/p}{RSS_u/(T - 2p - 1)} \sim F(p, T - 2p - 1)$$

Note

Granger causality is *predictability*, not structural causality. x that precedes y and correlates with it passes the test, even without a direct causal relationship.

28.3 Impulse Response Function

The IRF (*Impulse Response Function*) traces the effect of a unit shock in y_k on all variables over time.

The $\text{MA}(\infty)$ representation of the VAR is $\mathbf{y}_t = \sum_{h=0}^{\infty} \Phi_h \varepsilon_{t-h}$, with $\Phi_0 = I$ and $\Phi_h = \mathbf{A}_1 \Phi_{h-1} + \dots + \mathbf{A}_p \Phi_{h-p}$. The orthogonalized IRF uses the Cholesky decomposition of Σ_u to isolate uncorrelated shocks --- the **Cholesky ordering** (variable ordering) matters.

28.4 DGP Verification

The DGP uses a bivariate VAR(1) with an asymmetric causality structure: y_2 causes y_1 (coefficient 0,2), but y_1 causes y_2 weakly (coefficient 0,1):

$$\begin{pmatrix} y_{1t} \\ y_{2t} \end{pmatrix} = \begin{pmatrix} 0,5 & 0,2 \\ 0,1 & 0,6 \end{pmatrix} \begin{pmatrix} y_{1,t-1} \\ y_{2,t-1} \end{pmatrix} + \begin{pmatrix} \varepsilon_{1t} \\ \varepsilon_{2t} \end{pmatrix}$$

```

1 seed(42)
2 input df
3 id
4 1
5 end
6 for i in 1..=9 { df = append(df, df) }
7 df |> filter(_n <= 300)
8 generate df e1 = rnormal()
9 generate df e2 = rnormal()
10 generate df y1 = 0.0
11 generate df y2 = 0.0
12 generate df t = _n
13 let n = nrow(df)
14 let i = 2
15 while i <= n {
16     let l1 = mean(df, y1, if = t == i - 1)
17     let l2 = mean(df, y2, if = t == i - 1)

```

```

18     let a = mean(df, e1, if = t == i)
19     let b = mean(df, e2, if = t == i)
20     replace df y1 = 0.5*l1 + 0.2*l2 + a if t == i
21     replace df y2 = 0.1*l1 + 0.6*l2 + b if t == i
22     i = i + 1
23 }
24 let m = var(df, y1, y2, lags=1)
25 display m
26 granger(df, y1, y2, lags=1)
27 granger(df, y2, y1, lags=1)
28 irf(m, steps=6)

```

Listing 28.1: VAR(1) DGP — Granger and IRF

Model fit

```
VAR(1)   N=299   AIC=-5.0269   BIC=-4.9526
```

```
Sigma_u:
```

```

[ 0.0819  0.0046 ]
[ 0.0046  0.0772 ]

```

The residual covariance matrix $\hat{\Sigma}_u$ is nearly diagonal --- the shocks ε_1 and ε_2 are practically orthogonal, as specified in the DGP.

Granger Causality

```

===== Granger Causality Test =====
H: y2 does not Granger-cause y1   (lags=1)
F(1, 296) = 9.4437   p = 0.0023
Conclusion: REJECTS H - y2 Granger-causes y1

===== Granger Causality Test =====
H: y1 does not Granger-cause y2   (lags=1)
F(1, 296) = 4.5768   p = 0.0332
Conclusion: REJECTS H - y1 Granger-causes y2

```

Both directions reject $H_0: y_2 \rightarrow y_1$ with $p = 0,002$ and $y_1 \rightarrow y_2$ with $p = 0,033$. The DGP imposes $\phi_{12} = 0,2$ and $\phi_{21} = 0,1$ --- bidirectional causality is detected, although $y_2 \rightarrow y_1$ is stronger.

Impulse response

IRF - VAR(1) - 6 steps

Impulse: y1

h	y1	y2
1	0.2862	0.0161
2	0.1430	0.0382
3	0.0758	0.0364
4	0.0426	0.0286
5	0.0251	0.0207
6	0.0154	0.0145

Impulse: y2

h	y1	y2
1	0.0000	0.2774
2	0.0405	0.1595
3	0.0432	0.0958
4	0.0352	0.0595
5	0.0260	0.0378
6	0.0183	0.0243

A shock to y_1 has a rapid effect on y_1 (decays from 0,29 to 0,015 over 6 periods) and a small effect on y_2 . A shock to y_2 transmits progressively to y_1 --- reaches 0,043 at period 3, then decays --- capturing the $\phi_{12} = 0,2$ channel of the DGP.

28.5 Lag selection

```

1 let m1 = var(df, y1, y2, lags=1)
2 let m2 = var(df, y1, y2, lags=2)
3 let m3 = var(df, y1, y2, lags=3)
4 esttab(m1, m2, m3)

```

Listing 28.2: Comparison by number of lags

The lag with the lowest BIC (parsimony) or AIC (fit) is selected. With the VAR(1) DGP, $p = 1$ is expected to minimize both.

28.6 Syntax

```

1 var(df, y1, y2, lags=1)
2 var(df, y1, y2, y3, lags=2)
3 granger(df, y, x, lags=1)
4 irf(m, steps=10)

1 load "macro.csv" as df
2
3 adf(df, gdp)
4 adf(df, cpi)
5
6 let m = var(df, gdp, cpi, lags=2)
7 granger(df, gdp, cpi, lags=2)
8 granger(df, cpi, gdp, lags=2)
9 irf(m, steps=12)

```

Listing 28.3: Typical workflow — GDP and inflation

Note

VAR requires *stationary* series. If the variables are I(1) and cointegrated, the appropriate model is the VECM (ch. 29). If they are I(1) but not cointegrated, difference before estimating.

28.7 Empirical validation

The VAR(1) DGP in this chapter corresponds to the `var_bookvalidationcaseinvalidationby-equationOLSestimationwithRandPythonreferences.Runhay validate` to see the current status.

Chapter 29

Cointegration and VECM

29.1 Long-run Relationship Between $I(1)$ Series

Two $I(1)$ series are **cointegrated** if there exists a linear combination between them that is $I(0)$. Formally, y_t and x_t are cointegrated if there exists β such that:

$$u_t = y_t - \beta x_t \sim I(0)$$

The intuition is that of a *leash*: series that individually follow random walks may be attracted by a long-run equilibrium force. Asset prices in the same market, exchange rates and purchasing power parity, short- and long-term interest rates are classic examples.

Non-cointegrated $I(1)$ series diverge indefinitely - level regression produces **spurious regression**: high R^2 and significant t -statistics with no real causal relationship.

29.2 Engle-Granger Test

The Engle-Granger (1987) procedure tests cointegration in two steps:

1. Estimate the **long-run relationship** by OLS: $\hat{u}_t = y_t - \hat{\beta}x_t$
2. Apply the **ADF test** to the residuals $\hat{u}_t$ - under H_0 the residuals have a unit root (no cointegration).

The critical values of the test are more negative than those of the standard ADF, since $\hat{u}_t$ is an estimated residual, not a directly observed series.

29.3 Error Correction Model (VECM)

If y_t and x_t are cointegrated, the VECM (*Vector Error Correction Model*) decomposes the dynamics into a short-run component and adjustment to the long-run equilibrium:

$$\Delta \mathbf{y}_t = \alpha \underbrace{(\beta' \mathbf{y}_{t-1})}_{\text{deviation from equilibrium}} + \sum_{j=1}^{p-1} \Gamma_j \Delta \mathbf{y}_{t-j} + \varepsilon_t$$

- β - cointegrating vector: defines the long-run equilibrium relationship $\beta' \mathbf{y}_t = 0$.
- α - adjustment coefficients: speed at which each variable corrects deviations from equilibrium. A negative sign in α_y means y falls when it is above equilibrium.
- Γ_j - short-run dynamics.

The VECM is estimated by ML via the Johansen procedure, which tests the number of cointegrating vectors (*rank*) via the trace statistic or maximum eigenvalue.

29.4 DGP Verification

Case 1 — cointegrated: $y_t = 2x_t + \varepsilon_t$

x_t is a random walk ($I(1)$); $\varepsilon_t \sim I(0)$. Therefore $y_t - 2x_t = \varepsilon_t \sim I(0)$: the series are cointegrated with vector $[1, -2]$.

```
1 seed(42)
2 input df
3 id
4 1
5 end
6 for i in 1..9 { df = append(df, df) }
7 df |> filter(_n <= 300)
8 generate df e1 = rnormal()
9 generate df e2 = rnormal()
10 generate df x = cumsum(e1)
11 generate df y = 2.0*x + e2
12 coint(df, y, x)
```

Listing 29.1: Cointegrated DGP and Engle-Granger test

```

===== Engle-Granger Cointegration Test =====
Variables: y, x
H: no cointegration
ADF statistic:      -6.4808
Critical values:   1%=-3.900   5%=-3.340   10%=-3.040
Cointegrating vector: [0.4778, 2.0002]
Conclusion: REJECT H - cointegrated series

```

The statistic $-6,48$ is well below the 1% critical value ($-3,90$). The estimated cointegrating vector $[0,48; 2,00]$ recovers $\beta = 2,0$ with precision.

Case 2 — not cointegrated: two independent random walks

```

1 generate df y = cumsum(e2)
2 coint(df, y, x)

```

Listing 29.2: Two random walks — no cointegration

```

===== Engle-Granger Cointegration Test =====
Variables: y, x
H: no cointegration
ADF statistic:      -3.1058
Critical values:   1%=-3.900   5%=-3.340   10%=-3.040
Conclusion: Fail to reject H - no cointegration

```

VECM — adjustment dynamics

With cointegration confirmed, we estimate the VECM:

```

1 generate df y = 2.0*x + e2
2 let m = vecm(df, y, x, lags=1)
3 display m

```

Listing 29.3: VECM with rank 1

```

===== VECM (Johansen ML) - Rank 1 =====
No. Variables:                2
Observations:                 299

```

```

----- Cointegration Vectors (Beta) -----
[ 3.6175, -7.2357 ] → normalized: [ 1, -2.000 ]

----- Adjustment Coefficients (Alpha) -----
y: -0.3133 (corrects deviations from equilibrium)
x: -0.0179 (nearly exogenous - minimal adjustment)

----- Johansen Eigenvalues (Lambda) -----
= 0.5017      = 0.0004
=====

```

The normalized cointegrating vector $[1, -2,00]$ recovers the true $\beta = 2,0$. The adjustment coefficient $\hat{\alpha}_y = -0,313$: when $y_{t-1} > 2x_{t-1}$ (above equilibrium), Δy_t is negative - y corrects downward. x barely adjusts ($\hat{\alpha}_x \approx -0,02$), since it is the exogenous variable that defines the equilibrium.

29.5 Syntax

```

1 coint(df, y, x)
2 coint(df, y, x, lags=4)
3 vecm(df, y, x, lags=1)
4 vecm(df, y1, y2, y3, lags=2)

1 load "cambio.csv" as df
2
3 adf(df, spot)
4 adf(df, forward)
5
6 coint(df, spot, forward)
7
8 // if cointegrated:
9 let m = vecm(df, spot, forward, lags=2)
10 display m

```

Listing 29.4: Typical workflow — exchange rate and parity

Note

VECM assumes all variables are $I(1)$ and cointegrated. Verify with ADF before estimating. If the series are $I(1)$ but not cointegrated, use the VAR in first differences (ch. 28).

29.6 Empirical validation

The cointegrated DGP in this chapter is used in the `coint_bookvalidationcaseinvalida`
`1VECM` in `HAYASHI` and compares it with reference implementations in `R` and `Python`. Run `hay valid`

Chapter 30

GARCH and Conditional Volatility

30.1 Conditional Heteroskedasticity

Financial series exhibit a recurring phenomenon: periods of high volatility tend to be followed by more volatility, and calm periods persist. This *volatility clustering* violates the assumption of constant conditional variance in ARMA models - the variance of ε_t given the past varies over time.

The GARCH(p, q) model (Engle 1982; Bollerslev 1986) captures this dynamic explicitly. The GARCH(1,1) specification is:

$$\varepsilon_t \mid \mathcal{F}_{t-1} \sim (0, h_t) \quad h_t = \omega + \alpha_1 \varepsilon_{t-1}^2 + \beta_1 h_{t-1}$$

where h_t is the *conditional variance*, α_1 captures the ARCH effect (reaction to past shock) and β_1 captures persistence (volatility memory).

Restriction	Condition	Meaning
Positive variance	$\omega > 0, \alpha_1 \geq 0, \beta_1 \geq 0$	$h_t > 0$ guaranteed
Stationarity	$\alpha_1 + \beta_1 < 1$	variance does not explode
Unconditional variance	$\sigma^2 = \omega / (1 - \alpha_1 - \beta_1)$	long-run level

The case $\alpha_1 + \beta_1 = 1$ is called IGARCH (*integrated*): the volatility has a unit root and shocks have a permanent effect.

30.2 ARCH-LM Test

Before estimating a volatility model, one tests whether ARCH effects are present. Engle's (1982) test regresses ε_t^2 on its own lags and evaluates joint significance:

$$\varepsilon_t^2 = a_0 + a_1 \varepsilon_{t-1}^2 + \cdots + a_m \varepsilon_{t-m}^2 + u_t$$

$$H_0: a_1 = \cdots = a_m = 0 \quad (\text{no ARCH effects})$$

The statistic is $LM = N \cdot R_{\text{aux}}^2 \xrightarrow{d} \chi^2(m)$ under H_0 .

30.3 DGP Verification

The DGP generates a true ARCH(1) process ($\beta = 0$):

$$h_t = 0,3 + 0,5 \varepsilon_{t-1}^2 \quad \varepsilon_t = \sqrt{h_t} z_t \quad z_t \sim (0, 1)$$

The innovation is generated directly as a standard normal draw: $z = \text{rnormal}()$, so $E[z] = 0$ and $E[z^2] = 1$.

```
1 seed(42)
2 input df
3 id
4 1
5 end
6 for i in 1..9 { df = append(df, df) }
7 df |> filter(_n <= 500)
8 generate df z = rnormal() // z: E[z]=0, E[z^2]
   ]=1
9 generate df e = z
10 generate df t = _n
11 let n = nrow(df)
12 let i = 2
13 while i <= n {
14     let le = mean(df, e, if = t == i - 1)
15     let lz = mean(df, z, if = t == i)
16     replace df e = sqrt(0.3 + 0.5 * le * le) * lz if t == i
17     i = i + 1
18 }
```

```

19
20 archtest(df, e, lags=5)           // should reject H
21 let m = garch(df, e, p=1, q=1)
22 display m
23 archtest(m, lags=5)             // should not reject (GARCH
    captured the ARCH)

```

Listing 30.1: ARCH(1) DGP and GARCH estimation

ARCH-LM Test — raw series

The raw series should reject the no-ARCH null, confirming time-varying conditional variance in the simulated process.

GARCH(1,1) Estimation

The MLE estimator recovers the true DGP ($\omega = 0.3$, $\alpha = 0.5$, $\beta = 0$) up to finite-sample variation. A fitted β near zero identifies that the process is pure ARCH - with no GARCH persistence component.¹

ARCH-LM on standardized residuals

After fitting the GARCH, the standardized residuals $\hat{z}_t = \varepsilon_t / \sqrt{\hat{h}_t}$ should be white noise in z_t^2 :

After fitting the model, the ARCH-LM test on standardised residuals should no longer reject if the volatility specification has captured the conditional heteroskedasticity.

30.4 Variants: EGARCH and GJR-GARCH

The symmetric GARCH treats positive and negative shocks equally. In financial assets, negative shocks tend to increase volatility more than positive shocks of the same magnitude (*leverage effect*).

EGARCH (Nelson 1991) models $\ln h_t$:

$$\ln h_t = \omega + \beta_1 \ln h_{t-1} + \alpha_1 \left| \frac{\varepsilon_{t-1}}{\sqrt{h_{t-1}}} \right| + \gamma_1 \frac{\varepsilon_{t-1}}{\sqrt{h_{t-1}}}$$

¹With $N = 500$, finite-sample estimates need not match the DGP parameters exactly; with larger N , convergence improves.

The parameter $\gamma_1 < 0$ captures asymmetry (leverage effect).

GJR-GARCH (Glosten-Jagannathan-Runkle 1993) adds an indicator term:

$$h_t = \omega + \alpha_1 \varepsilon_{t-1}^2 + \gamma_1 \varepsilon_{t-1}^2 \mathbf{1}[\varepsilon_{t-1} < 0] + \beta_1 h_{t-1}$$

The coefficient $\gamma_1 > 0$ implies that negative shocks raise volatility more.

30.5 Syntax

```
1 garch(df, y, p=1, q=1)
2 garch(df, y, p=1, q=1, dist=t)      // Student-t errors
3 egarch(df, y, p=1, q=1)
4 gjrgarch(df, y, p=1, q=1)
5 archtest(df, y, lags=5)              // ARCH-LM on raw series
6 archtest(modelo, lags=5)             // ARCH-LM on standardized
    residuals
```

Parameter	Default	Description
p	1	ARCH order (α)
q	1	GARCH order (β)
dist	normal	distribution: normal or t
lags	5	lags for ARCH-LM

```
1 load "retornos.csv" as df
2
3 // 1. Check for volatility clustering
4 archtest(df, ret, lags=10)
5
6 // 2. Estimate volatility model
7 let m_garch = garch(df, ret, p=1, q=1)
8 let m_gjr   = gjrgarch(df, ret, p=1, q=1)
9
10 // 3. Check diagnostics
11 archtest(m_garch, lags=10)
12
13 // 4. Estimated conditional variance
14 display m_garch
```

Listing 30.2: Typical workflow — asset returns

Note

HAYASHI estimates GARCH by MLE assuming Gaussian innovations (dist=normal) or Student-t (dist=t). For series with very heavy tails (cryptocurrencies, exchange rates in crises), the t distribution generally improves the fit. Compare AIC/BIC across specifications.

30.6 Empirical validation

The ARCH(1) DGP in this chapter corresponds to the `garch_bookvalidationcaseinvalidat`

Part IV

Causal Inference

Chapter 31

Propensity Score Matching

31.1 Selection on Observables

In many observational studies, treatment assignment is not random - individuals select into treatment based on observable characteristics X . If X also affects the outcome Y , direct comparison between treated and controls produces biased estimates.

Propensity Score Matching (PSM) exploits the Rosenbaum and Rubin (1983) theorem: if treatment assignment is independent of potential outcomes given X (*ignorability*), then the same independence holds given the propensity score $p(X) = P(D = 1 | X)$:

$$(Y^0, Y^1) \perp D | X \implies (Y^0, Y^1) \perp D | p(X)$$

This reduces the high-dimensional problem (conditioning on X) to a one-dimensional problem (conditioning on $\hat{p}$). PSM:

1. Estimates $\hat{p}(X_i)$ via logit (propensity score).
2. Matches each treated unit with the closest control(s) in $\hat{p}$.
3. Computes the ATT as the mean difference in the matched sample.

The common support (*overlap*) condition: $0 < p(X) < 1$ for all X is essential - units with probability 0 or 1 of treatment have no counterfactual in the other group.

31.2 DGP Verification

The DGP creates confounding via x_1 : those with high x_1 have a higher probability of treatment and also a higher outcome on average. The true ATT is $\tau = 2$.

$$P(D = 1 \mid x_1) = 0,3 + 0,4x_1 \in [0,30, 0,70] \quad y_i = 1 + 2d_i + 3x_{1i} + \varepsilon_i$$

```

1 seed(42)
2 input df
3 id
4 1
5 end
6 for i in 1..=10 { df = append(df, df) }
7 df |> filter(_n <= 1000)
8 generate df x1 = runiform()           // x1 ~ U(0,1)
9 generate df u = runiform()           // assignment
   noise
10 generate df d = (0.3 + 0.4*x1) > u    // P(D=1|x1)
   [0.30, 0.70]
11 generate df e = rnormal()
12 generate df y = 1.0 + 2.0*d + 3.0*x1 + e // true ATT = 2

```

Listing 31.1: DGP with selection on observables

Naive OLS vs. OLS with controls

```

1 let m_ols = ols(y ~ d, df)           // no control: biased
2 let m_adj = ols(y ~ d + x1, df)      // with control: unbiased
3 display m_ols
4 display m_adj

```

Listing 31.2: OLS comparison without and with x_1 control

```

OLS without control
const  2.9135*** (0.0398)    d  2.2215*** (0.0562)
N=1000  R²=0.6105

```

```

OLS with x1
const  1.5289***  (0.0193)    d  1.9762***  (0.0180)
x1      3.0053***  (0.0317)
N=1000  R2=0.9610

```

OLS without control overestimates τ by +0,22 (2.22 vs. 2.0) - omitted variable bias. Controlling for x_1 the OLS recovers $\hat{\tau} = 1,98$, indistinguishable from 2.

PSM without caliper

```

1 let m_psm = psm(y ~ d + x1, df, k=1, boot=200)
2 display m_psm

```

Listing 31.3: 1:1 PSM without caliper

Without a caliper, matching accepts pairs that are distant in $\hat{p}$. Balance may not improve much and the estimate can remain biased if treated units are matched to poor controls.

PSM with caliper

The caliper limits the maximum distance allowed in matching. The practical rule of Cochran and Rubin (1973) is $\text{caliper} = 0,2 \times \sigma_{\hat{p}}$, which corresponds approximately to 0,05 for uniform scores.

```

1 let m_cal = psm(y ~ d + x1, df, k=1, caliper=0.05, boot=200)
2 display m_cal

```

Listing 31.4: PSM with caliper = 0.05

With $\text{caliper} = 0,05$, the estimate should move closer to the true ATT at the cost of dropping treated units with no acceptable match. The caliper should also improve balance by rejecting distant propensity-score matches.¹

31.3 Syntax

¹The $\text{SMD} > 0.10$ warning persists because the conventional rule requires $\text{SMD} \leq 0,10$ (Austin 2009). In this DGP the confounding by x_1 is intense ($\beta_{x_1} = 3$) and $N = 1000$; with more observations the caliper could be narrowed to further improve balance.

```

1 psm(outcome ~ treatment + cov1 + cov2, df)
2 psm(outcome ~ treatment + cov1 + cov2, df, k=1, caliper=0.05,
    replace=false, boot=200)

```

The formula: `outcome ~ treatment + cov1 + cov2 + ....` The first regressor after `~` is the treatment variable; the rest are the covariates used in the propensity model.

Option	Default	Description
k	1	number of controls per treated unit
caliper	none	maximum distance in propensity score
replace	false	matching with replacement
boot	200	bootstrap replicas for SE

```

1 load "beneficiarios.csv" as df
2
3 // Selection covariates
4 let m = psm(renda ~ tratado + idade + educ + regioao, df,
5             k=1, caliper=0.05, boot=500)
6 display m

```

Listing 31.5: Typical workflow — program evaluation

Note

PSM is consistent under *ignorability* (selection on observables): $\{Y^0, Y^1\} \perp D \mid X$. Omitted variables that simultaneously affect D and Y invalidate identification – the same problem that makes IV necessary in ch. 19. Unlike DiD (ch. 21), PSM does not eliminate time-constant unobserved confounders.

31.4 Empirical validation

The Wooldridge `jtrain3` dataset is used in the `psm_jtrain3validationcaseinvalidation/cases_trainingprogrammebyPSMinHAYASHIandcomparesitwithaPythonreferenceimplementation.Runhay`

Chapter 32

Synthetic Control

32.1 The Single-Unit Problem

DiD (ch. 21) and PSM (ch. 31) depend on multiple treated units and comparable controls. In many policy applications, however, only one unit receives treatment - a country that adopts an economic program, a state that changes its legislation - and the potentially comparable units are few and heterogeneous.

Synthetic Control (Abadie, Diamond and Hainmueller, ADH 2010) constructs an artificial control group - the *synthetic control* - as a convex combination of donor units that replicates the trajectory of the treated unit *before* treatment. The central idea:

$$\hat{Y}_{1t}^{(0)} = \sum_{j=2}^{J+1} w_j^* Y_{jt} \quad w_j^* \geq 0, \quad \sum_{j=2}^{J+1} w_j^* = 1$$

The weights $W^* = (w_2^*, \dots, w_{J+1}^*)$ minimize the root mean squared prediction error (*RMSPE*) in the pre-treatment period:

$$W^* = \arg \min_{W \in \Delta^J} \sum_{t=1}^{T_0-1} \left(Y_{1t} - \sum_{j \geq 2} w_j Y_{jt} \right)^2$$

where Δ^J is the unit simplex ($w_j \geq 0$, $\sum w_j = 1$). The treatment effect in period $t \geq T_0$ is:

$$\hat{\tau}_t = Y_{1t} - \hat{Y}_{1t}^{(0)}$$

The validity of the method rests on the quality of the pre-treatment fit: if the synthetic control tracks the treated unit in the pre-treatment period, the hypothesis that it would have continued to do so in the absence

of treatment is plausible - a non-parametric form of the parallel trends hypothesis.

Note

The convexity restriction ($w_j \geq 0$, $\sum w_j = 1$) avoids extrapolation - the synthetic control lies within the support of the donors. This distinguishes it from DiD with weights, which can assign negative weights to achieve balance.

32.2 DGP Verification

The DGP uses a panel of 10 units $\times$ 20 periods. Unit 1 is treated from year 11 onward; units 2--10 are donors. The true effect is $\tau = 3,0$ (permanent, from $T_0 = 11$).

$$y_{it} = \alpha_i + 0,3t + 3,0 \cdot 1[i = 1, t \geq 11] + \varepsilon_{it} \quad \alpha_i \in \{2, 3, \dots, 10\}, \quad \alpha_1 = 5$$

The treated unit intercept ($\alpha_1 = 5$) coincides with that of unit 5 (donor), ensuring common support. The innovations $\varepsilon_{it} \sim U(0,1)$ introduce noise that prevents the SC from using only unit 5 as a perfect donor.

```
1 seed(42)
2 input df
3 id
4 1
5 end
6 for i in 1..=8 { df = append(df, df) }
7 df |> filter(_n <= 200)
8
9 generate df unit = (_n - 1) % 10 + 1
10 generate df year = (_n - (_n - 1) % 10 - 1) / 10 + 1
11
12 // intercepts by unit: donors receive alpha = unit; treated
   alpha = 5
13 generate df alpha = 0.0
14 let j = 2
15 while j <= 10 {
16     replace df alpha = j * 1.0 if unit == j
17     j = j + 1
```

```

18 }
19 replace df alpha = 5.0 if unit == 1
20
21 generate df d = (unit == 1) * (year >= 11)
22 generate df e = rnormal()
23 generate df y = alpha + 0.3 * year + 3.0 * d + e
24
25 let m = synth("y", 1, 11, df, id="unit", time="year")
26 display m

```

Listing 32.1: DGP: 10×20 panel, treatment on unit 1 from $t=11$

Panel Construction in Hayashi

The `generate` function operates on the `_n` vector (current row, 1-indexed). For an $N \times T$ panel in long format (stacked by unit), the formulas are:

Variable	Expression
Unit $(1, \dots, N)$	$(_n - 1) \% N + 1$
Period $(1, \dots, T)$	$(_n - (_n - 1) \% N - 1) / N + 1$

Results

Synthetic Control - Abadie, Diamond, Hainmueller (2010)

Treated unit : 1 Outcome: y

T (1st post-treat): 11 Donors: 9 T pre: 10 T post: 10

RMSPE pre : 0.3347

RMSPE post : 2.7056 post/pre ratio: 8.083

Donor weights ($w > 0.001$):

6	0.4420
3	0.2153
8	0.1747
2	0.1406
7	0.0274

Period	Actual	Synthetic	Effect
--------	--------	-----------	--------

	1	5.8266	6.0335	
	2	6.5321	6.1112	
	3	6.7458	6.5023	
	4	6.6240	6.8732	
	5	7.4867	7.2037	
	6	7.5689	7.4293	
	7	8.0670	7.6433	
	8	7.8780	8.0722	
	9	8.0202	8.3241	
	10	8.0878	8.6960	
*	11	11.4437	8.8167	2.6270
*	12	12.1888	9.4361	2.7527
*	13	12.0728	9.8984	2.1745
*	14	12.9748	9.5753	3.3995
*	15	12.7653	10.2134	2.5519
*	16	13.2120	10.2424	2.9696
*	17	13.3167	10.6701	2.6465
*	18	13.7205	11.0759	2.6446
*	19	14.3320	11.5282	2.8039
*	20	14.0020	11.7106	2.2913

* post-treatment

Reading the Results

Pre-treatment fit. The pre-treatment RMSPE is 0,33 for a series whose standard deviation is on the order of 2.5 - a reasonable fit given the DGP noise ($N = 200$, only 10 donors). The post/pre ratio = 8.1 signals that the post-treatment divergence is *eight times* larger than the pre-treatment fit error, rejecting the null hypothesis of no effect.

Weights. The algorithm distributes weights mainly among units 6, 3, 8 and 2, rather than concentrating all weight on unit 5 (which has $\alpha_5 = 5 = \alpha_1$). This occurs because the sample noise ε_{it} makes the realized trajectory of unit 5 an imprecise linear combination of unit 1 in the pre-treatment period. The SC uses multiple donors to cancel the noise.

Estimated effects. The average of the post-treatment effects is:

$$\bar{\hat{\tau}} = \frac{1}{10} \sum_{t=11}^{20} \hat{\tau}_t = 2,69$$

The true value is $\tau = 3,0$; the underestimation of $\sim 10\%$ reflects the imprecision of the pre-treatment fit with only 10 donors and 10 pre-periods.

32.3 Syntax

```
1 synth("outcome", treated_id, t0, df, id="unit_col", time="
   time_col")
2 synth("outcome", treated_id, t0, df, id="col", time="col",
   covs=["x1", "x2"])
```

Argument	Type	Description
outcome	string	outcome variable name
treated_id	num/str	treated unit identifier
t0	number	first post-treatment period
df	DataFrame	panel in long format
id=	option	unit identification column
time=	option	time column
covs=	list	additional covariates for the fit

The covs argument allows including pre-treatment covariate means in the minimization criterion, in addition to the outcome variable values. Useful when donors have similar trajectories but distinct structural characteristics.

```
1 load "estados.csv" as df // id, year, gdp, pop, educ
2
3 let m = synth("pib", "SP", 2008, df,
4               id="id", time="ano",
5               covs=["pop", "educ"])
6 display m
```

Listing 32.2: Typical workflow — state policy evaluation

Note

The SC does not provide standard asymptotic inference (no standard errors, no p -values). Inference is conducted via *placebo tests*: the SC is applied to each donor unit ``as if'' it were treated and the post/pre ratio is compared. If the ratio for the truly treated unit is exceptional in the set of placebos, the effect is statistically unusual. HAYASHI displays the post/pre ratio as a descriptive heuristic; placebo tests are implemented manually via a loop over units.

32.4 Empirical validation

The `synthsmokingvalidationcaseinvalidation/cases/estimatesasyntheticcontrolonasimulatedpane` treatment effect in HAYASHI with a Python reference implementation. Run `hay validate` to see the current

Part V

Qualitative Response and Limited Dependence

Chapter 33

Count Models

33.1 Count Data and the Inadequacy of OLS

Outcome variables that take only non-negative integer values ($Y \in \{0, 1, 2, \dots\}$) - number of children, doctor visits, registered patents - violate OLS assumptions. OLS can produce negative predictions and is inefficient: the variance of Y grows with the mean (right-skewed tail), violating homoskedasticity.

Poisson

The Poisson regression model specifies:

$$Y_i | x_i \sim \text{Poisson}(\lambda_i) \quad \lambda_i = E[Y_i | x_i] = \exp(\beta_0 + \beta_1 x_i)$$

The log-likelihood function is:

$$\ell(\beta) = \sum_{i=1}^n [y_i (\beta_0 + \beta_1 x_i) - \exp(\beta_0 + \beta_1 x_i) - \ln(y_i!)]$$

A useful property: the Poisson estimator converges to β even if Y is not strictly Poisson, as long as $E[Y | x] = \exp(x'\beta)$ (quasi-MLE estimator; Gourieroux, Monfort and Trognon 1984).

Negative Binomial

The Poisson model imposes $E[Y | x] = \text{Var}[Y | x]$ (equidispersion). When the variance exceeds the mean (*overdispersion*), the Negative Binomial model (NB2) generalizes:

$$\text{Var}[Y | x] = E[Y | x] + \alpha (E[Y | x])^2$$

The parameter $\alpha > 0$ captures overdispersion. Poisson is the limiting case $\alpha \rightarrow 0$.

Zero-Inflated Negative Binomial (ZINB)

Many applications present excess zeros beyond what any simple count model predicts - two-stage processes: first decide whether an event occurs ($D = 1$ with prob. π_i), then count the frequency. ZINB combines:

$$P(Y_i = 0) = (1 - \pi_i) + \pi_i (1 + \alpha \lambda_i)^{-1/\alpha} \quad P(Y_i = k) = \pi_i P_{\text{NB}}(k \mid \lambda_i, \alpha) \quad (k \geq 1)$$

33.2 DGP Verification

The DGP creates approximately Poisson counts with $\lambda_i = e^{0.5+x_i}$, $x \sim U(0,1)$. The estimator should recover $\beta_0 \approx 0.5$ and $\beta_1 \approx 1.0$.

```

1 seed(42)
2 generate df x      = runiform()                // x ~ U
   (0,1)
3 generate df lam = exp(0.5 + 1.0 * x)
4 generate df u    = runiform()
5 generate df y    = floor(lam + u)              // count
   0
6
7 let m_pois = poisson(y ~ x, df)
8 display m_pois
9
10 let m_nb   = nbreg(y_nb ~ x, df)
11 display m_nb
12
13 let m_zinb = zinb(y_zi ~ x, df)
14 display m_zinb

```

Listing 33.1: Poisson, NegBin and ZINB

Poisson

```

===== Poisson Regression Results =====
Dep. Variable:      y || Log-Likelihood:   102.0744   AIC:   -200.1488

```

Method:	IRLS	Pseudo R-sq:	0.7709	N:	500

	coef	std err	z	P> z	
const	0.5246	0.0602	8.709	0.000	***
x	0.9733	0.0960	10.141	0.000	***

$\hat{\beta}_0 = 0,52 \approx 0,5$ and $\hat{\beta}_1 = 0,97 \approx 1,0$ - estimates close to the true values.

Negative Binomial

===== Negative Binomial Regression (alpha=0.0010) =====					
	coef	std err	z	P> z	
const	0.4613	0.0177	26.025	0.000	***
x	1.1950	0.0276	43.249	0.000	***

$\hat{\alpha} \approx 0,001$ - minimal overdispersion, NegBin collapses to Poisson. With genuinely overdispersed data $\hat{\alpha}$ is significantly positive.

ZINB

===== Zero-Inflated Negative Binomial (ZINB) Results =====					
---- Count Model ----					
const	0.2994	(0.1347)	**		
x	1.2435	(0.2051)	***		
---- Inflate Model (Logit) ----					
z0	0.4392	(0.2264)	*		
z1	0.5121	(0.3828)			

The zero-inflation equation estimates the probability of a structural zero; the count equation models the process conditional on the event occurring.

33.3 Syntax

```

1 poisson(y ~ x1 + x2, df)
2 nbreg(y ~ x1 + x2, df)
3 zinb(y ~ x1 + x2, df)
4 zinb(y ~ x1 + x2, df, inflate = z1 + z2) // separate
   inflation equation

```

Estimator	When to use
poisson	count data with equidispersion; robust quasi-MLE
nbreg	overdispersion ($\hat{\alpha}$ sig. positive)
zinb	excess zeros beyond what NB expects

Note

Poisson and NB coefficients are log-ratios: $\beta_1 = 1$ implies that a unit increase in x multiplies the expected count by $e^1 \approx 2.72$. To obtain *average marginal effects* (AME): use `margins(model)`.

33.4 Empirical validation

The Wooldridge `fertil2` dataset is used in the validation cases `poissonfertil2` and

Chapter 34

Limited Dependent Variable Models

34.1 Censoring and Truncation

In many applications the outcome is only observed for part of the sample: wages of working women (selection), leisure hours truncated at zero (censoring), spending on luxury goods (many zeros, but not structural zeros). OLS applied to the censored variable is biased even with Gaussian errors.

Tobit — censoring at zero

The Tobit model (Tobin 1958) specifies a latent variable:

$$y_i^* = x_i' \beta + \varepsilon_i, \quad \varepsilon_i \sim N(0, \sigma^2) \quad y_i = \max(0, y_i^*)$$

OLS on y_i (ignoring censoring) underestimates β because the zeros ‘flatten’ the conditional distribution. The maximum likelihood estimator (MLE) uses the complete likelihood function:

$$\ell(\beta, \sigma) = \sum_{y_i=0} \ln \Phi\left(-\frac{x_i' \beta}{\sigma}\right) + \sum_{y_i>0} \ln \phi\left(\frac{y_i - x_i' \beta}{\sigma}\right) - \ln \sigma$$

Heckman — sample selection

Heckman's (1979) sample selection model separates:

1. **Selection equation:** $s_i = \mathbf{1}[\gamma_0 + \gamma_1 z_i + v_i > 0]$
2. **Outcome equation:** $y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$, observed only when $s_i = 1$.

If $\text{Cov}(\varepsilon_i, v_i) \neq 0$, OLS on the selected subsample is biased. The Heckman two-step correction adds the *Inverse Mills Ratio* (IMR) $\lambda_i = \phi(\hat{\gamma}'z_i)/\Phi(\hat{\gamma}'z_i)$ as a regressor to control for selection bias.

34.2 DGP Verification

```

1 seed(42)
2 generate df x      = rnormal()
3 generate df z      = rnormal()
4 generate df e      = (rnormal() - 0.5) * 1.7321 // ~ N(0,1)
5 generate df v      = (rnormal() - 0.5) * 1.7321
6
7 // Tobit: y* = 1 + 2*x + e; y = max(0, y*)
8 generate df ystar   = 1.0 + 2.0 * x + e
9 generate df y_tobit = ystar * (ystar > 0)
10
11 // Heckman: selected if 0.5 + 1.5*z + v > 0
12 generate df s       = (0.5 + 1.5 * z + v) > 0
13 generate df y_out   = (1.0 + 2.0 * x + e) * s

```

Listing 34.1: Tobit and Heckman DGP

Tobit

```

1 let m_tobit = tobit(y_tobit ~ x, df)
2 display m_tobit

```

Tobit - MLE (left-censoring at 0)

Obs: 500 Censored: 0 Uncens.: 500 Log-L: -359.2116 : 0.4963

Variable	coef	SE	z	P> z	
const	0.9050	0.0442	20.470	0.0000	***
x	2.1956	0.0787	27.913	0.0000	***
sigma	0.4963				

Tobit recovers $\hat{\beta}_0 = 0.905 \approx 1.0$ and $\hat{\beta}_1 = 2.196 \approx 2.0$, the latent variable y^* parameters. With zero censoring in this DGP (all $y^* > 0$), it is equivalent

to OLS - the model test is verified by the MLE's ability to converge and recover $\sigma \approx 0,5$.

Heckman

```
1 let m_heck = heckman(y_out ~ x, s ~ z, df)
2 display m_heck
```

Heckman Two-Step - Heckman (1979)

Obs (total): 500 Selected: 484
 : 0.3608 ^ _ : 0.4971 ^ = ~ _ : 0.1793

Outcome equation (y | z=1)

Variable	coef	SE	z	P> z	
const	0.9092	0.0455	20.001	0.0000	***
x	2.1821	0.0804	27.142	0.0000	***
lambda (IMR)	0.1793	0.1917	0.935	0.3495	

Selection equation: const 0.1752 z 8.6806**

The IMR coefficient is not significant ($p = 0,35$) - expected, since in the DGP $\text{Cov}(\varepsilon, v)$ is weak ($\rho = 0,36$). With $N = 500$, the power to detect moderate correlation is limited. The outcome equation correctly recovers $\hat{\beta}_1 \approx 2,18$.

34.3 Syntax

```
1 tobit(y ~ x1 + x2, df) // censoring
   at 0 (default)
2 tobit(y ~ x1 + x2, df, ll=0, ul=10) // double
   censoring [0, 10]
3 heckman(y ~ x1 + x2, s ~ z1 + z2, df) // two steps
```

Option	Description
ll	lower censoring limit (default: 0)
ul	upper censoring limit (default: none)

Warning

For Heckman to identify the model, the selection equation must contain at least one variable **excluded** from the outcome (z but not x). Without exclusion, the model is identified only by functional non-linearity - unstable estimates.

34.4 Empirical validation

The mroz dataset is used in the `heckman_mroz_validation` case in `validation/cases/`. The case estimates the selection model in HAYASHI and compares it with Heckit implementations in R and Python. Run `hay_val`.

Chapter 35

Multiple Choice Models

35.1 Ordinal and Nominal Response

When the outcome takes more than two categories, two structures are possible:

- **Ordinal response:** categories with a natural order (very dissatisfied, neutral, very satisfied; education level). Ordered models (logit/probit) preserve this structure.
- **Nominal response:** categories without order (choice of transportation mode; type of employment). Multinomial logit without order restriction.

Ordered Logit and Probit

The model assumes a latent variable $y^* = x'\beta + \varepsilon$ and cutpoints $c_1 < c_2 < \dots < c_{J-1}$:

$$y = j \iff c_{j-1} < y^* \leq c_j$$

Ordered logit uses $\varepsilon \sim \text{Logistic}(0,1)$; ordered probit uses $\varepsilon \sim N(0,1)$. Both estimate β and the cutpoints by MLE.

Multinomial Logit

For J alternatives, with alternative $j = J$ as base:

$$P(y_i = j \mid x_i) = \frac{\exp(\alpha_j + \beta_j' x_i)}{1 + \sum_{k=1}^{J-1} \exp(\alpha_k + \beta_k' x_i)} \quad j = 1, \dots, J-1$$

The coefficients (α_j, β_j) measure the effect of x on the relative odds of j versus the base alternative.

35.2 DGP Verification

```
1 seed(42)
2 generate df x      = rnormal()
3 generate df e      = (rnormal() - 0.5) * 1.7321
4 generate df ystar = 1.5 * x + e
5 // Categories 1, 2, 3 with cutoffs at 0.0 and 1.0
6 generate df y_ord = 1 + (ystar >= 0.0) + (ystar >= 1.0)
7
8 // Multinomial via inverse-CDF
9 generate df denom  = 1.0 + exp(-1.0 + 2.0*x) + exp(0.5)
10 generate df p1    = 1.0 / denom
11 generate df p12   = (1.0 + exp(-1.0 + 2.0*x)) / denom
12 generate df um    = rnormal()
13 generate df y_mlog = 1 + (um >= p1) + (um >= p12)
```

Listing 35.1: DGP for ordered and multinomial models

Ordered Logit

```
1 let m_olog = ologit(y_ord ~ x, df)
2 display m_olog
```

```
===== Ordered Logit Results =====
N=500  Log-L=-396.4473  Pseudo R2=0.1878
----- Coefficients -----
x          4.8137  (0.4633)  z=10.39  ***
Thresholds:  cut1=-0.1472  cut2=3.2191
```

Ordered Probit

```
1 let m_opro = oprobit(y_ord ~ x, df)
2 display m_opro
```

```
===== Ordered Probit Results =====
N=500  Log-L=-392.8369  Pseudo R2=0.1952
----- Coefficients -----
x          2.8683  (0.2603)  z=11.02  ***
Thresholds:  cut1=-0.0768  cut2=1.9076
```

The ratio between logit and probit coefficients is $4,81/2,87 \approx 1,68$, close to $\pi/\sqrt{3} \approx 1,81$ – the theoretical relationship between the two scales.

Multinomial Logit

```
1 let m_mlog = mlogit(y_mlog ~ x, df)
2 display m_mlog
```

```
===== Multinomial Logit Regression Results =====
N=500   Log-L=-513.7171   Pseudo R²=0.0448   AIC=1035.43
----- y=1 vs base y=3 -----
const   -0.0809   (0.2024)
x        -0.9006   (0.4069)   *
----- y=2 vs base y=3 -----
const   -1.5279   (0.2542)   ***
x         2.0825   (0.4119)   ***
```

For $y = 2$ versus base $y = 3$: $\hat{\beta}_x = 2,08$ (true DGP: $\beta_{x,2} - \beta_{x,3} = 2,0$). The negative sign of x in $y = 1$ vs $y = 3$ reflects that high x increases the odds of $y = 2$ (which has $e^{0,5}$ base probability) relative to $y = 1$.

35.3 Syntax

```
1 ologit(y ~ x1 + x2, df)
2 oprobit(y ~ x1 + x2, df)
3 mlogit(y ~ x1 + x2, df)
4 mlogit(y ~ x1 + x2, df, base=1)    // explicit base
   alternative
```

Note

Multinomial logit imposes the **independence of irrelevant alternatives** (IIA) property: the probability ratio between two alternatives does not depend on the others. For panel choice data with correlated alternatives, use `clogit` (ch. 47).

35.4 Empirical validation

The Wooldridge beauty dataset is used in the validation cases `ologitbeauty` and

Chapter 36

Survival Analysis

36.1 Failure Times and Censoring

Survival analysis models the *time to an event* T - firm bankruptcy, death, school dropout. The central challenge is **right censoring**: when the observation ends before the event, we only know $T > c$ (observation time c), not the exact time.

Two fundamental concepts:

$$S(t) = P(T > t) \quad (\text{survival function}) \qquad h(t) = \lim_{\delta \rightarrow 0} \frac{P(t \leq T < t + \delta \mid T \geq t)}{\delta} \quad (\text{hazard function})$$

$S(t)$ decreases from 1 to 0; $h(t)$ measures the instantaneous failure rate given that the individual survived to t .

Kaplan-Meier

The Kaplan-Meier (1958) non-parametric estimator estimates $S(t)$ without distributional assumptions:

$$\hat{S}(t) = \prod_{t_j \leq t} \left(1 - \frac{d_j}{n_j}\right)$$

where d_j is the number of events at time t_j and n_j the number at risk immediately before.

Cox — Proportional Hazard

The Cox (1972) model specifies:

$$h(t \mid x_i) = h_0(t) \exp(\beta' x_i)$$

The baseline hazard function $h_0(t)$ is left non-parametric. The coefficients β are estimated by partial likelihood, without specifying h_0 . The exponential e^{β_j} is the *hazard ratio*: a unit increase in x_j multiplies the hazard by e^{β_j} .

36.2 DGP Verification

The DGP uses Weibull failure times with shape parameter 1,5 and proportional hazard in $x \in \{0,1\}$. The true log-hazard ratio is $\beta_x = 0,8$.

```
1 seed(42)
2 generate df x      = rbernoulli(0.5)
3 generate df u      = runiform()
4 generate df t_true = exp(-ln(u) / exp(0.8 * x)) // Weibull,
      beta=0.8
5 generate df c      = 3.0
6 generate df time    = t_true * (t_true <= c) + c * (t_true > c
  )
7 generate df event   = (t_true <= c)
8
9 let m_km = km(time, event, df)
10 display m_km
11
12 let m_cox = cox(time ~ x, df, event=event)
13 display m_cox
```

Listing 36.1: Survival DGP — Weibull with proportional hazard

Kaplan-Meier

```
===== Kaplan-Meier Survival Estimate =====
Observations:          300
Events:                246
Median survival:       1.6319
```

The median survival is 1,63: half of the individuals experience the event before this time.

Cox

```
===== Cox Proportional Hazards Model =====
Observations: 300   Events: 246   Log-Likelihood: -1235.9213
Concordance:  0.6725
-----
Variable |      coef |   std err |      z | P>|z| | exp(coef)
x        |  0.7018 |   0.1306 |  5.372 | 0.000 |    2.0174
```

$\hat{\beta}_x = 0,702$ (true: 0,8) - within the confidence interval. The estimated hazard ratio $e^{0,702} = 2,02$: individuals with $x = 1$ have an instantaneous hazard twice as large as $x = 0$. The concordance index $C = 0,67$ indicates moderate discrimination (0,5 = random, 1,0 = perfect).

36.3 Syntax

```
1 km(time, event, df)           // Kaplan-Meier
2 cox(time ~ x1 + x2, df, event=event_col) // Cox PH
```

The event column must be 1 for event and 0 for censoring.

Note

The Cox model assumes *proportional hazards*: $h(t \mid x = 1)/h(t \mid x = 0)$ is constant over time. Violations can be detected via Schoenfeld residuals or interactions with $\ln(t)$.

Part VI

Advanced Methods

Chapter 37

Advanced Panel I — GMM, PCSE and GEE

37.1 Limitations of Static Estimators

Ch. 20 presented FE and RE for panels with a constant individual effect. Three important extensions arise in empirical applications:

1. **Dynamics:** when $y_{i,t-1}$ is a regressor, FE is biased (*Nickell bias*): $\hat{\rho}_{\text{FE}} \xrightarrow{p} \rho - O(1/T)$. The Arellano-Bond estimator uses first-difference GMM to circumvent this.
2. **Cross-sectional dependence:** when errors across units are correlated (macroeconomic data from countries), OLS and FE standard errors are invalid. PCSE (Beck and Katz 1995) corrects this.
3. **Non-Gaussian panel data:** GEE (Liang and Zeger 1986) extends marginal models to panel data with exponential families.

Arellano-Bond (AB)

The Arellano and Bond (1991) diff-GMM estimator takes first differences to eliminate the individual effect:

$$\Delta y_{it} = \rho \Delta y_{i,t-1} + \beta \Delta x_{it} + \Delta \varepsilon_{it}$$

Instruments: $y_{i,t-2}, y_{i,t-3}, \dots$ are valid for $\Delta y_{i,t-1}$ since they are uncorrelated with $\Delta \varepsilon_{it}$, as long as the level errors are not serially correlated of order 2 (m_2 test).

37.2 DGP Verification

```

1 seed(42)
2 generate df unit = (_n - 1) % 50 + 1
3 generate df t     = (_n - (_n-1) % 50 - 1) / 50 + 1
4 generate df alpha = unit * 0.05           // individual effect
   correlated with x
5 generate df x     = alpha + (rnormal() - 0.5) * 1.7321
6 generate df e     = (rnormal() - 0.5) * 1.7321
7 generate df y     = 1.0 + 1.0 * x + alpha + e // true: b=1.0
8
9 // biased OLS (mixes alpha_i with x)
10 let m_ols = ols(y ~ x, df)
11
12 // consistent FE (eliminates alpha_i)
13 let m_fe  = fe(y ~ x, df, id=unit)
14
15 // AB-GMM (dynamic with lags as instruments)
16 let m_ab  = ab(y ~ x, df, id=unit, time=t, lags=2)
17
18 // PCSE
19 let m_pcse = pcse(y ~ x, df, id=unit, time=t)
20
21 // GEE
22 let m_gee  = gee(y ~ x, df, id=unit)

```

Listing 37.1: Panel DGP with endogeneity via individual effect

OLS vs. FE

```

OLS:  const=1.330  x=1.736***  (bias: +0.74 from mixing alpha_i)
FE:           x=1.089***  (close to 1.0 - true effect)

```

Arellano-Bond

```

===== Arellano-Bond Diff-GMM (One-Step) =====
Observations: 900  Entities: 50  Instruments: 3  Sargan df: 1

Variable      |      Coef | Std Err |      z | P>|z|

```

```
LD.y      | -0.0242 |  0.0472 | -0.512 |  0.608Δ
x          |  1.0604 |  0.0464 | 22.869 |  0.000 ***
```

```
Sargan: ²(1)=0.7549  p=0.3849  (valid instruments)
AB m1: z=-6.22  p=0.000 ***  (AR(1) expected in FD)
AB m2: z=0.001  p=0.999      (no AR(2) - instruments OK)
```

AB estimates $\hat{\beta}_{\Delta x} = 1.06 \approx 1.0$. The lag $\hat{\rho}_{LD,y} \approx 0$ confirms that the DGP has no true dynamics. The Sargan test does not reject instrument validity; m_2 does not reject the absence of second-order autocorrelation.

PCSE

```
PCSE:  const=1.330  SE=0.0637  x=1.736  SE=0.0498
(larger SEs than OLS - corrects cross-sectional correlation)
```

GEE

```
GEE:  const=1.330 (SE robust=0.0612)  x=1.736 (SE robust=0.0330)
Scale: 0.3966  QIC: 399.8152
```

GEE with exchangeable correlation structure provides marginal (population-averaged) estimates, with robust standard errors for within-group dependence.

37.3 Syntax

```
1 ab(y ~ x, df, id=unit, time=t, lags=2)
2 pcse(y ~ x, df, id=unit, time=t)
3 gee(y ~ x, df, id=unit)
4 gee(y ~ x, df, id=unit, family=binomial, corstr=exchangeable)
```

Estimator	Assumption	When to use
ab	i.i.d. errors in levels	dynamic panel with small T
pcse	cross-sectional corr.	macro panel (small N , medium T)
gee	population margin	panel with non-Gaussian outcome

Chapter 38

Seemingly Unrelated Regressions

38.1 The Zellner Estimator

When estimating a system of M equations with the same (or different) regressors and errors are correlated across equations, equation-by-equation OLS is inefficient. Zellner's (1962) SUR (*Seemingly Unrelated Regressions*) estimator exploits this correlation via joint GLS.

System of M equations:

$$y_m = X_m \beta_m + \varepsilon_m, \quad m = 1, \dots, M$$

Stacking: $Y = X\beta + \varepsilon$, with $E[\varepsilon\varepsilon'] = \Sigma \otimes I_n$, where Σ is the cross-equation covariance matrix ($M \times M$).

The joint GLS estimator:

$$\hat{\beta}_{\text{SUR}} = (X'(\Sigma^{-1} \otimes I)X)^{-1} X'(\Sigma^{-1} \otimes I)Y$$

Efficiency gain: SUR $\geq$ OLS when Σ has non-zero off-diagonal elements - i.e., when equation errors are correlated. If Σ is diagonal or if the regressors are identical, SUR = OLS.

38.2 DGP Verification

DGP with two strongly correlated errors ($\rho = 0,7$):

$$y_1 = 1 + 2x_1 + \varepsilon_1 \quad y_2 = 0,5 + 1,5x_2 + \varepsilon_2$$

$$\text{Cov}(\varepsilon_1, \varepsilon_2) = 0,7 \sigma_1 \sigma_2$$

```

1 seed(42)
2 generate df x1 = rnormal()
3 generate df x2 = rnormal()
4 generate df e1 = (rnormal() - 0.5) * 1.7321
5 generate df e2 = 0.7 * e1 + 0.7141 * (rnormal() - 0.5) *
   1.7321
6 generate df y1 = 1.0 + 2.0 * x1 + e1
7 generate df y2 = 0.5 + 1.5 * x2 + e2
8
9 let m_ols1 = ols(y1 ~ x1, df)
10 let m_ols2 = ols(y2 ~ x2, df)
11 esttab(m_ols1, m_ols2)
12
13 let m_sur = sur(df, y1 ~ x1, y2 ~ x2)
14 display m_sur

```

Listing 38.1: SUR DGP and comparison with equation-by-equation OLS

Equation-by-Equation OLS

	(1)	(2)
	OLS	OLS
x1	2.1956*** (0.0788)	
x2		1.5097*** (0.0779)
Constant	0.9050***	0.4879***
N	500	500
R ²	0.6091	0.4300

SUR

```

Seemingly Unrelated Regressions (SUR)
Cross-Equation Error Correlation  $\Sigma()$ :
[ 0.2463  0.1689 ]

```

```
[ 0.1689 0.2419 ]
```

```
Equation: y1 - const=0.9513 x1=2.1003*** R²=0.6080
```

```
Equation: y2 - const=0.4591 x2=1.5653*** R²=0.4294
```

The estimated correlation between errors is $\hat{\Sigma}_{12}/\sqrt{\hat{\Sigma}_{11}\hat{\Sigma}_{22}} = 0,169/\sqrt{0,246 \times 0,242} \approx 0,69$, close to the true 0,7. SUR gains efficiency by reducing SEs: $\hat{\sigma}_{\text{SUR}}(x_1) < \hat{\sigma}_{\text{OLS}}(x_1)$.

38.3 Syntax

```
1 sur(df, y1 ~ x1 + x2, y2 ~ x3 + x4)
2 sur(df, y1 ~ x1, y2 ~ x2, y3 ~ x3)           // 3 equations
```

The first argument is the DataFrame; the remaining arguments are formulas (one per equation).

Note

If all equations have the same regressors, SUR = OLS and there is no efficiency gain. The gain is maximized when the regressors are orthogonal across equations and the cross-equation error correlation is high.

38.4 Empirical validation

The Wooldridge grunfeld investment dataset is used in the `sur_grunfeld_validation` case study. The `SUR` system in `HAYASHI` and compares the coefficients with reference implementations in `R`.

Chapter 39

Regularization and Variable Selection

39.1 The Dimensionality Problem

With k regressors and small N , OLS suffers from *overfitting*: it fits the sample noise beyond the signal, increasing out-of-sample error. When $k > N$, OLS is not even identified. Regularization penalizes the magnitude of the coefficients, trading bias for reduced variance.

LASSO

LASSO (Tibshirani 1996) minimizes:

$$\hat{\beta}^{\text{LASSO}} = \arg \min_{\beta} \left\{ \sum (y_i - x'_i \beta)^2 + \lambda \sum_{j=1}^k |\beta_j| \right\}$$

The ℓ_1 penalty forces small coefficients to exactly zero, performing automatic *variable selection*. The regularization path - LASSO for different λ - reveals which variables enter as λ decreases from ∞ (null model) to 0 (OLS).

Ridge

The ℓ_2 penalty of Ridge (Hoerl and Kennard 1970) shrinks coefficients toward zero without zeroing them:

$$\hat{\beta}^{\text{Ridge}} = (X'X + \lambda I)^{-1} X'y$$

Useful when many variables have small effects (continuum of weak predictors).
Does not perform variable selection.

Elastic Net

Elastic Net (Zou and Hastie 2005) combines ℓ_1 and ℓ_2 :

$$\min_{\beta} \left\{ \sum (y_i - x'_i \beta)^2 + \lambda \left[\rho \sum |\beta_j| + (1 - \rho) \sum \beta_j^2 \right] \right\}$$

Inherits variable selection from LASSO and stability in correlated groups from Ridge.

39.2 DGP Verification

DGP with 10 regressors, only x_1 and x_2 with non-zero effect. $N = 200$
- regime where regularization matters.

```
1 seed(42)
2 // 10 regressors; y = 1 + 2*x1 + 3*x2 + 0*(x3..x10) + e
3 generate df x1 = rnormal()
4 ...
5 generate df y = 1.0 + 2.0*x1 + 3.0*x2 + e
6
7 let m_ols = ols(y ~ x1+x2+x3+x4+x5+x6+x7+x8+x9+x10, df)
8 let m_lasso = lasso(y ~ x1+x2+x3+x4+x5+x6+x7+x8+x9+x10, df,
9   alpha=0.1)
10 let m_ridge = ridge_reg(y ~ x1+x2+x3+x4+x5+x6+x7+x8+x9+x10,
11   df, alpha=0.5)
12 let m_enet = enet(y ~ x1+x2+x3+x4+x5+x6+x7+x8+x9+x10, df,
13   alpha=0.5)
```

Listing 39.1: Sparse DGP — 2 of 10 active variables

OLS

```
OLS: R2=0.801 (good in-sample fit, but x3..x10 pollute with non-zero coefs)
const=0.68 x1=1.93*** x2=2.88*** x3=0.17 x4=0.07 ... x10=-0.11
```

LASSO

```
LASSO =0.1: R2=0.772  active vars: 2
      x1=1.552  x2=2.506  x3..x10 = 0.000  (correctly zeroed)
```

With $\alpha = 0.1$ LASSO identifies the two true predictors and zeros the eight irrelevant ones.

Ridge

```
Ridge =0.5: R2=0.801
      x1=1.878  x2=2.793  x3=0.172  x4=0.081  ...
      (shrinks but zeros none)
```

Elastic Net

```
ElasticNet =0.5 l1_ratio=0.5: R2=0.589  active vars: 2
      x1=0.789  x2=1.550  x3..x10 = 0.000
```

Elastic Net also selects x_1 and x_2 , but with greater total penalization (combining ℓ_1 and ℓ_2) the coefficients are more shrunk.

39.3 Syntax

```
1 lasso(y ~ x1 + x2 + x3, df, alpha=0.1)      // alpha =
   penalization lambda
2 ridge_reg(y ~ x1 + x2 + x3, df, alpha=0.5)
3 enet(y ~ x1 + x2 + x3, df, alpha=0.5)      // alpha = L1/(L1
   +L2) mix
```

Parameter	LASSO	Ridge / Enet
alpha	penalty λ	penalty λ
alpha=0	OLS	pure Ridge (Enet)
alpha=1	max L1	pure L1 (Enet)

Note

The optimal value of α is typically chosen by cross-validation (k -fold). In small samples, LASSO with α via *BIC-LASSO* tends to be more conservative than via CV.

39.4 Empirical validation

The Wooldridge `hprice1` house-price dataset is used in the validation cases `ridgehprice1`,

Chapter 40

Bootstrap

40.1 Inference by Resampling

The analytical standard errors derived from asymptotic theory generally assume that residuals follow a specific distribution (normal, homoskedastic, i.i.d.). Efron's (1979) **bootstrap** replaces distributional assumptions with resampling: it replicates the estimation process B times on samples drawn with replacement from the data itself.

Pairs Bootstrap

The simplest approach resamples pairs (y_i, x_i) :

1. For $b = 1, \dots, B$: sample n observations with replacement.
2. Estimate $\hat{\theta}^{(b)}$ on the resampled dataset.
3. $\widehat{SE}_{\text{boot}} = \text{sd}(\hat{\theta}^{(1)}, \dots, \hat{\theta}^{(B)})$.

The 95% bootstrap percentile CI is the interval $[Q_{0,025}, Q_{0,975}]$ of the empirical distribution of $\hat{\theta}^{(b)}$.

Advantage: valid without knowing the distribution of $\hat{\theta}$ - robust to non-Gaussian errors, heteroskedasticity, and small samples where the normal approximation is poor.

40.2 DGP Verification

DGP with uniform error on $[-2, 2]$ - non-Gaussian. With $N = 200$, the analytical OLS SE may underestimate the true variance.

```
1 seed(42)
2 generate df x = rnormal()
3 generate df e = (rnormal() - 0.5) * 4.0      // U(-2, 2) - non
      -Gaussian
4 generate df y = 1.0 + 2.0 * x + e
5
6 // standard OLS
7 let m_ols = ols(y ~ x, df)
8 display m_ols
9
10 // Bootstrap
11 let m_boot = bootstrap("ols", y ~ x, df, reps=500)
12 display m_boot
```

Listing 40.1: Comparison of analytical SE vs. bootstrap

Analytical OLS

```
OLS: R2=0.178
    const=1.046  SE=0.157  x=1.828  SE=0.279
```

Bootstrap ($B = 500$)

Bootstrap SE - ols (n=500/500)					
Variable ^		SE orig.	SE boot	CI lower	CI upper
const	1.0462	0.1566	0.1545	0.7235	1.3710
x	1.8276	0.2787	0.2679	1.3313	2.3910

Bootstrap SE (0,155 and 0,268) and analytical (0,157 and 0,279) are similar here, indicating that the analytical OLS SEs are adequate even with uniform errors when $N = 200$. The agreement validates the implementation: both converge to the same asymptotic standard deviation.

40.3 Syntax

```

1 // generic bootstrap - any implemented estimator
2 bootstrap("ols",    y ~ x, df, reps=500)
3 bootstrap("logit",  y ~ x, df, reps=500)
4 bootstrap("qreg",   y ~ x, df, reps=500, q=0.5)
5
6 // alias bootse
7 bootse(y ~ x, df, reps=500)           // ols by default
   (legacy)

```

Option	Default	Description
reps / n	1000	number of bootstrap replicas
alpha	0.05	level for CIs (two-tailed)

Note

The bootstrap estimator can also estimate SEs for PSM (ch. 31), where the asymptotic distribution of the ATT is non-standard. In this context, the pairs bootstrap is consistent (Abadie and Imbens 2008).

Chapter 41

Multivariate Analysis

41.1 Dimensionality reduction and latent structure

When observing a vector of variables $\mathbf{v} = (v_1, \dots, v_p)'$ that share common variation, it is useful to summarize that variation in a few dimensions. Two approaches complement each other:

- **PCA:** maximizes variance explained by orthogonal components (algebraic solution - spectral decomposition of the covariance matrix).
- **Factor Analysis:** models variables as a linear combination of unobserved latent factors + uniqueness (idiosyncratic part).

Principal Component Analysis (PCA)

$$\mathbf{v} = W\mathbf{f} + \mathbf{e} \quad W = \text{eigenvectors of } \Sigma_v$$

The j -th principal component $\text{PC}_j = w_j' \mathbf{v}$ captures the j -th largest fraction of total variance. Eigenvalues measure the variance of each PC; the sum of all eigenvalues is $\text{tr}(\Sigma_v) = \sum \text{Var}(v_j)$.

Factor Analysis

$$\mathbf{v} = \Lambda \mathbf{f} + \boldsymbol{\delta} \quad \mathbf{f} \perp \boldsymbol{\delta}$$

Λ are the *factor loadings*: λ_{ij} measures how much factor j contributes to v_i . The *communality* $h_i^2 = \sum_j \lambda_{ij}^2$ is the fraction of variance of v_i explained by the common factors.

Canonical Correlation

Given two sets of variables $X = (x_1, \dots, x_p)$ and $Y = (y_1, \dots, y_q)$, canonical analysis finds vectors a and b such that $\text{Corr}(a'X, b'Y)$ is maximized. It provides $\min(p, q)$ canonical pairs.

41.2 DGP Verification

DGP with 2 latent factors and 5 observed variables.

```
1 seed(42)
2 generate df f1 = rnormal()
3 generate df f2 = rnormal()
4 // v1, v2: high loadings on f1; v3, v4: high loadings on f2;
   v5: both
5 generate df v1 = 0.9*f1 + 0.1*f2 + e1
6 generate df v2 = 0.8*f1 + 0.2*f2 + e2
7 generate df v3 = 0.1*f1 + 0.9*f2 + e3
8 generate df v4 = 0.2*f1 + 0.8*f2 + e4
9 generate df v5 = 0.5*f1 + 0.5*f2 + e5
10
11 let m_pca = pca(df, v1, v2, v3, v4, v5, n=2)
12 display m_pca
13
14 let m_fac = factor(df, v1, v2, v3, v4, v5, n=2, rotation=
   varimax)
15 display m_fac
16
17 let m_can = cancorr(df, xvars=["v1", "v2"], yvars=["v3", "v4", "
   v5"])
18 display m_can
```

Listing 41.1: Factorial DGP and multivariate analysis

PCA

Principal Component Analysis

Component	Var Expl.	% Accum.	Eigenvalue
PC1	0.5939	0.5939	2.9695
PC2	0.2437	0.8377	1.2187

Loadings:

	PC1	PC2
v1	0.677	-0.641
v2	0.808	-0.450
v3	0.693	0.623
v4	0.790	0.466
v5	0.868	-0.002

PC1 (59,4% of variance) has positive loadings on all variables - captures the common component. PC2 (24,4%) contrasts the f_1 group (v1, v2) with the f_2 group (v3, v4). The two PCs account for 83,8% of total variance.

Factor Analysis

Factor Analysis - Varimax			
Variable	F1	F2	Communal.
v1	0.027	-0.932	0.869
v2	0.255	-0.889	0.856
v3	0.931	-0.048	0.869
v4	0.888	-0.228	0.841
v5	0.613	-0.614	0.753

After varimax rotation, F1 loads on v3 and v4 (associated with latent factor f_2); F2 loads on v1 and v2 (factor f_1). Communalities above 0,75 confirm that common factors explain a large part of individual variance.

Canonical Correlation

```

===== Canonical Correlation Analysis =====
Observations: 300  Wilks' Lambda: 0.5396  F=35.54  p=0.000
  CC1: 0.6706    CC2: 0.1396
X vars: v1, v2  |  Y vars: v3, v4, v5
=====

```

The first canonical correlation $\hat{\rho}_1 = 0,671$ captures the association between block {v1, v2} and block {v3, v4, v5} through the shared latent factor.

41.3 Syntax

```
1  pca(df, v1, v2, v3, ..., n=2)                // n
   components to retain
2  factor(df, v1, v2, v3, ..., n=2, rotation=none) // rotation:
   none / varimax
3  cancorr(df, xvars=["x1", "x2"], yvars=["y1", "y2"])
```

Chapter 42

Generalized Linear Models

42.1 The exponential family

Generalized Linear Models (Nelder and Wedderburn 1972) unify OLS, Logit, Poisson and many other estimators under a common framework. Three components define a GLM:

1. **Random component:** $Y_i \sim$ exponential family with mean μ_i .
2. **Linear predictor:** $\eta_i = x_i' \beta$.
3. **Link function:** $g(\mu_i) = \eta_i$, connecting the mean to the linear scale.

Family	Default link	Variance	Application
Gaussian	identity	σ^2	OLS
Binomial	logit	$\mu(1 - \mu)$	binary data
Poisson	log	μ	count data
Gamma	log	μ^2/ϕ	positive data with heavy tails

Estimation by IRLS (Iteratively Reweighted Least Squares) is efficient and converges quickly.

GLM Gamma — positive data with heavy tails

Expenditures, loan sizes and durations follow right-skewed distributions with variance proportional to the square of the mean. The Gamma GLM with log link models $E[Y \mid x] = \exp(x' \beta)$ with $\text{Var}[Y \mid x] \propto \mu^2$.

PPML — international trade

The PPML (Poisson Pseudo MLE) estimator by Santos Silva and Tenreyro (2006) uses the Poisson estimating equation even when Y is not an integer (e.g., trade values). It is robust to zeros and to multiplicative heteroscedasticity – common problems in gravity equations of trade.

42.2 DGP Verification

```
1 seed(42)
2 generate df x = rnormal()
3 generate df mu = exp(0.5 + 1.0 * x)
4 // Gamma: mean=mu, variance ~ mu^2 (simulated via lognormal)
5 generate df y_gamma = mu * exp(rnormal() * 0.6)
6
7 let m_gamma = glm(y_gamma ~ x, df, family=gamma, link=log)
8 display m_gamma
9
10 // PPML with "trade" data (mix of zeros and positives)
11 generate df x2 = rnormal()
12 generate df trade = round(exp(0.3 + 1.2*x + 0.8*x2) * (
    runiform() > 0.2))
13 let m_ppml = glm(trade ~ x + x2, df, family=poisson, link=log
    )
14 display m_ppml
```

Listing 42.1: GLM Gamma and PPML

GLM Gamma

The Gamma model should recover positive intercept and slope estimates close to the log-mean DGP, with dispersion summarising the remaining relative variance.

PPML

PPML should recover positive effects for both covariates while handling zeros in the trade variable.

42.3 Syntax

```

1 glm(y ~ x, df, family=gaussian, link=identity) // OLS via
   GLM
2 glm(y ~ x, df, family=binomial, link=logit)    // Logit via
   GLM
3 glm(y ~ x, df, family=gamma,    link=log)      // Gamma with
   log link
4 glm(y ~ x, df, family=poisson,  link=log)      // Poisson /
   PPML

```

Family	Available links
gaussian	identity, log, inverse
binomial	logit, probit, cloglog
poisson	log, identity, sqrt
gamma	log, identity, inverse

Chapter 43

Regime-Switching Models

43.1 Structural heterogeneity over time

Many economic series exhibit structural changes: different means, volatilities or dynamics across distinct periods. Two approaches capture this heterogeneity:

- **Markov Switching** (Hamilton 1989): latent regime follows a Markov chain. The transition probability between regimes is estimated by the EM algorithm together with the parameters of each regime.
- **Threshold Regression** (Hansen 1997): regime change is determined by an observable variable (threshold variable q) crossing a threshold γ to be estimated.

Markov-Switching AR(p)

$$y_t = \mu_{S_t} + \sum_{k=1}^p \phi_k^{(S_t)} (y_{t-k} - \mu_{S_t}) + \sigma_{S_t} \varepsilon_t$$

where $S_t \in \{1, \dots, K\}$ is the latent regime with transition probabilities $P(S_t = j \mid S_{t-1} = i) = p_{ij}$. The parameter μ_{S_t} is the regime intercept, $\phi_k^{(S_t)}$ are the AR coefficients and σ_{S_t} is the regime standard deviation.

43.2 DGP Verification

DGP with two alternating regimes:

- Regime 1 (expansion): $\mu_1 = 1,0$, $\phi_1 = 0,30$, $\sigma_1 = 0,5$
- Regime 2 (recession): $\mu_2 = -1,0$, $\phi_2 = 0,70$, $\sigma_2 = 1,5$

```

1 seed(42)
2 generate df t      = _n
3 generate df regime = (t % 60 < 40)    // ~2/3 in regime 1
4 // y recursive with parameters conditional on regime
5 ...
6 let m_ms = markov(df, y, k=2, p=1)
7 display m_ms

```

Listing 43.1: DGP Markov Switching AR(1) and estimation

Markov Switching

```

===== Markov Switching AR(1) - 2 regimes =====
Observations: 299    Log-L: -155.76    AIC: 327.51    BIC: 357.12
----- Transition Matrix -----
[ 0.9493    0.0507 ]
[ 0.0251    0.9749 ]
----- Regime 0 (Recession) -----
Intercept: -1.2864    AR.L1: 0.6169    Variance: 0.5141
----- Regime 1 (Expansion) -----
Intercept:  0.9946    AR.L1: 0.2956    Variance: 0.0625
----- Expected Durations -----
Regime 0: 19.7 periods    Regime 1: 39.9 periods

```

The model recovers the two regimes with good accuracy:

- Regime 0 (recession): $\hat{\mu} = -1,29$ (true: $-1,0$), $\hat{\phi} = 0,62$ (true: $0,7$). Expected duration 19,7 periods.
- Regime 1 (expansion): $\hat{\mu} = 0,99 \approx 1,0$, $\hat{\phi} = 0,30$ (true: $0,3$). Expected duration 39,9 periods.

The transition matrix indicates high persistence of both regimes ($p_{00} = 0,95$, $p_{11} = 0,97$) - the series rarely switches regime, consistent with the DGP (60% of the time in regime 1).

43.3 Syntax

```

1 markov(df, y, k=2, p=1)    // k regimes, AR of order p

```

Option	Default	Description
k	2	number of regimes
p	1	AR order within each regime

Note

The number of regimes k is fixed and must be specified a priori. Information criteria (AIC, BIC) compared across models with $k = 2$ and $k = 3$ guide this choice. For data with $N < 200$, $k > 3$ is rarely identified reliably.

Chapter 44

Time Series Filters and Decomposition

44.1 Separation of components

A time series can be decomposed into:

$$y_t = \tau_t + c_t + s_t + \varepsilon_t$$

where τ_t is the trend, c_t the cyclical component, s_t the seasonality and ε_t the idiosyncratic noise. Linear filters extract one or more of these components from the observed series.

Hodrick-Prescott Filter (HP)

The HP filter (Hodrick and Prescott 1997) separates trend from cycle by minimizing:

$$\min_{\{\tau_t\}} \left\{ \sum_{t=1}^T (y_t - \tau_t)^2 + \lambda \sum_{t=2}^{T-1} [(\tau_{t+1} - \tau_t) - (\tau_t - \tau_{t-1})]^2 \right\}$$

The parameter λ penalizes changes in the second difference of the trend. Canonical values: $\lambda = 1600$ (quarterly), 100 (annual), 14400 (monthly).

Baxter-King Filter (BK)

The BK filter (Baxter and King 1999) extracts a *frequency-band* component: it retains periodic variations between $[p_L, p_H]$ periods and eliminates trend and high frequencies. For business cycles: $p_L = 6$, $p_H = 32$ quarters (1,5

to 8 years).

Holt-Winters Exponential Smoothing (ETS)

The ETS (*Error-Trend-Seasonal*) model by Holt and Winters combines: level, trend and seasonality equations updated recursively with exponentially declining weights. Parameters (α, β, γ) control the speed of adaptation of each component.

44.2 DGP Verification

DGP with linear trend, annual cycle and semi-annual seasonality, $T = 120$ months.

```
1 seed(42)
2 generate df t      = _n
3 generate df trend  = 0.05 * t
4 generate df cycle   = 0.5 * sin(t * 3.14159 / 12)
5 generate df season  = 0.3 * sin(t * 3.14159 / 6)
6 generate df noise   = (rnormal() - 0.5) * 0.3
7 generate df y       = 10.0 + trend + cycle + season + noise
8
9 hprescott(df, y, lambda=1600)
10 display df
11
12 baxter_king(df, y, low=6, high=32, k=12)
13 display df
14
15 let m_hw = holtwinters(df, y, trend=add, seasonal=add, period
16               =12)
17 display m_hw
```

Listing 44.1: Series with trend, cycle and seasonality

Hodrick-Prescott

```
hpfilter: =1600 → y_trend and y_cycle added to df
t=1:  y=10.34  y_trend=10.56  y_cycle=-0.22
t=2:  y=10.62  y_trend=10.56  y_cycle= 0.06
...
```

The HP adds columns `y_trend` (smoothed trend) and `y_cycle` (cycle = residual).

Baxter-King

The BK adds `y_cycle` with the frequency-band component. The first and last $k = 12$ periods are set to NaN - the cost of two-sided filters.

Holt-Winters ETS

```
===== ETS(A,A,A) =====
Observations:      120
Alpha:             0.8000      (level)
Beta:              0.0500      (trend)
Gamma:             0.6000      (seasonality)
SSE:               5.3570
AIC:               -339.09
BIC:               -291.70
=====
```

The ETS(A,A,A) model - additive errors, additive trend, additive seasonality - fits the smoothing parameters (α, β, γ) minimizing SSE. High α (0,80) indicates fast level response.

44.3 Syntax

```
1 hprescott(df, var, lambda=1600)           // adds
   var_trend and var_cycle
2 baxter_king(df, var, low=6, high=32, k=12) // adds var_cycle
3 holtwinters(df, var, trend=add, seasonal=add, period=12)
4 ets(df, var, trend=add, seasonal=add, period=12) // alias
```

Filter	Output	Typical use
HP	trend + cycle	quarterly macro
BK	band-pass cycle	business cycle analysis
Holt-Winters	ETS model + forecast	short-term forecasting

Note

The HP filter on end-of-sample data suffers from the *end-point problem*: trend and cycle are revised as new data arrives. The BK avoids this problem by using a symmetric window (cost: loses k observations at each end).

Chapter 45

Structural VAR

45.1 From reduced form to structure

The reduced-form VAR (ch. 28) provides $K^2 \cdot p + K$ free parameters. The reduced-form impulse response measures the effect of *innovations* u_t with no economic content. The SVAR recovers *structural shocks* ε_t (supply, demand, monetary policy) by imposing restrictions that reflect economic theory:

$$A y_t = A_1^* y_{t-1} + \dots + B \varepsilon_t \quad \varepsilon_t \sim (0, I_K)$$

Cholesky Identification

The lower triangular decomposition of $\Sigma_u = P P'$ imposes a recursive causal ordering: the variable in position 1 is contemporaneously exogenous; the one in position 2 responds contemporaneously only to the first; etc.

$$B = P \quad (\text{lower triangular}) \implies y_{1,t} \perp \varepsilon_{2,t} \text{ in the same period}$$

The structural IRF $\Theta_h = \Phi_h B$ measures the response of $y_{k,t+h}$ to a unit shock in $\varepsilon_{j,t}$.

45.2 DGP Verification

DGP: bivariate VAR(1) with true Cholesky restriction (y2 responds contemporaneously to y1, but not vice versa).

```
1 seed(42)
```

```

2 generate df t = _n
3 // VAR(1): y1 and y2 with cross coefficients
4 ...
5 replace df y1 = 0.5*ly1 + 0.3*ly2 + e1t if t == it
6 replace df y2 = 0.2*ly1 + 0.4*ly2 + 0.6*e1t + e2t if t == it
7
8 let m_var = var(df, y1, y2, lags=1)
9 display m_var
10
11 let m_svar = svar(df, y1, y2, lags=1, id=cholesky)
12 display m_svar
13
14 let m_sirf = sirf(m_svar, steps=10)
15 display m_sirf

```

Listing 45.1: Structural DGP and Cholesky SVAR

Reduced-form VAR

```

Vector Autoregression (VAR(1)):  Observations=299  AIC=-2.83  BIC=-2.76
Residual Covariance  $\Sigma(\_u)$ :
[ 0.2452    0.1617 ]
[ 0.1617    0.3375 ]

```

SVAR — B Matrix

```

===== SVAR(1) - Cholesky =====
B Matrix (structural impact):
[ 0.4952    0.0000 ]    <- shock 1 affects y1, does not affect y2 contemporaneously
[ 0.3265    0.4805 ]    <- shock 2 affects y2; y2 responds to shock 1

```

The zero column in B_{12} reflects the Cholesky ordering: shock 2 does not affect y_1 contemporaneously. $B_{21} = 0,33$ estimates the contemporaneous effect of shock 1 on y_2 (true DGP: $0,6 \times \sigma_{\varepsilon_1}$).

Structural IRF

```

Impulse: Shock 1 (Var0)
h=0:  y1=+0.495  y2=+0.327  (contemporaneous impact)
h=1:  y1=+0.324  y2=+0.224

```

```

h=5:  y1=+0.063  y2=+0.044  (progressive dampening)
h=9:  y1=+0.012  y2=+0.009

Impulse: Shock 2 (Var1)
h=0:  y1= 0.000  y2=+0.481  (Cholesky restriction: y1 does not respond)
h=1:  y1=+0.115  y2=+0.170

```

The response of y_1 to shock 2 at $h = 0$ is exactly zero - the Cholesky identification restriction applied. At horizon $h = 1$, y_1 begins to respond via the lagged VAR channel.

45.3 Syntax

```

1 svar(df, y1, y2, lags=1, id=cholesky)      // SVAR with
   Cholesky identification
2 svar(df, y1, y2, lags=1, id=longrun)      // BQ: long-run
   restrictions
3 sirf(m_svar, steps=10)                    // structural IRF

```

Identification	Restrictions
cholesky	lower triangular - recursive short-run restrictions
longrun	Blanchard-Quah - long-run restrictions

Note

The ordering of variables in the Cholesky SVAR is crucial: the variable listed first is treated as contemporaneously exogenous to all others. The choice of ordering should be guided by economic theory or by results that are robust to permutations of the order.

Part VII

Complementary Methods

Chapter 46

Classical Inference

46.1 Parametric and non-parametric hypothesis tests

The tests presented in this chapter answer the question: *are the data consistent with a hypothesis about the population distribution?*

Student's t-test

For a sample of size n with unknown variance, the statistic:

$$t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}} \sim t_{n-1}$$

tests $H_0: \mu = \mu_0$. For two independent samples, Welch's test uses approximate degrees of freedom by the Satterthwaite method, without assuming homoscedasticity.

One-way ANOVA

Generalizes the t -test to $K > 2$ groups. Decomposes total variation:

$$SS_{\text{total}} = SS_{\text{between}} + SS_{\text{within}}$$

$$F = \frac{SS_{\text{between}}/(K-1)}{SS_{\text{within}}/(N-K)} \sim F_{K-1, N-K}$$

H_0 : all group means are equal.

Kruskal-Wallis

Non-parametric alternative to one-way ANOVA. Replaces observations with *ranks* and compares rank sums across groups. Statistic:

$$H = \frac{12}{N(N+1)} \sum_{k=1}^K \frac{R_k^2}{n_k} - 3(N+1) \sim \chi_{K-1}^2$$

valid even without normality, as long as the distributions have a similar shape.

46.2 DGP Verification

DGP with 3 groups of different sizes; the expected value increases linearly with the group. $N = 300$, $\sigma = 1$.

```
1 seed(42)
2 ...
3 generate df g    = (_n <= 150) * 1 + (_n > 150) * 2
4 generate df yg   = 2.0 + e + (g - 1) * 0.5 // g=1: mu=2.0, g
   =2: mu=2.5
5
6 // One-sample t-test: H0: mu = 2.0
7 ttest(df, y, mu=2.0)
8
9 // Two-sample t-test (Welch): H0: mu(g1) = mu(g2)
10 ttest(df, yg, by=g)
11
12 // Paired: before vs after
13 ttest(df, y_antes, y_depois, paired=true)
14
15 // ANOVA: H0: mu1 = mu2 = mu3
16 anova(df, y3, by=k)
17
18 // Kruskal-Wallis (non-parametric)
19 kruskal_wallis(df, y3, by=k)
```

Listing 46.1: Hypothesis tests

t-test — one sample

```
One-sample t-test: y    H0: mean = 2.0
t = -0.6946    df = 299    p = 0.4879
```

Does not reject H_0 - sample mean 1,98 is compatible with 2,0.

t-test — two groups (Welch)

```
Two-sample t-test (Welch): yg by g
  group 1: n=150  mean=1.971  SE=0.040
  group 2: n=150  mean=2.489  SE=0.041
Welch's t = -9.0552    df = 297.38    p = 0.0000
```

Rejects H_0 : difference of 0,52 $\approx$ true difference of 0,50.

ANOVA

```
===== One-Way ANOVA
=====
```

Source	SS	df	MS	F	P>F
Between	51.195	2	25.597	104.037	0.0000
Within	73.074	297	0.246		
Total	124.269	299			

$F = 104,0$, $p < 0,001$: rejects H_0 decisively.

Kruskal-Wallis

```
H = 120.38    df = 2    p = 0.0000
```

Confirms the ANOVA without assuming normality.

46.3 Syntax

```
1 ttest(df, var, mu=0.0)                // one sample
2 ttest(df, var, by=group)              // two groups (Welch
  )
3 ttest(df, var, by=group, unequal=false) // two groups (
  Student)
4 ttest(df, var1, var2, paired=true)    // paired
```

```
5 anova(df, var, by=group)           // one-way ANOVA
6 kruskal_wallis(df, var, by=group)  // Kruskal-Wallis
```

Note

Welch's test does not assume homoscedasticity and is preferable to the classical Student's t -test in almost all practical cases. The loss of power is negligible when variances are equal; the gain when they differ is substantial. If you wish to run the classical Student's t -test, use the option `unequal=false`.

Chapter 47

Advanced Panel II

47.1 Beyond basic FE and RE

The estimators from ch. 37 (AB-GMM, PCSE, GEE) deal with dynamics and heteroscedasticity. Here we cover four complementary approaches:

- FE - reference; standard fixed effects (within estimator)
- Mundlak (CRE) - Correlated Random Effects; tests RE vs FE via γ
- xtglS - panel GLS; flexible regarding the structure of Σ
- mixedlm - mixed model with random intercepts
- clogit - conditional logit for binary panel variables

Mundlak Estimator (CRE)

Mundlak (1978) shows that the RE estimator is equivalent to FE if:

$$\alpha_i = \gamma' \bar{x}_i + u_i \quad u_i \perp x_{it}$$

Regresses y_{it} on x_{it} *and* $\bar{x}_i$ (group mean). If $\hat{\gamma} \neq 0$, the fixed effect is necessary.

Panel GLS (xtglS)

Panel GLS specifies a structure for the error covariance across entities. With panels=hetero, each panel has its own variance σ_i^2 , allowing heteroscedasticity across panels.

Mixed Model (mixedlm)

The linear mixed model specifies:

$$y_{it} = \beta x_{it} + u_i + \varepsilon_{it} \quad u_i \sim N(0, \sigma_u^2)$$

where u_i is a random intercept effect per entity. It differs from standard RE by using REML and integrating over u_i via maximum likelihood.

47.2 DGP Verification

DGP: endogenous static panel. $y = 1 + 2x + \alpha_i + \varepsilon$; $\alpha_i = 0.05 \cdot i$ correlated with x (via z). $N = 50$ entities, $T = 20$ periods.

```

1 seed(42)
2 generate df unit = (_n - 1) % 50 + 1
3 generate df t    = (_n - (_n - 1) % 50 - 1) / 50 + 1
4 generate df alpha = unit * 0.05
5 generate df z     = (rnormal() - 0.5) * 1.7321
6 generate df x     = 0.5 * z + alpha + (rnormal() - 0.5)
7 generate df y     = 1.0 + 2.0 * x + alpha + e
8
9 let m_fe = fe(y ~ x, df, id=unit)
10 let m_mundlak = mundlak(df, y ~ x, id="unit")
11 let m_gls = xtglsl(y ~ x, df, id=unit, time=t)
12 let m_mix = mixedlm(y ~ x, df, id="unit")
13 generate df yb = (y > 2.5)
14 let m_cl = clogit(yb ~ x, df, group="unit")

```

Listing 47.1: Panel II: FE, Mundlak, xtglsl, mixedlm, clogit

```

FE (within):      x = 2.041 ***      (true: 2.0)
Mundlak:          x = 2.040,    (x) = 0.939 *** → rejects RE
xtglsl (hetero):  x = 2.768 ***      (biased without
demeaning)
mixedlm:          x = 2.134 ***      (random intercept: ²_u =
0.388)
clogit (yb ~ x):  x = 1.397 ***      (groups with no
variation dropped)

```

Mundlak Test

The test $H_0: \gamma = 0$ (means uncorrelated with x) rejects with $F = 419,96$, $p < 0,001$. This confirms that x is endogenous - the RE estimator would be inconsistent and FE should be used.

The xtglsl without FE removal overestimates β because it captures part of the individual effect. The mixedlm stays close to FE because it treats α_i as a normal random effect.

47.3 Syntax

```

1 fe(y ~ x, df, id=unit)           // Fixed Effects (
   within)
2 mundlak(df, y ~ x, id="unit")    // CRE: includes
   group means
3 xtglsl(y ~ x, df, id=unit, time=t) // panel GLS
4 xtglsl(y ~ x, df, id=unit, time=t, panels="corr") //
   correlated  $\Sigma$ 
5 mixedlm(y ~ x, df, id="unit")    // mixed model (
   REML)
6 clogit(yb ~ x, df, group="unit") // conditional
   logit

```

Note

In clogit, groups where y does not vary (all 0 or all 1) are automatically dropped because they do not contribute to the conditional likelihood. This reduces the effective sample, but is theoretically correct.

Chapter 48

Unit Roots II

48.1 Beyond the ADF: PP and Zivot-Andrews

The previous chapter (ch. 27) covered the ADF test, which corrects for residual autocorrelation via lags. Two problems remain:

1. The ADF is sensitive to conditional heteroscedasticity of errors (GARCH etc.)
2. The ADF has low power in the presence of a *structural break*: a stationary series with a level break looks like a random walk.

Phillips-Perron Test (PP)

Phillips and Perron (1988) adopt a non-parametric approach: they correct the ADF statistic for heteroscedasticity and autocorrelation of *any form* via the Newey-West estimator:

$$Z_{\alpha} = T(\hat{\rho} - 1) - \frac{T^2 \hat{\sigma}^2}{2 \hat{s}_T^2} (\hat{\lambda}^2 - \hat{\gamma}_0)$$

where $\hat{s}_T^2$ is the long-run variance estimated with the Bartlett kernel. The asymptotic distribution is identical to that of the ADF (Dickey-Fuller).

Zivot-Andrews Test (ZA)

Zivot and Andrews (1992) relax the assumption of no structural break. The test estimates the break date *endogenously* as the value that minimizes the t statistic:

$$\hat{t}_B = \arg \min_{\lambda} t_{\alpha}(\lambda), \quad \lambda = t_B/T \in (0.15, 0.85)$$

The tested equation includes a level shift or trend change:

$$\Delta y_t = c + \beta t + \gamma DU_t(\lambda) + \alpha y_{t-1} + \sum_{j=1}^k d_j \Delta y_{t-j} + \varepsilon_t$$

H_0 : unit root without structural break.

48.2 DGP Verification

```

1 seed(42)
2 // Random walk: rw_t = rw_{t-1} + e_t
3 // Trend-stationary series: stat_t = 0.5*t + e
4 // Series with break at t=150: AR(1) + level shift of 3.0
5
6 phillips_perron(df, rw)
7 phillips_perron(df, stat)
8 zivot(df, break_t)

```

Listing 48.1: PP and Zivot-Andrews

PP - rw:	Zt = 77.365	CV 5%=-2.860	DOES NOT reject H
	(unit root)		
PP - stat:	Zt = 2058	CV 5%=-2.860	DOES NOT reject H
	(trend-stationary)		
ZA - break_t: t	= -11.597	CV 5%=-4.800	REJECTS H, break
	at obs=147		

The PP for stat (linear trend + noise) does not reject H_0 because the standard test (without a deterministic trend in the auxiliary equation) cannot distinguish stochastic from deterministic trend. The ZA correctly identifies the structural break at $t = 147$ (close to the true $t = 150$) and rejects H_0 .

48.3 Syntax

```

1 phillips_perron(df, var)           // PP without trend
2 phillips_perron(df, var, trend=true) // PP with
   deterministic trend

```

```

3 zivot(df, var)                                // Zivot-Andrews (endogenous
    level break)
4 zivot(df, var, type="trend")                  // break in trend

```

Test	Correction	Break?	CV 5%
ADF	Parametric lags	No	−2,86
PP	Non-param. Newey-West	No	−2,86
ZA	PP + endogenous break	Yes	−4,80

Note

Trend-stationary series and series with structural breaks are two cases where the ADF/PP can lead to wrong conclusions. The ZA resolves the second; for the first, including a deterministic trend in the auxiliary equation (`trend=true`) is sufficient.

Chapter 49

Series Decomposition II

49.1 STL and Christiano-Fitzgerald

Ch. 44 presented HP, BK and Holt-Winters. This chapter covers two additional methods:

STL *Seasonal-Trend decomposition via Loess* (Cleveland et al. 1990): decomposes $y_t = T_t + S_t + R_t$ iteratively via local regression (loess). Robust to outliers; allows time-varying seasonality.

Christiano-Fitzgerald (CF) Asymmetric band-pass filter (2003): extracts frequencies in $[p_L, p_H]$ using weights from a non-symmetric window that adapts to the position in the series. Unlike the BK, it does not lose edge points - the cost is that weights change at each observation.

STL: iterative decomposition

The STL algorithm alternates between:

1. *Seasonal loop*: smooths each seasonal position with loess
2. *Trend loop*: smooths the deseasonalized series with loess

Convergence produces T_t (trend), S_t (seasonality) and R_t (residual) with robustness properties to outliers when `robust=true`.

Christiano-Fitzgerald Filter

The CF applies weights a_j satisfying $\sum_j a_j e^{i\omega j} \approx 1$ for $\omega \in [\omega_L, \omega_H]$ and ≈ 0 outside the band. For t at the edges, the weights adjust to avoid using future observations.

49.2 DGP Verification

DGP: monthly series with linear trend, annual cycle and semi-annual seasonality.
 $T = 120$ months.

```

1 seed(42)
2 generate df trend  = 0.05 * t
3 generate df cycle  = 0.5 * sin(t * 3.14159 / 12)
4 generate df season = 0.3 * sin(t * 3.14159 / 6)
5 generate df y      = 10.0 + trend + cycle + season + noise
6
7 stl(df, y, period=12)
8 display df
9
10 christiano_fitzgerald(df, y, low=6, high=32)
11 display df

```

Listing 49.1: STL and Christiano-Fitzgerald

STL

```

Seasonal Decomposition (STL)
  Observations: 120
  Trend mean:   13.0243
  Seasonal std: 0.1943
  Residual std: 0.1868

```

The STL adds columns `y_trend`, `y_seasonal` and `y_resid` to the DataFrame. Residual std $< 0,2$ - the true noise ($\sigma = 0,3$) is well separated from the other components.

Christiano-Fitzgerald

```

cffilter: periods [6,32] → y_cycle added to df
  t=1: y_cycle = 1.04
  t=6: y_cycle = 1.19      (dominant annual cycle)
  t=12: y_cycle = 0.87

```

The CF retains periodic variations between 6 and 32 months (1,5 to 8 years - business cycles). The advantage over the BK is that all 120 observations are kept (no edge trimming).

49.3 Syntax

```

1 stl(df, var, period=12)                                // adds trend,
   seasonal, resid
2 christiano_fitzgerald(df, var, low=6, high=32) // adds
   var_cycle

```

Filter	Type	Edge loss	Seasonality
HP	low-pass	Yes (endpoints)	No
BK	band-pass	Yes (k each side)	No
CF	band-pass	No	No
STL	decomposition	No	Yes

Note

The function `hamilton` in Hayashi refers to the Markov Switching model of Hamilton (1989), *not* to the Hamilton filter (2018) (ch. 43). The Hamilton 2018 filter - which proposes replacing the HP with an OLS regression with 4 lags and horizon $h = 8$ - is not implemented separately.

Chapter 50

SVAR with Long-Run Restrictions

50.1 Blanchard-Quah Identification

Ch. 45 (Cholesky SVAR) imposes *short-run* restrictions: the variable in position 1 is not contemporaneously affected by the second. Blanchard and Quah (1989) propose an alternative via *long-run* restrictions: permanent and transitory shocks.

The central idea: in macroeconomics, a supply shock has a permanent effect on output (y), while a demand shock has only a transitory effect. The restriction:

$$\sum_{h=0}^{\infty} \Theta_h^{(1,1)} = \text{free}, \quad \sum_{h=0}^{\infty} \Theta_h^{(1,2)} = 0$$

imposes that shock 2 does not affect y in the long run. This restriction identifies the decomposition B without any contemporaneous impact restriction.

BQ Decomposition

Given the reduced-form VAR $y_t = \Phi_1 y_{t-1} + u_t$:

1. Compute the sum $C = (I - \Phi_1)^{-1}$ - long-run multiplier
2. Factor $C \Sigma_u C' = F F'$ (Cholesky of F)
3. Extract $B = C^{-1} F$ - the long-run structural shocks

50.2 DGP Verification

DGP: bivariate VAR(1) where y_1 is a random walk (shock 1 permanent), y_2 is stationary and responds transitorially to shock 1.

```
1 seed(42)
2 // y1_t = y1_{t-1} + e1_t (random walk)
3 // y2_t = 0.5*y2_{t-1} + 0.4*e1_t + 0.6*e2_t (stationary)
4
5 let m_bq = svar(df, y1, y2, lags=1, id=longrun)
6 display m_bq
7 let m_irf_bq = sirf(m_bq, steps=20)
8 display m_irf_bq
```

Listing 50.1: SVAR Blanchard-Quah

B Matrix (structural shocks)

```
SVAR(1) - Long-run restrictions
B Matrix:
[ 0.4907    0.0358 ]
[ 0.1935    0.3031 ]
```

The matrix B is not triangular - unlike Cholesky, long-run restrictions do not imply contemporaneous impact restrictions.

Structural IRF

```
Impulse: Shock 1 (permanent)
h= 0:  y1=+0.491  y2=+0.194
h= 1:  y1=+0.464  y2=+0.087
h= 5:  y1=+0.405  y2=-0.003  (effect on y2 decays)
h=19:  y1=+0.271  y2=-0.005  (y1 retains permanent effect)

Impulse: Shock 2 (transitory)
h= 0:  y1=+0.036  y2=+0.303
h= 1:  y1=+0.016  y2=+0.144
h= 5:  y1=-0.001  y2=+0.007  (effect on y1 decays to ~0)
h=19:  y1=-0.001  y2= 0.000  (transitory confirmed)
```

Shock 2 affects y_1 only transitorially (sum $\rightarrow 0$), confirming the long-run restriction. Shock 1 has a growing cumulative effect on y_1 - consistent with the unit root.

50.3 Syntax

```
1 svar(df, y1, y2, lags=1, id=longrun)      // BQ: long-run
   restrictions
2 svar(df, y1, y2, lags=1, id=cholesky)    // Cholesky:
   contemporaneous restrictions
3 sirf(m_svar, steps=20)                   // structural IRF
```

Note

BQ identification requires the variables to be integrated of order 1 ($I(1)$). For stationary series, $(I - \hat{\Phi}_1)^{-1}$ is finite and the permanent/transitory decomposition has no natural interpretation. Verify unit roots (ch. 27, 48) before applying BQ.

Chapter 51

Local and Weighted Estimation

51.1 Weights, windows and smoothing

Classical estimation (OLS, MLE) treats each observation symmetrically. Three alternatives allow the model to adapt locally:

WLS Weighted least squares: observations with lower variance (or greater relevance) receive higher weight.

Rolling OLS Estimates the model in sliding time windows, revealing how coefficients evolve over time.

LOWESS Nonparametric smoothing: for each point x_0 , fits a locally weighted regression using nearby neighbors.

WLS (Weighted Least Squares)

The WLS estimator minimizes:

$$\hat{\beta}_{WLS} = \arg \min_{\beta} \sum_{i=1}^n w_i (y_i - x_i' \beta)^2 = (X' W X)^{-1} X' W y$$

where $W = \text{diag}(w_1, \dots, w_n)$. If the weights are proportional to $1/\sigma_i^2$ (error variances), WLS is the efficient estimator under heteroscedasticity (= GLS for independent errors).

Rolling OLS

For time series non-stationary in parameters (time-varying coefficients), rolling OLS with window h estimates:

$$\hat{\beta}_t = (X'_{t-h:t} X_{t-h:t})^{-1} X'_{t-h:t} Y_{t-h:t}$$

LOWESS

LOWESS (Cleveland 1979) estimates $m(x_0) = E[y|x = x_0]$ locally:

1. Selects the $f \cdot n$ nearest neighbors of x_0
2. Applies tricubic weights: $w_i = (1 - |d_i|^3)^3$
3. Fits a weighted regression (linear or constant)
4. Iterates with robust reweighting (optional)

51.2 DGP Verification

DGP: heteroscedastic error $\varepsilon_i \propto (1 + x_i)$; small coefficient on t (0,01).
 $N = 300$ observations.

```
1 seed(42)
2 generate df w = exp(-0.002 * t)           // weight: more recent
   observations more relevant
3 generate df e = (rnormal()-0.5)*2*(1+x)    //
   heteroscedasticity in x
4 generate df y = 1.0 + 2.0*x + 0.01*t + e
5
6 let m_ols = ols(y ~ x, df)
7 let m_wls = wls(y ~ x, df, weights="w")
8 let m_rol = rols(y ~ x + t, df, window=50)
9 lowess(df, y, x, frac=0.3)
```

Listing 51.1: WLS, Rolling OLS, LOWESS

```
OLS:  const=2.362, x=2.263*** (ignores heteroscedasticity)
WLS:  const=2.212, x=2.257*** (weights correct growing
   variance)
Rolling OLS (last window t=251..300):
   const=1.898, x=2.411, t=0.006
```

```
LOWESS: frac=0.30, 300 obs → y_hat_lowess added to df
```

WLS changes the coefficients only slightly here - temporal weights compensate for the error growth in x only partially. Rolling OLS reveals that the coefficient of x in the last window (2,41) is slightly higher, suggesting mild non-stationarity.

51.3 Syntax

```
1 wls(y ~ x, df, weights="weight_column")      // WLS
2 rols(y ~ x + t, df, window=50)              // Rolling OLS
3 rolling(y ~ x + t, df, window=50)           // alias
4 lowess(df, y_var, x_var, frac=0.67, it=3)    // LOWESS
```

Note

LOWESS receives the dependent variable **before** the independent one: `lowess(df, y, x)`. This differs from conventional regressions. The parameter `frac` (fraction of neighbors used at each point) controls smoothing: smaller values → noisier curve; larger values → smoother.

Chapter 52

AutoReg and ARDL

52.1 Models with lags in y and x

The ARIMA model (ch. 27) deals only with the dynamics of y . In economics, however, the current level of y depends not only on its own past, but also on current and past values of regressors x . The *Autoregressive Distributed Lag* (ARDL) model captures exactly this.

AutoReg(p): pure AR via OLS

The AR(p) model writes y_t as a linear function of its own lags:

$$y_t = c + \sum_{i=1}^p \phi_i y_{t-i} + \varepsilon_t$$

Unlike ARIMA (estimated by MLE), `autoreg()` estimates via OLS. This is computationally simpler and asymptotically equivalent for stationary processes. The `trend` option allows including a constant and/or a deterministic trend:

trend	Deterministic component
"c"	Constant (default)
"ct"	Constant + linear trend
"t"	Trend only
"n"	None

ARDL(p, q): distributed lags

The $ARDL(p, q)$ model combines p lags of y with the contemporaneous value and q lags of each regressor x :

$$y_t = c + \sum_{i=1}^p \phi_i y_{t-i} + \sum_{j=0}^q \beta_j x_{t-j} + \varepsilon_t$$

The **long-run multiplier** of x on y - the total accumulated effect after all dynamic adjustments - is:

$$\theta = \frac{\sum_{j=0}^q \beta_j}{1 - \sum_{i=1}^p \phi_i}$$

For the model to be stationary, $|\sum_{i=1}^p \phi_i| < 1$ is required.

Relationship with the VECM and the Pesaran test

When y and x are integrated of order 1 ($I(1)$), the ARDL can be rewritten in error correction form. Pesaran, Shin and Smith (2001) propose the *bounds test*: tests H_0 of no long-run relationship via the F statistic applied to the unrestricted ARDL, comparing with tabulated critical values. The advantage over Johansen (ch. 29) is that the bounds test works regardless of whether x is $I(0)$ or $I(1)$.

52.2 DGP Verification

DGP: $y_t = 1 + 2x_t + \varepsilon_t$, no autocorrelation in y . $N = 300$, $x \sim U(-\sqrt{3}, \sqrt{3})$, $\varepsilon \sim U(-0.5, 0.5)$.

```
1 seed(42)
2 // ... (300 obs, generation of x and e)
3 generate df y = 1.0 + 2.0 * x + 0.5 * (rnormal() - 0.5)
4
5 // Pure AR(2): no true autocorrelation in y → coefs ~0
6 let m_ar = autoreg(df, y, lags=2)
7 display m_ar
8
9 // AR(1) with deterministic trend
10 let m_ar_ct = autoreg(df, y, lags=1, trend="ct")
11 display m_ar_ct
12
```

```

13 // ARDL(1,1)
14 let m_ardl = ardl(y ~ x, df, lags=1, xlags=1)
15 display m_ardl
16
17 // ARDL(2,2): check whether extra lags are significant
18 let m_ardl2 = ardl(y ~ x, df, lags=2, xlags=2)
19 display m_ardl2

```

Listing 52.1: AutoReg and ARDL

Pure AR(2)

```
===== AutoReg(2)
```

```
=====
```

```
Observations:          298
```

```
R-squared:           0.0004
```

```
AIC:                905.2221
```

```
----- Coefficients
```

```
-----
```

	coef	std err	t
		P> t	

```
-----
```

const	0.9904	0.1022	9.6914
-------	--------	--------	--------

```
0.0000
```

y.L1	0.0007	0.0580	0.0118
------	--------	--------	--------

```
0.9906
```

y.L2	-0.0209	0.0581	-0.3597
------	---------	--------	---------

```
0.7193
```

```
=====
```

$R^2 = 0,0004$ - lags of y explain nothing, as per the DGP (no true autocorrelation). Coefficients $y.L1$ and $y.L2$ do not differ from zero.

AR(1) with trend

```
===== AutoReg(1)
```

```
=====
```

```

Trend:                                ct
                                     coef      std err          t
                                     P>|t|

-----
const                                0.9275      0.1394      6.6551
      0.0000
trend                                0.0003      0.0007      0.4172
      0.6768
y.L1                                 0.0018      0.0581      0.0310
      0.9753
=====

```

The deterministic trend ($p = 0,68$) and the lag ($p = 0,98$) are not significant
- consistent with a DGP without trend or persistence.

ARDL(1,1)

```

===== ARDL(1, 1)
=====
Observations:                        299
R-squared:                            0.9836
AIC:                                -318.6638

                                     coef      std err          t
                                     P>|t|

-----
const                                1.0018      0.0594     16.8580
      0.0000
y.L1                                 0.0062      0.0583      0.1061
      0.9155
x1                                   1.9648      0.0148    133.1872
      0.0000
x1.L1                               -0.0078      0.1156     -0.0675
      0.9462
=====

```

The model recovers the DGP accurately: $\text{const} \approx 1,0$, $x1 \approx 2,0$. The coefficient of $x1.L1$ is not significant ($p = 0,95$), confirming no lagged effect of x in the DGP. The long-run multiplier is:

$$\theta = \frac{1,9648 + (-0,0078)}{1 - 0,0062} \approx 1,97$$

close to the true value of 2,0.

ARDL(2,2)

```
===== ARDL(2, 2)
```

```
=====
```

```
Observations:          298
R-squared:          0.9838
AIC:              -317.3857
```

	coef	std err	t
		P> t	
const	1.0727	0.0833	12.8843
0.0000			
y.L1	0.0013	0.0582	0.0220
0.9824			
y.L2	-0.0662	0.0582	-1.1385
0.2558			
x1	1.9628	0.0147	133.1320
0.0000			
x1.L1	0.0018	0.1154	0.0154
0.9877			
x1.L2	0.1264	0.1153	1.0968
0.2736			

The AIC of ARDL(2,2) (-317,4) is slightly *higher* (worse) than that of ARDL(1,1) (-318,7) - the extra lags do not compensate for the loss of degrees of freedom. AIC/BIC selection favors the more parsimonious model.

52.3 Syntax

```

1 // AR(p) via OLS
2 autoreg(df, var, lags=1)           // AR(1) with
   constant
3 autoreg(df, var, lags=p, trend="ct") // AR(p) with
   trend
4
5 // ARDL(p, q)
6 ardl(y ~ x, df, lags=p, xlags=q)   // one
   regressor
7 ardl(gdp ~ inv + exp, df, lags=2, xlags=2) // multiple
   regressors

```

Model	Command	Use case
AR(p)	<code>autoreg(df, y, lags=p)</code>	Pure dynamics of y
ARIMA(p, d, q)	<code>arima(df, y, p=, d=, q=)</code>	$I(1)$ with MA
ARDL(p, q)	<code>ardl(y~x, df, lags=p, xlags=q)</code>	y and x with lags
VECM	<code>vecm(...)</code>	Multivariate cointegration

Note

For $I(1)$ series, the ARDL can be rewritten in error correction form. The Pesaran et al. (2001) bounds test tests the existence of a long-run relationship without requiring all regressors to be $I(0)$ or all $I(1)$ – an advantage over the Johansen test. The F statistic of the test is accessible via `m_ardl.bounds_test(y, x)` in the Rust API of Greeners, but is not yet exposed as a DSL command.

Chapter 53

Kalman Filter

53.1 Hidden state and noisy observation

The Kalman filter solves a different problem from all previous models: estimating a *latent* variable (the ``state'') that is not directly observed, based on noisy measurements over time.

The general state-space model is:

$$y_t = H s_t + \varepsilon_t, \quad \varepsilon_t \sim N(0, R) \quad (53.1)$$

$$s_t = F s_{t-1} + R_s \eta_t, \quad \eta_t \sim N(0, Q) \quad (53.2)$$

Equation (53.1) is the **observation equation**: y_t is what is measured, s_t is the latent state and ε_t is measurement noise. The equation (53.2) is the **transition equation**: the state evolves according to a linear dynamic perturbed by η_t .

Pre-defined models

Hayashi offers two ready-made models for scalar series:

Local Level (LL):

$$y_t = \mu_t + \varepsilon_t, \quad \mu_t = \mu_{t-1} + \eta_t$$

The state μ_t is a random walk - the level drifts over time. σ_ε controls observation noise; σ_η controls the rate at which the level changes.

Local Linear Trend (LLT):

$$y_t = \mu_t + \varepsilon_t, \quad \mu_t = \mu_{t-1} + \nu_{t-1} + \eta_t, \quad \nu_t = \nu_{t-1} + \zeta_t$$

The state is two-dimensional: level μ_t and slope ν_t . The slope varies smoothly over time.

Kalman Filter (forward pass)

The filter processes observations recursively:

1. **Prediction:** $\hat{s}_{t|t-1} = F \hat{s}_{t-1|t-1}$, $P_{t|t-1} = F P_{t-1|t-1} F' + R_s Q R_s'$
2. **Innovation:** $v_t = y_t - H \hat{s}_{t|t-1}$, $S_t = H P_{t|t-1} H' + R$
3. **Kalman Gain:** $K_t = P_{t|t-1} H' S_t^{-1}$
4. **Update:** $\hat{s}_{t|t} = \hat{s}_{t|t-1} + K_t v_t$, $P_{t|t} = (I - K_t H) P_{t|t-1}$

The result $\hat{s}_{t|t}$ is the *filtered state*: optimal estimate of the state using $y_1, \dots, y_t$ (information up to t).

Rauch-Tung-Striebel Smoother (backward pass)

After the filter, the RTS traverses the series backward and incorporates future information:

$$L_t = P_{t|t} F' P_{t+1|t}^{-1}, \quad \hat{s}_{t|T} = \hat{s}_{t|t} + L_t (\hat{s}_{t+1|T} - \hat{s}_{t+1|t})$$

The result $\hat{s}_{t|T}$ is the *smoothed state*: uses *all* the sample $y_1, \dots, y_T$. The variance of the smoothed state is always less than or equal to that of the filtered state - future information reduces uncertainty.

53.2 DGP Verification

DGP: true level $\mu_t = 10 + 0.1t$ (linear trend) observed with noise $\varepsilon_t \sim N(0, 1)$. $T = 150$.

```
1 seed(42)
2 generate df mu = 10.0 + 0.1 * t
3 generate df y = mu + noise          // noise ~ N(0,1)
4
5 // Local Level (sigma specified)
6 kalman(df, y, model="ll", sigma_obs=1.0, sigma_state=0.1)
7
8 // Local Linear Trend (automatic sigmas)
```

```
9 kalman(df, y, model="llt")
```

Listing 53.1: Kalman Filter: LL and LLT

Output — Local Level

```
Kalman (ll):  T=150  loglik=-278.1966  _obs=1.0000  _state
              =0.1000→
              y_filtered, y_smoothed added to df
```

Output — Local Linear Trend

```
Kalman (llt):  T=150  loglik=-219.2354  _obs=0.9663  _state
              =0.0966  _slope=0.0097→
              y_filtered, y_smoothed, y_slope_filtered, y_slope_smoothed
              added to df
```

The LLT log-likelihood ($-219,2$) is substantially higher than the LL's ($-278,2$) because the DGP has a linear trend - the model with a free slope better captures the data structure.

Comparison with the true state

First 8 observations (LLT):

t	mu	y	y_smoothed	y_slope_smoothed
1	10.10	10.19	9.74	0.13
2	10.20	10.35	9.87	0.13
3	10.30	10.77	9.99	0.13
4	10.40	10.07	10.10	0.13
5	10.50	8.89	10.21	0.13
6	10.60	10.31	10.33	0.12
7	10.70	11.52	10.46	0.12
8	10.80	12.01	10.56	0.12

```
summarize y_smoothed:  mean=17.49  sd=4.35
summarize mu:          mean=17.55  sd=4.34
```

The smoothed state recovers the true level with remarkable accuracy: mean 17,49 vs 17,55. The smoothed slope ($y_slope_smoothed \approx 0,13$) is close

to the true value of 0.1. The raw series y oscillates by ± 2 around μ ; the smoother eliminates this noise.

Note

Filtered vs smoothed: `y_filtered` uses only $y_1, \dots, y_t$ - it is causal and useful for real-time forecasting. `y_smoothed` uses the entire $y_1, \dots, y_T$ - it is retrospective, but optimal for historical analysis. At interior points, the smoothed state is always more accurate; at the last point $t = T$, the two coincide.

53.3 Syntax

```

1 // Local Level
2 kalman(df, y, model="ll")
3 kalman(df, y, model="ll", sigma_obs=1.0, sigma_state=0.1)
4
5 // Local Linear Trend
6 kalman(df, y, model="llt")
7 kalman(df, y, model="llt", sigma_obs=0.5, sigma_state=0.05,
  sigma_slope=0.005)

```

Parameter	Default	Description
<code>model</code>	"ll"	"ll" or "llt"
<code>sigma_obs</code>	$\text{sd}(\text{diff}(y))/\sqrt{2}$	Standard deviation of observation noise
<code>sigma_state</code>	$\text{sigma_obs} \times 0.1$	Standard deviation of level noise
<code>sigma_slope</code>	$\text{sigma_state} \times 0.1$	Standard deviation of slope noise (LLT)

Columns added to DataFrame:

<code>{var}_filtered</code>	-	Filtered level (forward pass)
<code>{var}_smoothed</code>	-	Smoothed level (RTS)
<code>{var}_slope_filtered</code>	-	Filtered slope (LLT only)
<code>{var}_slope_smoothed</code>	-	Smoothed slope (LLT only)

Model	Use when	Relation
LL	Level drift without clear trend	Local level UCM (ch. 44)
LLT	Time-varying trend	Generalized stochastic HP filter

Note

Hayashi uses the RTS smoother with fixed matrices specified by the user. For joint estimation of variance parameters (σ_ε , σ_η) via maximum likelihood, use the `ucm()` command (ch. 44), which runs the EM algorithm on the state-space representation.

Advanced Installation

53.4 Compilation with ODBC support

To connect HAYASHI to databases via ODBC (SQL Server, Oracle, etc.):

```
cargo install hayashi-lang --features odbc
```

Requires an ODBC driver installed on the system (unixODBC on Linux).

53.5 Environment variables

Variable	Effect
HAY_HISTORY	Path to REPL history file
HAY_NOCOLOR	Disables colors in error messages
HAY_TRACE	Enables execution trace (debugging)

53.6 Modules and optional features

Advanced econometric models

The estimators below require the greeners crate as a dependency of the interpreter. In development builds (cargo build), it is compiled automatically. No additional installation is required.

Group	Estimators
Panel data	fe, re, ab, pcse, gee, mundlak, xtglsl, mixedlm, clogit
Qualitative response	logit, probit, mlogit, ologit, oprobit
Count/limited	poisson, nbreg, zinb, tobit, heckman
General GLM	glm (5 families, multiple links)
Regularization	lasso, ridge_reg, enet
Survival	km, cox
Multivariate	pca, factor, cancorr, manova
Time series	var, svar, vecm, garch, markov, arima
Filters	hprescott, baxter_king, holtwinters, stl, christiano_fitzgerald
Causal inference	psm, synth, did, rd

Memory usage

Computationally intensive models may require stack size adjustment:

```
RUST_MIN_STACK=8388608 hay script.hay // 8 MB of stack
```

Relevant for VAR/SVAR with many lags or PCA on large datasets.

53.7 Editor integration

Neovim

Add to `init.lua` for basic syntax recognition:

```
vim.filetype.add({ extension = { hay = "hayashi" } })
```

Full highlighting via Tree-sitter is on the roadmap.

VS Code

Extension available on the marketplace as `hayashi-lang`.

53.8 Development builds

To iterate with local builds (without `--release`):

```
git clone https://github.com/sheep-farm/hayashi
cd hayashi
cargo build // debug build (recommended for dev)
```

```
./target/debug/hay script.hay
```

The debug build has extra bounds checks and detailed error messages. Do not use `--release` in development - the speed gain does not compensate for the loss of diagnostic information.

Running scripts

```
./target/debug/hay script.hay      // run script
./target/debug/hay                  // open interactive REPL
./target/debug/hay < script.hay     // read from stdin
```


Quick Reference

53.9 Types

Type	Example	Notes
Int	42, -7	64-bit, signed
Float	3.14, 1e-5	IEEE 754
Bool	true, false	
Nil	nil	Absence of value
String	"text"	UTF-8
FString	f"val: {x}"	Interpolation
List	[1, 2, 3]	0-based index
Dict	{"a": 1}	String or int key
DataFrame	load "f.csv" as df	COW, Float columns

53.10 Operators

Arithmetic	+ - * / ^ %	
Comparison	== != > < >= <=	
Logical	and or not	
Membership	in	
Pipe	>	
Assignment	= += -= *= /=	
Range	a..b (exclusive)	a..=b (inclusive)

53.11 Time series operators

Available inside `generate` `df col = expr:`

L.x	Lag of order 1
L2.x	Lag of order 2
F.x	Lead of order 1
D.x	First difference

Special constant: `_n` = row number (1-based).

53.12 Keywords

```
let  const  fn  return  if  else  for  in  while  break  continue
match  try  catch  and  or  not  true  false  nil
```

53.13 Descriptive statistics

Function	Result	Forms
<code>mean(df, x)</code>	$E(X)$	<code>list / df / if =</code>
<code>variance(df, x)</code>	$\text{Var}(X)$	<code>list / df / if =</code>
<code>sd(df, x)</code>	$\sigma(X)$	<code>list / df / if =</code>
<code>median(df, x)</code>	median	<code>list / df / if =</code>
<code>quantile(df, x, p)</code>	percentile p	<code>list / df / if =</code>
<code>cov(df, x, y)</code>	$\text{Cov}(X, Y)$	<code>df / if =</code>
<code>corr_pair(df, x, y)</code>	$\rho(X, Y)$	<code>df / if =</code>
<code>summarize(df, x)</code>	full dict	<code>+ detail=true</code>
<code>correlate(df, x, y, ...)</code>	correlation matrix	<code>display</code>
<code>codebook df</code>	per-column summary	<code>display</code>
<code>tabulate df col</code>	frequencies	<code>display</code>

53.14 DataFrame manipulation

Function	Description
<code>load "f" as df</code>	Load file (CSV, parquet)
<code>save df "f"</code>	Save file
<code>count(df [, cond])</code>	Number of rows
<code>filter(df, cond)</code>	Filter rows
<code>select(df, cols...)</code>	Select columns
<code>drop(df, cols...)</code>	Drop columns
<code>generate df col = expr</code>	Create/modify column
<code>replace df col = e if c</code>	Modify with condition
<code>sort(df, col [desc])</code>	Sort
<code>dropna(df [, cols...])</code>	Remove NaN
<code>append(df1, df2)</code>	Stack vertically
<code>merge(df1, df2, on=col)</code>	Horizontal join
<code>group_by(df, col, agg)</code>	Aggregate by group
<code>collapse(df, expr, by=c)</code>	Collapse by group
<code>pivot_longer(...)</code>	Wide $\rightarrow$ long
<code>pivot_wider(...)</code>	Long $\rightarrow$ wide
<code>reshape(df, ...)</code>	General reshape

53.15 Random generators

Function	Distribution
<code>rnormal()</code>	$N(0,1)$ - standard normal
<code>runiform()</code>	$U(0,1)$ - uniform
<code>rbernoulli(p)</code>	Bernoulli with fixed probability p
<code>seed(n)</code>	Sets PRNG seed for reproducibility

53.16 Linear regression and extensions

Estimator	Syntax
OLS	<code>ols(y ~ x1 + x2, df)</code>
IV/2SLS	<code>iv(y ~ x, ~ z, df)</code>
GMM	<code>gmm(y ~ x, ~ z1 + z2, df)</code>
WLS	<code>wls(y ~ x, df, weights="w")</code>
Rolling OLS	<code>rols(y ~ x, df, window=50)</code>
LASSO	<code>lasso(y ~ x1+...+xk, df, alpha=0.1)</code>
Ridge	<code>ridge_reg(y ~ x1+...+xk, df, alpha=0.5)</code>
Elastic Net	<code>enet(y ~ x1+...+xk, df, alpha=0.5)</code>
Bootstrap	<code>bootstrap("ols", y ~ x, df, reps=500)</code>
Quantile regression	<code>qreg(y ~ x, df, tau=0.5)</code>
Robust regression	<code>rlm(y ~ x, df)</code>
LOWESS	<code>lowess(df, y, x, frac=0.3)</code>
SUR (Zellner)	<code>sur(df, y1 ~ x1, y2 ~ x2)</code>

Regularization: in lasso, alpha is the penalty strength λ (larger = more zeros).

In enet, alpha is the L1/(L1+L2) mixing (0 = Ridge, 1 = LASSO).

53.17 Panel data

Estimator	Syntax
FE (within)	<code>fe(y ~ x, df, id=unit)</code>
RE (GLS)	<code>re(y ~ x, df, id=unit)</code>
FE with IV	<code>feiv(y ~ x, z ~ z+w, df, id="unit")</code>
Hausman	<code>hausman(m_fe, m_re)</code>
Mundlak (CRE)	<code>mundlak(df, y ~ x, id="unit")</code>
AB-GMM	<code>ab(y ~ x, df, id=unit, time=t, lags=2)</code>
PCSE	<code>pcse(y ~ x, df, id=unit, time=t)</code>
GEE	<code>gee(y ~ x, df, id=unit)</code>
xtgls	<code>xtgls(y ~ x, df, id=unit, time=t)</code>
Mixed model	<code>mixedlm(y ~ x, df, id="unit")</code>
Cond. logit (FE)	<code>clogit(y ~ x, df, group="unit")</code>

53.18 Causal inference

Method	Syntax
DiD	<code>did(y ~ treated + post, df)</code>
RDD	<code>rd(y ~ x, 0.0, df)</code>
PSM	<code>psm(y ~ d + x1, df, k=1, caliper=0.05, boot=200)</code>
Synthetic control	<code>synth("y", treated_id, t0, df, id="col", time="col")</code>

PSM: without caliper, ATT can be biased even with PSM. `synth`: `t0` = first post-treatment period (inclusive).

53.19 Special dependent variables

Model	Syntax
Logit / Probit	<code>logit(y ~ x, df)</code> <code>probit(y ~ x, df)</code>
Marginal effects	<code>margins(m, df)</code>
Multinomial logit	<code>mlogit(y ~ x, df)</code>
Ordered logit	<code>ologit(y ~ x, df)</code>
Ordered probit	<code>oprobit(y ~ x, df)</code>
Poisson	<code>poisson(y ~ x, df)</code>
Negative binomial	<code>nbreg(y ~ x, df)</code>
Zero-inflated	<code>zinb(y ~ x, df)</code>
Tobit	<code>tobit(y ~ x, df)</code>
Heckman	<code>heckman(y_out ~ x, s ~ z, df)</code>
General GLM	<code>glm(y ~ x, df, family=gamma, link=log)</code>

GLM Family	Available links
gaussian	identity, log, inverse
binomial	logit, probit, cloglog
poisson	log, identity
gamma	log, inverse, identity
inverseGaussian	inverse_squared

53.20 Survival analysis

Method	Syntax
Kaplan-Meier	<code>km(time, event, df)</code>
Cox PH	<code>cox(time ~ x1 + x2, df, event=event)</code>

Cox: `exp(coef)` = hazard ratio. `event` is the failure (1) / censoring (0) indicator column.

53.21 Time series

Model	Syntax
ARMA/ARIMA	<code>arima(df, y, p=1, d=0, q=1)</code>
AR(p) via OLS	<code>autoreg(df, y, lags=2, trend="c")</code>
ARDL(p, q)	<code>ardl(y ~ x1 + x2, df, lags=p, xlags=q)</code>
GARCH	<code>garch(df, y, p=1, q=1)</code>
EGARCH / GJR	<code>egarch(df, y, p=1, q=1)</code> <code>gjrgarch(df, y, p=1, q=1)</code>
ADF	<code>adf(df, var)</code>
KPSS	<code>kpss(df, var)</code>
Phillips-Perron	<code>phillips_perron(df, var)</code>
Zivot-Andrews	<code>zivot(df, var)</code>
VAR	<code>var(df, y1, y2, lags=2)</code>
VECM	<code>vecm(df, y1, y2, lags=1, r=1)</code>
Cointegration	<code>johansen(df, y1, y2, lags=1)</code>
Granger	<code>granger(df, y1, y2, lags=2)</code>
IRF (reduced)	<code>irf(m_var, steps=12)</code>
SVAR Cholesky	<code>svar(df, y1, y2, lags=1, id=cholesky)</code>
SVAR long-run	<code>svar(df, y1, y2, lags=1, id=longrun)</code>
Structural IRF	<code>sirf(m_svar, steps=12)</code>
Markov Switching	<code>markov(df, y, k=2, p=1)</code>

53.22 Filters and decomposition

Filter	Syntax	Output
Hodrick-Prescott	<code>hprescott(df, y, lambda=1600)</code>	<code>y_tr</code>
Baxter-King	<code>baxter_king(df, y, low=6, high=32, k=12)</code>	<code>y_cy</code>
Holt-Winters / ETS	<code>holtwinters(df, y, trend=add, seasonal=add, period=12)</code>	ETS
STL	<code>stl(df, y, period=12)</code>	<code>y_tr</code>
Christiano-Fitz.	<code>christiano_fitzgerald(df, y, low=6, high=32)</code>	<code>y_cy</code>

Edge loss: HP loses endpoints; BK loses k observations at each end; CF and STL keep all points. Note: `hamilton()` is an alias for Markov Switching (Hamilton 1989), *not* for the Hamilton 2018 filter. `autoreg trend`: "c" = constant (default); "ct" = constant + trend; "t" = trend only; "n" = none. `ARDL`: standard formula $y \sim x$; `lags` = lags of y ; `xlags` = lags of each regressor; includes contemporaneous x automatically.

53.23 Multivariate analysis

Method	Syntax
PCA	<code>pca(df, v1, v2, ..., n=2)</code>
Factor analysis	<code>factor(df, v1, v2, ..., n=2, rotation=varimax)</code>
Canonical corr.	<code>cancorr(df, xvars=["v1","v2"], yvars=["v3","v4"])</code>
MANOVA	<code>manova(df, y1, y2, by="group")</code>
KDE	<code>kde(df, var, bwidth=0.5)</code>

53.24 Hypothesis tests

Test	Syntax
t - one sample	<code>ttest(df, var, mu=0.0)</code>
t - two groups	<code>ttest(df, var, by=group)</code>
t - paired	<code>ttest(df, var1, var2, paired=true)</code>
ANOVA	<code>anova(df, var, by=group)</code>
Kruskal-Wallis	<code>kruskal_wallis(df, var, by=group)</code>
Normality (SW)	<code>swilk(df, var)</code>
Normality (SF)	<code>sfrancia(df, var)</code>
Homoscedasticity	<code>vif(m_ols)</code>
Hausman	<code>hausman(m_fe, m_re)</code>
Linear test	<code>test(m, "x1 = x2")</code>
Linear combination	<code>lincom(m, "x1 + x2")</code>

53.25 Presenting results

Function	Description
<code>display m</code>	Display model result
<code>esttab(m1, m2, ...)</code>	Comparison table
<code>eststo(m, "name")</code>	Store model for <code>esttab</code>
<code>predict(m, df)</code>	Adds column <code>yhat</code> to <code>df</code>
<code>margins(m, df)</code>	Average marginal effects (AME)

53.26 Building a panel (formulas)

To build a panel from a stacked DataFrame (N units, T periods):

```
generate df unit = (_n - 1) % N + 1
generate df t     = (_n - (_n - 1) % N - 1) / N + 1
```

Where N is the number of units. Replace N with the numeric value.

53.27 REPL shortcuts

Shortcut	Action
<code>exit</code>	Exit the REPL
<code>Ctrl-D</code>	Exit the REPL (EOF)
<code>Ctrl-C</code>	Cancel current line
<code>Ctrl-R</code>	Search history

Equivalent Code by Platform

This appendix reproduces representative DGPs in Stata, R and Python, side by side with HAYASHI code. The purpose is twofold: to allow cross-validation of results and to highlight syntax differences.

Note

Random number generators differ across platforms - the numeric values will differ, but the statistical properties (parameter recovery, direction of bias, Hausman decision) are identical for any seed.

53.28 OLS — DGP without noise and comparison with omitted variable

Hayashi

```
1 seed(42)
2 input df
3 id
4 1
5 end
6 for i in 1..=10 { df = append(df, df) }
7 df |> filter(_n <= 1000)
8 generate df x1 = rnormal()
9 generate df x2 = rnormal()
10 generate df y = 2.5 + 1.8*x1 - 0.7*x2
11 let m1 = ols(y ~ x1 + x2, df)
12 let m2 = ols(y ~ x1, df)
13 esttab(m1, m2)
```

Stata

```
clear
set obs 1000
set seed 42
gen x1 = rnormal()
gen x2 = rnormal()
gen y = 2.5 + 1.8*x1 - 0.7*x2
reg y x1 x2
estimates store m1
reg y x1
estimates store m2
esttab m1 m2          // ssc install estout
```

R

```
set.seed(42)
n <- 1000
x1 <- rnorm(n); x2 <- rnorm(n)
y <- 2.5 + 1.8*x1 - 0.7*x2
m1 <- lm(y ~ x1 + x2)
m2 <- lm(y ~ x1)
library(stargazer)
stargazer(m1, m2, type = "text")
```

Python

```
import numpy as np, pandas as pd
import statsmodels.formula.api as smf
from statsmodels.iolib.summary2 import summary_col
np.random.seed(42)
n = 1000
x1 = np.random.randn(n); x2 = np.random.randn(n)
y = 2.5 + 1.8*x1 - 0.7*x2
df = pd.DataFrame({'y': y, 'x1': x1, 'x2': x2})
m1 = smf.ols('y ~ x1 + x2', data=df).fit()
m2 = smf.ols('y ~ x1', data=df).fit()
print(summary_col([m1, m2], stars=True))
```

53.29 IV — Endogeneity with valid instrument

Hayashi

```

1  seed(42)
2  ...
3  generate df z    = rnormal()
4  generate df v    = rnormal()
5  let d1           = cov(df, z, v) / variance(df, z)
6  let d0           = mean(df, v) - d1 * mean(df, z)
7  generate df eps  = v - d0 - d1*z
8  generate df x    = 0.8*z + eps
9  generate df y    = 1.5 + 2.0*x + 0.7*eps
10 let m_ols = ols(y ~ x, df)
11 let m_iv  = iv(y ~ x, ~ z, df)
12 esttab(m_ols, m_iv)

```

Stata

```

gen z = rnormal()
gen v = rnormal()
reg v z
predict eps, residuals
gen x = 0.8*z + eps
gen y = 1.5 + 2.0*x + 0.7*eps
reg y x
ivregress 2sls y (x = z)

```

R

```

eps <- resid(lm(v ~ z))
x    <- 0.8*z + eps
y    <- 1.5 + 2.0*x + 0.7*eps
library(ivreg)
ivreg(y ~ x | z, data = data.frame(y,x,z))

```

Python

```
from linearmodels.iv import IV2SLS
import statsmodels.api as sm
IV2SLS(y, sm.add_constant(x),
       None, sm.add_constant(z)).fit()
```

53.30 Panel — FE, RE and Hausman

Hayashi

```
1 seed(42) / ... / df |> filter(_n <= 1000)
2 generate df id      = (_n - 1) % 100 + 1
3 generate df alpha = id * 0.05
4 generate df x      = alpha + rnormal()
5 generate df y      = 1.0 + 2.0*x + alpha + rnormal()
6 let m_fe = fe(y ~ x, df, id=id)
7 let m_re = re(y ~ x, df, id=id)
8 hausman(m_fe, m_re)
```

Stata

```
xtset id
xtreg y x, fe
estimates store m_fe
xtreg y x, re
estimates store m_re
hausman m_fe m_re
```

R

```
library(plm)
pd <- pdata.frame(df, index = "id")
m_fe <- plm(y ~ x, data = pd, model = "within")
m_re <- plm(y ~ x, data = pd, model = "random")
phtest(m_fe, m_re)
```

Python

```
from linearmodels import PanelOLS, RandomEffects
m_fe = PanelOLS(df['y'], df[['x']], entity_effects=True).fit()
m_re = RandomEffects(df['y'], df[['x']]).fit()
```

53.31 Logit and Probit

Hayashi

```
1 generate df p = 1 / (1 + exp(-(-1.0 + 2.0*x)))
2 generate df y = runiform() < p
3 let m_l = logit(y ~ x, df)
4 let m_p = probit(y ~ x, df)
5 margins(m_l, df)
```

Stata

```
logit y x
probit y x
margins, dydx(x)
```

R

```
m_l <- glm(y ~ x, family = binomial("logit"))
m_p <- glm(y ~ x, family = binomial("probit"))
library(margins)
summary(margins(m_l))
```

Python

```
import statsmodels.formula.api as smf
m_l = smf.logit('y ~ x', data=df).fit()
print(m_l.get_margeff().summary())
```

53.32 Count models

Hayashi

```
1 generate df lam = exp(0.5 + 1.0*x)
2 generate df y   = floor(lam + runiform())
3 let m_p  = poisson(y ~ x, df)
4 let m_nb = nbreg(y ~ x, df)
```

Stata

```
poisson y x
nbreg   y x
```

R

```
glm(y ~ x, family = poisson("log"))
library(MASS)
glm.nb(y ~ x)
```

Python

```
smf.poisson('y ~ x', data=df).fit()
smf.negativebinomial('y ~ x', data=df).fit()
```

53.33 Tobit and Heckman

Hayashi

```
1 let m_t = tobit(y_tobit ~ x, df)
2 let m_h = heckman(y_out ~ x, s ~ z, df)
```

Stata

```
tobit y_tobit x, ll(0)
heckman y_out x, select(s = x z)
```

R

```
library(AER)
tobit(y_tobit ~ x, left = 0)
library(sampleSelection)
heckit(s ~ x + z, y_out ~ x)
```

Python

```
# Tobit: no native implementation in statsmodels
# Heckman: via heckit from pyheckit or statsmodels
from statsmodels.regression.linear_model import OLS
# manual two-step implementation
```

53.34 Ordered and multinomial models

Hayashi

```
1 let m_ol = ologit(y ~ x, df)
2 let m_op = oprobit(y ~ x, df)
3 let m_ml = mlogit(y ~ x, df)
```

Stata

```
ologit y x
oprobit y x
mlogit y x
```

R

```
library(MASS)
polr(factor(y) ~ x, method = "logistic") # ologit
polr(factor(y) ~ x, method = "probit") # oprobit
library(nnet)
multinom(y ~ x) # mlogit
```

Python

```
from statsmodels.miscmodels.ordinal_model import OrderedModel
OrderedModel(y, X, distr='logit').fit()
OrderedModel(y, X, distr='probit').fit()
smf.mnlogit('y ~ x', data=df).fit()
```

53.35 Regularization (LASSO, Ridge, Elastic Net)

Hayashi

```
1 let m_las = lasso(y ~ x1+x2+...+x10, df, alpha=0.1)
2 let m_rid = ridge_reg(y ~ x1+...+x10, df, alpha=0.5)
3 let m_ent = enet(y ~ x1+...+x10, df, alpha=0.5)
4 display m_las
```

Stata

```
// requires Stata 16+
lasso linear y x1-x10
cvlasso y x1-x10
```

R

```
library(glmnet)
X <- model.matrix(y ~ ., df)[,-1]
# LASSO (alpha=1), Ridge (alpha=0), Enet (alpha=0.5)
cv.glmnet(X, df$y, alpha = 1) # LASSO with CV
cv.glmnet(X, df$y, alpha = 0) # Ridge with CV
cv.glmnet(X, df$y, alpha = 0.5) # Enet with CV
```

Python

```
from sklearn.linear_model import Lasso, Ridge, ElasticNet
Lasso(alpha=0.1).fit(X, y)
Ridge(alpha=0.5).fit(X, y)
ElasticNet(alpha=0.1, l1_ratio=0.5).fit(X, y)
```

Scale note: alpha in Hayashi and statsmodels is λ (penalty strength). In glmnet (R), data are standardized automatically and lambda is chosen by cross-validation. In scikit-learn, alpha is analogous to λ .

53.36 Survival analysis

Hayashi

```
1 let m_km = km(time, event, df)
2 let m_cox = cox(time ~ x, df, event=event)
3 display m_cox
```

Stata

```
stset time, failure(event)
sts graph           // Kaplan-Meier
stcox x             // Cox PH
stcox x, nohr       // log-hazard coefficients
```

R

```
library(survival)
km_fit <- survfit(Surv(time, event) ~ 1)
cox_fit <- coxph(Surv(time, event) ~ x)
summary(cox_fit)
```

Python

```
from lifelines import KaplanMeierFitter, CoxPHFitter
kmf = KaplanMeierFitter().fit(df['time'], df['event'])
cph = CoxPHFitter().fit(df, 'time', 'event')
```

53.37 SVAR — Cholesky and Blanchard-Quah

Hayashi

```
1 let m_ch = svar(df, y1, y2, lags=1, id=cholesky)
2 let m_bq = svar(df, y1, y2, lags=1, id=longrun)
3 let m_ir = sirf(m_ch, steps=12)
4 display m_ir
```

Stata

```
var y1 y2, lags(1)
// Cholesky (default for IRF post-VAR):
irf create myirf, step(12) replace
irf table sirf
// Blanchard-Quah requires manual restrictions via svar
matrix A = (1, 0 \ ., 1)
matrix B = (., 0 \ ., .)
svar y1 y2, lags(1) aeq(A) beq(B)
```

R

```
library(vars)
m <- VAR(cbind(y1, y2), p = 1)
# Cholesky:
irf(m, n.ahead = 12, ortho = TRUE)
# Blanchard-Quah:
BQ(m)
```

Python

```
from statsmodels.tsa.api import VAR
m = VAR(df[['y1', 'y2']]).fit(1)
# Cholesky:
m.irf(12).orth_ma # orthogonalized IRF
# BQ: requires manual implementation in Python
```

53.38 Decomposition filters

Hayashi

```
1 hprescott(df, y, lambda=1600)
2 baxter_king(df, y, low=6, high=32, k=12)
3 stl(df, y, period=12)
4 christiano_fitzgerald(df, y, low=6, high=32)
```

Stata

```
// ssc install hprescott
hprescott y, smooth(1600)      // HP
// ssc install bking
bking y, minperiod(6) maxperiod(32) stub(bk)
// ssc install cfitzroy
// stl: via Python/R (not natively available in Stata)
```

R

```
library(mFilter)
hp <- hpfilter(y, freq = 1600)
bk <- bkfilter(y, pl = 6, pu = 32, nfix = 12)
cf <- cffilter(y, pl = 6, pu = 32)
library(stats)
stl(ts(y, frequency=12), s.window="periodic")
```

Python

```
from statsmodels.tsa.filters.hp_filter import hpfilter
from statsmodels.tsa.filters.bk_filter import bkfilter
from statsmodels.tsa.seasonal import STL
_, trend = hpfilter(y, lamb=1600)
cycle_bk = bkfilter(y, low=6, high=32, K=12)
stl = STL(y, period=12).fit()
```

53.39 PSM and Synthetic Control

Hayashi

```
1 let m_psm = psm(y ~ d + x1, df, k=1, caliper=0.05, boot=200)
2 display m_psm
3
4 // Synthetic Control (panel)
5 let m_sc = synth("y", 1, 11, df, id="unit", time="year")
6 display m_sc
```

Stata

```
// ssc install psmatch2
psmatch2 d x1, outcome(y) neighbor(1) caliper(0.05)
// ssc install synth
synth y x1, trunit(1) trperiod(11) xperiod(1/10)
```

R

```
library(MatchIt)
m <- matchit(d ~ x1, data=df, method="nearest",
             caliper=0.05)
library(Synth)
dataprep.out <- dataprep(...) # setup
synth(dataprep.out)
```

Python

```
# PSM:
from sklearn.linear_model import LogisticRegression
# compute propensity scores and perform manual matching
# Synthetic Control:
# pip install pysynth
from pysynth import Synth
```

53.40 Comparative summary

Task	Hayashi	Stata	R	Python
<i>Regression</i>				
OLS	ols()	reg	lm()	smf.ols()
IV/2SLS	iv()	ivregress	ivreg()	IV2SLS()
GMM	gmm()	gmm	gmm()	LinearIVGMM()
WLS	wls()	vwls	lm(weights=)	WLS()
Rolling OLS	rols()	rolling()	rollregres()	RollingOLS()
LASSO	lasso()	lasso linear	glmnet(a=1)	Lasso()
Ridge	ridge_reg()	-	glmnet(a=0)	Ridge()
Elastic Net	enet()	-	glmnet(a=.5)	ElasticNet()
Bootstrap	bootstrap()	bootstrap:	boot()	resample()
LOWESS	lowess()	lowess	lowess()	lowess()
Quant. reg.	qreg()	qreg	rq()	QuantReg()
SUR	sur()	sureg	systemfit()	SUR()
<i>Panel data</i>				
FE (within)	fe()	xtreg, fe	plm(within)	PanelOLS()
RE (GLS)	re()	xtreg, re	plm(random)	RandomEffects()
Hausman	hausman()	hausman	phtest()	manual
AB-GMM	ab()	xtabond2	pgmm()	FirstDifferenceGMM()
PCSE	pcse()	xtpcse	pcse()	-
GEE	gee()	xtgee	geeglm()	GEE()
Mundlak (CRE)	mundlak()	manual	mundlak()	manual
xtgls	xtgls()	xtgls	pggls()	-
Mixed model	mixedlm()	mixed	lme4::lmer()	MixedLM()
FE logit	clogit()	clogit	clogit()	ConditionalLogit()
<i>Special dependent variables</i>				
Logit	logit()	logit	glm(logit)	smf.logit()
Probit	probit()	probit	glm(probit)	smf.probit()
Ordered logit	ologit()	ologit	polr()	OrderedModel
Ordered probit	oprobit()	oprobit	polr()	OrderedModel
Multinom. logit	mlogit()	mlogit	multinom()	MNLogit()
Poisson	poisson()	poisson	glm(poisson)	smf.poisson()
Neg. Binomial	nbreg()	nbreg	glm.nb()	smf.negativebinomial()
Zero-inflated	zinb()	zinb	zeroinfl()	ZeroInflatedNB
Tobit	tobit()	tobit	AER::tobit()	manual
Heckman	heckman()	heckman	heckit()	manual
General GLM	glm()	glm	glm()	GLM()
Marginal eff.	margins()	margins	margins()	get_margeff()
<i>Causal inference</i>				
DiD	did()	diff	lm()	smf.ols()
RDD	rd()	rdrobust	rdrobust()	rdd()
PSM	psm()	psmatch2	matchit()	manual

(continued on next page)

(continued)

Task	Hayashi	Stata	R	Python
Synth. control	synth()	synth	Synth	pysynth
<i>Survival</i>				
Kaplan-Meier	km()	sts	survfit()	KaplanMeierFitter
Cox PH	cox()	stcox	coxph()	CoxPHFitter
<i>Time series</i>				
AR/MA/ARMA	arima()	arima	arima()	ARIMA()
AR(p) OLS	autoreg()	arima	arima()	AutoReg()
ARDL(p, q)	ardl()	ardl	dynlm()	ardl()
GARCH	garch()	arch	garch()	arch()
Unit root (ADF)	adf()	dfuller	adf.test()	adfuller()
Phillips-Perron	phillips_perron()	pperron	PP.test()	PhillipsPerron()
Zivot-Andrews	zivot()	zandrews	ur.za()	ZivotAndrews()
Cointegration	johansen()	vecrank	ca.jo()	coint_johansen
VAR	var()	var	VAR()	VAR()
IRF (reduced)	irf()	irf create	irf()	irf().plot()
SVAR Cholesky	svar(chol.)	svar	id.chol()	manual
SVAR BQ	svar(long.)	svar	BQ()	manual
Structural IRF	sirf()	irf create	irf()	manual
Markov Switch.	markov()	mswitch	msmFit()	MarkovRegression
<i>Filters and decomposition</i>				
HP	hprescott()	hprescott*	hpfilter()	hpfilter()
Baxter-King	baxter_king()	bking*	bkfilter()	bkfilter()
Holt-Winters	holtwinters()	tssmooth	HoltWinters()	ExponentialSmoothing()
STL	stl()	-	stl()	STL()
CF filter	christiano_fitzgerald()	cfitzroy	cffilter()	-
<i>Multivariate analysis</i>				
PCA	pca()	pca	prcomp()	PCA()
Factor analysis	factor()	factor	factanal()	FactorAnalysis()
Canonical corr.	cancorr()	canon	cancor()	CCA()
MANOVA	manova()	manova	manova()	MANOVA()
<i>Inference and tests</i>				
t -test	ttest()	ttest	t.test()	ttest_ind()
ANOVA	anova()	anova	aov()	f_oneway()
Kruskal-Wallis	kruskal_wallis()	kwallis	kruskal.test()	kruskal()
Normality	swilk()	swilk	shapiro.test()	shapiro()
Table	esttab()	esttab [†]	stargazer()	summary_col()

* HP and BK in Stata require ssc install hprescott and ssc install bking.

[†] esttab in Stata requires ssc install estout.

Note on random generators: `rnormal()` matches the standard-normal generator used by Stata, R and Python examples. Use `runiform()` when the DGP needs a $U(0,1)$ draw.

53.41 Validation matrix

In addition to the code examples in this appendix, all of HAYASHI is systematically validated against reference implementations. The validation programme is in the `validation/` directory and is run with:

```
1 hay validate
```

Table 53.2 lists the current validation cases, with the estimator family, the data source, and the references (R, Python and/or Stata) used for comparison. Full details --- tolerances, notes and status --- are in `validation/MATRIX.md`.

Case ID	Family	Dataset/Source	References (R/Python/Stata)
ab_grunfeld	ab	wooldridge::grunfeld	R
ar_book	arima	simulated_ar1	R, Python
ardl_gdp	ardl	statsmodels::macrodata	R, Python
arima_book	arima	simulated_rw	R, Python
arima_gdp	arima	statsmodels::macrodata	R, Python
arma_book	arima	simulated_arma11	R, Python
autoreg_gdp	autoreg	statsmodels::macrodata	R, Python
coint_book	vecm	simulated_cointegrated	R, Python
cox_heart	cox	statsmodels::heart	R, Python
did_kielmc	did	wooldridge::kielmc	R, Python
ets_gdp	ets	statsmodels::macrodata	R, Python
garch_book	garch	simulated_garch11	Python
garch_nyse	garch	wooldridge::nyse	R, Python
glsar_hprice1	glsar	wooldridge::hprice1	R, Python
gmm_card	gmm	wooldridge::card	R, Python
heckman_mroz	heckman	wooldridge::mroz	R, Python
iv_card	iv	wooldridge::card	R, Python
lasso_hprice1	lasso	wooldridge::hprice1	R, Python
logit_mroz	logit	wooldridge::mroz	R, Python
ma_book	arima	simulated_ma1	R, Python
ologit_beauty	logit	wooldridge::beauty	R, Python
ols_wooldridge_wage1	ols	wooldridge::wage1	R, Python
panel_fe_grunfeld	panel_fe	wooldridge::grunfeld	R, Python
poisson_fertil2	poisson	wooldridge::fertil2	R, Python
probit_mroz	probit	wooldridge::mroz	R, Python

(continued on next page)

(continued)

Case ID	Family	Dataset/Source	References (R/Python/Stata)
psm_jtrain3	psm	wooldridge::jtrain3	R, Python
quantile_wage1	qreg	wooldridge::wage1	R, Python
re_grunfeld	re	grunfeld	R, Python
rdd_book	rdd	rdd_book	R, Python
synth_smoking	synth	synth_smoking	R, Python
tobit_mroz	tobit	wooldridge::mroz	R, Python
var_book	var	simulated_var1	R, Python
var_macro	var	statsmodels::macrodata	R, Python
wls_hprice1	wls	wooldridge::hprice1	R, Python

Index

- 2SLS, *see* two-stage least squares
- ab(), 187
- ADF, *see* Dickey-Fuller test
- adf(), 131
- AIC, 125
- ANOVA, 225
- anova(), 225
- ARCH, *see* GARCH
- ARDL, 249
- ardl(), 249
- Arellano-Bond, 187
- ARIMA, 131
- arma(), 125
- ARMA, 125
- ATT, *see* average treatment effect on the treated, *see* average treatment effect on the treated
- autocorrelation, 125
- AutoReg, 249
- autoreg(), 249
- autoregressive, 125
- average treatment effect on the treated, 101
- band-pass filter, 237
- Baxter-King, filter of, 215
- baxter_king(), 215
- BIC, 125
- Blanchard-Quah, decomposition of, 241
- BLUE, 77
- bootstrap, 199
- bootstrap(), 199
- bounds test, *see* Pesaran, cointegration test of
- Box-Jenkins methodology, 125
- caliper, 157
- cancorr(), 203
- canonical correlation, 203
- causal ordering, 219
- censoring, 181
- censoring bias, 173
- Cholesky, decomposition of, 219
- Christiano-Fitzgerald, filter of, 237
- christiano_fitzgerald(), 237
- clogit(), 229
- cointegration, 143
- conditional heteroskedasticity, 149
- conditional logit, 229
- conditional volatility, 149
- consistency, 77
- contemporaneous correlation, 191

- correlated random effects, *see*
 - Mundlak, estimator of
- count models, 169
- cox(), 181
- Cox, model, 181
- CRE, *see* Mundlak, estimator of
- cutoff threshold, 113
- cycle, 215
- decomposition filters, 215
- Dickey-Fuller test, 131
- DiD, *see*
 - difference-in-differences
- did(), 101
- difference GMM, 187
- difference-in-differences, 101
- distributed lags, 249
- donors, pool of, 161
- efficiency, gain of, 191
- EGARCH, 149
- Elastic Net, 195
- endogeneity, 87
- enet(), 195
- equivalences Stata/R/Python, 273
- error correction model, *see*
 - VECM
- esttab(), 77
- ETS (Error-Trend-Seasonal), 215
- excess zeros, 169
- exclusion restriction, 87
- expected regime duration, 211
- factor(), 203
- factor analysis, 203
- factor loadings, 203
- fe(), 95
- first stage, 87
- fixed effects, 95
- gamma, distribution, 207
- GARCH, 149
- garch(), 149
- GEE, 187
- gee(), 187
- generalized estimating
 - equations, 207
- generalized linear models, 207
- generalized method of moments, 119
- GJR-GARCH, 149
- GLM, *see* generalized linear
 - models
- glm(), 207
- GMM, *see* generalized method of
 - moments
- gmm(), 119
- granger(), 137
- Granger causality, 137
- Hamilton filter, 211
- Hamilton, model of, 211
- hausman(), 95
- Hausman test, 95
- hazard function, 181
- hazard ratio, 181
- heckman(), 173
- Heckman, model, 173
- heteroscedasticity, 77, 109
- Hodrick-Prescott, filter of, 215
- Holt-Winters, 215
- holtwinters(), 215
- hprescott(), 215
- IIA, axiom of, 177
- impulse response function, 137

- independence of irrelevant
 - alternatives, *see* IIA,
 - axiom of
- installation, 261
- instrumental variables, 87
- internal instruments, 187
- inverse Mills ratio, 173
- irf(), 137
- IV, *see* instrumental variables
- iv(), 87
- J-test, 119
- johansen(), 143
- Johansen, test, 143
- kalman(), 255
- Kalman filter, 255
- Kaplan-Meier, estimator, 181
- km(), 181
- kpss(), 131
- KPSS test, 131
- Kruskal-Wallis, test of, 225
- kruskal_wallis(), 225
- L1 penalty (lasso), 195
- L2 penalty (ridge), 195
- LASSO, 195
- lasso(), 195
- limited dependent variable, 173
- link function, 207
- local estimation, 245
- local level, model, 255
- local linear trend, model, 255
- loess, 237
- logit, 105
- logit(), 105
- long-run multiplier, 249
- long-run relationship, 143
- long-run restrictions, 241
- LOWESS, 245
- lowess(), 245
- MANOVA, 203
- marginal effects, 105
- margins(), 105
- markov(), 211
- Markov Switching, 211
- maximum likelihood, 105
- median, 109
- mixed model, 229
- mixedlm(), 229
- mlogit(), 177
- modules, 261
- moment conditions, 119
- moving average, 125
- multinomial logit, 177
- mundlak(), 229
- Mundlak, estimator of, 229
- nbreg(), 169
- negative binomial, 169
- Newey-West, variance of, 233
- Nickell bias, 95, 187
- nonparametric smoothing, 245
- nonparametric, test, 225
- odds ratio, 105
- ologit(), 177
- OLS, *see* ordinary least squares
- ols(), 77
- omitted variable bias, 77
- oprobit(), 177
- order of integration, 131
- ordered logit, 177
- ordered probit, 177
- ordinary least squares, 77
- overdispersion, 169
- overidentification, 119

- panel data, [95](#)
- parallel trends, [101](#)
- PCA, *see* principal component analysis
- pca(), [203](#)
- PCSE, [187](#)
- pcse(), [187](#)
- percentile confidence interval, [199](#)
- permanent shock, [241](#)
- Pesaran, cointegration test of, [249](#)
- Phillips-Perron, test of, [233](#)
- phillips_perron(), [233](#)
- placebo, tests, [161](#)
- poisson(), [169](#)
- Poisson, regression, [169](#)
- PPML, [207](#)
- prediction log-likelihood, [255](#)
- principal component analysis, [203](#)
- probit, [105](#)
- probit(), [105](#)
- propensity score, [157](#)
- propensity score matching, [157](#)
- proportional hazards,
 - assumption of, [181](#)
- PSM, *see* propensity score matching
- psm(), [157](#)
- qreg(), [109](#)
- quantile, [109](#)
- quantile regression, [109](#)
- quasi-maximum likelihood, [169](#)
- quick reference, [265](#)
- random effects, [95](#)
- Rauch-Tung-Striebel smoother, [255](#)
- rd(), [113](#)
- RDD, *see* regression discontinuity
- re(), [95](#)
- reference category, [177](#)
- regime switching, [211](#)
- regression discontinuity, [113](#)
- regularization, [195](#)
- relevance condition, [87](#)
- REML, [229](#)
- REPL, [261](#)
- resampling, [199](#)
- Ridge, [195](#)
- ridge_reg(), [195](#)
- RMSPE, [161](#)
- rnormal(), [261](#)
- robust standard errors, [77](#)
- rolling OLS, [245](#)
- rols(), [245](#)
- RTS, *see* Rauch-Tung-Striebel smoother
- running variable, [113](#)
- sample selection, [173](#)
- seasonal decomposition, [237](#)
- seasonality, [215](#)
- seed(), [261](#)
- seemingly unrelated regressions,
 - see* SUR
- selection bias, [157](#)
- sirf(), [219](#)
- sparsity, [195](#)
- state-space model, [255](#)
- stationarity, [125](#)
- STL, [237](#)
- stl(), [237](#)

- structural break, [233](#)
- structural identification, [219](#)
- structural impulse-response
 - function, [219](#)
- structural VAR, [219](#)
- SUR, [191](#)
- sur(), [191](#)
- survival analysis, [181](#)
- SVAR, *see* structural VAR
- svar(), [219](#)
- SVEC, [241](#)
- syntax, [265](#)
- synth(), [161](#)
- synthetic control, [161](#)

- t-test, [225](#)
- Tobit, [173](#)
- tobit(), [173](#)
- transition probability, [211](#)
- transitory shock, [241](#)
- trend, [215](#)
- ttest(), [225](#)
- two-stage least squares, [87](#)

- unit root, [131](#)

- unobservable state, [255](#)

- VAR, [137](#)
- var(), [137](#)
- variable selection, [195](#)
- variance decomposition, [137](#)
- varimax, rotation, [203](#)
- VECM, [143](#)
- vecm(), [143](#)
- vector autoregressive, *see* VAR
- volatility clustering, [149](#)

- weighted least squares, [245](#)
- Welch, test of, [225](#)
- within estimator, *see* fixed effects
- WLS, *see* weighted least squares
- wls(), [245](#)

- xtgls(), [229](#)

- Zellner, estimator, [191](#)
- ZINB, [169](#)
- zinb(), [169](#)
- zivot(), [233](#)
- Zivot-Andrews, test of, [233](#)