Statistical Computing with R

Lecture 9: general programming tips; tracking execution time; making your code faster; probability theory in R

Mirko Signorelli
: mirkosignorelli.github.io
: statcompr [at] gmail.com

Mathematical Institute
Leiden University

Master in Statistics and Data Science (2023-2024)

Universiteit Leiden

Announcements

- Assignments:
  1. feedback A1 published through Brightspace
  2. A2 published last week, deadline: this Sunday
  3. A3 will be published after lecture 11 (tentative: Nov. 24)
- Reminders:
  1. question hours: see Brightspace for dates and sign-up form
  2. course email: statcompr[at]gmail.com

Controlling output width in R Markdown pdfs

- Problem: when compiling a pdf, `R` console output may get printed out of the document margins. How can we prevent that?
- Solution: use `option(width = k)`, where k is a suitable integer (try different values until you get the desired result)
- Simple hack: put this in a chunk at the very beginning of the document (with option `echo = F`), and you will not need to worry about it for the rest of the document
- NB: this works in almost all situations, but there may be specific instances where it doesn't (like in slide 25 of L8)
- More about this topic: see Section 5.3 of the R Markdown cookbook

Recap

Lecture 8:

- the `is` and `as` families
- `message( )`, `warning( )` and `stop( )`
- `any( )` and `all( )` (*self-study*)
- dataviz (part 3): comparing groups and visualizing (many) correlations

Today:

- general programming tips
- tracking computing time
- making your code faster
- probability calculus in `R`
- discrete distributions
- continuous distributions

General programming tips

Tracking computing time

Probability calculus in R

Discrete distributions

Continuous distributions

General programming tips

- Programming is "less standardized" than math: different programmers often do things differently...
- and in general: there's nothing wrong about it! (Remember what we said about "diversity & programming"?)

However:

- it is possible to draw some general recommendations / guidelines / tips about programming

Three major goals:

1. code neatness, so that you and others can more easily understand it
2. reproducibility = every time you rerun the code, you get the same results
3. computationally efficient solutions (significantly lower computing time / less memory usage)

1: don't edit code in the console

Tip #1: edit code in an R script rather than in the console

- Write your code in R scripts[1]
- If you get it wrong, edit the code inside the script, not in the console
- Run the script line by line to debug it
  1. Check how things change after every step (= line of code)
  2. Try to understand where the problem arises, and why is that happening
  3. If the task coded is very complex: try to use toy examples to check what is going on
- At the end, you will have a script that executes all your calculations correctly and in the desired order! ☺

[1] or, if you prefer, in R Markdown. But if you use R Markdown, knit the file regularly to avoid last minute problems. Don't use R Markdown as if it was an R script!

2. Make use of comments

Tip #2: use # to document your code inside an R script

- Whatever comes after # is ignored by R
- Use #s to add information about what the code is doing!
- You may add a title and a short description to any function you create
- Use comments to explain what "obscure" lines of code are doing!

⚠ When using R Markdown, well-written explanations placed outside of code chunks are usually preferable to long comments inside a chunk! ⚠

3. Use spaces and indentation

Tip #3: make your code neater by using:

- space and line breaks to make code easier to read
- indentation to better display nested code
- consistent and informative variable names

See example in the next slides!

Example

- Example of badly organized code[2]:

```
total.interest1=function(depo) {
if (depo<=3000) { interest=0.01
} else if (depo<=10000) { interest=0.013
} else {interest=0.015
}
depo*(1+interest)^3-depo
}
```

[2] code used during lecture 4

Example (cont'd)

- Improved code that implements tip # 3:

```
total.interest2 = function(depo) {
  if (depo <= 3000) {
    interest = 0.01
  }
  else if (depo <= 10000) {
    interest = 0.013
  }
  else {
    interest = 0.015
  }
  depo*(1 + interest)^3 - depo
}
```

NB: here { and } are redundant and could be dropped, but I included them to illustrate the concept of indentation more extensively

Example (cont'd)

- The 2 functions yield the same output:

```
total.interest1(1500); total.interest2(1500)
```

```
## [1] 45.4515
## [1] 45.4515
```

- ...but `total.interest2( )` is much more readable!

Making your code faster

- When programming, different solutions to a problem are often possible
- Not all solutions are equivalent. Faster (and possibly less memory-intensive) solutions should be preferred, especially when implementing complex algorithms
- A few guidelines:
  1. R is designed to work with vectors, so operations on whole vectors are usually faster than element-wise operations
  2. preallocating memory for an object is (usually) more efficient than expanding the size of the object several times
  3. try to avoid nested loops, which look like this:

```
for (i in 1:1000) {
  for (j in 1:400) {
    ...
  }
}
```

Time tracking

- Several ways to track time in R
- Most basic relies on `Sys.time()`:

```
t1 = Sys.time()
x = rnorm(1000)
t2 = Sys.time()
# time difference:
t2 - t1
```

```
## Time difference of 0.02382898 secs
```

The rbenchmark package

- A more systematic and organized way to compare computing time is through the `benchmark` function from the package `rbenchmark`:

```
library(rbenchmark)
```

```
set.seed(9); n = 10000
x = rpois(n, lambda = 4)
y = rnorm(n, mean = 2, sd = 1)
t.eval = benchmark(
  'for loop sum' = {
    z = c()
    for (i in 1:n) {
      z = c(z, x[i] + y[i])
    }
  },
  'vectorized sum' = {
    z = x + y
  },
  replications = 100 # repeat the evaluation 100 times
)
```

The rbenchmark package (cont'd)

```
t.eval
```

```
##              test replications elapsed relative
## 1   for loop sum          100  11.103   5551.5
## 2 vectorized sum          100   0.002      1.0
##   user.self sys.self user.child sys.child
## 1     8.430    2.665          0         0
## 2     0.002    0.000          0         0
```

- Several ways to measure time. The one relevant for you is the `elapsed` time, which measures the total time needed to execute each expression k times (as specified through the `replications` argument)
- `relative` column shows ratios of the elapsed computing times (default behaviour)
- More `replications` = more accurate estimates of computing times (but you'll need to wait longer!)

Why is there such a big difference?

The first expression is 5551.5 slower than the second. Why?

1. element-wise sum `x[i] + y[i]` inside the `for` loop slower than vectorized sum `x+y` ⇒ if possible, use vectorized operations instead of looping
2. size of `z` is augmented at every iteration of the `for` loop, whereas it is created in one go in the second expression ⇒ pre-allocate memory beforehand, rather than asking for extra memory at every iteration!

Your turn

Exercises

Consider the previous example, where we used two expressions - one based on a `for` loop, and the other on a vectorised sum.

1. Rewrite the `for` loop solution by pre-allocating `z` before the `for` loop
2. Compare this alternative implementation to the previous two: is it faster? Why?

Object preallocation: example

```
t.eval2 = benchmark(
  'for without preallocation' = {
    z = c()
    for (i in 1:n) {
      z = c(z, x[i] + y[i])
    }
  },
  'for with preallocation' = {
    z = rep(NA, n)
    for (i in 1:n) {
      z[i] =  x[i] + y[i]
    }
  },
  'vectorized sum' = {
    z = x + y
  },
  replications = 100
)
```

Object preallocation: example (cont'd)

```
t.eval2
```

```
##                          test replications elapsed
## 2    for with preallocation          100   0.212
## 1 for without preallocation          100  16.557
## 3            vectorized sum          100   0.003
##   relative user.self sys.self user.child sys.child
## 2   70.667     0.210    0.002          0         0
## 1 5519.000    12.953    3.594          0         0
## 3    1.000     0.002    0.001          0         0
```

Conclusions:

1. Pre-allocation has made the `for` loop considerably faster!

```
t.eval2[2, 'elapsed'] / t.eval2[1, 'elapsed']
```

```
## [1] 78.09906
```

2. Vectorized sum still the fastest in this (basic) example

Two take-home messages

I know that this is getting repetitive, but let me say this one more time:

- when using a loop, ask yourself: do I actually need it, or can I do without it?
- if you decide to / have to use `for` loops, preallocate the objects you are going to use in the loop!

More about programming tips

You can read more about general programming tips in the book of Braun and Murdoch (2021)[3]:

- Section 4.4: "Miscellaneous programming tips"
- Section 4.5: "Some general programming guidelines"
- Section 4.6: "Debugging and maintenance"
- Section 4.7: "Efficient programming"

[3] Braun, W. J., & Murdoch, D. J. (2021). A First Course in Statistical Programming with R. *Cambridge University Press*.

Why are we covering this subject "again"?

- Probability theory and calculus are at the basis of most statistical methods
- A good understanding of probability theory is essential to
  1. understand how a statistical method works
  2. figure out if, and why, something may go wrong when applying a statistical method to a specific dataset / problem
  3. interpret the results obtained from statistical models
- The theoretical part was covered during *Statistics and probability*, and you are expected to be familiar with it by now
- Here I will show you how to perform some basic probability calculus with `R`, highlighting the link with the theory it is based on

Discrete vs continuous random variables

Let Ω denote the support (= set of possible realizations) of a random variable X

- X is called discrete if Ω is either
  1. a finite set (e.g., $\{1, 2, 3, 4, 5\}$)
  2. a countably infinite set (e.g., $\mathbb{N}$)
- If Ω isn't a countable set, X is continuous (e.g., $\mathbb{R}$)

Discrete random variables

If X is discrete, we define:

- its probability mass function (pmf) $f_X(x) = P(X = x)$
- its cumulative distribution function (cdf) $F_X(x) = P(X \leq x)$
- its expected value
$$E(X) = \sum_{x \in \Omega} x f_X(x)$$
- its variance
$$Var(X) = E\left[(X - E(X))^2\right]$$

Properties of the pmf and cdf

To be a pmf, f_X must satisfy:

1. $f_X(x) \geq 0$ for $x \in \Omega$
2. $\sum_{x \in \Omega} f_X(x) = 1$

The cdf F_X is a non-decreasing function for which

$$\lim_{x \to -\infty} F_X(x) = 0 \text{ and } \lim_{x \to \infty} F_X(x) = 1$$

Binomial distribution

- Let $Y \in \{0, 1\}$ denote the outcome of a binary experiment that can result in a success $(y = 1)$ with probability $p \in (0, 1)$, and in a failure $(y = 0)$ with probability $1 - p$
- Y is called Bernoulli random variable
- The binomial distribution counts the number of successes x out of n independent Bernoulli trials:

$$f(x) = \binom{n}{x} p^x (1-p)^{n-x}, \; x = 0, 1, ..., n$$

$$E(X) = np$$

$$Var(X) = np(1-p)$$

Example: binomial distribution

Let $n = 7$ and $p = 0.4$. We can compute f_X using

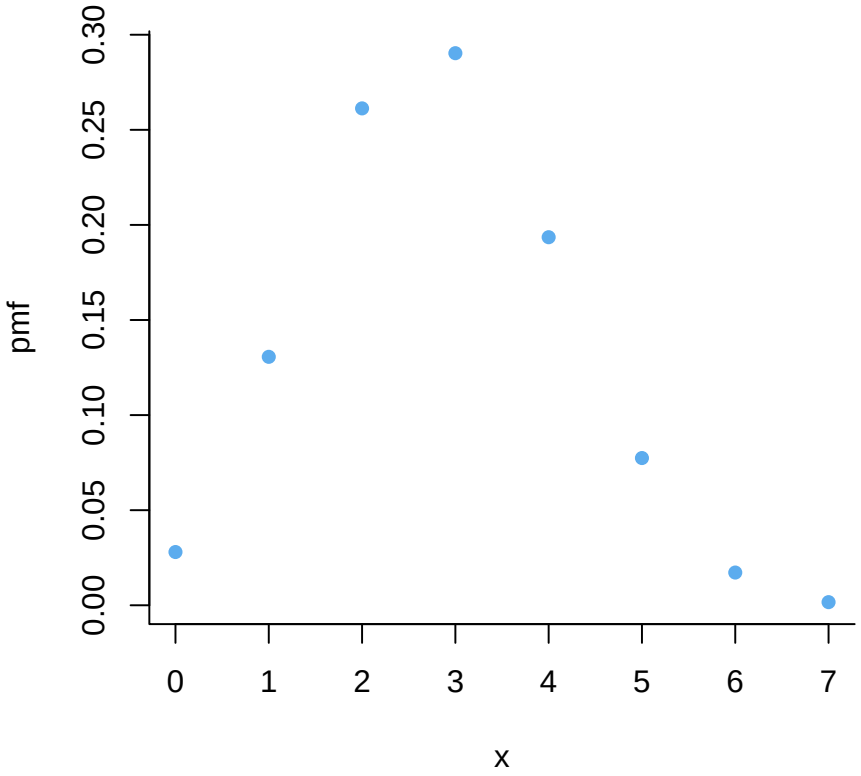
```
n = 7; p = 0.4
omega = 0:n
pmf = dbinom(x = omega, size = n, prob = p)
bin.distr = data.frame(x = omega, pmf)
bin.distr
```

```
##   x       pmf
## 1 0 0.0279936
## 2 1 0.1306368
## 3 2 0.2612736
## 4 3 0.2903040
## 5 4 0.1935360
## 6 5 0.0774144
## 7 6 0.0172032
## 8 7 0.0016384
```

Example: binomial distribution (cont'd)

Here's how the pmf of X looks like:

```
plot(bin.distr, pch = 16, bty = "l", col = 'steelblue2')
```



Example: binomial distribution (cont'd)

To compute the cdf we can either use:

1. `pbinom( )`

```
cdf1 = pbinom(q = omega, size = n, prob = p)
```

2. `cumsum( )` applied to the output or `dbinom( )`

```
cdf2 = cumsum(pmf)
```

Do we get the same result? ✓

```
cdf1; cdf2
```

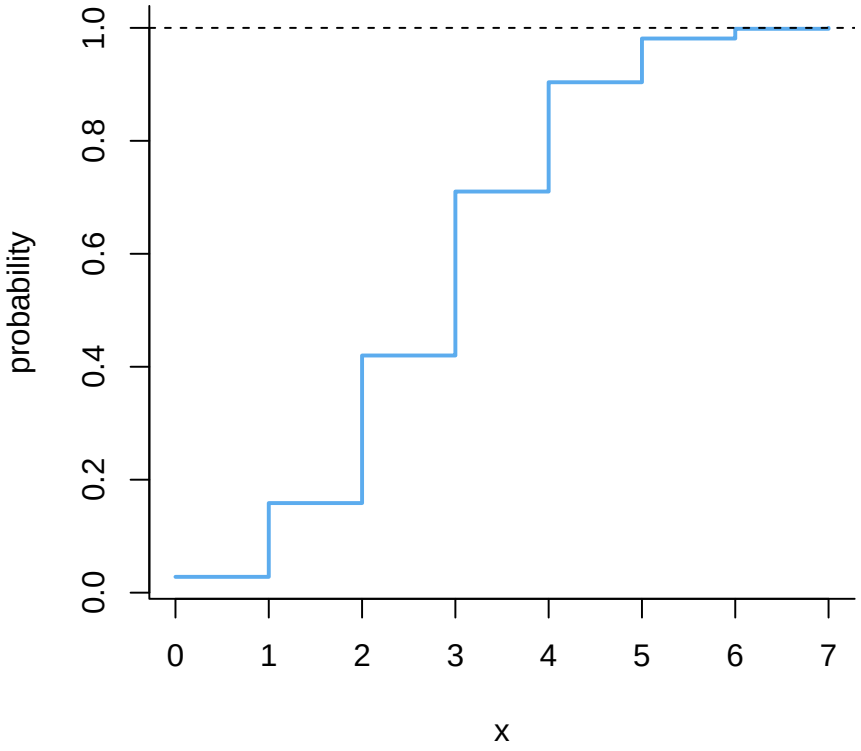
```
## [1] 0.0279936 0.1586304 0.4199040 0.7102080 0.9037440
## [6] 0.9811584 0.9983616 1.0000000
```

```
## [1] 0.0279936 0.1586304 0.4199040 0.7102080 0.9037440
## [6] 0.9811584 0.9983616 1.0000000
```

Example: binomial distribution (cont'd)

Lastly, here's how we can visualize the cdf:

```
plot(omega, cdf1, type = 's', bty = "l",
     xlab = 'x', ylab = 'probability',
     col = 'steelblue2', lwd = 2)
abline(h = 1, lty = 2)
```



Common discrete distributions

Functions	Distribution
`dbinom`, `pbinom`, `rbinom`	binomial
`dgeom`, `pgeom`, `rgeom`	geometric
`dhyper`, `phyper`, `rhyper`	hypergeometric
`dpois`, `ppois`, `rpois`	Poisson
`dnbinom`, `pnbinom`, `rnbinom`	negative binomial

The first letter tells you what a function does:

- `d` computes the pmf / pdf ("density");
- `p` computes the cdf ("probability");
- `r` draws random numbers from this distribution.

⚠ NB: the `d` / `p` names make more sense if you think about continuous random variables! ⚠

Your turn

Exercises

1. Compute the pmf of $X \sim Poi(\lambda = 3)$ for $x = 0, 1, 2, ..., 20$
2. Visualize the pmf

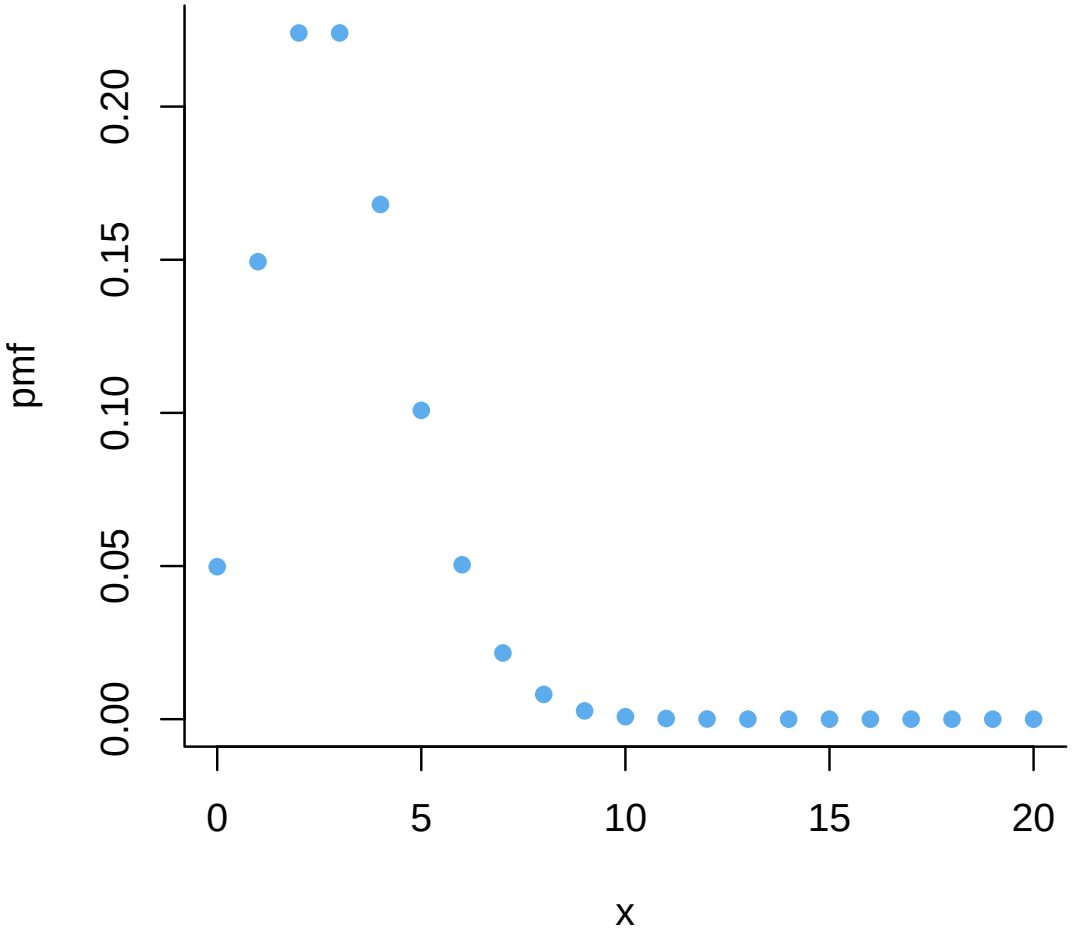
Solution

```r
# Ex. 1
x = 0:20
pmf = dpois(x = x, lambda = 3)
poi.distr = data.frame(x = x, pmf)
head(poi.distr, 10)
```

```
##    x         pmf
## 1  0 0.049787068
## 2  1 0.149361205
## 3  2 0.224041808
## 4  3 0.224041808
## 5  4 0.168031356
## 6  5 0.100818813
## 7  6 0.050409407
## 8  7 0.021604031
## 9  8 0.008101512
## 10 9 0.002700504
```

Solution (cont'd)

```
plot(poi.distr, pch = 16, bty = "l", col = 'steelblue2')
```



Continuous random variables

If X is continuous, we define:

- its probability density function (pdf) $f_X(x)$ as the function for which $P(X \in A) = \int_{X \in A} f_X(t)dt$
- its cumulative distribution function (cdf)

$$F_X(x) = P(X \leq x) = \int_{-\infty}^{x} f_X(t)dt$$

- its expected value

$$E(X) = \int_{-\infty}^{\infty} x f_X(x)dx$$

- its variance

$$Var(X) = E\left[(X - E(X))^2\right]$$

Properties of the pdf and cdf

To be a pdf, f_X must satisfy:

1. $f_X(x) \geq 0$ for $x \in \Omega$
2. $\int_{t \in \Omega} f_X(t)dt = 1$

The cdf F_X is a non-decreasing function for which

$$\lim_{x \to -\infty} F_X(x) = 0 \text{ and } \lim_{x \to \infty} F_X(x) = 1$$

Normal distribution

- A random variable X is said to follow a normal distribution with mean μ and variance σ^2 if

$$f_X(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}, \; x \in (-\infty, \infty)$$

- No closed form expression available for the cdf: use `pnorm( )`!

$$E(X) = \mu, \; Var(X) = \sigma^2$$

- Standardization: $Z = \frac{X-\mu}{\sigma} \sim N(0,1)$ is the standard normal distribution

Example: normal distribution

- `dnorm( )` evaluates the pdf. This is ⚠ NOT a probability! ⚠

```r
dnorm(x = 1, mean = 2, sd = 0.5)
```

```
## [1] 0.1079819
```

```r
dnorm(x = 2, mean = 2, sd = 0.1)
```

```
## [1] 3.989423
```

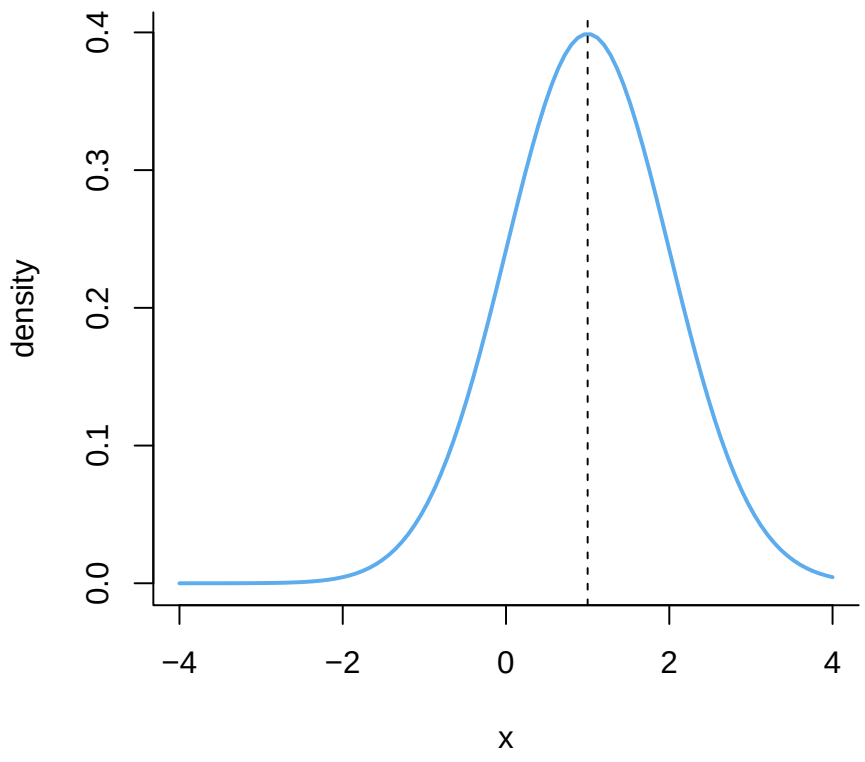
- `pnorm( )` evaluates the cdf:

```r
pnorm(q = 1, mean = 2, sd = 0.5)
```

```
## [1] 0.02275013
```

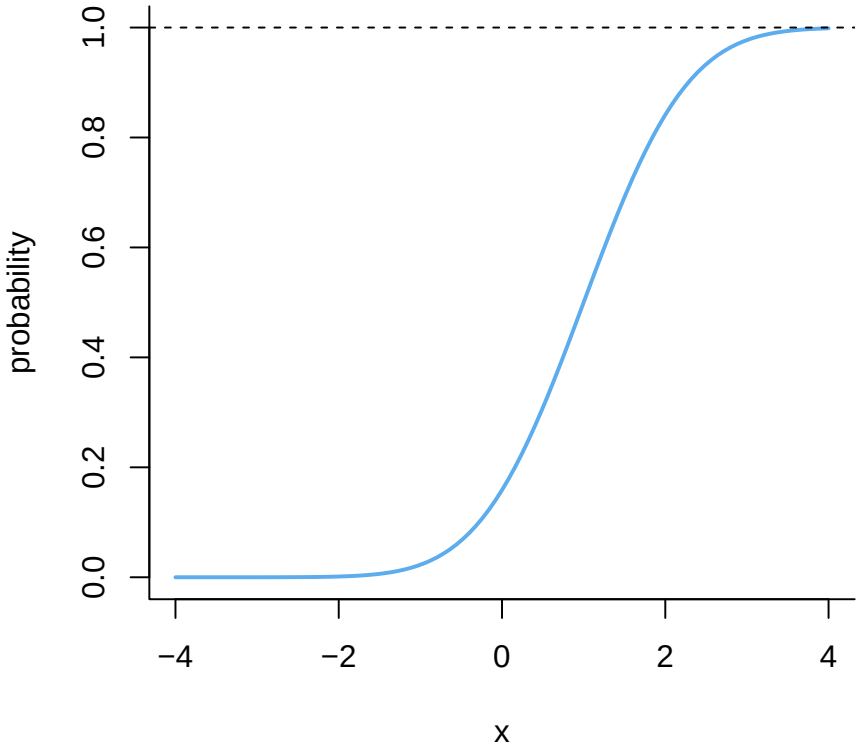
Example: normal distribution (cont'd)

```
curve(dnorm(x, mean = 1, sd = 1), xlim = c(-4, 4),
      bty = 'l', ylab = 'density',
      col = 'steelblue2', lwd = 2)
abline(v = 1, lty = 2)
```



Example: normal distribution (cont'd)

```
curve(pnorm(x, mean = 1, sd = 1), xlim = c(-4, 4),
      bty = 'l', ylab = 'probability',
      col = 'steelblue2', lwd = 2)
abline(h = 1, lty = 2)
```



Common continuous distributions

Functions	Distribution
`dnorm`, `pnorm`, `rnorm`	normal
`dbeta`, `pbeta`, `rbeta`	beta
`dgamma`, `pgamma`, `rgamma`	gamma
`dexp`, `pexp`, `rexp`	exponential
`dchisq`, `pchisq`, `rchisq`	χ^2
`dt`, `pt`, `rt`	Student's T
`df`, `pf`, `rf`	Fisher–Snedecor's F

Useful readings

- Programming tips, debugging and efficient coding: Sections 4.4-4.7 of Braun and Murdoch (2021)
- Review of probability theory: Sections 2.0-2.3 of Rizzo (2019)