Week 8: Audio and Image Processing

MY472: Data for Data Scientists
https://lse-my472.github.io/

Autumn Term 2025

Ryan Hübert
Associate Professor of Methodology

Slides last updated: 19 November 2025

Introduction

The standard social science dataset is tabular data

But lots of interesting “data” in the world isn’t tabular

Today we’ll discuss audio, images and video

We’ll cover:

- How “analog” formats are digitised and stored
- Conceptualising these data types quantitatively
- Ingesting these data types into R
- Doing some simple manipulations/visualisations

Introduction

There is lots of *analysis* you can do with these kinds of data

- Recall: this class isn't about that!
- Some promising applications use machine learning/AI to analyse data at scale (e.g. Francisco Cantú's 2019 article)

This lecture is about giving you vocabulary and core skills to begin working with this kind of data

Warning: some of following R packages aren't well developed

Outline

1. Audio data
2. Visual data
3. Digital photos
4. Video data
5. Base64 encoding

The basics of sound

Human ears are auditory organs that sense **sound** by detecting fluctuations in air pressure

The "data" the brain processes (from each ear) is a one-dimensional time series with wave-like structure—a **waveform**

The link between sound waves and human hearing is complex

- For example: how the brain combines data from each ear into interpret a single "sound"

We'll focus on two key properties: **pitch** and **loudness**

- **Frequency** (**wavelength**) $\rightarrow$ pitch
- **Amplitude** $\rightarrow$ loudness

The basics of sound

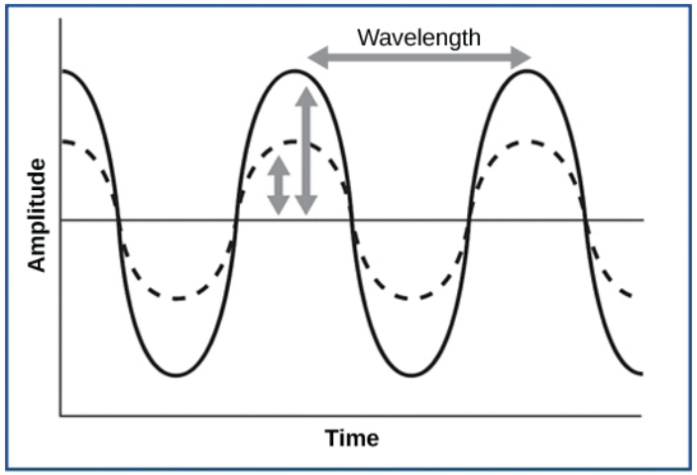


Figure 1. For sound waves, wavelength corresponds to pitch. Amplitude of the wave corresponds to volume. The sound wave shown with a dashed line is softer in volume than the sound wave shown with a solid line. (credit: NIH)

Source: https://books.psychstat.org/rdata/audio-data.html

Sound pitch can be measured using frequency in Hertz (Hz)

- If a sound wave completes one cycle in one second = 1 Hz
- Humans can hear sounds from 20 Hz to 20,000 Hz

The basics of sound

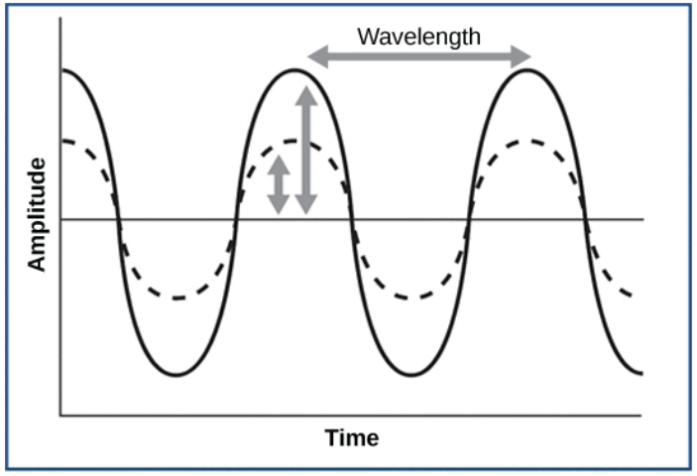


Figure 1. For sound waves, wavelength corresponds to pitch. Amplitude of the wave corresponds to volume. The sound wave shown with a dashed line is softer in volume than the sound wave shown with a solid line. (credit: NIH)

Source: https://books.psychstat.org/rdata/audio-data.html

Loudness can be measured in decibels (dB)

- A conversation is around 60 dB
- Hearing loss can occur with prolonged exposure > 80 dB

Sound digitisation

Sound (as it exists in air) is a time-continuous **analog signal**

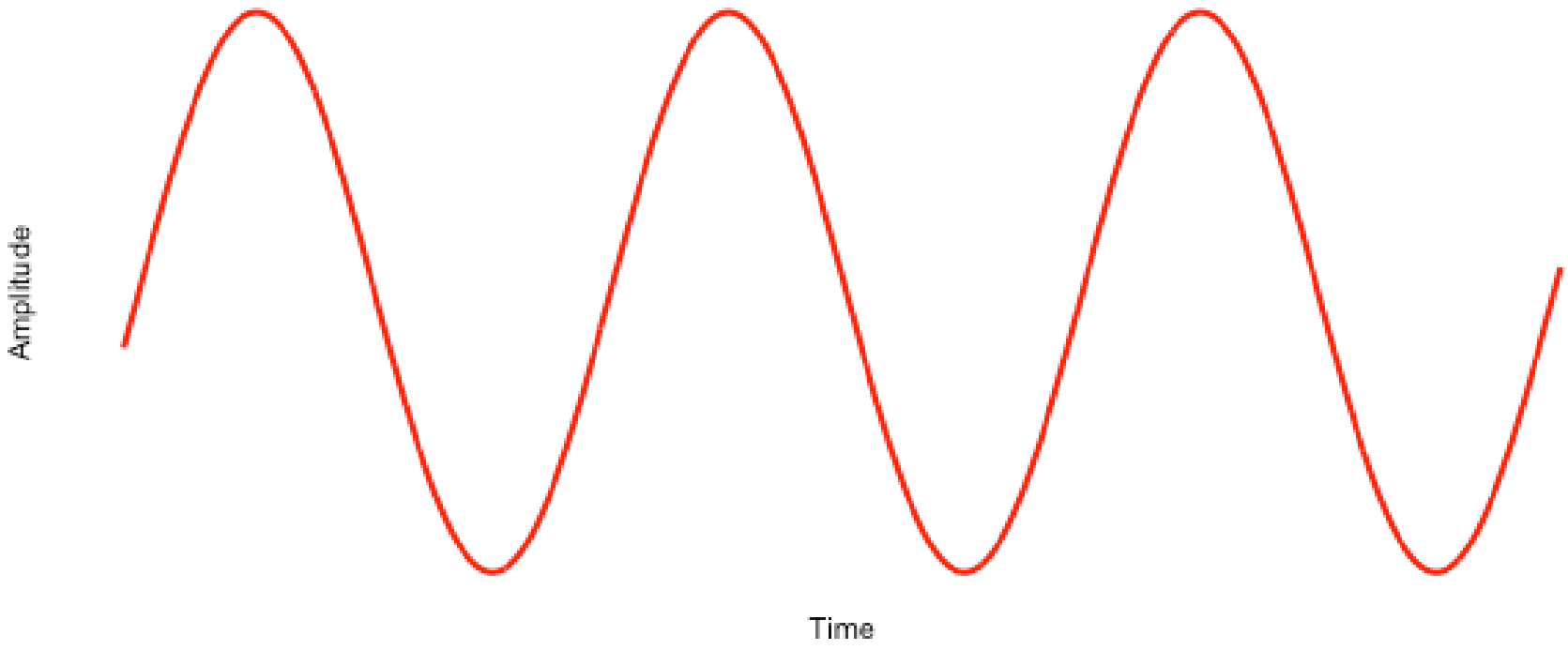
But computer-readable data is represented discretely

- Digitising sound involves discretising it
- This process is known as **pulse-code modulation (PCM)**, which results in a **PCM stream**

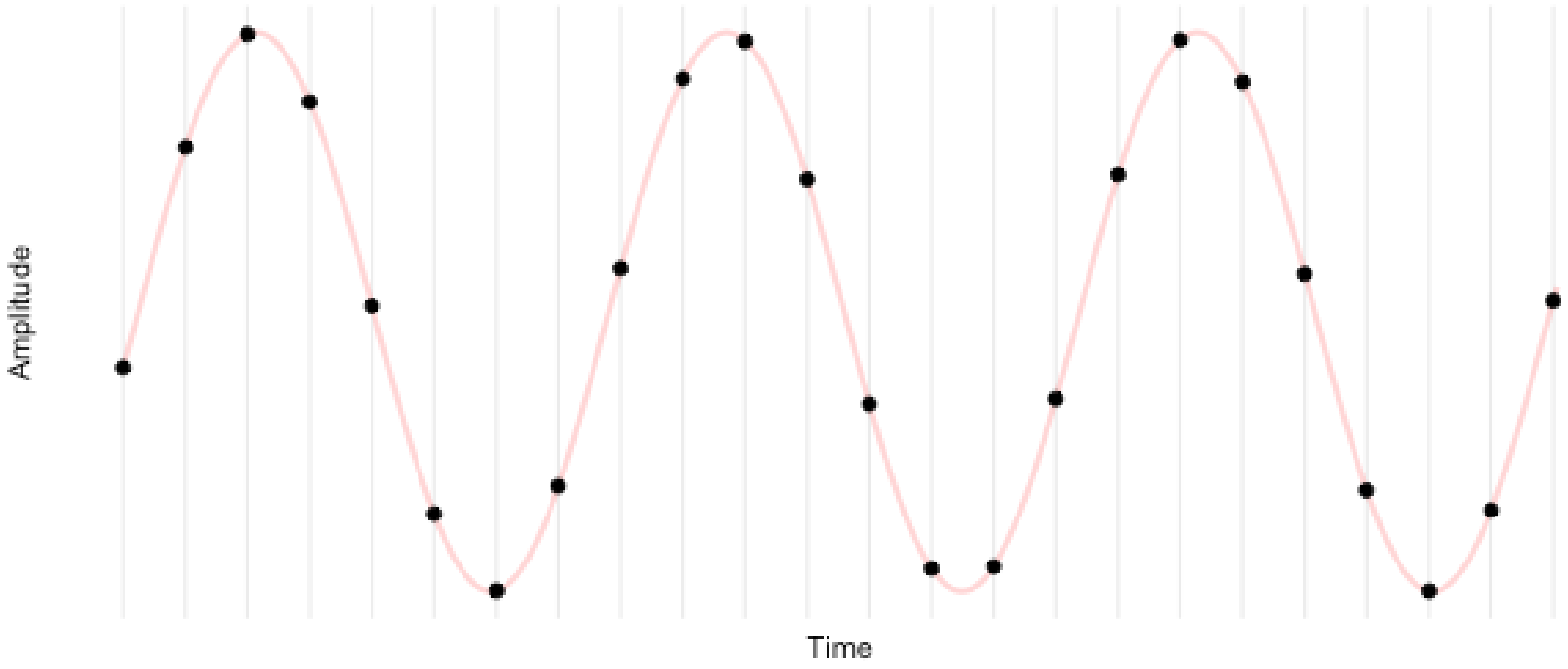
Digitisation happens via **sampling** (not in statistical sense):

- **Sampling frequency**: samples per second, measured in Hz
- **Quantising**: how are amplitude readings "rounded"?
  - Digitised sound is quantised in **bit depth**, so 4-bit digitised sound measures amplitude in $2^4 = 16$ intervals
- Sampling causes information loss and lowers quality

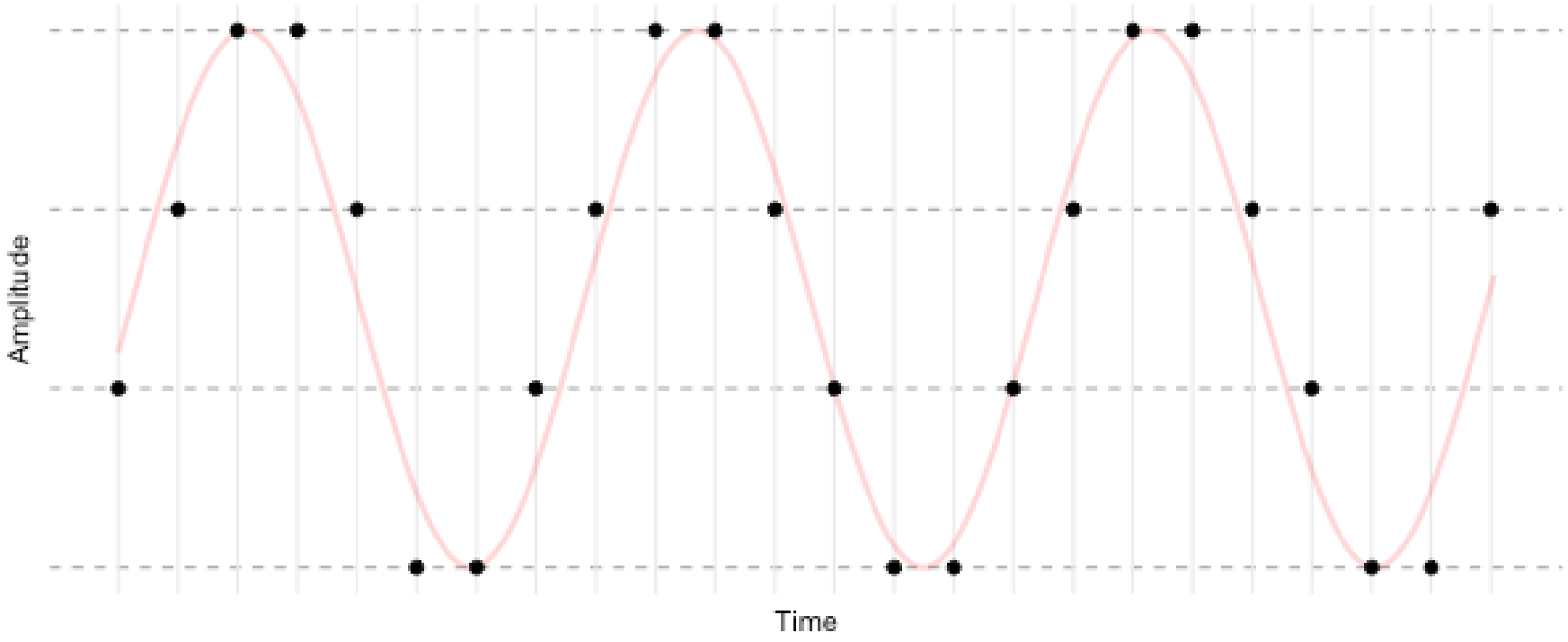
Digital audio sampling



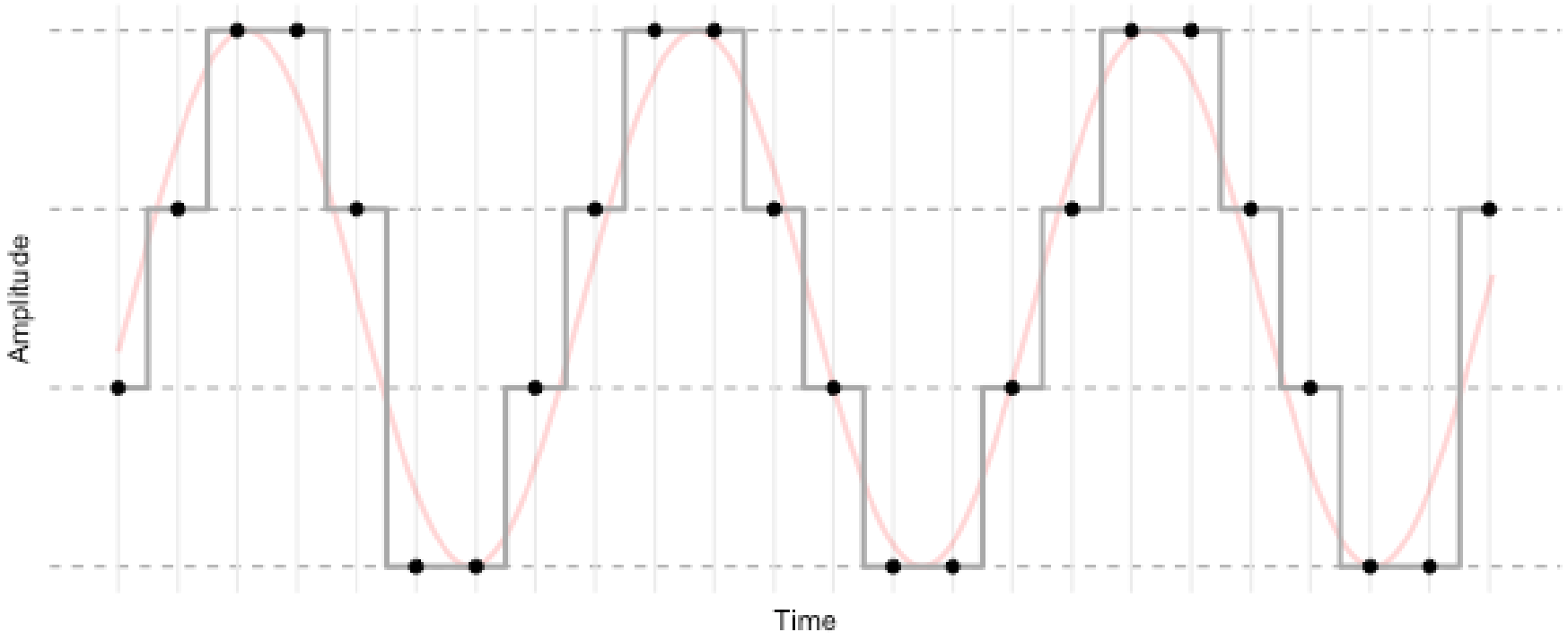
Digital audio sampling



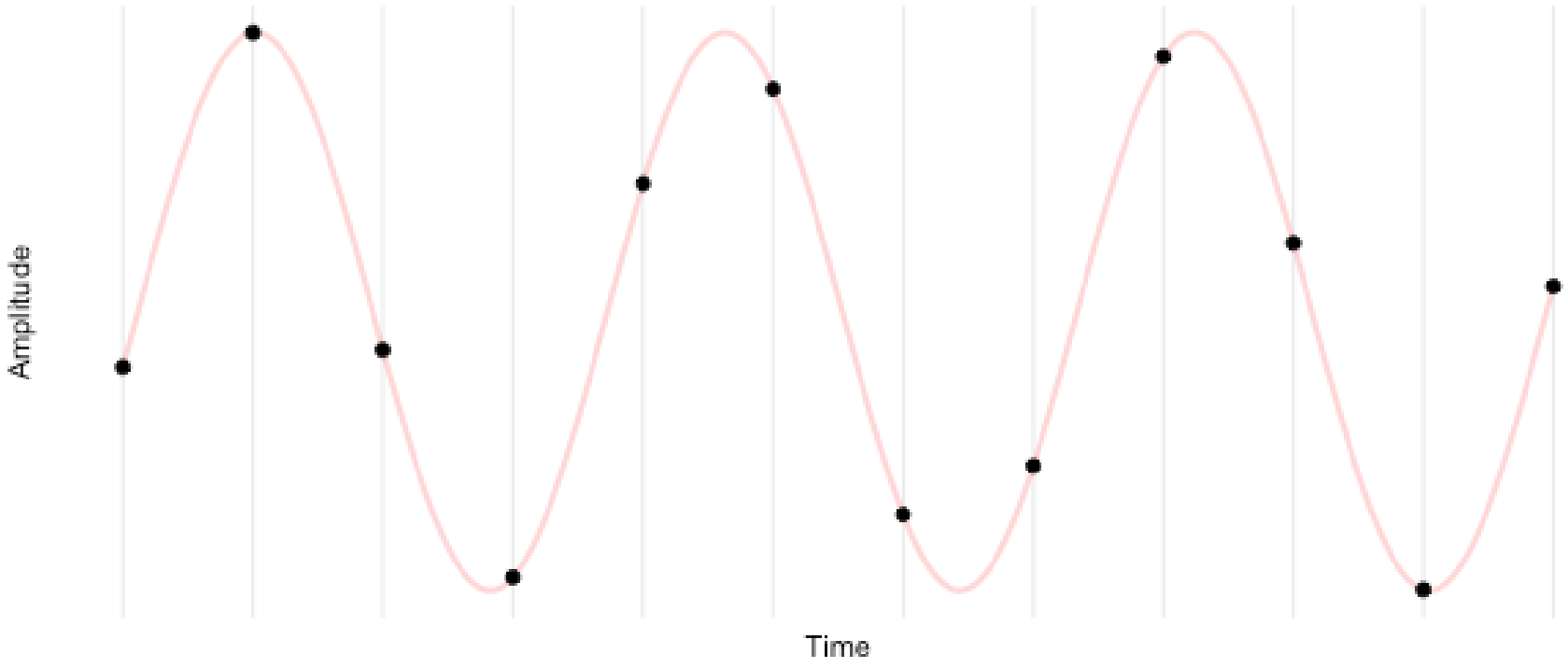
Digital audio sampling



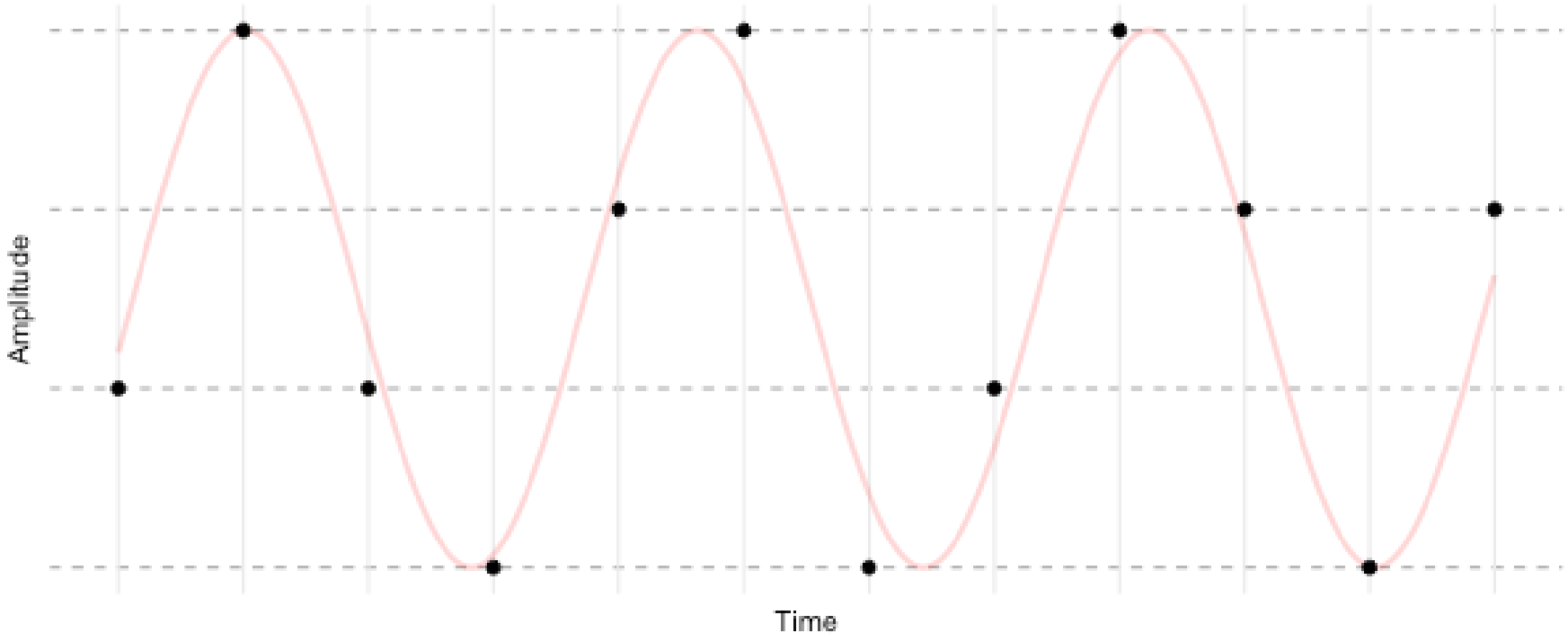
Digital audio sampling



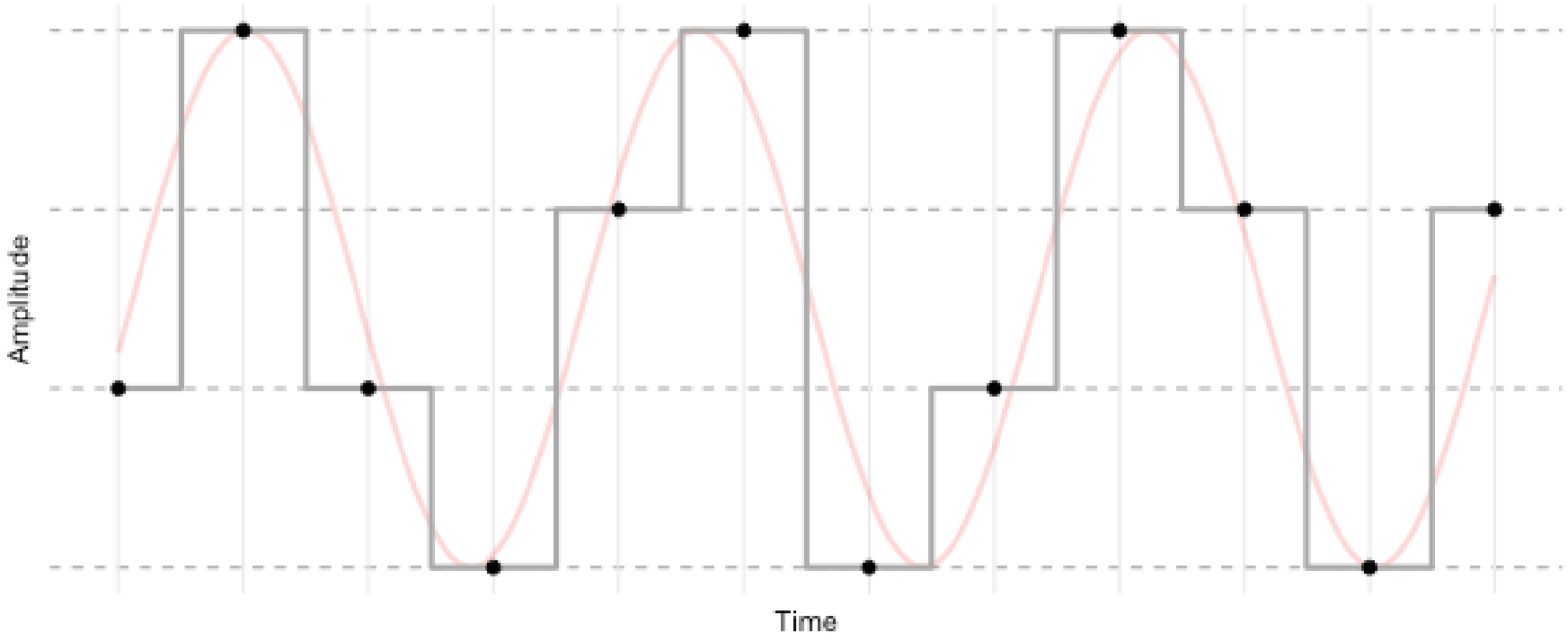
Digital audio sampling



Digital audio sampling



Digital audio sampling



Modern digital audio sampling

"Standard":

- Sample rate of 44,100 Hz (44.1 kHz)
- Bit depth: 16-bit

High resolution (usually for music)

- Apple Music: up to 24-bit/192 kHz (source)
- Spotify: up to 24-bit/44.1 kHz (source)

For audio on video streaming:

- Usually a sampling rate of 48 kHz (source)

For telephones: can use much lower resolution sampling

- Sampling rate of 8 kHz, bit depth of 8-bits (source)

Information loss in digitisation

Digitisation involves “fundamental” information loss

- Analog: infinite data; digital: finite data
- (Side note: brain also samples from analog signal!)

Even after digitisation: often too much data to store/transmit

Digital audio data is often **compressed**: some data is dropped

- **Lossy compression**: drop data humans can’t hear (roughly)
  - Based on psychoacoustic models, depends on audio type
  - Irreversible data loss
- **Lossless compression**: uses prediction models and entropy coding to reduce file size, so no data is lost
  - Useful analogy: UTF-8 is lossless compression of UTF-32

Digital audio encoding

Compression of a PCM stream is known as **encoding**

- **Decoding** reverses the encoding
- Encoding/decoding occur via **codec** software (see huge list)

Common lossless codecs:

- **Apple Lossless Audio Codec (ALAC)**—used by Apple Music
- **Free Lossless Audio Codec (FLAC)**—used by Spotify

Common lossy codecs:

- **MP3** — somewhat outdated now
- **Advanced Audio Coding (AAC)**—used by Apple Music, Spotify web player
- **E-AC3 (Dolby Digital Plus)**—used by streaming video apps

Digital audio file types

Digital audio is typically stored in file types called **containers**

- Container files contain the audio data plus some metadata
- Audio data can be compressed or not
- Many other file types (e.g., video) are also containers

Most common file types for uncompressed data are:

- **Waveform Audio File Format (WAV)**, stored in `.wav` files
- **Audio Interchange File Format (AIFF)**, stored in `.aiff` files

Digital audio file types

For compressed data, common file types are:

- MP3-encoded (lossy) data is usually stored in `.mp3`
- AAC-encoded (lossy) data can be stored in `.m4a`, `.mp4`, `.aac`
- E-AC3 encoded (lossy) data can be stored in `.mp4`
- ALAC-encoded (lossless) data can be stored in `.m4a` or `.mp4`
- FLAC-encoded (lossless) data can be stored in `.flac`

Note: file type $\neq$ encoding!

Audio data in R

For audio data in R: `{tuneR}` and `{seewave}`

- Audio data is a sequence of recorded amplitude readings plus sampling information (sampling frequency and bits)
- In `{tuneR}` data is loaded as a `Wave` object
  - `@samp.rate` gives sampling frequency; `@bits` gives bit depth
  - Access amplitude readings via `@left` and `@right`
  - If mono, there will only be readings in `@left`

Side note: **mono** versus **stereo** refers to how many "channels" are in digitised audio—mono is 1 channel, stereo is 2 channels

- Many stereo speakers/headphones will play mono audio through both speakers

Audio data in R

In seminar exercises: MLK's "I have a dream" speech

- https://ia801605.us.archive.org/25/items/MLKDream/MLKDream.mp3

```
mlk <- readMP3(mlk.file)
print(mlk)
```

```
Wave Object
    Number of Samples:      21684288
    Duration (seconds):     983.41
    Samplingrate (Hertz):   22050
    Channels (Mono/Stereo): Stereo
    PCM (integer format):   TRUE
    Bit (8/16/24/32/64):    16
```

Audio data in R

```
str(mlk)
```

```
Formal class 'Wave' [package "tuneR"] with 6 slots
  ..@ left     : int [1:21684288] 0 0 0 0 0 0 0 0 0 0 ...
  ..@ right    : int [1:21684288] 0 0 0 0 0 0 0 0 0 0 ...
  ..@ stereo   : logi TRUE
  ..@ samp.rate: num 22050
  ..@ bit      : num 16
  ..@ pcm      : logi TRUE
```

```
mlk@left[1000:1010]
```

```
[1] -24 -18  -8  -5 -15 -20 -18 -17 -19 -14  -8
```

```
mlk@right[1000:1010]
```

```
[1] -24 -18  -8  -5 -15 -20 -18 -17 -19 -14  -8
```

Visualising audio data

Since sound is generated by waves, the “data” of sound allows you to (approximately) reconstruct the waves

- Remember: sampling+quantising+compression=info loss!

But we have a dimensionality problem:

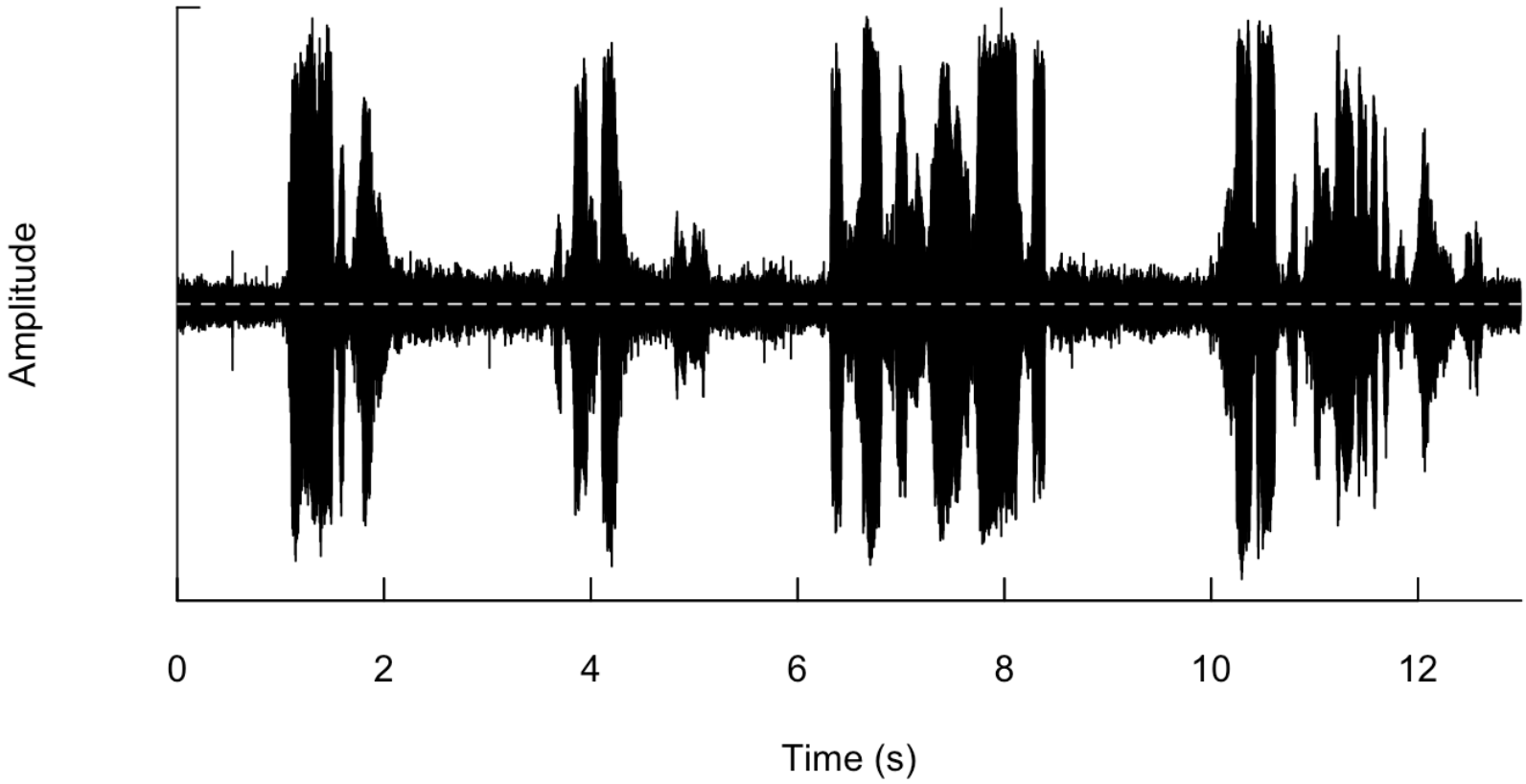
- time
- amplitude
- frequency

Three basic types of visualisations are useful: **oscillogram**, **periodogram** and a **spectrogram**

- Use the `{seewave}` package to visualise

Visualising audio data

1. An **oscillogram** is simply a plot of a digitised sound wave
   - ▷ But: a little difficult to “see” the frequency

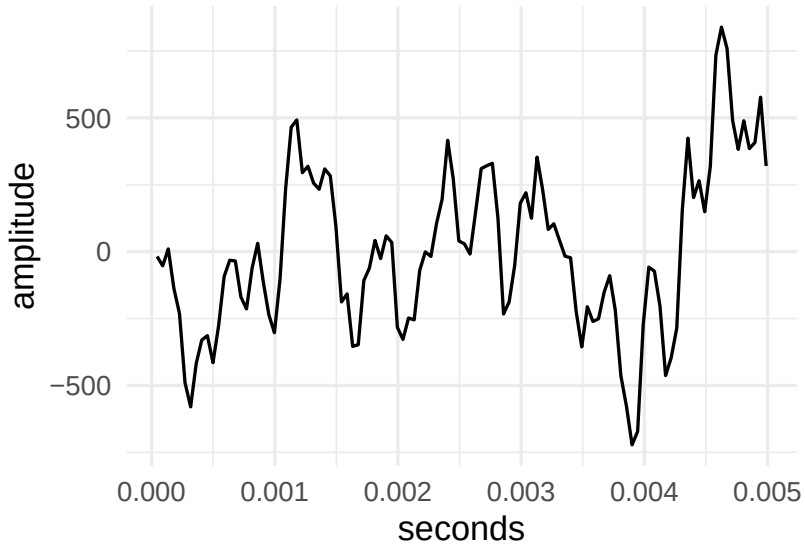


Visualising audio data

You can also plot oscillograms manually using `{ggplot2}`

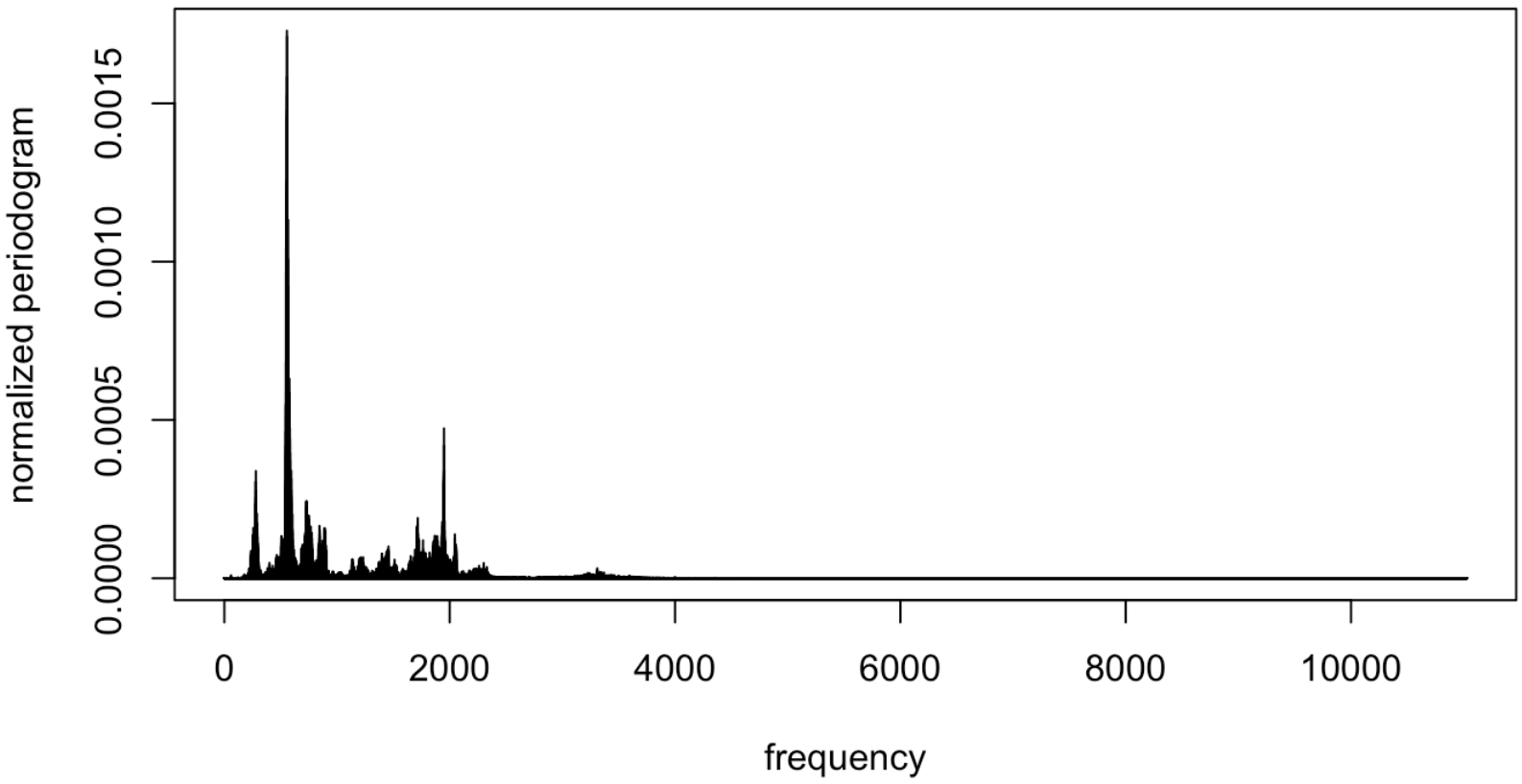
- Below is an example of a (very) small clip 110 samples

```
mlk.samp <- mlk@left[1:110]
tp <- tibble(seconds = (1:length(mlk.samp))/mlk@samp.rate,
             amplitude = mlk.samp)
```



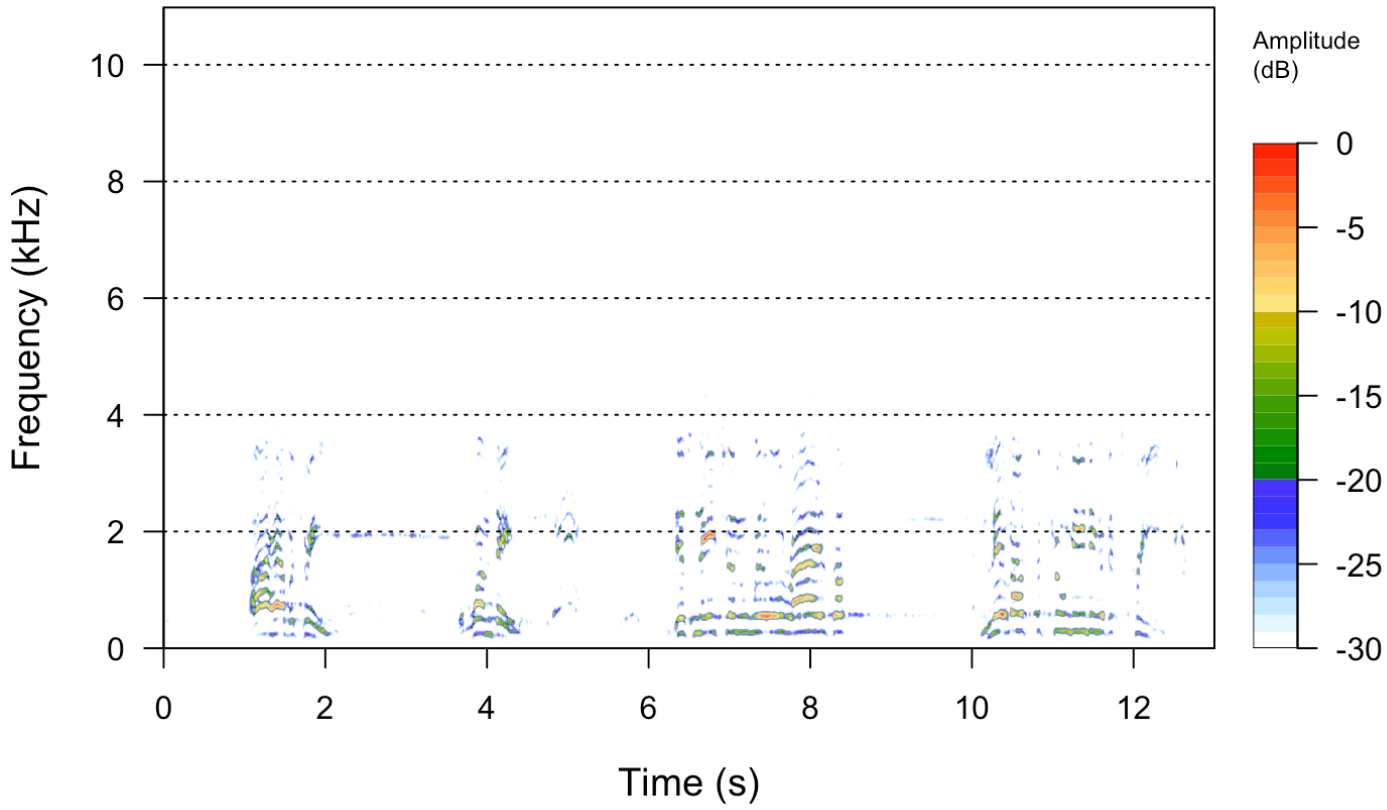
Visualising audio data

2. A **periodogram** shows the "dominant" frequencies in a recording, i.e. frequencies that contribute most to amplitude
   - ▷ But: it doesn't show time



Visualising audio data

3. A **spectrogram** is a “heat map” that depicts variation in amplitude *and* frequency over time



Outline

The basics of vision

Human vision is enabled by “light waves”

- **Frequency** (or **wavelength**) determines **colour**
  - Short wavelengths $\rightarrow$ blue/violet
  - Long wavelengths $\rightarrow$ red
- **Amplitude** (or **intensity**) determines **brightness**
  - Brightness $\approx$ how much light energy hits the retina

Unlike sound, light is **electromagnetic**, not mechanical

- EM waves oscillate too quickly for eyes to detect the waveform
- Retinas only detect light intensity
- Retinas have three types of sensors (“cones”) detecting light intensity across different wavelengths (colours)

Digital images

Digial images are always displayed with **pixels**

- Small, discrete units arranged in a 2D grid displaying visual data, such as colour and brightness
- E.g., screens contain pixels displaying brightness & colour
- Roughly speaking, human eyes also process data in this way

But display $\neq$ storage: need to balance file size against visual fidelity

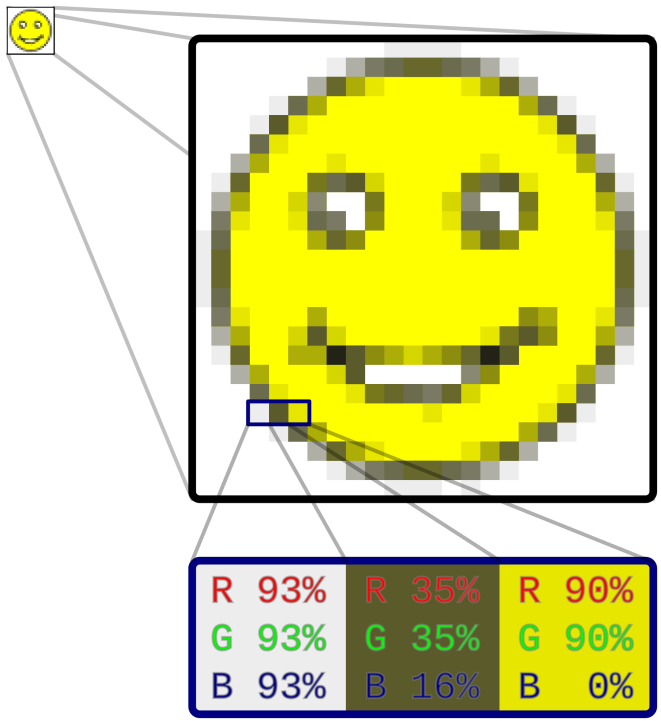
Visual fidelity priorities:

1. Colour richness
2. Possiblity for re-sizing

Prioritising colour richness: raster image data

Raster (or bitmap) graphics store images as a grid

- Many digital images: `.tif`, `.jpg`, `.gif`, `.png`



Source: https://en.wikipedia.org/wiki/Raster_graphics

Prioritising colour richness: raster image data

Raster graphics have three important components:

1. A grid (or **tessellation**) of equal size cells
2. A single value in each cell of the grid
3. A **lookup table** that maps cell values to colors

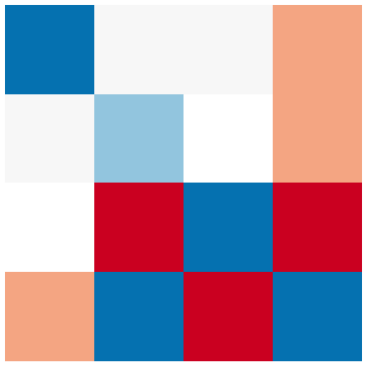
A. Cell IDs

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

B. Cell values

92	55	48	21
58	70	NA	37
NA	12	94	11
36	83	4	88

C. Colored values



Source: https://r.geocompx.org/spatial-class

Prioritising colour richness: raster image data

Most raster images use the **RGB colour space**

- Each raster cell's display colour is determined by a mixture of the three primary colours: red, green and blue

A raster image using the RGB colour space has (at least) three **channels**

- A channel is an array storing one layer of pixel data
- So, for RGB colour space: each channel is a colour (red, green and blue)
  - Can have additional "alpha" channel (transparency)
- Resulting image layers channels to generate rich visual

Prioritising colour richness: raster image data

Prioritising colour richness: raster image data

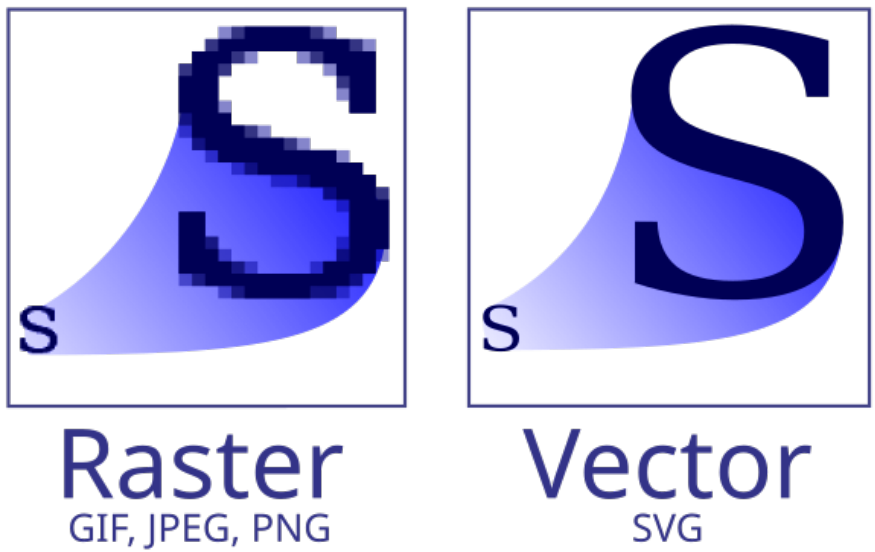
Prioritising colour richness: raster image data



Prioritising re-sizing: vector image data

Vector graphics store images as geometric objects, such as points, line segments, polygons, etc.

- Common file types: `.svg`, `.pdf`, and `.eps`



Source: https://en.wikipedia.org/wiki/Vector_graphics

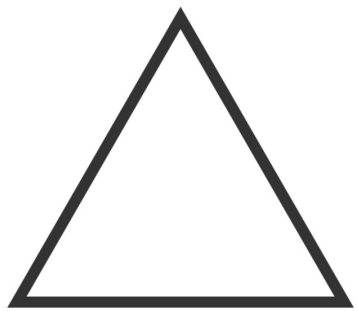
Prioritising re-sizing: vector image data

Vector graphics are defined by **geometries**: points, line segments, polygons, curves, etc.

Roughly speaking: vector graphics "trace" images with geometries

Polygons are a common way to make shapes, defined by:

- **Edges**: line segments
- **Nodes**/**vertices**: where line segments meet



Use cases

Raster graphics: images with continuous tones/colors, and/or when efficiency/size is important (e.g., websites)

- “Real-life” photos and videos
- Maps with aerial and sensor imaging
- Topographic maps

Vector graphics: images with geometric shapes and no continuous tones/colors, and/or when rescaling is important

- Maps of borders and other geographical shapes
- Cartoons, clip art, logos, etc.
- Typography (e.g., PDFs)

Outline

Digital photo data

Digital cameras have sensors that measure brightness (luminance) information

- Brightness information is quantised by bit depth
- Data from sensors combine to form pixels

A **monochrome camera** captures only brightness

- Generate black and white (technically **greyscale**) images

Sensors in a **colour camera** have filters for red, green and blue

- The sensors still capture brightness, but for a specific colour
- Each pixel is contains data on brightness for each colour

Digital photos are stored as raster data (usually compressed)

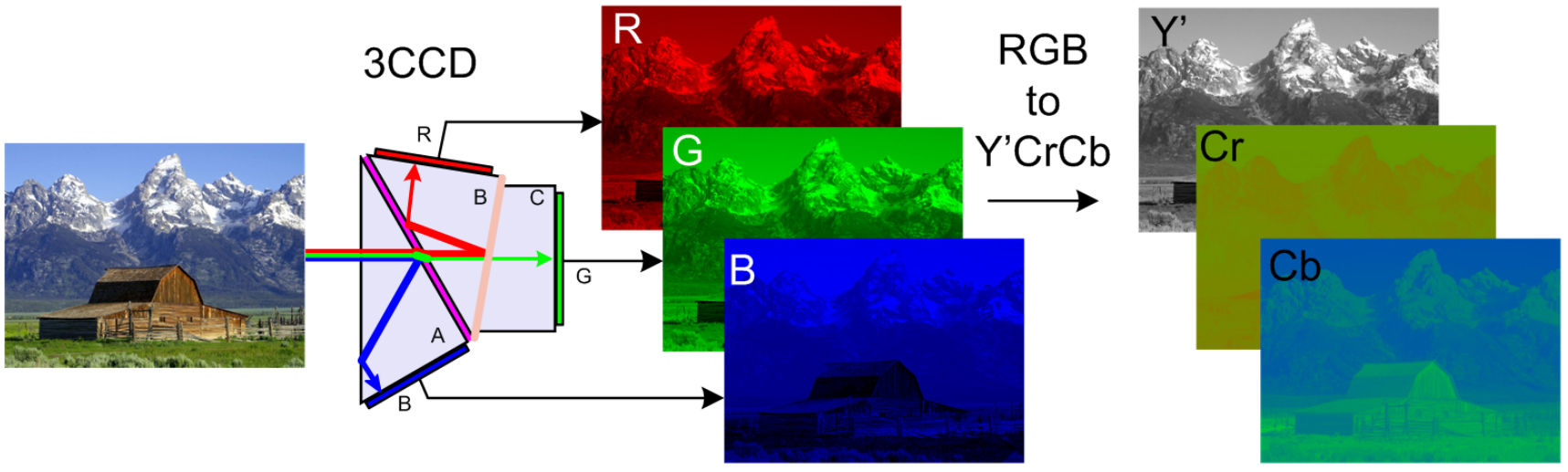
Digital photo compression

JPEG is a common standard for encoding photo data

These are (roughly) the steps for JPEG encoding

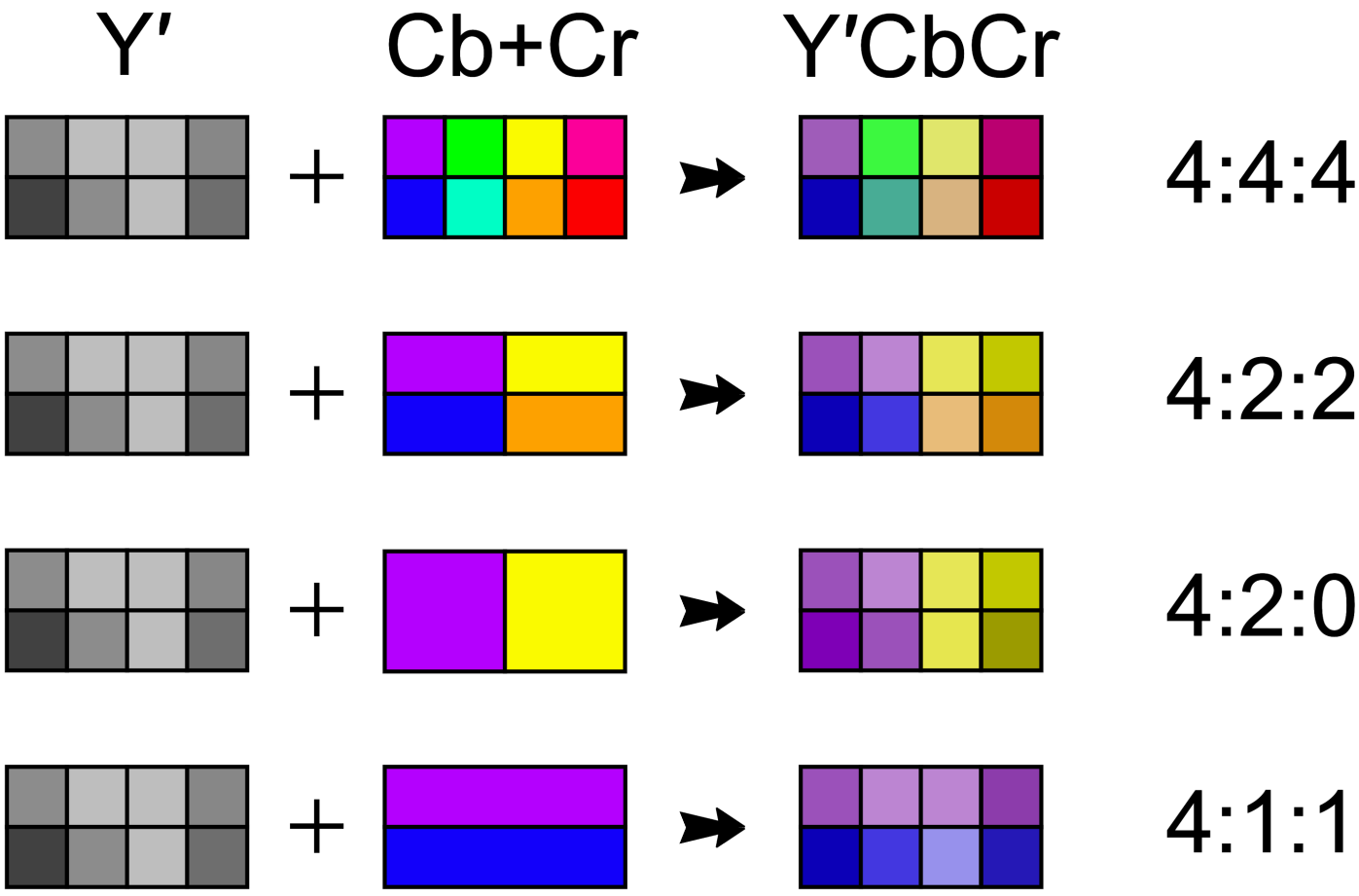
1. **Convert**: RGB channels to YCbCr channels (*lossless*)
2. **Subsample**: Chroma subsampling (*lossy*)
3. **Transform**: Discrete cosine transform (DCT) applied to blocks of pixels across each channel, yielding frequency coefficients (*lossless*)
4. **Quantise**: Resulting frequency coefficients are quantised, where bit depth varies across frequencies based on model of human vision (*lossy*)
5. **Compress**: Further *lossless* compression

Digital photo compression



Source: https://en.wikipedia.org/wiki/YCbCr#/media/File:CCD.png

Digital photo compression



Source: https://en.wikipedia.org/wiki/Chroma_subsampling#/media/File:Common_chroma_subsampling_ratios_YCbCr_CORRECTED.svg

Digital photo quality

Digital photos can be low quality for many reasons

- Core issue: information loss during capture or storage

Three major sources:

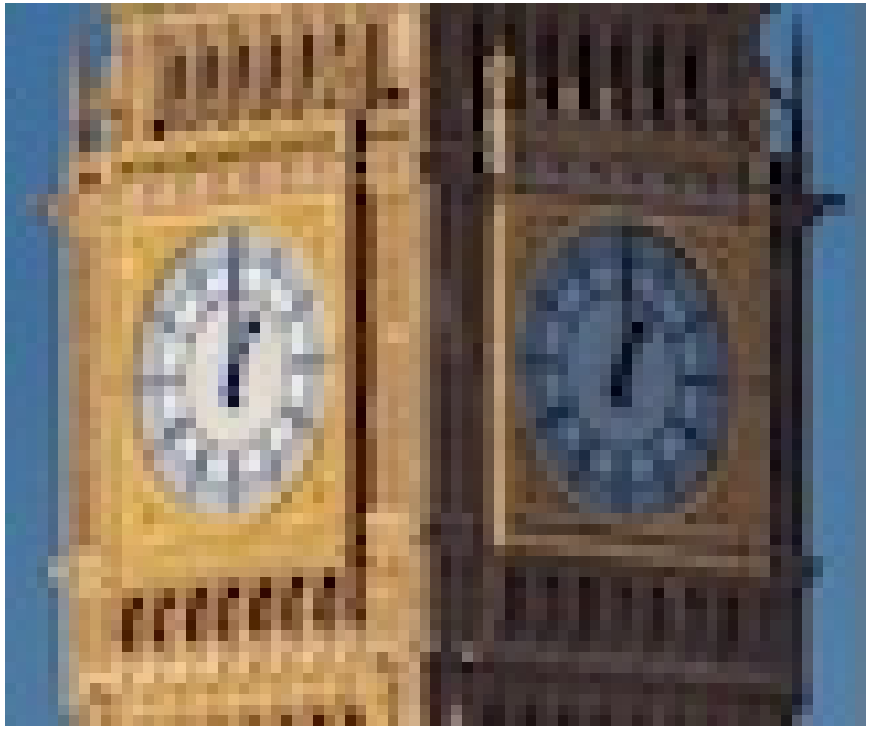
1. Excessively low pixel density or blur (happens during capture or file resize)
2. Chroma subsampling (happens during encoding)
3. Quantising (happens during capture and encoding)

Some of these are deliberate, some are not!

Digital photo quality

Raster graphics with low pixel density are **pixelated**

- “too few” cells in the grid to represent image well

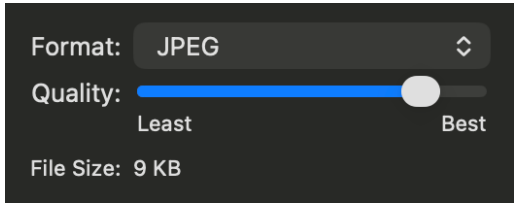


Source: https://en.wikipedia.org/wiki/Big_Ben

Digital photo quality

When saving a JPEG, often have to select “quality”

- that setting changes the quantising



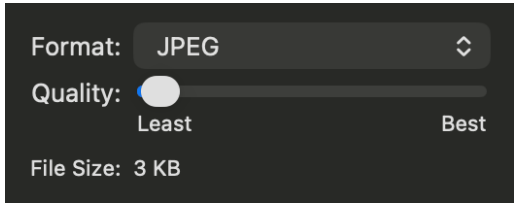


Photo data in R

There are multiple ways to import `.jpg` photo data into R, but we will use `{jpeg}` and `{exifr}`

- `{exifr}`: extract a photo's metadata
- `{jpeg}`: import a `.jpg` photo and work with its raster data

Reading a `.jpg` into R with `{jpeg}` decodes the JPEG file and yields an array object in R

- A collection of three matrices
- Each matrix is a raster grid corresponding to one of the three colour channels (RGB)
- Each cell in each matrix is a value from 0 to 1—the proportions of red, green and blue, respectively

When image data is "plotted", each pixel's colour is determined by mixing red, blue and green in the specified proportions

Photo data in R

```
library("exifr")
metadata <- read_exif(ryan.jpg.file)
names(metadata)
```

```
 [1] "SourceFile"           "ExifToolVersion"
 [3] "FileName"             "Directory"
 [5] "FileSize"             "FileModifyDate"
 [7] "FileAccessDate"       "FileInodeChangeDate"
 [9] "FilePermissions"      "FileType"
[11] "FileTypeExtension"    "MIMEType"
[13] "JFIFVersion"          "ResolutionUnit"
[15] "XResolution"          "YResolution"
[17] "ImageWidth"           "ImageHeight"
[19] "EncodingProcess"      "BitsPerSample"
[21] "ColorComponents"      "YCbCrSubSampling"
[23] "ImageSize"            "Megapixels"
```

Photo data in R

```
library("jpeg")
my.img <- readJPEG(ryan.jpg.file)
str(my.img)
```

```
 num [1:200, 1:200, 1:3] 0.345 0.341 0.333 0.322 0.31 ...
```

```
dim(my.img)
```

```
[1] 200 200   3
```

```
# Top corner of the red channel
my.img[1:3, 1:3, 1]
```

```
          [,1]      [,2]      [,3]
[1,] 0.3450980 0.3372549 0.3176471
[2,] 0.3411765 0.3333333 0.3137255
[3,] 0.3333333 0.3215686 0.3019608
```

Photo data in R

```
# Pixel data for top left pixel
tl.red <- my.img[1,1,1]
tl.green <- my.img[1,1,2]
tl.blue <- my.img[1,1,3]
print(c(tl.red,tl.green,tl.blue))
```

```
[1] 0.3450980 0.4392157 0.2431373
```

```
library("scales")
tl.pixel <- rgb(tl.red, tl.green, tl.blue)
tl.pixel
```

```
[1] "#58703E"
```

Photo data in R

```
show_col(tl.pixel, cex_label = 5)
```

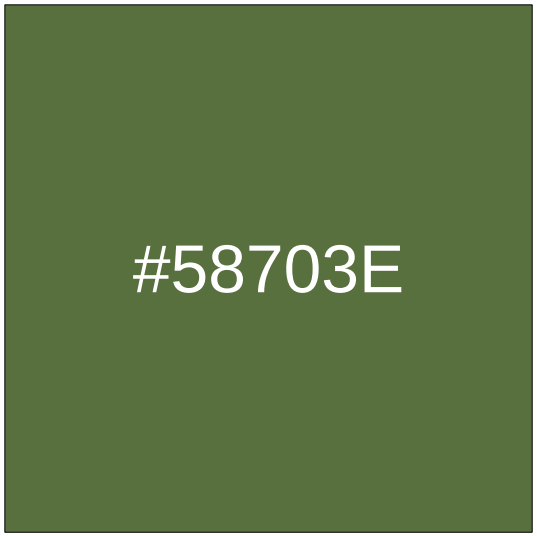


Photo data in R

```
library("tidyverse")
H <- dim(my.img)[1]
W <- dim(my.img)[2]
tp <- tibble(expand.grid(y = 1:H, x = 1:W))
tp$r <- as.vector(my.img[,,1])
tp$g <- as.vector(my.img[,,2])
tp$b <- as.vector(my.img[,,3])
tp$fill <- rgb(tp$r,tp$g,tp$b)
head(tp)
```

```
# A tibble: 6 x 6
      y     x     r     g     b fill
  <int> <int> <dbl> <dbl> <dbl> <chr>
1     1     1     1 0.345 0.439 0.243 #58703E
2     2     1     1 0.341 0.435 0.239 #576F3D
3     3     1     1 0.333 0.427 0.231 #556D3B
4     4     1     1 0.322 0.416 0.220 #526A38
5     5     1     1 0.310 0.404 0.216 #4F6737
6     6     1     1 0.306 0.4   0.212 #4E6636
```

Photo data in R

```
ggplot(tp, aes(x = x, y = y, fill = fill)) +
  geom_raster() +
  scale_fill_identity() +
  scale_y_reverse() +
  coord_equal(ratio = 1) +
  theme_void()
```

Other raster data file types

- Tag Image File Format (TIFF, `.tif` and `.tiff`)
  - Common file type for “raw” scanned data
  - Typically lossless compression (or uncompressed)
  - Very large files, not suitable for web
  - Technically a container—can use `.tif` for lossy compressed images
- Graphics Interchange Format (GIF)—`.gif`
  - The “ASCII of graphics” — only encodes 256 colours
  - Lossless compression (but still limited by palette)
- Portable Network Graphics (PNG)—`.png`
  - Full colour (but limited by bit depth)
  - Good for screenshots, text, diagrams, logos, line art
  - Not good for very large photos

Raster maps


Lerwick
Kirkwall
HEBRIDES
Elgin
Inverness
Portree
SCOTLAND
Aberdeen
Dundee
St Andrews
Stirling
Glasgow
Edinburgh
United Kingdom
Newcastle upon Tyne
NORTHERN IRELAND
Sligo
Isle of Man
Dundalk
Drogheda
Great Britain
Leeds
Dublin
Liverpool
Manchester
Galway
Ireland
Limerick
Kilkenny
Tralee
Waterford
Dingle
Killarney
Cork
Birmingham
Norwich
Cambridge
WALES
ENGLAND
Oxford
Cardiff
Bristol
London
Exeter
Bournemouth
Plymouth
English Channel
Guernsey
Jersey
Caen
Rouen
North Sea
Geopark Sunnhordland
Haugesund
Stavanger
Sandnes
Sandefjord
Fredrikstad
Oslo
Drammen
Karlstad
Örebro
Kristiansand
Trollhättan
Skövde
Linköping
Gothenburg
Jönköping
Hirtshals
Vendsyssel
Aalborg
Växjö
Halmstad
Herning
Aarhus
Helsingborg
Billund
Copenhagen
Denmark
Sjælland
Flensburg
Kiel
Stralsund
Kołobrzeg
Rostock
Lübeck
Świnoujście
Hamburg
Szczecin
Groningen
Bremen
Gorzów Wielkopolski
Amsterdam
Hanover
Potsdam
Berlin
Rotterdam
Netherlands
Bielefeld
Magdeburg
Zielona Gór
Essen
Dortmund
Antwerp
Düsseldorf
Kassel
Leipzig
Germany
Brussels
Cologne
Dresden
Lille
Liège
Belgium
Frankfurt
Prague
Luxembourg
Pilsen
Czechia
Mannheim
Nuremberg

Raster maps

For raster maps in R: `terra`

- Defines spatial data using a raster grid + projection
- Data is loaded as an `SpatRaster` object that is its own class:
  - dimensions: how many cells in the grid (the “resolution”)
  - matrix: stores values of cells in raster grid
- The values in the raster grid are “scaled” by the projection

Raster spatial data is stored as raster graphics, usually in `.tif` file format

Outline

Combining sound and images

Digital videos combine audio and still images

Digital videos are stored in containers with:

- Frames (images)
- Audio
- Other data like: syncronisation information, subtitles and metadata

For analysis:

- Can use standard audio analysis and image analysis tools
- Can study motion data

Tools are limited in R: we'll do some basic stuff

Combining sound and images

Core challenge: *huge* amount of data

Video cameras captures sequence of still digital images (**frames**), usually 24–60 per second

- If uncompressed: minutes of raw HD video can be hundreds of GBs
- Plus: record audio track(s), which can also be large

Video codecs encode video files to enable watching and transferring over a network

- Audio encoding is separate (and as above)

Lots of video codecs; we'll focus on core ideas

Video encoding: the idea

Compression for video data happens on two dimensions:

- **Intra-frame** compression
  - Similar in spirit to JPEG
  - Lossy compression around “imperceptable” colour variations + some lossless compression
- **Inter-frame** compression
  - Most frames are very similar
  - Instead of storing all data from each frame: store changes between frames
  - Much more efficient!

Video encoding: the idea

1. All frames: convert to YCbCr, then do chroma subsampling
2. Then, do frame prioritisation:
   - **I-frames** (intra): store compressed still image (similar to JPEG)—largest
   - **P-frames** (predicted): stores motion vectors and residuals for prediction (based on predcition forward)
   - **B-frames** (bidirectional): stores motion vectors and residuals for prediction (based on prediction forward and backward)
3. For all I, P and B frames: do (lossless) transformation to frequencies, quantise frequencies and do final lossless compression

Video data in R

Most video files you work with are containers

- The distinction between them are important for videographers, less important here

You can load data from video containers into R

- Unfortunately: R is less robust for video data
- This is partly because video data is quite large
- But also: codecs are complicated and someone has to write the R wrapper

In seminar, we'll look at some simple tools for ingesting video then:

- Extracting the audio track
- Extracting individual frames

Outline

Encoding machine readable data

A lot of data you use is machine-readable (binary)

For example, the data in a `.jpg` file is not encoded text:

```
read_file(ryan.jpg.file)      # not human readable!
```

```
[1] "\xff\xd8\xff\xe0"
```

```
read_file_raw(ryan.jpg.file)[1:15]  # raw bytes (first 10)
```

```
 [1] ff d8 ff e0 00 10 4a 46 49 46 00 01 01 01 00
```

But, sometimes you need to encode binary data into encoded characters

One reason: transporting data over a text-only network

Encoding machine readable data

One standard way to do this is **Base64** encoding

- Each collection of 6 bits is encoded with a character
- So, need $2^6 = 64$ characters for Base64 encoding

```
library("openssl")
ryan.b64 <- base64_encode(read_file_raw(ryan.jpg.file)[1:15])
ryan.b64
```

```
[1] "/9j/4AAQSkZJRgABAQEA"
```

Encoding machine readable data

This will *increase* the storage space required for your data

- Each Base64 character represents six bits, but requires 8 bits for storage (UTF-8)—Base64 encoded data is 33% larger!

A (very silly!) example:

```
charToRaw("LSE") # three bytes (in UTF-8)
```

```
[1] 4c 53 45
```

```
encoded <- base64_encode("LSE")
encoded
```

```
[1] "TFNF"
```

```
charToRaw(encoded) # four bytes (in UTF-8)
```

```
[1] 54 46 4e 46
```