Lecture 4: From Data to Models
Programming for Economics—ECNM10106

Albert Rodriguez-Sala & Jacob Adenbaum
School of Economics, The University of Edinburgh

Winter 2023

This Class

- Random Numbers
- Monter Carlo Simulation
- Numerical Integration
  - Numerical integration by Monte Carlo
  - Gauss-Hermite quadrature rule
- Time-dependent Random Processes
  - Ar(1) Model
  - Markov Chains

Random Numbers

Economics and random variables

In statistics and economics we work with random variables. Computers allow us to simulate random variables which helps us to

- Study probabilistic problems.
- Study the behavior of certain statistics or estimators.
- To solve stochastic models: Study the behavior of models that depend on random variables (shocks, heterogeneity). Solve models by simulation.
- Compute integrals.

Generating random numbers

Computers in general cannot generate completely random numbers. However, we have **pseudo-random numbers:** Numbers that looks close enough to being random although they are generated from deterministic algorithms.

Since they are good as random, from now on we will just refer to them as random numbers.

```
# Let's generate 1000 draws form a x~N(0,1)
x = np.random.normal(loc=0,scale=1,size=1000)
x[0:4]
>> [ 0.66698806,  0.02581308, -0.77761941,  0.94863382]
```

Generating random numbers: set a seed

For **replicability**, we might want that on each run the same sequence of random numbers is now generated. For that we need to set a **seed**—from where random numbers are generated.

```
# Setting a seed
np.random.seed(68)
x1 = np.random.normal(loc=0,scale=1,size=1000)
>>[-0.94447636  0.19994362 -1.54035342  0.83804177, ...]
x2 = np.random.normal(loc=0,scale=1,size=1000)
>> [ 0.59233049 -0.85910019 -0.45770469 -0.31261349, ...]
np.random.seed(68)
x3 = np.random.normal(loc=0,scale=1,size=1000)
>> [-0.94447636  0.19994362 -1.54035342  0.83804177, ...]
```

Set the seed only once and at the beginning of the file.

Random numbers from special distributions

- From a Normal distribution.

```
# Normal distribution x~N(mu, sigma^2)
x = np.random.normal(mu,sigma,N)
```

- From a Uniform distribution.

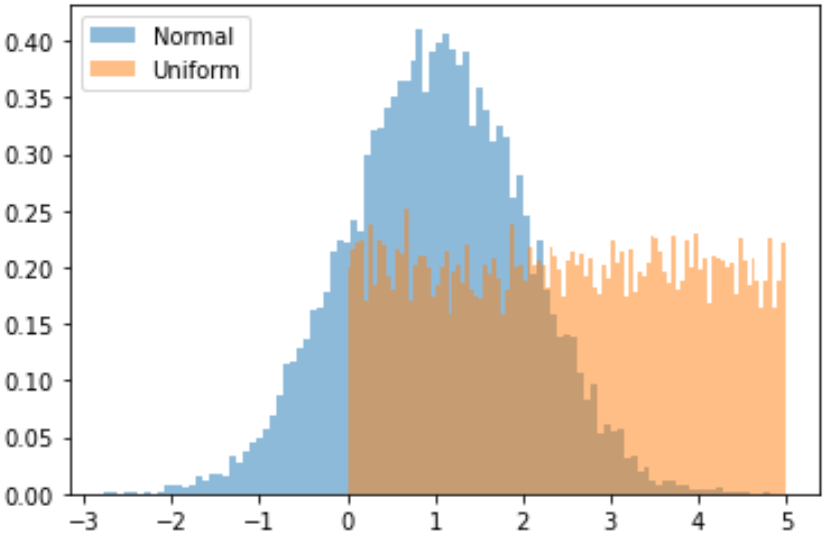
```
# Uniform distribution x~U(a,b)
y = np.random.uniform(low=a,high=b,size=N)
```

- from a Log-Normal distribution.

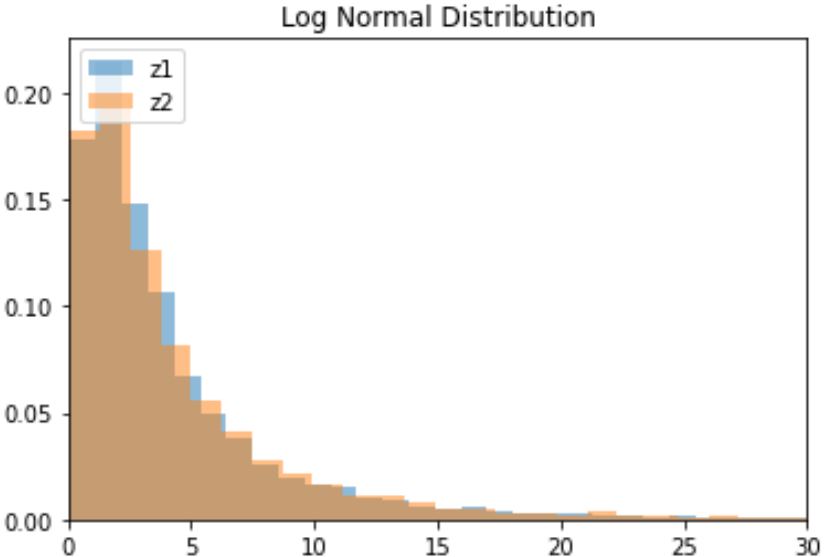
```
#  Log-normal distribution: z st ln(z)~N(mu, sigma^2)
z = np.random.lognormal(mean=mu, sigma=sigma, size=N)
# alternatively
z2 = np.exp(x)
```

- Other default distributions: Beta, Poisson, Weibull, etc.

Normal vs Uniform Distribution



Log-Normal distribution



Random numbers from a user-created distribution

We can also draw random numbers from **our own defined distribution** with *np.random.choice*. Or in general, we can use np.random.choice to draw random numbers from a distribution without an explicit form.

```
# A variable v on support v_s{0,1..,4} with probabilities v_p=[0.4, 0.1, 0.1, 0.2,0.2]
#  Define support of the distribution
v_sup = np.array([0,1,2,3,4])
# Define probabilities (positive numbers, sum must be equal to 1)
v_p = [0.4, 0.1, 0.1, 0.2,0.2]
print('Prob of v sum to 1?', sum(v_p)==1)
#  Get draws from distribution
v = np.random.choice(v_sup,size=N,p=v_p)
v[0:6]
>> [0 0 3 0 4 3]
```

Monter Carlo Simulation

Monte Carlo simulation

Monte Carlo simulation consists of repeated random sampling to obtain numerical results.

The key idea of Monte Carlo simulation is that **by the law of large numbers**, **integrals** described by the expected value of some random variable can be approximated by taking the **sample mean of large samples**. Thus, in principle, Monte Carlo methods can be used to solve any problem having a probabilistic interpretation.

Monte Carlo Simulation is mostly useful when it is difficult or impossible to use other approaches.

Monte Carlo simulation: some cases

Some cases of when we use Monte Carlo simulation

- Study the behavior of a given statistic of interest.
- Numerical integration: to compute integrals or expectations that are complicated or do not have a closed-form solution.
- Solve models (by simulation): models with many heterogeneous agents. Probabilistic models, infection models, models without closed-form solutions, etc.

Example: OLS asymptotic behavior

Exercise: Proof that the OLS estimator is distributed asymptotically normal using Monte Carlo simulation.

$$\sqrt{n}\left(\hat{\beta}-\beta\right) \overset{as}{\sim} N\left(0, \sigma^2\left(\mathbb{E}(x_i x_i')\right)^{-1}\right)$$

Solution:

Preliminary steps: generate a sample from the model and estimate it by OLS.

- 1. Consider the following model or data generating process

$$y_i = 2 + 0.5x_{2,i} + e_i$$
$$e_i \sim N(0, 25)$$

  Second create a sample with size N=100, and a ficticious $x_2 \sim N(0,1)$ from the model.

- 3. Estimate the model by OLS. We found our $\hat{\beta}$.

Monte Carlo Simulation of $\hat{\beta}$ OLS

Now let's use Monte Carlo simulation to show the asymptotic properties of $\hat{\beta}$.
Monte Carlo Simulation:

- **Step 1:** set the number T of repetitions of the simulation.
- **Step 2:** create an empty list (or array) to store results of the simulation.
- **Step 3**: create a loop where for each repetition $t = \{1, ..., T\}$ of the experiment:
  - Generate a new sample.
  - Run an OLS regression on the sample.
  - Store the OLS coefficients in the list.
- **Step 4:** Results: Plot the distribution of $\hat{\beta}_0$ and of $\hat{\beta}_1$. There we have the asymptotic properties of OLS.

Results: Monte Carlo Simulation of $\hat{\beta}$ OLS

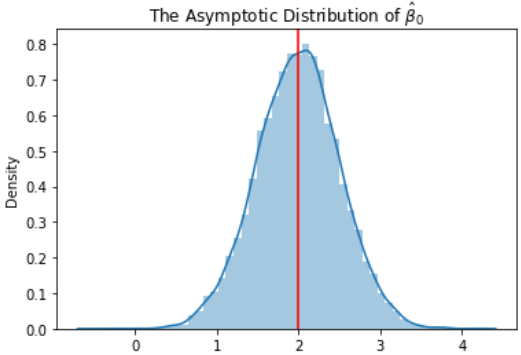


Figure 1: $\hat{\beta}_0$ distribution

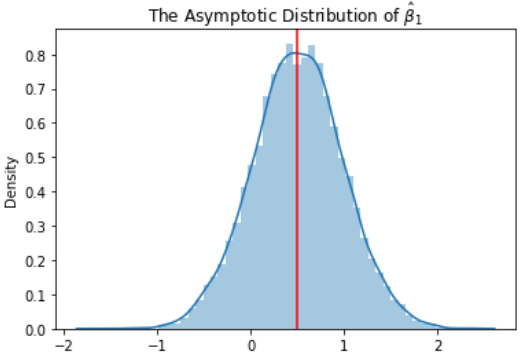


Figure 2: $\hat{\beta}_1$ distribution

Numerical Integration

Numerical Integration by Monte Carlo

Numerical integration consists of computing

$$\mathbb{E}\left[g(x)\right] \quad x \sim F$$

where F is a known distribution and $g()$ is a function (known or unknown). Note that the expectation is defined as the following integral

$$\mathbb{E}[g(x)] = \int_{\mathcal{X}} g(x) dF(x) = \int_{\mathcal{X}} g(x) f(x) dx$$

where f is the density—PDF–of the distribution (CDF) F and $\mathcal{X}$ is the domain of x.

Numerical integration by Monte Carlo

Relying on the law of large numbers we **approximate** the true integral with the **average of a simulated sample**:

$$\mathbb{E}[g(x)] \approx \frac{1}{N} \sum_{i=1}^{N} g(x_i)$$

where x_i is drawn from F using a random number generator.

Example: Numerical integration by Monte Carlo

Compute the expected value of $g(x) = (x-1)^4$ where $x \sim N(0,1)$ using Monte Carlo integration.

```
def g(x):
    return (x-1)**(4)
# 1. simulate g(x) for a big sample N
N = 100000
X = np.random.normal(0,1,N)
g_X = g(X)
# 2.  compute the sample average of g(X)
mc_integr_1 = np.mean(g_X)
# result
print('Monte Carlo Integration of g(x)')
print('E[g(x)] =', mc_integr_1)
>>
Monte Carlo Integration of g(x)
E[g(x)] = 9.984409
```

Monte Carlo integration: greatness and pitfalls

Numerical integration by Monte Carlo is a very powerful tool since it is **very robust method**. In principle, for any problem—and for any underlying distribution—we can integrate by Monte Carlo.

However, Monte Carlo integration relies on generating big samples and taking averages. **When the problems get complex**, Monte Carlo integration might be too **slow** and **imprecise** to be implemented.

- ▷ When the underlying distribution is normal, a solution is to use Gauss-Hermite Quadrature rules.

Gauss-Hermite quadrature rule

Idea: Instead of using a sample average—many points equally weighted—the **Gauss-Hermite quadrature rule** allows us to **approximate integrals computing a weighted average** in a smaller number of points,

$$\mathbb{E}\left[g(x)\right] \approx \sum_{i=1}^{N} \omega_i g(\varepsilon_i)$$

where $\{\varepsilon_i\}_{i=1}^{N}$ are the **quadrature nodes**—points—and $\{\omega_i\}_{i=1}^{N}$ are the **quadrature weights** that depend on original distribution $x \sim N(\mu, \Sigma)$.

- The *function gauss_hermite_1d(N, μ, Σ)* in the file *functions_albert.py* computes the quadrature nodes $\{\varepsilon_i\}_{i=1}^{N}$ and quadrature weights $\{\omega_i\}_{i=1}^{N}$ given a number of nodes N, a vector of means of means μ and a variance-covariange matrix Σ for a vector of variables X.

Example: Integration by Gauss-Hermite quadrature

Compute the expected value of $g(x) = (x-1)^4$ where $x \sim N(0,1)$ using Gauss-Hermite quadrature.

```
from functions_albert import gauss_hermite_1d  #file functions_albert must be in your WD
# The function
def g(x):
    return (x-1)**(4)
# 1. Compute the quadrature nodes and weights for x~N(0,1)
eps, w = gauss_hermite_1d(10,0,1)
# 2. Compute the weighted average
int_gh = np.sum(w*g(eps))
print('Gauss-Hermite Integration of g(x)')
print('E[g(x)] =', round(int_gh,6))
>>
Gauss-Hermite Integration of g(x)
 E[g(x)] = 10.0
```

Comparison: numerical integration by Monte Carlo vs Gauss-Hermite

Compare the speed and accuracy of Monte-Carlo integration vs Gauss-Hermite integration using the previous example.

- The true value of $\mathbb{E}[g(x)]$ is 10.
- Play with N in Monte-Carlo and N in Gauss-Hermite to get a precision of at least 2 decimals——i.e. a number between (0.990, 10.009).
- To check the time:

```
import time
tic = time.clock()
## your computations
toc = time.clock()
print('Elapsed time: ', round(toc-tic,6))
```

Time-dependent Random Processes

Ar(1) Model

We have seen how to simulate variables that are (time) independent. Yet, many economic variables follow processes with time dependency —i.e. GDP, inflation, wages, etc.

A simple class of time-dependent stochastic processes is the **AutoRegressive model of order 1**. **AR(1)** model is used to represent the dynamics of series such as

- Labor income.
- Stocks values.
- Productivity.
- In Macro time-series: VAR(j) for GDP, inflation, oil prices, etc.

AR(1) process

A simple **univariate autoregressive model of order 1** takes the following form

$$y_{t+1} = a + \rho y_t + \varepsilon_t$$
$$\varepsilon_t \sim N(0, \sigma^2)$$

where the output variable is linearly dependent on its own previous values and on a white noise term.

- This law of motion generates a **time series** as soon as we specify an initial condition y_0.
- AR(1) processes can take negative values. To have an AR(1) with **only positive values** take the exponential of the process. That is for variable x_t, assume $log(x_t) \sim AR(1)$.
- **Let's simulate some AR(1) processes with Python.**

Some AR(1) processes

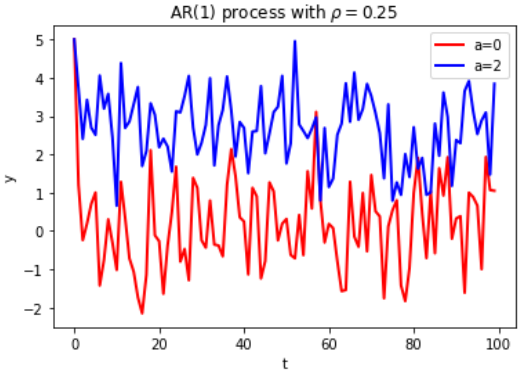


Figure 3: AR(1) with $\rho = 0.25$

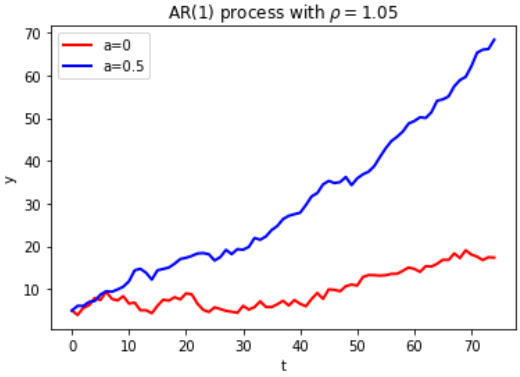


Figure 4: AR(1) with $\rho = 1.02$

Distribution dynamics of y_t

- First note that by iterating backwards, y_t can be rewritten as

$$y_t = \rho^t y_0 + a \sum_{r=0}^{t-1} \rho^r + \sum_{r=0}^{t-1} \varepsilon_{t-1}$$

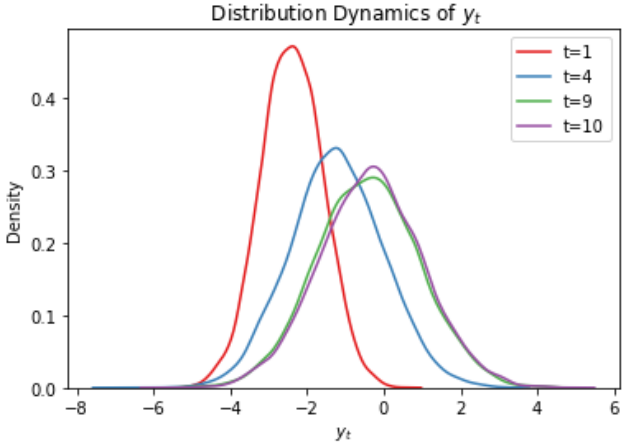
- Let ψ_t be the**distribution** of the random variable y in a point of time t.
- Since ε_t follows a normal, and y_t is a linear combination of independent normal random variables, y_t **follows a normal distribution**

$$y_t \sim \psi_t = N(\mu_t, v_t) \tag{1}$$

$$\text{and } \mu_{t+1} = \rho\mu_t + a \; v_t = \rho^2 v_t$$

Distribution dynamics of y_t: plotting ψ_t along t periods

Let y follow an AR(1) with $\rho = 0.8, y_0 = -5, \sigma_\varepsilon = 0.8$. Simulate 1000 y for 10 periods. Plot the distribution of y_t for $t = \{1, 4, 9, 10\}$. **Do we observe convergence in ψ_t?**



Stationary Distribution

A **stationary distribution** is a distribution that is a fixed point of the update rule for distributions. In other words, if ψ_t^* is stationary, then, $\psi_t^* = \psi_{t+j}$ for any $j > 0$.

▷ Stationary distributions have a natural interpretation as **stochastic steady states**.

In AR(1) models

- **If** $|\rho| < 1$: then the stochastic process is **stationary**—i.e. converges to a unique stationary distribution ψ^*

$$\psi^* = N(\mu^*, v^*)$$

and

$$\mu^* := \frac{a}{1-\rho},\ v^* := \frac{\sigma_\varepsilon^2}{1-\rho^2}$$

- **If** $|\rho| >= 1$: the stochastic process is **not stationary**, and the variance of the time-series y explodes.

Stochastic Matrices

A **stochastic matrix** (or Markov matrix) is an $n \times n$ square matrix P such that:

1. Each element is not negative.
2. Each row of P sums to 1.

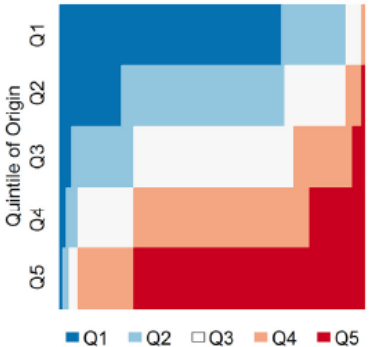
Each row of P can be regarded as a probability mass function over possible outcomes.

Example: income transition probabilities

Example: Let the rows denote the state today, and the columns the state tomorrow.

$$\begin{array}{c} \\ \textit{Poor} \\ \textit{Middle} \\ \textit{Rich} \end{array} \begin{array}{c} \begin{array}{ccc} \textit{Poor} & \textit{Middle} & \textit{Rich} \end{array} \\ \begin{pmatrix} 0.8 & 0.15 & 0.05 \\ 0.25 & 0.5 & 0.25 \\ 0.05 & 0.35 & 0.6 \end{pmatrix} \end{array}$$

Real world example: income mobility in the US. From De Magalhães, L., Santaeulàlia-Llopis, R. (2018)[a]



[a] Income transition probabilities in the US. Quintiles of the income distribution using 2004-06 PSID data.

Discrete Markov chains

Let $\mathcal{S}$ be a finite set with n elements $\{s_1, ..., s_n\}$. $\mathcal{S}$ is the **state space**, $s_1, ..., s_n$ are the **state values**.

A **Discrete Markov chain** $\{X_t\}$ on $\mathcal{S}$ is a sequence of random variables on $\mathcal{S}$ such that accomplish the **Markov property**. That is, for any t and state s_i

$$P(X_{t+1} = s_l | X_t) = P(X_{t+1} = s_i | X_t, X_{t-1}...)$$

Therefore, a discrete Markov chain (or process) is a sequence of random variables on a fixed finite set S with the probabilities defined by a stochastic matrix P.

P is also refered as the **transition matrix**. where each element p_{ij} is called the **transition probability** from state s_i to state s_j.

Examples Markov Chains

Example 1: $\mathcal{S}$ = income distribution states. s_1=poor, s_2=medium, s_3=rich. P_1 is the following stochastic matrix

$$P_1 = \begin{array}{c} \\ \text{Poor} \\ \text{Middle} \\ \text{Rich} \end{array} \begin{array}{c} \begin{array}{ccc} \text{Poor} & \text{Middle} & \text{Rich} \end{array} \\ \begin{pmatrix} 0.8 & 0.15 & 0.05 \\ 0.25 & 0.5 & 0.25 \\ 0.05 & 0.35 & 0.6 \end{pmatrix} \end{array}$$

Example 2: Let $\mathcal{S} = \{u, e\}$ where u denotes a person in unemployment state, e denotes a person in an employment state. The unemployment/employment transitions are given by the following stochastic matrix P_2.

$$P_2 = \begin{array}{c} \\ \text{u} \\ \text{e} \end{array} \begin{array}{c} \begin{array}{cc} \text{u} & \text{e} \end{array} \\ \begin{pmatrix} 0.4 & 0.6 \\ 0.8 & 0.2 \end{pmatrix} \end{array}$$

Markov chains properties: dynamics

- Markov chain is called **periodic** if it cycles in a predictable way, and **aperiodic** otherwise.
- Note from the definition of the Markov chain, the distribution ψ_t across states s_t evolves as follows

$$\psi_{t+1} = \psi_t P$$

- Given an initial distribution of the states ψ_0 the distribution of states in period t, ψ_t is the following

$$\psi_t = \psi_0 P^t$$

Markov chains properties: stationarity

Stationary Distribution. The stationary distribution of a Markov chain is characterized by a probability distribution ψ^* such that

$$\psi^* = \psi^* P$$

- **Theorem 1:** Every stochastic matrix has **at least one stationary distribution**, ψ^*.
- **Theorem 2:** If P is both aperiodic and irreducible(i.e. can go from any state to any state), then,
  1. P as **a unique stationary distribution** $\psi^* = \psi^* P$.
  2. for any initial marginal distribution, ψ_0, For a large enough t $\psi_0 P^t$ converges to ψ^*.

Stationarity and dynamics in Markov chains

Given the previous properties, we can answer the following questions from our examples

- Over the long run, what is the proportion of people that are poor, middle income or rich?
- Over the long run, what is the proportion of people that are unemployed?
- What is the average duration of unemployment given P2?

Let's answer these questions in Python!

The quantecon project has a great library on Markov chains. Install the quantecon package and let's answer the questions!

The Rouwenhorst method: discretize AR(1) into Markov chains

Markov chains are fundamental for quantitative modeling. In particular, with **Markov chains allow us to approximate AR(1) process**—continuous, infinite points—to discrete states.

The Rouwenhorst method allows us to convert an AR(1) process into a discrete Markov chain. We can implement this method using the following routine in the quantecon library

- quantecon.rouwenhorst(mu=a,rho=ρ, sigma=σ_ε, n=N).
  Where a, ρ, σ_ε are the parameters in our AR(1). N is in how many states s_i we want to discretize our AR(1).

Exercise: Let's go to Python and discretize the two AR(1) processes in Figure 3 into Markov chains of 5 states.