

Postponed exam ECON3170/4170 Fall 2025

1. Voting in the 2025 Eurovision Song Contest

a)

```
library(tidyverse)
esc <- read_csv("esc.csv")
dim(esc) - c(0,4) # 4 variables in addition to point giving countries
```

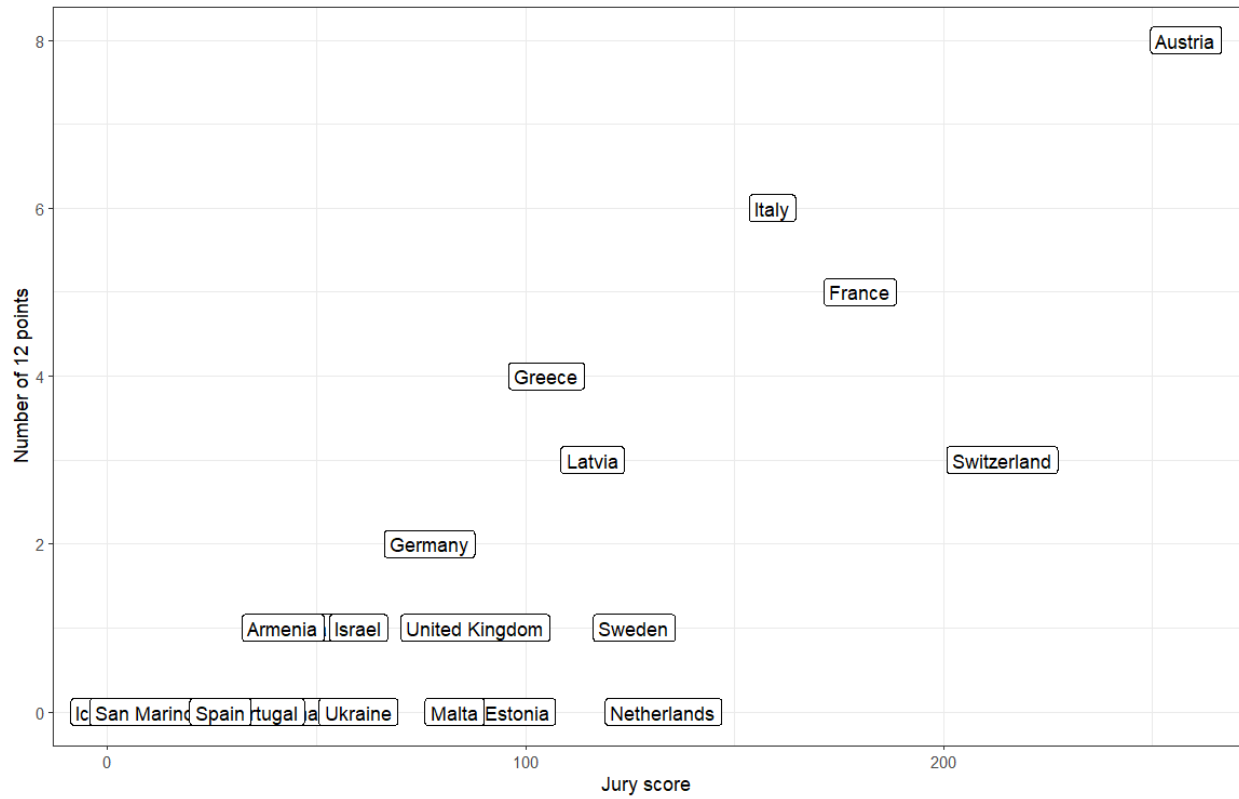
26 participants and 37 point awarders

b)

```
esc_long <- esc |>
  pivot_longer(cols = Sweden:Switzerland,
               names_to = 'awarding',
               values_to = 'score') |>
  replace_na(list(score = 0))
```

c)

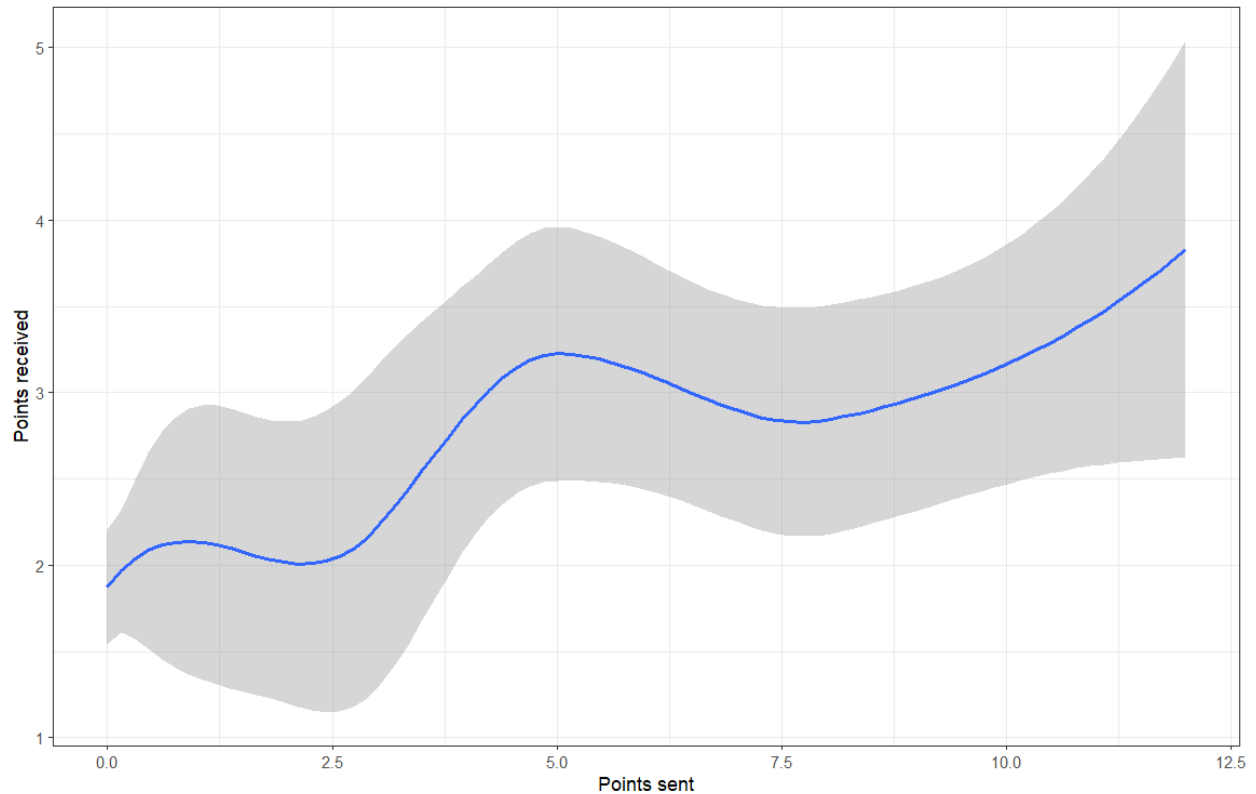
```
esc_long |>
  mutate(douze = score==12) |>
  group_by(contestant, jury_score) |>
  summarise(n_douze = sum(douze)) |>
  ggplot(aes(x = jury_score, y = n_douze)) +
  geom_label(aes(label = contestant)) +
  labs(x = 'Jury score', y = 'Number of 12 points') +
  theme_bw()
```



d)

```
esc_reverse <- esc_long |>
  select(-ends_with('_score')) |>
  rename(score_reverse = score,
         contestant = awarding,
         awarding = contestant)

esc_long |>
  left_join(esc_reverse) |>
  ggplot(aes(x=score, y=score_reverse)) +
  geom_smooth() +
  labs(x = "Points sent", y = "Points received") +
  theme_bw()
```



There is a weak tendency of a positive relationship, but not very strong. Hence only moderate support for the claim.

e)

```
library(readxl)
televote <- read_excel("esc.xlsx",
                      sheet = "Televote", skip = 2) |>
  select(-ends_with('vote')) |>
  select(-ends_with('World')) |>
  rename(contestant = `...2`,
         total_score = `...3`,
         jury_score = `...4`,
         televote_score = `...5`) |>
  filter(!is.na(contestant)) |>
  pivot_longer(cols = Sweden:Switzerland,
               names_to = 'awarding',
               values_to = 'televote') |>
  replace_na(list(televote = 0))
```

f)

```
esc_long |>
  left_join(televote) |>
  mutate(abs_diff = abs(score-televote)) |>
  group_by(contestant) |>
  summarise(abs_diff=mean(abs_diff)) |>
  arrange(-abs_diff) |>
  slice_head(n = 5)
```

	contestant	abs_diff
	<chr>	<dbl>
1	Israel	6.46
2	Switzerland	5.78
3	Estonia	5.30
4	France	4.54
5	Austria	4.46

2. Integration

a)

```
library(tidyverse)
f <- function(x) {
  return((x-1)/log(x))
}

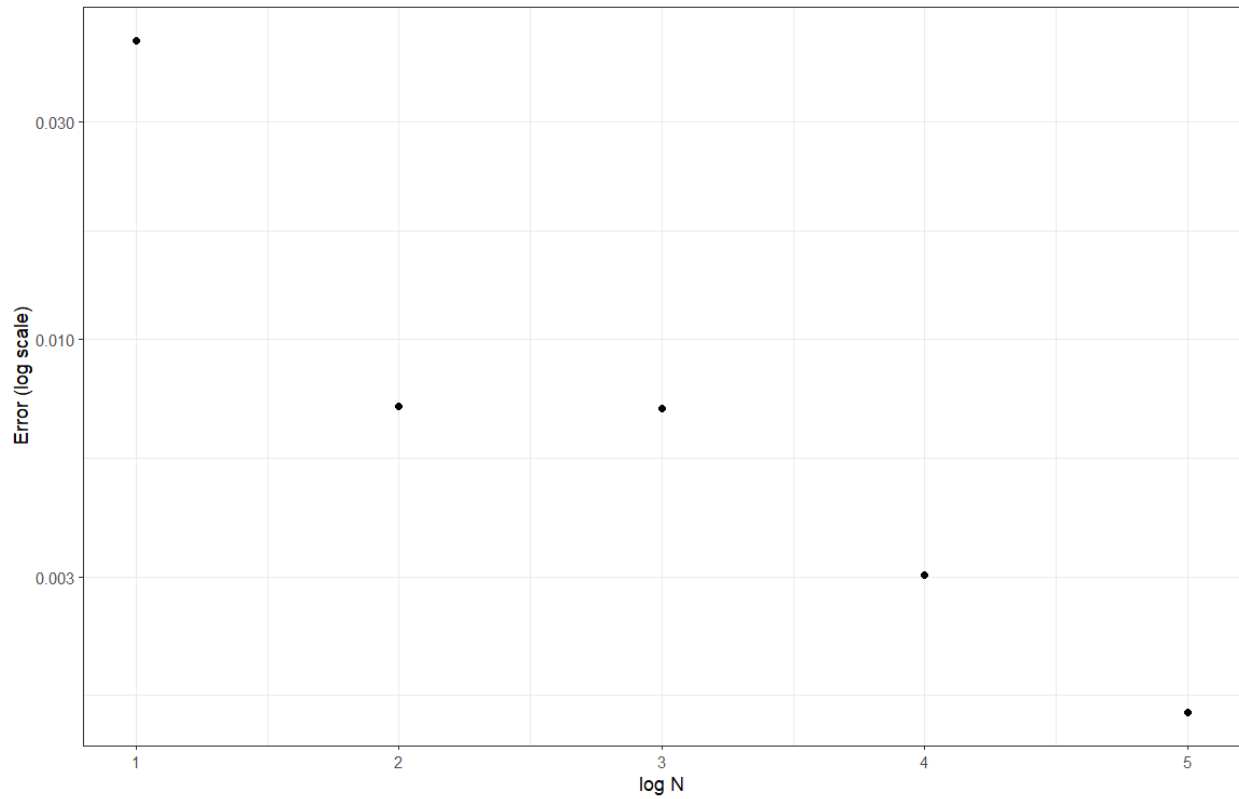
set.seed(12345)
I <- mean(f(runif(1000)))
```

We get a value of 0.7053

b)

```
Ns <- NULL
In <- NULL
for (N in 1:5) {
  set.seed(12345)
  Ns <- c(Ns,N)
  In <- c(In, mean(f(runif(10^N))))
}

tibble(
  N = Ns,
  err=abs(In-log(2))
) |>
  ggplot(aes(x=N, y=err)) +
  geom_point() +
  scale_y_log10() +
  labs(x = "log N", y = "Error (log scale)") +
  theme_bw()
```



There seems to be a negative relationship, so higher N leads to smaller errors. Increasing the number of simulations typically increase the precision of MC integration.

c)

```
u <- function(c) {  
  return((c^(1-sigma)-1)/(1-sigma))  
}
```

```
set.seed(12345)  
sigma <- 2
```

```
N <- 1000 # Number of draws  
b <- 5  
e <- exp(rnorm(N))
```

```
mean(u(b+e))
```

The computed expected utility is 0.8423969

3. Text encoding

ASCII was the traditional way to store text as binary on computers. It assigns each character a number from 0–127, stored in one byte (with the top bit 0). For example, “A” is decimal 65 or binary 01000001. This covers English letters, digits, punctuation, and some control characters. A string like "Hi! becomes the bytes 01001000 01101001 00100001.

The main problem: 128 characters is far too few for all the world’s writing systems and symbols. Different regions tried to fix this with “extended ASCII” encodings (ISO 8859-1, Windows-1252, etc.), using bytes 128–255 differently. The same byte could mean different characters depending on the encoding, so text often appeared garbled if opened with the wrong setting. There was no single, global character set.

Unicode addresses this by defining a universal set of code points for virtually all characters and symbols. But Unicode by itself is abstract: it doesn’t specify how these code points are stored as bytes. That’s what encodings like UTF-8, UTF-16, and UTF-32 do.

UTF-8 is a variable-length encoding for Unicode using 1–4 bytes per code point:

- ASCII range: 1 byte 0xxxxxxx
- Higher code points: 2, 3, or 4 bytes, with a leading pattern like 110xxxxx, 1110xxxx, 11110xxx and continuation bytes 10xxxxxx.

Crucially, ASCII bytes map to exactly the same values in UTF-8. So any valid ASCII text is automatically valid UTF-8, with no conversion needed. Non-ASCII characters are encoded as multi-byte sequences. For example, `é` becomes two bytes, and Chinese `中` becomes three bytes.

Thus ASCII is a subset of UTF-8 with identical byte values. Unicode defines code points; UTF-8 is one specific way of representing those code points as bytes. UTF-8 solves ASCII’s core problems by providing:

- A single, standardized encoding for all Unicode characters.
- Backward compatibility with ASCII.
- Consistent interpretation of byte sequences across systems.

Shortcomings of UTF-8 include:

- Variable-length: 1–4 bytes per code point makes indexing by “character” more complex; you can’t safely cut strings at arbitrary byte boundaries.
- Size for some languages: many East Asian characters take 3 bytes, sometimes larger than in older local encodings.
- General Unicode complexity (combining characters, emoji sequences, normalization) remains; UTF-8 doesn’t remove that, it just encodes the code points reliably.

Despite this, UTF-8 has become the dominant text encoding on the web and in many systems because its benefits and compatibility greatly outweigh its drawbacks.