# MAS2905: Probability and Bayesian Methods with R

Newcastle University
lee.fawcett@ncl.ac.uk

Semester 1, 2020/21

# What you should have done so far...

- Completed all work from week 1, including:
  - Work from our first synchronous online lecture last Wednesday, 21st October
  - All work associated with the videos **Home Study 1** and **Home Study 2**
- You should have worked through **Examples 2.1–2.4**; solutions to Example 2.4 will be revealed at the start of the **Home Study 3** video, to be released later this week
- This all means we have covered everything up to, and including, Section 2.5 of the course notes

# What you should have done so far...

- Completed all work from week 1, including:
  - Work from our first synchronous online lecture last Wednesday, 21st October
  - All work associated with the videos **Home Study 1** and **Home Study 2**
- You should have worked through **Examples 2.1–2.4**; solutions to Example 2.4 will be revealed at the start of the **Home Study 3** video, to be released later this week
- This all means we have covered everything up to, and including, Section 2.5 of the course notes

# What you should have done so far...

- Completed all work from week 1, including:
  - Work from our first synchronous online lecture last Wednesday, 21st October
  - All work associated with the videos **Home Study 1** and **Home Study 2**
- You should have worked through **Examples 2.1–2.4**; solutions to Example 2.4 will be revealed at the start of the **Home Study 3** video, to be released later this week
- This all means we have covered everything up to, and including, Section 2.5 of the course notes

# Today's practical work

**Main Aim:** Recap **functions**, **for loops** and **if statements** in R.

We have three main activities to complete:

1. **The sticker album**: Self-directed reading and computing, covering Section 2.6 of the course notes ($\sim$30 minutes)
2. A look at **sampling distributions**: Independent work on for loops and functions, with 4 questions to answer ($\sim$30 minutes)
3. **Monte Carlo integration**: Guided example, and then another for you to try out independently! ($\sim$30 minutes)

All exercises should get you up-to-speed with functions, for loops and if statements.

# Today's practical work

**Main Aim:** Recap **functions**, **for loops** and **if statements** in R.

We have three main activities to complete:

1. **The sticker album**: Self-directed reading and computing, covering Section 2.6 of the course notes (∼30 minutes)
2. A look at **sampling distributions**: Independent work on for loops and functions, with 4 questions to answer (∼30 minutes)
3. **Monte Carlo integration**: Guided example, and then another for you to try out independently! (∼30 minutes)

All exercises should get you up-to-speed with functions, for loops and if statements.

# Today's practical work

**Main Aim:** Recap **functions**, **for loops** and **if statements** in R.

We have three main activities to complete:

1. **The sticker album**: Self-directed reading and computing, covering Section 2.6 of the course notes ($\sim$30 minutes)
2. A look at **sampling distributions**: Independent work on for loops and functions, with 4 questions to answer ($\sim$30 minutes)
3. **Monte Carlo integration**: Guided example, and then another for you to try out independently! ($\sim$30 minutes)

All exercises should get you up-to-speed with functions, for loops and if statements.

# Part 1: Sticker album

This part of the practical will take you through Section 2.6 of the course notes, thus completing our R Refresher.

The 2018 *Panini* sticker album for the Fifa world cup football finals had a whopping 682 stickers to collect (including 40 awesome glitter stickers!). Each packet of stickers cost 80p and contained 5 stickers.

The obvious question to ask is:

*"How much does it cost to fill the sticker album?"*

# Part 1: Sticker album

Try out the following line of code a few times, and make sure you understand what's going on! Then move to the next slide.

Initially, to simplify things, we'll assume there are only **50** stickers to collect.

```
# Let's buy a packet of stickers!

# Random sample of size 5 from {1, 2, ..., 50}
(packet = sample(1:50, 5, replace=TRUE))
 [1] 17 28 34  8  1
# We got stickers 1, 8, 17, 28 and 34!

(packet = sample(1:50, 5, replace=TRUE))
 [1] 10 23 20 48 11
# Now we got stickers 10, 11, 20, 23 and 48!
```

# Part 1: Sticker album

Try out the following line of code a few times, and make sure you understand what's going on! Then move to the next slide.

Initially, to simplify things, we'll assume there are only **50** stickers to collect.

```
# Let's buy a packet of stickers!

# Random sample of size 5 from {1, 2, ..., 50}
(packet = sample(1:50, 5, replace=TRUE))
 [1] 17 28 34  8  1
# We got stickers 1, 8, 17, 28 and 34!

(packet = sample(1:50, 5, replace=TRUE))
 [1] 10 23 20 48 11
# Now we got stickers 10, 11, 20, 23 and 48!
```

# Part 1: Sticker album

Now try out the next two blocks of code - again, make sure you follow what's going on.

```
# Create a vector for the sticker sequence: 50 stickers
(stickers = seq(1, 50))
  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
 [17] 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
 [33] 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
 [49] 49 50
```

```
# As a way of keeping track of the stickers we have collected, we could set the
# corresponding sticker sequence entry to zero
stickers[packet[1]] = 0        # Will set stickers[10] to 0
stickers[packet[2]] = 0        # Will set stickers[23] to 0
stickers[packet[3]] = 0        # Will set stickers[20] to 0
stickers[packet[4]] = 0        # Will set stickers[48] to 0
stickers[packet[5]] = 0        # Will set stickers[11] to 0
```

# Part 1: Sticker album

The last block of code consisted of five lines, each one doing the same thing!

This is crying out for a **for loop**:

```
for(j in 1:5){
  stickers[packet[j]] = 0}

# Now look at what's happened to stickers (yours will be different!)
stickers
  [1]  1  2  3  4  5  6  7  8  9  0  0 12 13 14 15 16
 [17] 17 18 19  0 21 22  0 24 25 26 27 28 29 30 31 32
 [33] 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47  0
 [49] 49 50
```

# Part 1: Sticker album

The last block of code consisted of five lines, each one doing the same thing!

This is crying out for a **for loop**:

```
for(j in 1:5){
  stickers[packet[j]] = 0}

# Now look at what's happened to stickers (yours will be different!)
stickers
  [1]  1  2  3  4  5  6  7  8  9  0  0 12 13 14 15 16
 [17] 17 18 19  0 21 22  0 24 25 26 27 28 29 30 31 32
 [33] 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47  0
 [49] 49 50
```

# Part 1: Sticker album

Now:

- Try buying another packet of stickers using the `sample` command
- Run this packet of stickers through the `for` loop
- Watch how the `stickers` vector is updated – you should see more zeros being added!
- The aim is to keep buying packets of stickers and setting the corresponding entry in the `sticker` sequence to zero as a way of keeping track of the stickers we have collected...
- ... Once again, we are doing the same thing repeatedly, so another **for loop** is required!

# Part 1: Sticker album

Now:

- Try buying another packet of stickers using the `sample` command
- Run this packet of stickers through the `for` loop
- Watch how the `stickers` vector is updated – you should see more zeros being added!
- The aim is to keep buying packets of stickers and setting the corresponding entry in the `sticker` sequence to zero as a way of keeping track of the stickers we have collected...
- ... Once again, we are doing the same thing repeatedly, so another **for loop** is required!

# Part 1: Sticker album

Now:

- Try buying another packet of stickers using the `sample` command
- Run this packet of stickers through the `for` loop
- Watch how the `stickers` vector is updated – you should see more zeros being added!
- The aim is to keep buying packets of stickers and setting the corresponding entry in the `sticker` sequence to zero as a way of keeping track of the stickers we have collected...
- ... Once again, we are doing the same thing repeatedly, so another **for loop** is required!

# Part 1: Sticker album

```
# Re-set the sticker sequence
(stickers = seq(1, 50))
  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
 [17] 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
 [33] 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
 [49] 49 50

# Start a big for loop to repeatedly buy packets of stickers
for(i in 1:10000){          # 10,000 – something large!
  packet = sample(1:50, 5, replace=TRUE)

  for(j in 1:5){
    stickers[packet[j]] = 0}# Close the j loop
    }# Close the i loop

# Take a look at stickers:
stickers
  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 [26] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

# Part 1: Sticker album

Looking at the listing above, it looks like we've filled the sticker album after buying 10,000 packets of stickers – all entries in the sticker sequence vector are zero

- But, we don't know exactly when this happened
- Did it happen after we bought the 10,000th packet?
- Or after we bough the 200th packet?
- Or the 1524th packet?!

We need to 'break' the code as soon as all entries in the sticker sequence are zero.

# Part 1: Sticker album

Looking at the listing above, it looks like we've filled the sticker album after buying 10,000 packets of stickers – all entries in the sticker sequence vector are zero

- But, we don't know exactly when this happened
- Did it happen after we bought the 10,000th packet?
- Or after we bough the 200th packet?
- Or the 1524th packet?!

We need to 'break' the code as soon as all entries in the sticker sequence are zero.

# Part 1: Sticker album

Looking at the listing above, it looks like we've filled the sticker album after buying 10,000 packets of stickers – all entries in the sticker sequence vector are zero

- But, we don't know exactly when this happened
- Did it happen after we bought the 10,000th packet?
- Or after we bough the 200th packet?
- Or the 1524th packet?!

We need to 'break' the code as soon as all entries in the sticker sequence are zero.

# Part 1: Sticker album

Looking at the listing above, it looks like we've filled the sticker album after buying 10,000 packets of stickers – all entries in the sticker sequence vector are zero

- But, we don't know exactly when this happened
- Did it happen after we bought the 10,000th packet?
- Or after we bough the 200th packet?
- Or the 1524th packet?!

We need to 'break' the code as soon as all entries in the sticker sequence are zero.

# Part 1: Sticker album

Looking at the listing above, it looks like we've filled the sticker album after buying 10,000 packets of stickers – all entries in the sticker sequence vector are zero

- But, we don't know exactly when this happened
- Did it happen after we bought the 10,000th packet?
- Or after we bough the 200th packet?
- Or the 1524th packet?!

We need to 'break' the code as soon as all entries in the sticker sequence are zero.

# Part 1: Sticker album

Looking at the listing above, it looks like we've filled the sticker album after buying 10,000 packets of stickers – all entries in the sticker sequence vector are zero

- But, we don't know exactly when this happened
- Did it happen after we bought the 10,000th packet?
- Or after we bough the 200th packet?
- Or the 1524th packet?!

We need to 'break' the code as soon as all entries in the sticker sequence are zero.

# Part 1: Sticker album

Now edit your code, with the addition of a `break` command within an **if statement**:

```
stickers = seq(1, 50)

for(i in 1:10000){
  packet = sample(1:50, 5, replace=TRUE)

  for(j in 1:5){
    stickers[packet[j]] = 0}

# Break the code when all entries in the sticker sequence are zero
  if(sum(stickers)==0) {
    break}
  }

# When did the code break?
i
 [1] 37
```

# Part 1: Sticker album

So this time it only took 37 packets of stickers to completely fill the album! (Yours may well be different)

Of course, in this toy example we only have 50 stickers to collect.

For the *Panini* football sticker album, there are 682!

Really, what we'd like to do is run the code in the above listing repeatedly, to see what happens for different 'people' trying to fill their sticker albums.

It might be a good idea to put all of this into a **function** now.

# Part 1: Sticker album

So this time it only took 37 packets of stickers to completely fill the album! (Yours may well be different)

Of course, in this toy example we only have 50 stickers to collect.

For the *Panini* football sticker album, there are 682!

Really, what we'd like to do is run the code in the above listing repeatedly, to see what happens for different 'people' trying to fill their sticker albums.

It might be a good idea to put all of this into a **function** now.

# Part 1: Sticker album

So this time it only took 37 packets of stickers to completely fill the album! (Yours may well be different)

Of course, in this toy example we only have 50 stickers to collect.

For the *Panini* football sticker album, there are 682!

Really, what we'd like to do is run the code in the above listing repeatedly, to see what happens for different 'people' trying to fill their sticker albums.

It might be a good idea to put all of this into a **function** now.

# Part 1: Sticker album

So this time it only took 37 packets of stickers to completely fill the album! (Yours may well be different)

Of course, in this toy example we only have 50 stickers to collect.

For the *Panini* football sticker album, there are 682!

Really, what we'd like to do is run the code in the above listing repeatedly, to see what happens for different 'people' trying to fill their sticker albums.

It might be a good idea to put all of this into a **function** now.

# Part 1: Sticker album

So this time it only took 37 packets of stickers to completely fill the album! (Yours may well be different)

Of course, in this toy example we only have 50 stickers to collect.

For the *Panini* football sticker album, there are 682!

Really, what we'd like to do is run the code in the above listing repeatedly, to see what happens for different 'people' trying to fill their sticker albums.

It might be a good idea to put all of this into a **function** now.

# Part 1: Sticker album

Now use the following code to make your `FillAlbum` function.

```
# N = something large, k = number of stickers to collect
FillAlbum = function(N, k){
  stickers = seq(1, k)

  for(i in 1:N){
  packet = sample(1:k, 5, replace=TRUE)

  for(j in 1:5){
    stickers[packet[j]] = 0}

  if(sum(stickers)==0) {
    break}
  }
  return(i)}
```

# Part 1: Sticker album

Notice, from the code listing above, the general structure of a function in R:

- The word `function` (line 2), followed by round brackets and some "arguments" to be passed through the function (arguments that we must specify when we execute the function)
- An appropriate name for the function (`FillAlbum` here, see line 2)
- The use of braces surrounding the operations within the function (lines 2 and 14; all other braces relate to the for loops/if statement)
- A `return` statement (line 14), declaring what we would like the function to spit out

# Part 1: Sticker album

Notice, from the code listing above, the general structure of a function in R:

- The word `function` (line 2), followed by round brackets and some "arguments" to be passed through the function (arguments that we must specify when we execute the function)
- An appropriate name for the function (`FillAlbum` here, see line 2)
- The use of braces surrounding the operations within the function (lines 2 and 14; all other braces relate to the for loops/if statement)
- A `return` statement (line 14), declaring what we would like the function to spit out

# Part 1: Sticker album

Notice, from the code listing above, the general structure of a function in R:

- The word `function` (line 2), followed by round brackets and some "arguments" to be passed through the function (arguments that we must specify when we execute the function)
- An appropriate name for the function (`FillAlbum` here, see line 2)
- The use of braces surrounding the operations within the function (lines 2 and 14; all other braces relate to the for loops/if statement)
- A `return` statement (line 14), declaring what we would like the function to spit out

# Part 1: Sticker album

Notice, from the code listing above, the general structure of a function in R:

- The word `function` (line 2), followed by round brackets and some "arguments" to be passed through the function (arguments that we must specify when we execute the function)
- An appropriate name for the function (`FillAlbum` here, see line 2)
- The use of braces surrounding the operations within the function (lines 2 and 14; all other braces relate to the for loops/if statement)
- A `return` statement (line 14), declaring what we would like the function to spit out

# Part 1: Sticker album

Now execute your `FIllAlbum` function – first for 50 stickers, and then for the full 682!

```
# Let's try for an album containing just 50 stickers again
FillAlbum(10000, 50)
  [1] 52
FillAlbum(10000, 50)
  [1] 32
FillAlbum(10000, 50)
  [1] 40

# Now let's try for the full album of 682 stickers!
FillAlbum(10000, 682)
  [1] 1290
FillAlbum(10000, 682)
  [1] 1611
FillAlbum(10000, 682)
  [1] 1252
```

# Part 1: Sticker album

Now execute your `FIllAlbum` function – first for 50 stickers, and then for the full 682!

```
# Let's try for an album containing just 50 stickers again
FillAlbum(10000, 50)
 [1] 52
FillAlbum(10000, 50)
 [1] 32
FillAlbum(10000, 50)
 [1] 40

# Now let's try for the full album of 682 stickers!
FillAlbum(10000, 682)
 [1] 1290
FillAlbum(10000, 682)
 [1] 1611
FillAlbum(10000, 682)
 [1] 1252
```

# Part 1: Sticker album

Referring to the code in the listing above, we have simulated three people trying to fill the full *Panini* sticker album.

Respectively, they needed to buy 1,290, 1,611 and 1,252 packets of stickers to do this.

We could now execute this function many times, within another `for` loop.

# Part 1: Sticker album

Referring to the code in the listing above, we have simulated three people trying to fill the full *Panini* sticker album.

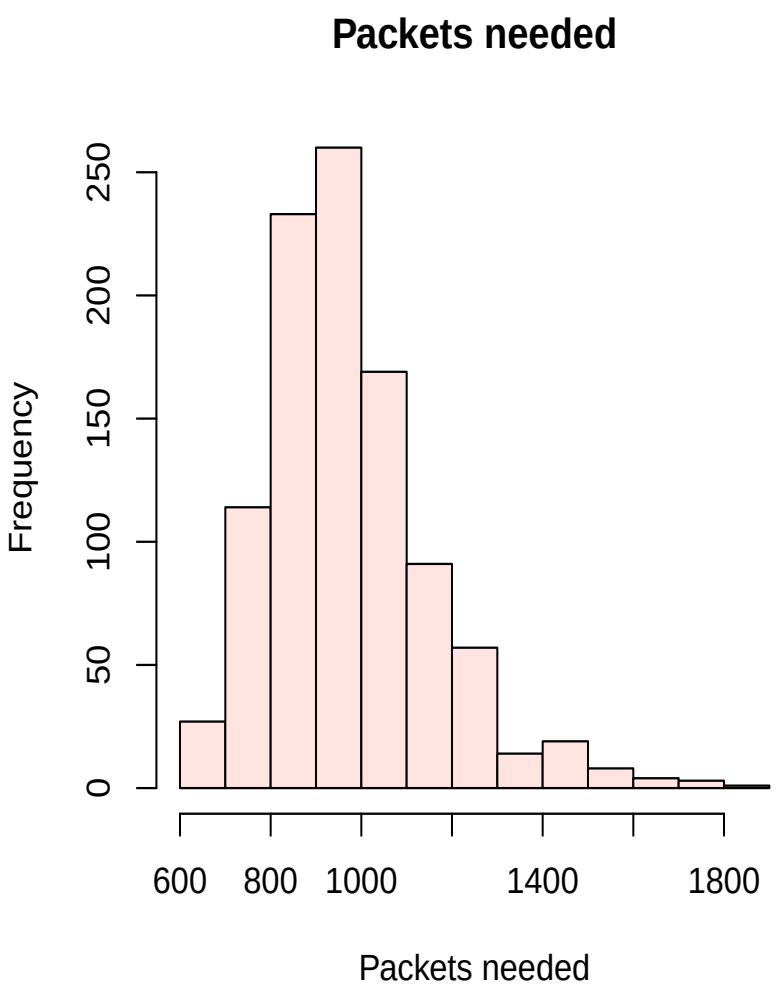
Respectively, they needed to buy 1,290, 1,611 and 1,252 packets of stickers to do this.

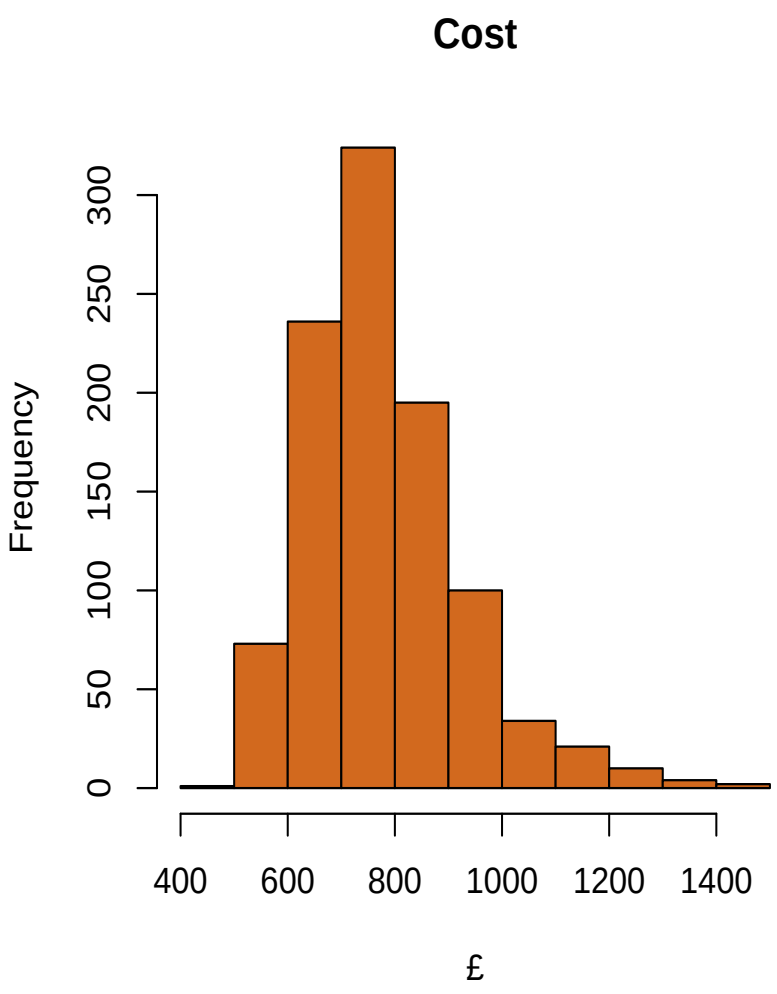
We could now execute this function many times, within another `for` loop.

# Part 1: Sticker album

```
# Let's simulate 1000 people trying to fill their albums
packets_needed = vector("numeric", 1000)

for(k in 1:1000){
  packets_needed[k] = FillAlbum(10000, 682)}

cost = packets_needed*0.8        # Each packet costs 80p

mean(packets_needed); mean(cost)
 [1] 974.609
 [1] 779.6872

sd(packets_needed); sd(cost)
 [1] 181.351
 [1] 145.0808

par(mfrow=c(1,2))
hist(packets_needed, col="misty rose", xlab="Packets",
    main="Packets needed")
hist(cost, col="chocolate", xlab="Pounds",main="Cost")
```

# Part 1: Sticker album



# Part 1: Sticker album

This analysis is really insightful!

- We can see that, on average, it takes around 975 packets of stickers to fill the album (standard deviation 181)
- There is an associated cost of around £780 (standard deviation £145)
- The histograms reveal the full distribution for the number of packets needed and the associated cost
- The larger the number of people simulated (we used 1,000 here), the more accurate this distribution will be!

# Part 1: Sticker album

This analysis is really insightful!

- We can see that, on average, it takes around 975 packets of stickers to fill the album (standard deviation 181)
- There is an associated cost of around £780 (standard deviation £145)
- The histograms reveal the full distribution for the number of packets needed and the associated cost
- The larger the number of people simulated (we used 1,000 here), the more accurate this distribution will be!

# Part 1: Sticker album

This analysis is really insightful!

- We can see that, on average, it takes around 975 packets of stickers to fill the album (standard deviation 181)
- There is an associated cost of around £780 (standard deviation £145)
- The histograms reveal the full distribution for the number of packets needed and the associated cost
- The larger the number of people simulated (we used 1,000 here), the more accurate this distribution will be!

# Part 1: Sticker album

This analysis is really insightful!

- We can see that, on average, it takes around 975 packets of stickers to fill the album (standard deviation 181)
- There is an associated cost of around £780 (standard deviation £145)
- The histograms reveal the full distribution for the number of packets needed and the associated cost
- The larger the number of people simulated (we used 1,000 here), the more accurate this distribution will be!

# Part 1: Sticker album

This analysis is really insightful!

- We can see that, on average, it takes around 975 packets of stickers to fill the album (standard deviation 181)
- There is an associated cost of around £780 (standard deviation £145)
- The histograms reveal the full distribution for the number of packets needed and the associated cost
- The larger the number of people simulated (we used 1,000 here), the more accurate this distribution will be!

# Part 2: Sampling distributions

The results of this exercise support material we will consider towards the end of the semester, but the exercise itself lends itself well to checking your basic programming skills in R!

(a) Commit the following code carefully to RStudio, and then produce a histogram of `x`. What can you say about the shape of the distribution you see?

```
x = rexp(10000, 2)
```

Do you know what the function `rexp` is actually doing? (don't worry if not!) Can you hazard a guess?

# Part 2: Sampling distributions

The results of this exercise support material we will consider towards the end of the semester, but the exercise itself lends itself well to checking your basic programming skills in R!

(a) Commit the following code carefully to RStudio, and then produce a histogram of `x`. What can you say about the shape of the distribution you see?

```
x = rexp(10000, 2)
```

Do you know what the function `rexp` is actually doing? (don't worry if not!) Can you hazard a guess?

# Part 2: Sampling distributions

(b) We now want to see what the distribution of the sample mean from `x` looks like.

(i) Use the `sample` command to draw a random sample from `x` of length 30. Find the mean of this sample, and write it down.

(ii) Repeat what you did in (i) a further four times.

(iii) Now you should repeat what you did in part (i) <u>1000</u> times... but you should automate the process by using a for loop!

First of all, type the following code into RStudio, which will set up a vector of length 1000 to take sample means from repeated random samples from `x`:

```
xbar = vector("numeric", 1000)
```

Now at each iteration $i$ in your for loop, $i = 1, \ldots, 1000$, store the sample mean in position $i$ of the vector `xbar`.

# Part 2: Sampling distributions

(b) We now want to see what the distribution of the sample mean from `x` looks like.

(i) Use the `sample` command to draw a random sample from `x` of length 30. Find the mean of this sample, and write it down.

(ii) Repeat what you did in (i) a further four times.

(iii) Now you should repeat what you did in part (i) <u>1000</u> times... but you should automate the process by using a for loop!

First of all, type the following code into RStudio, which will set up a vector of length 1000 to take sample means from repeated random samples from `x`:

```
xbar = vector("numeric", 1000)
```

Now at each iteration $i$ in your for loop, $i = 1, \ldots, 1000$, store the sample mean in position $i$ of the vector `xbar`.

# Part 2: Sampling distributions

(b) We now want to see what the distribution of the sample mean from `x` looks like.

(i) Use the `sample` command to draw a random sample from `x` of length 30. Find the mean of this sample, and write it down.

(ii) Repeat what you did in (i) a further four times.

(iii) Now you should repeat what you did in part (i) <u>1000</u> times... but you should automate the process by using a for loop!

First of all, type the following code into RStudio, which will set up a vector of length 1000 to take sample means from repeated random samples from `x`:

```
xbar = vector("numeric", 1000)
```

Now at each iteration $i$ in your for loop, $i = 1, \ldots, 1000$, store the sample mean in position $i$ of the vector `xbar`.

# Part 2: Sampling distributions

(b) We now want to see what the distribution of the sample mean from `x` looks like.

(i) Use the `sample` command to draw a random sample from `x` of length 30. Find the mean of this sample, and write it down.

(ii) Repeat what you did in (i) a further four times.

(iii) Now you should repeat what you did in part (i) <u>1000</u> times... but you should automate the process by using a for loop!

First of all, type the following code into RStudio, which will set up a vector of length 1000 to take sample means from repeated random samples from `x`:

```
xbar = vector("numeric", 1000)
```

Now at each iteration $i$ in your for loop, $i = 1, \ldots, 1000$, store the sample mean in position $i$ of the vector `xbar`.

# Part 2: Sampling distributions

(b) We now want to see what the distribution of the sample mean from `x` looks like.

(i) Use the `sample` command to draw a random sample from `x` of length 30. Find the mean of this sample, and write it down.

(ii) Repeat what you did in (i) a further four times.

(iii) Now you should repeat what you did in part (i) <u>1000</u> times... but you should automate the process by using a for loop!

First of all, type the following code into RStudio, which will set up a vector of length 1000 to take sample means from repeated random samples from `x`:

```
xbar = vector("numeric", 1000)
```

Now at each iteration $i$ in your for loop, $i = 1, \ldots, 1000$, store the sample mean in position $i$ of the vector `xbar`.

# Part 2: Sampling distributions

(c) The *Central Limit Theorem*, which we will come back to later in this module, states that – regardless of the distribution of the population – the distribution of the sample mean will be Normal, with mean $\mu$ and variance $\sigma^2/n$, where

- $\mu$ is the population mean
- $\sigma^2$ is the population variance
- $n$ is the sample size (e.g. $n = 30$ in part (b))

Suppose your vector `x` contains the population. Can you verify the Central Limit Theorem for the distribution of your sample means obtained in (b)(iii)?

(d) Write a function that explores the Central Limit Theorem for different size samples from `x`, where the sample size $n$ is an argument to be passed in the function.

# Part 2: Sampling distributions

(c) The *Central Limit Theorem*, which we will come back to later in this module, states that – regardless of the distribution of the population – the distribution of the sample mean will be Normal, with mean $\mu$ and variance $\sigma^2/n$, where

- $\mu$ is the population mean
- $\sigma^2$ is the population variance
- $n$ is the sample size (e.g. $n = 30$ in part (b))

Suppose your vector `x` contains the population. Can you verify the Central Limit Theorem for the distribution of your sample means obtained in (b)(iii)?

(d) Write a function that explores the Central Limit Theorem for different size samples from `x`, where the sample size $n$ is an argument to be passed in the function.

# Part 2: Sampling distributions

(c) The *Central Limit Theorem*, which we will come back to later in this module, states that – regardless of the distribution of the population – the distribution of the sample mean will be Normal, with mean $\mu$ and variance $\sigma^2/n$, where

- $\mu$ is the population mean
- $\sigma^2$ is the population variance
- $n$ is the sample size (e.g. $n = 30$ in part (b))

Suppose your vector `x` contains the population. Can you verify the Central Limit Theorem for the distribution of your sample means obtained in (b)(iii)?

(d) Write a function that explores the Central Limit Theorem for different size samples from `x`, where the sample size $n$ is an argument to be passed in the function.

# Part 3: Monte Carlo integration

Suppose the random variable $Z$ follows a **standard Normal distribution**, that is, $Z \sim N(0, 1)$.

Then $Z$ has probability density function given by

$$f_Z(z) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}z^2}, \quad z \in \mathcal{R}.$$

Commit the following code to RStudio, and make sure you understand what's going on.

```
z = seq(-4, 4, 0.001)
d = 1/(sqrt(2*pi))*exp(-(z^2)/2)
plot(z, d, xlab = "z", ylab = "density", type = "l")
```

# Part 3: Monte Carlo integration

Now suppose we want to find $\Pr(-2 < Z < 1)$.

The old-school way of doing this might be to annotate a diagram such as that produced on the previous slide, and then use **statistical tables**!

We're not going to do this. We're going to use *Monte Carlo integration*!

# Part 3: Monte Carlo integration

Consider throwing random points at the plot you produced earlier, across a "simulation grid" that covers the area you're interested in.

First, draw the "simulation grid":

```
# Draw the simulation grid on your plot
# Lower bound for Pr(-2 < Z < 1)
abline(v = -2, lty = 2, col = "red")

# Upper bound for Pr(-2 < Z < 1)
abline(v = 1, lty = 2, col = "red")

# Covers the top of the Normal PDF
abline(h = 0.4, lty = 2, col = "red")
```

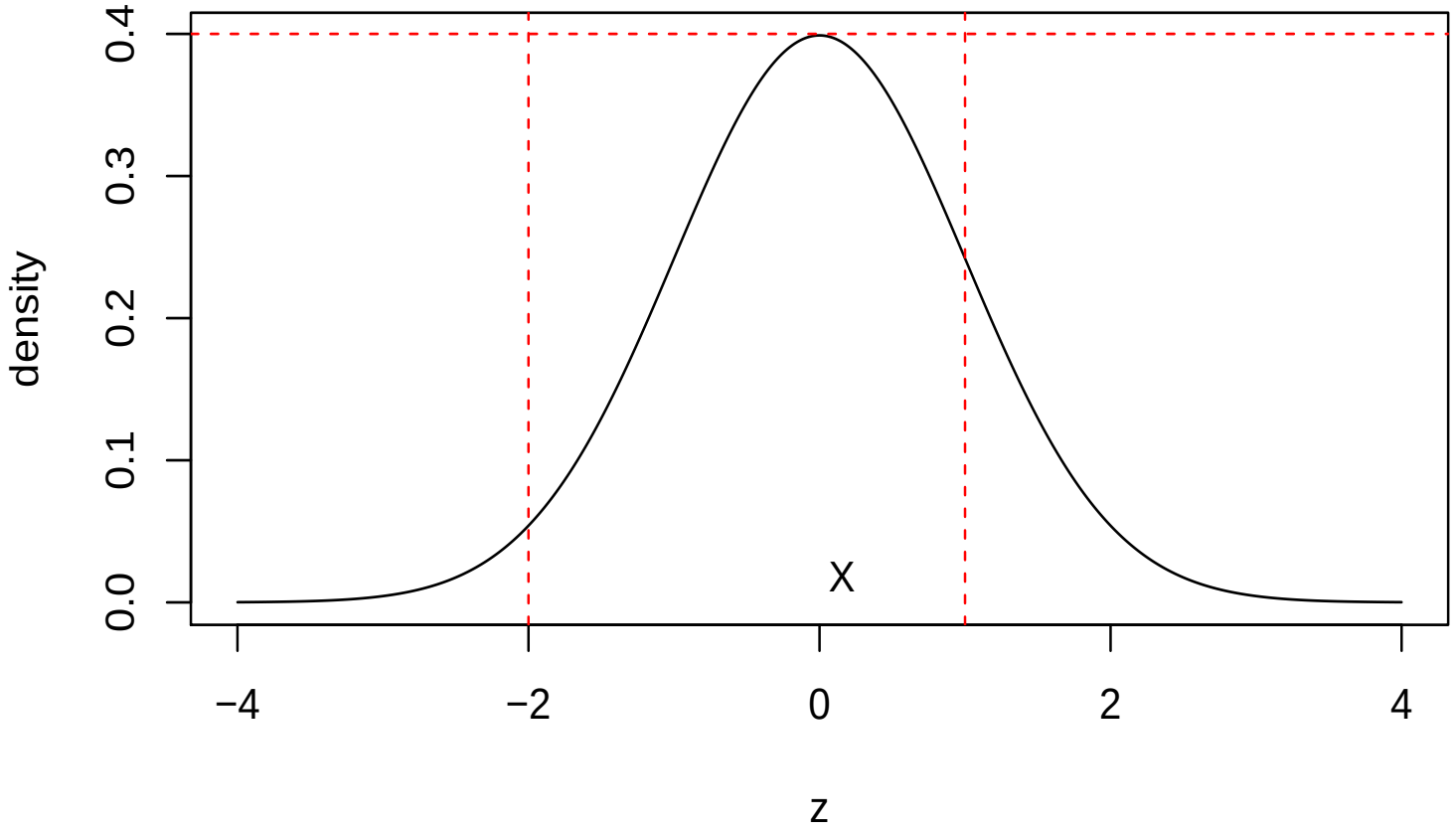
# Part 3: Monte Carlo integration

Now throw random points at the grid, repeatedly – and watch how your plot updates.

```
# Generate a sequence for the x co-ordinate (between -2 and 1)
xseq = seq(-2, 1, 0.0001)

# Generate sequence for the y co-ordinate (between 0 and 0.4)
yseq = seq(0, 0.4, 0.0001)
```

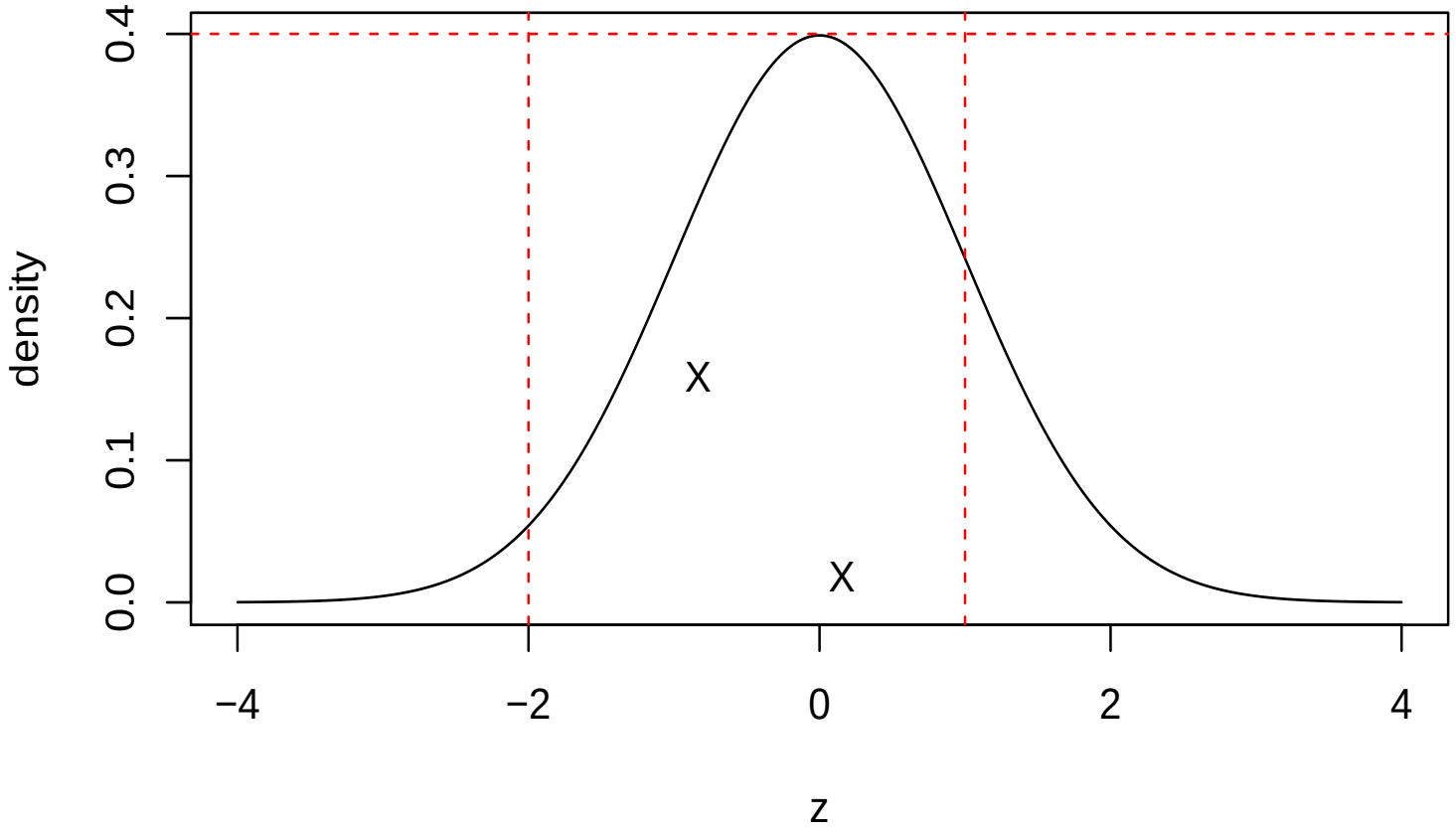
# Part 3: Monte Carlo integration

```r
# Throw your first point at the grid!
lines(sample(xseq,1),sample(yseq,1),type="p",pch="X")
```



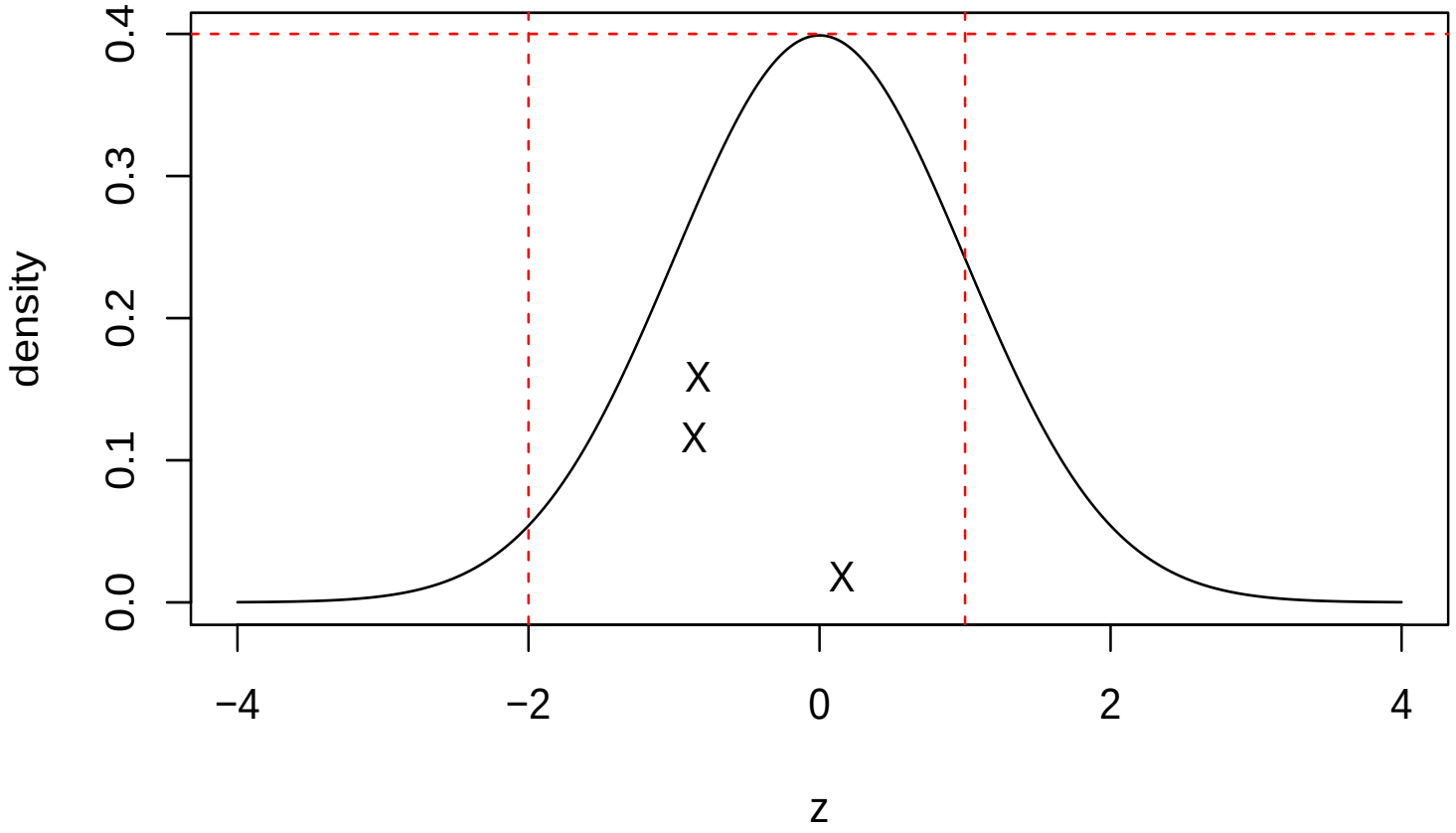
# Part 3: Monte Carlo integration

```r
# And another!
lines(sample(xseq,1),sample(yseq,1),type="p",pch="X")
```



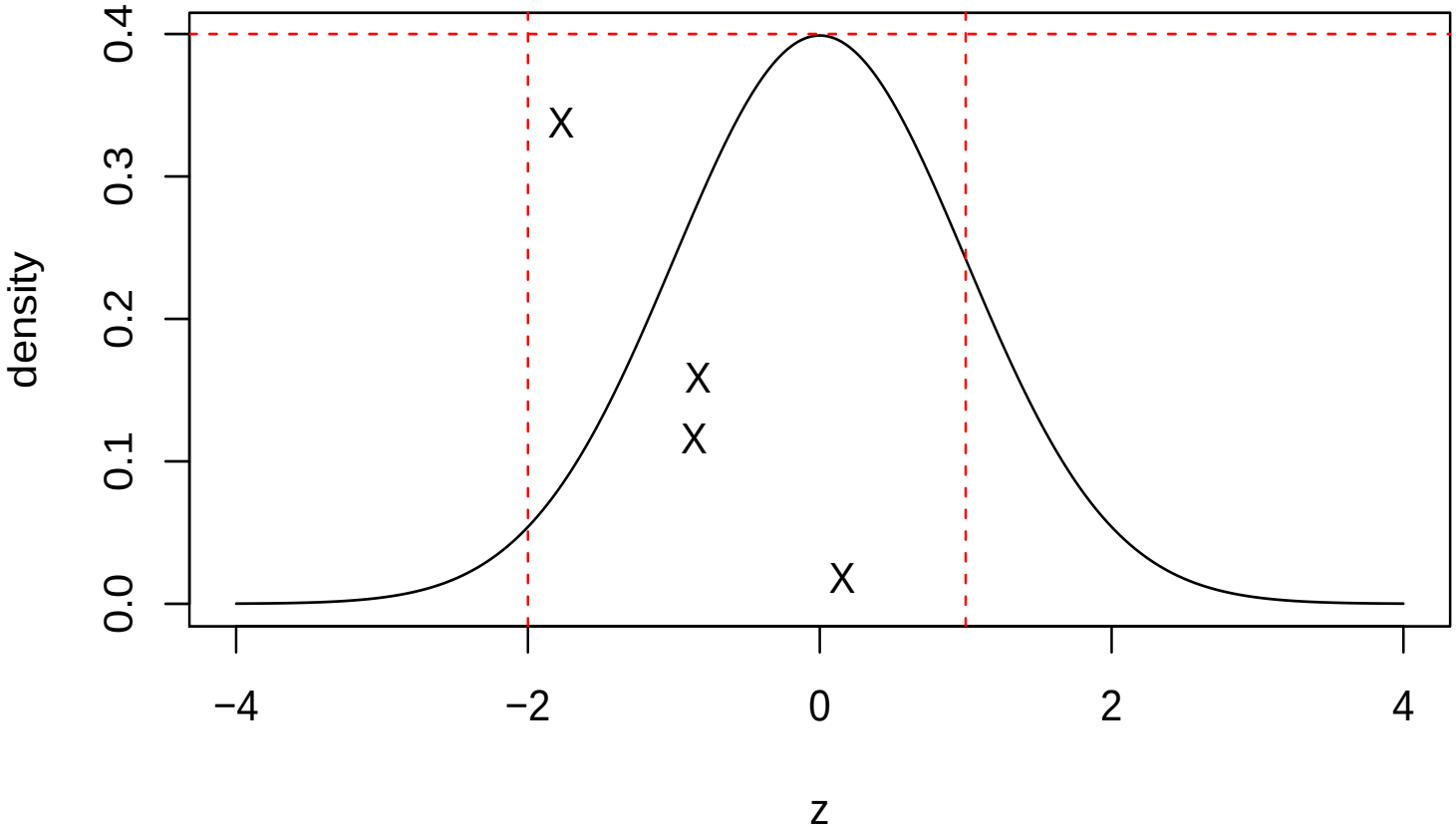
# Part 3: Monte Carlo integration

```r
# Keep going...
lines(sample(xseq,1),sample(yseq,1),type="p",pch="X")
```



# Part 3: Monte Carlo integration

```
# One more for now...!
lines(sample(xseq,1),sample(yseq,1),type="p",pch="X")
```



# Part 3: Monte Carlo integration

**Observations**

- As you throw random points at the grid — some will lie under the curve, some will lie above the curve
- What's the probability that a point will lie under the curve (call this $A$)?

$$\Pr(A) = \frac{\text{Area under the curve}}{\text{Area of simulation grid}}.$$

- Therefore, by rearranging, we can see that

$$\begin{aligned} \text{Area under the curve} &= \Pr(A) \times \text{Area of simulation grid} \\ \Pr(-2 < Z < 1) &\approx \frac{\#\text{points under the curve}}{\#\text{points simulated}} \times 1.2. \end{aligned}$$

# Part 3: Monte Carlo integration

**Observations**

- As you throw random points at the grid — some will lie under the curve, some will lie above the curve
- What's the probability that a point will lie under the curve (call this $A$)?

$$\Pr(A) = \frac{\text{Area under the curve}}{\text{Area of simulation grid}}.$$

- Therefore, by rearranging, we can see that

$$\begin{aligned} \text{Area under the curve} &= \Pr(A) \times \text{Area of simulation grid} \\ \Pr(-2 < Z < 1) &\approx \frac{\#\text{points under the curve}}{\#\text{points simulated}} \times 1.2. \end{aligned}$$

# Part 3: Monte Carlo integration

**Observations**

- As you throw random points at the grid — some will lie under the curve, some will lie above the curve
- What's the probability that a point will lie under the curve (call this $A$)?

$$\Pr(A) = \frac{\text{Area under the curve}}{\text{Area of simulation grid}}.$$

- Therefore, by rearranging, we can see that

$$\begin{aligned} \text{Area under the curve} &= \Pr(A) \times \text{Area of simulation grid} \\ \Pr(-2 < Z < 1) &\approx \frac{\#\text{points under the curve}}{\#\text{points simulated}} \times 1.2. \end{aligned}$$

# Part 3: Monte Carlo integration

**Observations**

- As you throw random points at the grid — some will lie under the curve, some will lie above the curve
- What's the probability that a point will lie under the curve (call this $A$)?

$$\Pr(A) = \frac{\text{Area under the curve}}{\text{Area of simulation grid}}.$$

- Therefore, by rearranging, we can see that

$$\begin{aligned}\text{Area under the curve} &= \Pr(A) \times \text{Area of simulation grid} \\ \Pr(-2 < Z < 1) &\approx \frac{\#\text{points under the curve}}{\#\text{points simulated}} \times 1.2.\end{aligned}$$

# Part 3: Monte Carlo integration

**Observations**

- As you throw random points at the grid — some will lie under the curve, some will lie above the curve
- What's the probability that a point will lie under the curve (call this $A$)?

$$\Pr(A) = \frac{\text{Area under the curve}}{\text{Area of simulation grid}}.$$

- Therefore, by rearranging, we can see that

$$\text{Area under the curve} = \Pr(A) \times \text{Area of simulation grid}$$

$$\Pr(-2 < Z < 1) \approx \frac{\#\text{points under the curve}}{\#\text{points simulated}} \times 1.2.$$

# Part 3: Monte Carlo integration

**Observations**

- As you throw random points at the grid — some will lie under the curve, some will lie above the curve
- What's the probability that a point will lie under the curve (call this $A$)?

$$\Pr(A) = \frac{\text{Area under the curve}}{\text{Area of simulation grid}}.$$

- Therefore, by rearranging, we can see that

$$\begin{aligned} \text{Area under the curve} &= \Pr(A) \times \text{Area of simulation grid} \\ \Pr(-2 < Z < 1) &\approx \frac{\#\text{points under the curve}}{\#\text{points simulated}} \times 1.2. \end{aligned}$$

# Part 3: Monte Carlo integration

So, very crudely,

$$\Pr(-2 < Z < 1) \approx \frac{3}{4} \times 1.2 = 0.9,$$

according to the points on my plot a few slides ago.

- This approximiation should improve as we throw more and more points at the grid
- We can do this easily – many times – by using a **for loop**
- We can check our answers using tables (or even using one of R's built-in functions)
- Generally, such **Monte Carlo integration** is really useful if our integration problem is too hard to do by hand

# Part 3: Monte Carlo integration

So, very crudely,

$$\Pr(-2 < Z < 1) \approx \frac{3}{4} \times 1.2 = 0.9,$$

according to the points on my plot a few slides ago.

- This approximiation should improve as we throw more and more points at the grid
- We can do this easily – many times – by using a **for loop**
- We can check our answers using tables (or even using one of R's built-in functions)
- Generally, such **Monte Carlo integration** is really useful if our integration problem is too hard to do by hand

# Part 3: Monte Carlo integration

So, very crudely,

$$\Pr(-2 < Z < 1) \approx \frac{3}{4} \times 1.2 = 0.9,$$

according to the points on my plot a few slides ago.

- This approximiation should improve as we throw more and more points at the grid
- We can do this easily – many times – by using a **for loop**
- We can check our answers using tables (or even using one of R's built-in functions)
- Generally, such **Monte Carlo integration** is really useful if our integration problem is too hard to do by hand

# Part 3: Monte Carlo integration

So, very crudely,

$$\Pr(-2 < Z < 1) \approx \frac{3}{4} \times 1.2 = 0.9,$$

according to the points on my plot a few slides ago.

- This approximiation should improve as we throw more and more points at the grid
- We can do this easily – many times – by using a **for loop**
- We can check our answers using tables (or even using one of R's built-in functions)
- Generally, such **Monte Carlo integration** is really useful if our integration problem is too hard to do by hand

# Part 3: Monte Carlo integration

So, very crudely,

$$\Pr(-2 < Z < 1) \approx \frac{3}{4} \times 1.2 = 0.9,$$

according to the points on my plot a few slides ago.

- This approximiation should improve as we throw more and more points at the grid
- We can do this easily – many times – by using a **for loop**
- We can check our answers using tables (or even using one of R's built-in functions)
- Generally, such **Monte Carlo integration** is really useful if our integration problem is too hard to do by hand

# Part 3: Monte Carlo integration

So, very crudely,

$$\Pr(-2 < Z < 1) \approx \frac{3}{4} \times 1.2 = 0.9,$$

according to the points on my plot a few slides ago.

- This approximiation should improve as we throw more and more points at the grid
- We can do this easily – many times – by using a **for loop**
- We can check our answers using tables (or even using one of R's built-in functions)
- Generally, such **Monte Carlo integration** is really useful if our integration problem is too hard to do by hand

# Part 3: Monte Carlo integration

```
MCintegrate = function(N){
  no_of_hits = 0
  for(i in 1:N){
    # Generate the x/y co-ordinates in a better way:
    x = runif(1, -2, 1)
    y = runif(1, 0, 0.4)

    # Evaluate the Normal density at x:
    d = 1/(sqrt(2*pi))*exp(-(x^2)/2)
    if(y<d){        # Is the point under the curve?
      no_of_hits = no_of_hits+1
    }
  }
  P = no_of_hits/N                          # Find P
  area_under_curve = P*(3*0.4)              # Find the area:
  return(area_under_curve)}
```

# Part 3: Monte Carlo integration

(a) Run the function `MCintegrate` using `N` $= 10$, 100, 1000 and 10,000.

(b) Can you generalise the function so that it can be used to find probabilities from the standard Normal distribution between *any* two points (as opposed to just between –2 and 1)?

(c) Explore the R's built-in function `pnorm` to evaluate the integral exactly, and compare to your answers in part (a).