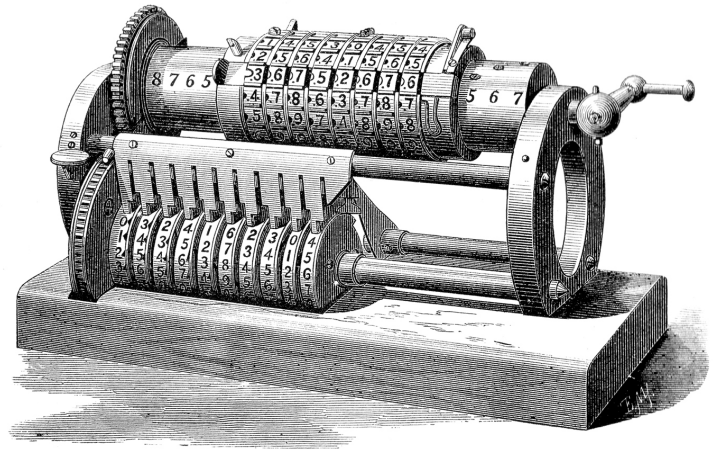
LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Programming in R

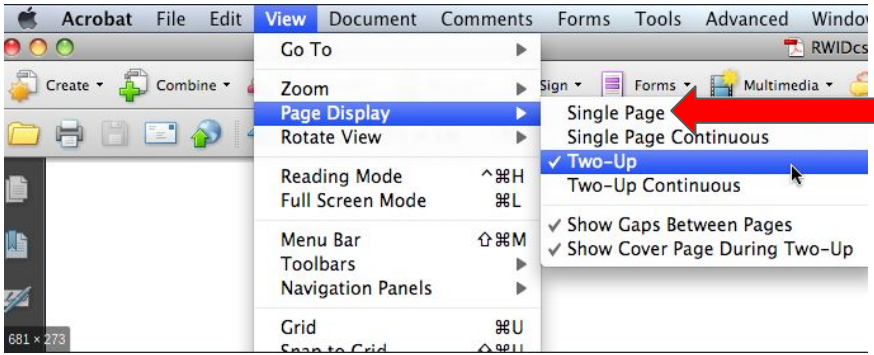
Binder

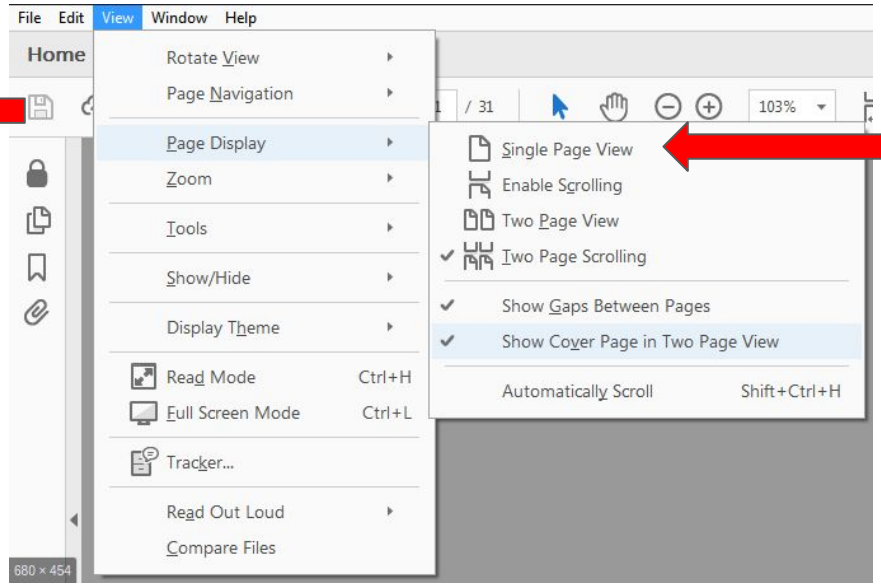


Unit 6:
Profiling & Benchmarks

About these Slides

The best way to view these slides (and to get the most out of the animations) is to view it in "presentation mode" or "single page view". See the images below where to set up single page view in your system.





Profiling, Tests, Good Software

Remember, this course has multiple goals:

- Learn things about the R language: "R"
- Get to know nice tools to use: "Tools"
- Learn things about software development in general: "Dev"

This unit:

- "Tools" Track: Profiling and Performance: `profviz`, `microbenchmark`

Tools Track

Benchmarks and Profiling:

`profvis`, `microbenchmark`

Profiling

Profiling and Benchmarking

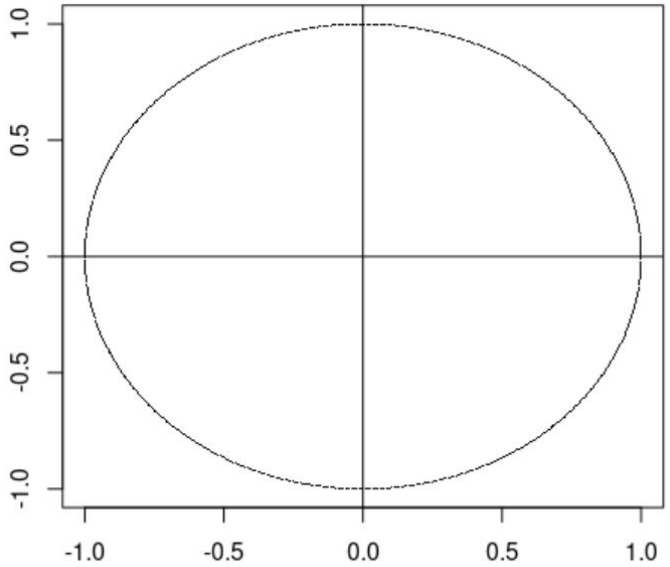
We have written code, it works well, but it is slow. What do we do?

=> Profiling

Profiling and Benchmarking

Example use case: estimating **π** (~3.1416) by estimating the area of a circle with radius 1

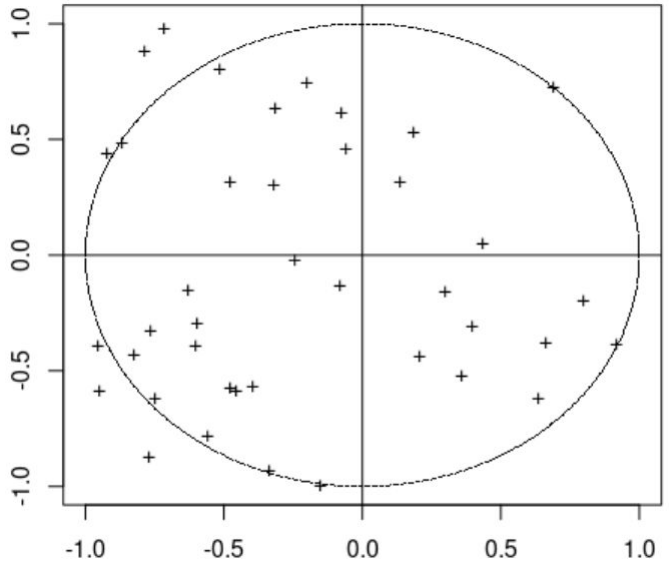
(I don't expect you to know this approach but I think it is a cool idea so I'm showing it here)



Profiling and Benchmarking

Example use case: estimating **π** (~3.1416) by estimating the area of a circle with radius 1

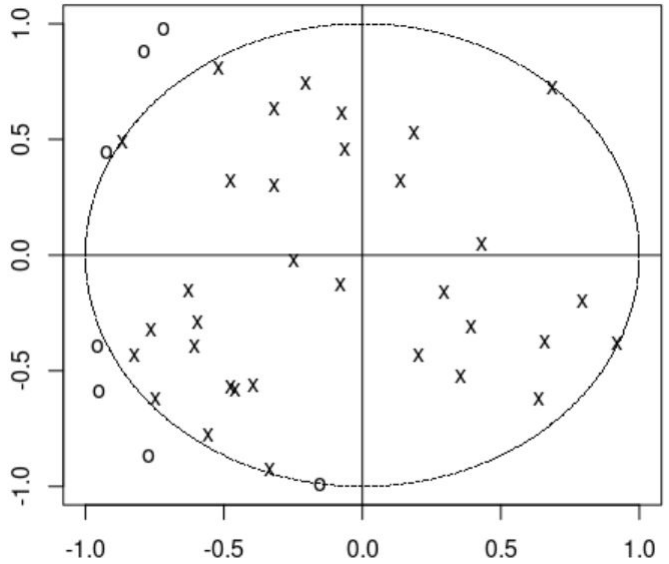
- sample points uniformly from [-1, 1] x [-1, 1]



Profiling and Benchmarking

Example use case: estimating **π** (~3.1416) by estimating the area of a circle with radius 1

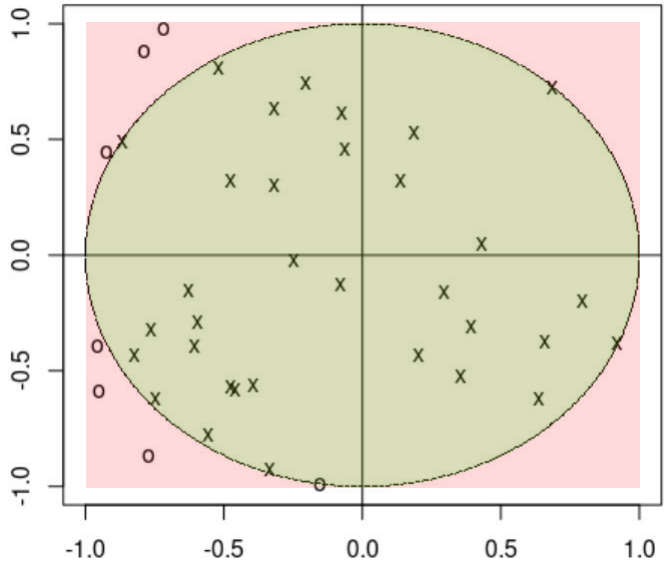
- sample points uniformly from [-1, 1] x [-1, 1]
- Check whether each point lies within the circle (by checking x^2+y^2 < 1)



Profiling and Benchmarking

Example use case: estimating **π** (~3.1416) by estimating the area of a circle with radius 1

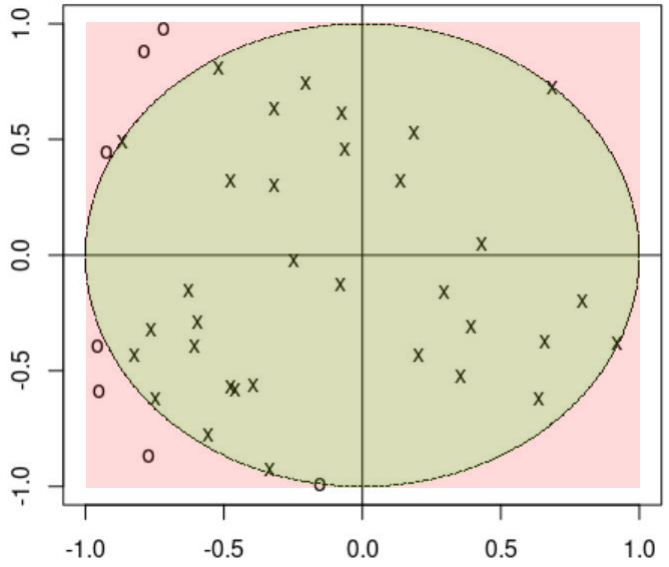
- sample points uniformly from [-1, 1] x [-1, 1]
- Check whether each point lies within the circle (by checking x^2+y^2 < 1)
- Fraction of points that are *within* the circle is approximately the fraction of the area of the circle relative to the area of the [-1, 1] x [-1, 1]-square (which is 4)



Profiling and Benchmarking

Example use case: estimating **π** (~3.1416) by estimating the area of a circle with radius 1

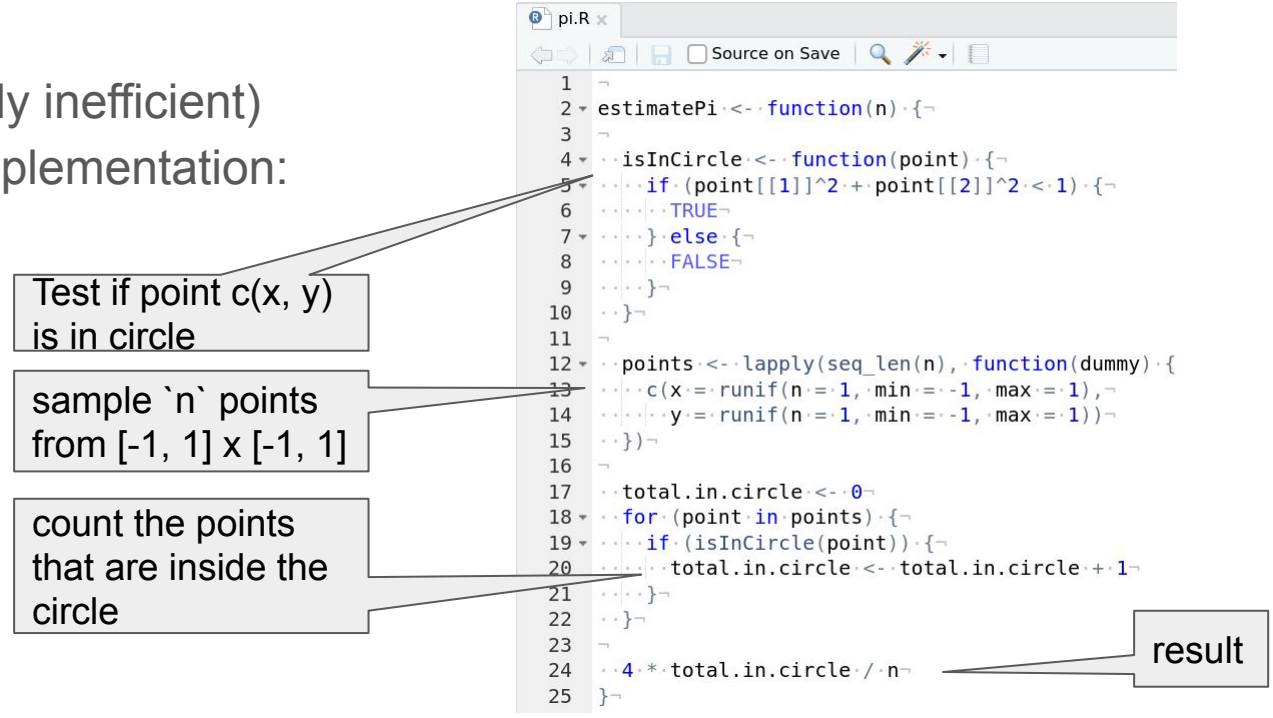
- sample points uniformly from [-1, 1] x [-1, 1]
- Check whether each point lies within the circle (by checking x^2+y^2 < 1)
- Fraction of points that are *within* the circle is approximately the fraction of the area of the circle relative to the area of the [-1, 1] x [-1, 1]-square (which is 4)
- => 4 * points.in.circle / points.total ≈ **π**



Profiling and Benchmarking

Example use case: estimating **π** (~3.1416) by estimating the area of a circle with radius 1

- (purposefully inefficient) example implementation:



Profiling and Benchmarking

Example use case: estimating **π** (~3.1416) by estimating the area of a circle with radius 1

- (purposefully inefficient) example implementation:

```
> estimatePi(1000000)
[1] 3.143032
> estimatePi(1000000)
[1] 3.143356
> system.time(estimatePi(1000000))
   user  system elapsed
  6.973   0.059   7.071
```

pi.R

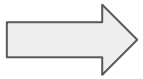
Source on Save

```
1
2  estimatePi <- function(n) {
3
4    isInCircle <- function(point) {
5      if (point[[1]]^2 + point[[2]]^2 < 1) {
6        TRUE
7      } else {
8        FALSE
9      }
10   }
11
12   points <- lapply(seq_len(n), function(dummy) {
13     c(x = runif(n = 1, min = -1, max = 1),
14       y = runif(n = 1, min = -1, max = 1))
15   })
16
17   total.in.circle <- 0
18   for (point in points) {
19     if (isInCircle(point)) {
20       total.in.circle <- total.in.circle + 1
21     }
22   }
23
24   4 * total.in.circle / n
25 }
```

Profiling and Benchmarking

`profvis` package

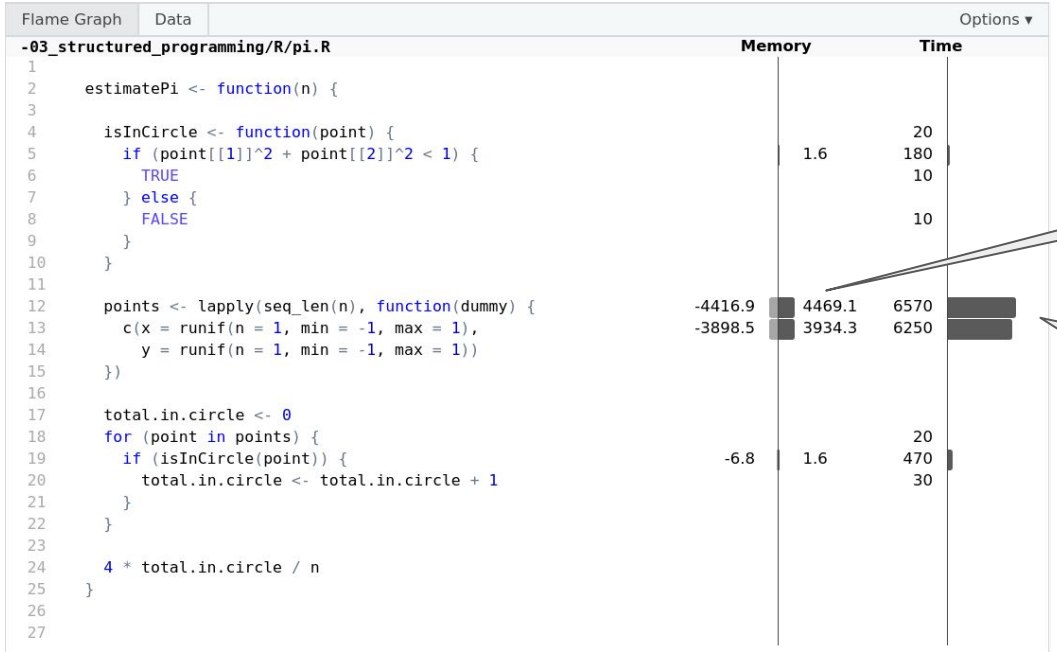
```
> profvis::profvis(estimatePi(1000000))
```



Profiling and Benchmarking

`profvis` package

```
> profvis::profvis(estimatePi(1000000))
```



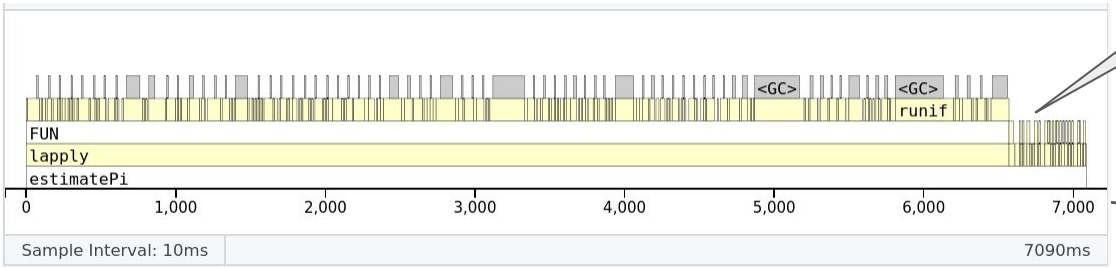
memory allocated / freed

time spent at the line (ms)
Line 13 is a continuation of line 12, so the 6.2 seconds in line 13 are also counted in the 6.6 seconds in line 12.

Profiling and Benchmarking

`profvis` package

```
> profvis::profvis(estimatePi(1000000))
```



what function was being executed at each time point?

time axis

We learn that we should put our first effort into making the "lapply" faster, or try to avoid it altogether.

Profiling and Benchmarking

`profvis` package

One thing to be careful about: lazy evaluation! Function arguments only start to get calculated once they are "needed". This can make for confusing profiles.

Profiling and Benchmarking

`profvis` package

Example:

```
myPrint <- function(x) {
  cat("Pi is about... ")
  assertAtomic(x)
  print(x)
}

calcPi <- function() {
  myPrint(estimatePi(10000000))
}
```

This is where 'x' is needed the first time within "myPrint", so this is where it gets evaluated. What is x, however? It is `estimatePi(10000000)`, which takes long!

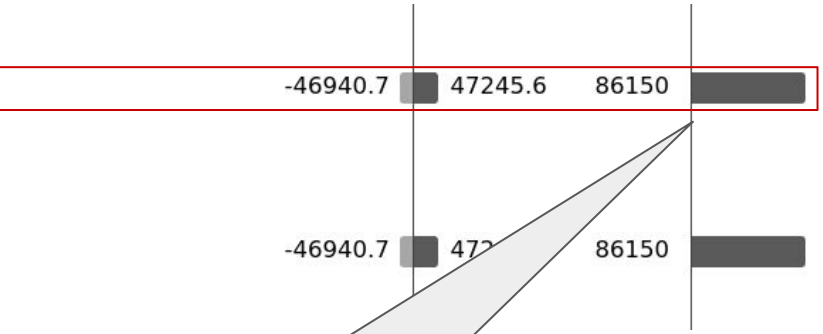
```
> profvis::profvis(calcPi())
```

Profiling and Benchmarking

`profvis` package

Example:

```
48  myPrint <- function(x) {
49    cat("Pi is about... ")
50    assertAtomic(x)
51    print(x)
52  }
53
54  calcPi <- function() {
55    myPrint(estimatePi(10000000))
56  }
57
```



This is where 'x' is needed the first time within "myPrint", so this is where it gets evaluated. What is x, however? It is `estimatePi(10000000)`, which takes long!

checkmate is not that slow!

Profiling and Benchmarking

`profvis` package

Proof:

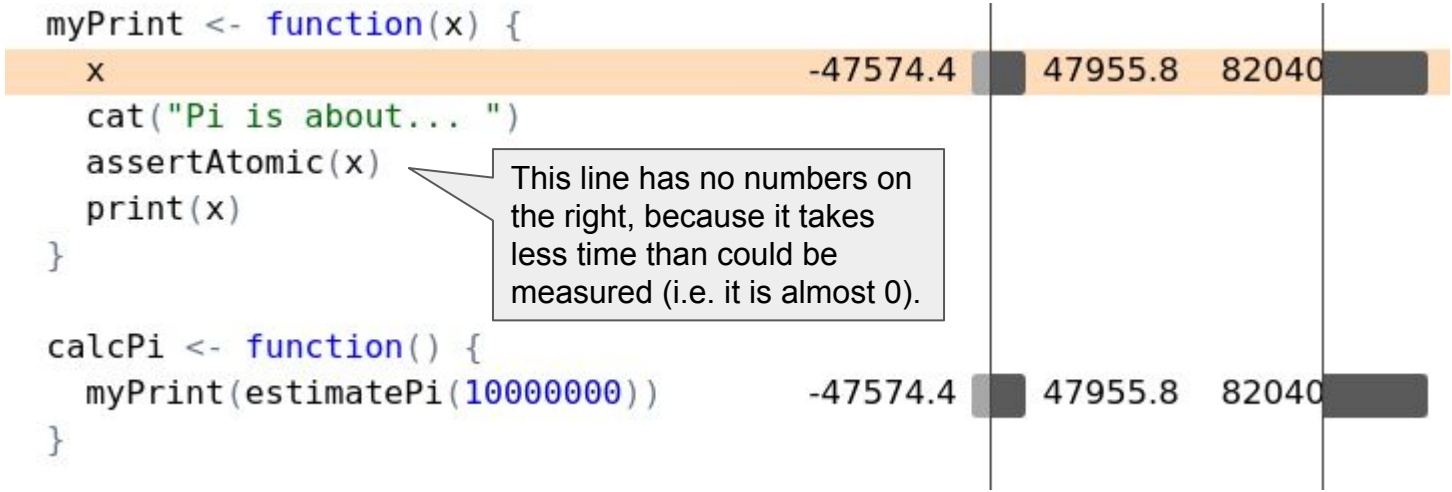
```
myPrint <- function(x) {
  x
  cat("Pi is about... ")
  assertAtomic(x)
  print(x)
}

calcPi <- function() {
  myPrint(estimatePi(10000000))
}

> profvis::profvis(calcPi())
```

`x` (i.e. `estimatePi(10000000)`) will now get evaluated here!
(this is called "forcing")

Profiling and Benchmarking

`profvis` package

Proof:



Profiling and Benchmarking

`profvis` package

- Uses Rprof (part of R) internally
- *sampling profiler*
  - samples every 10 ms (can be changed, it is the 'interval' parameter or profvis()) where exactly the execution currently is
  - ==> non-deterministic result
  - ==> lines / functions that run very fast may be missed
- ?profvis::profvis
- online documentation

Benchmarking

Profiling and Benchmarking

We want to write fast code, but we don't know what is fast

=> Benchmarking

Profiling and Benchmarking

Benchmarking

- A single evaluation: system.time() function

```
> system.time(estimatePi(1000000))
   user  system elapsed
  6.973   0.059   7.071
```

  - "user": total CPU-time that R was run
  - "system": time taken by the system (Windows / Mac OS / Linux) for some reason -- this may be relevant if you are doing a lot with graphics, network, or files
  - "elapsed": total time that ran during the evaluation
    - this can be less than "user" if you have code that runs on multiple CPUs

Profiling and Benchmarking

Benchmarking

- A single evaluation: system.time() function

```
> system.time(estimatePi(1000000))
   user  system elapsed
  6.973   0.059   7.071
```

  - "user": total CPU-time that R was run
  - "system": time taken by the system (Windows / Mac OS / Linux) for some reason -- this may be relevant if you are doing a lot with graphics, network, or files
  - "elapsed": total time that ran during the evaluation
    - this can be less than "user" if you have code that runs on multiple CPUs
  - "gcFirst"-argument: whether to free up memory before evaluating the call. Setting this to FALSE makes the benchmark less deterministic but more efficient.

```
> system.time(estimatePi(1000000), gcFirst = FALSE)
```

Profiling and Benchmarking

Benchmarking

`microbenchmark` package

- run multiple repetitions of different functions
- calculate statistics about runtime
- this is useful for comparison of different approaches
- *micro*benchmark, because it should be used on small parts of approaches you are considering
- Arguments:
  - the expressions to be evaluated, possibly named
  - "times": how often to evaluate each expression
  - "check": check that results are equal? (can be set to "equal", "equivalent", "identical" or NULL)
  - ... others; see ?microbenchmark::microbenchmark

Profiling and Benchmarking

Benchmarking

`microbenchmark` package example: compare different ways of looping over vectors:

```
xs = rnorm(10000)
mb <- microbenchmark::microbenchmark(
  vectorized = xs^2,
  vapply = vapply(xs, function(x) x^2, 0),
  pre.alloc = {
    res <- numeric(length(xs))
    for (i in seq_along(xs)) {
      res[[i]] <- xs[[i]]^2
    }
    res
  },
  append = {
    res <- numeric(0)
    for (x in xs) {
      res <- c(res, x^2)
    }
    res
  },
  times = 10
)
```

Setting "times" to less than the default (100) to speed things up

Profiling and Benchmarking

Benchmarking

`microbenchmark` package example: compare different ways of looping over vectors:

```
xs = rnorm(10000)
mb <- microbenchmark::microbenchmark(
  vectorized = xs^2,
  vapply = vapply(xs, function(x) x^2, 0),
  pre.alloc = {
    res <- numeric(length(xs))
    for (i in seq_along(xs)) {
      res[[i]] <- xs[[i]]^2
    }
    res
  },
  append = {
    res <- numeric(0)
    for (x in xs) {
      res <- c(res, x^2)
    }
    res
  },
  times = 10
)
```

```
> mb
Unit: microseconds
       expr        min         lq        mean      median         uq        max neval
 vectorized     12.324     14.327     24.4309     15.2035     22.191     84.819    10
     vapply   4977.244   5078.013   5998.0094   5270.6895   5461.182  11825.186    10
  pre.alloc   2837.942   3054.548   3111.4351   3105.1235   3165.486   3398.774    10
     append 138246.516 142391.590 143718.6829 144297.3290 145392.458 146827.529    10
```

Profiling and Benchmarking

Benchmarking

`microbenchmark` package example: compare different ways of looping over vectors:

```
xs = rnorm(10000)
mb <- microbenchmark::microbenchmark(
  vectorized = xs^2,
  vapply = vapply(xs, function(x) x^2, 0),
  pre.alloc = {
    res <- numeric(length(xs))
    for (i in seq_along(xs)) {
      res[[i]] <- xs[[i]]^2
    }
    res
  },
  append = {
    res <- numeric(0)
    for (x in xs) {
      res <- c(res, x^2)
    }
    res
  },
  times = 10
)
```

```
> mb
Unit: microseconds
       expr        min         lq        mean      median         uq        max neval
 vectorized     12.324     14.327     24.4309     15.2035     22.191     84.819    10
     vapply   4977.244   5078.013   5998.0094   5270.6895   5461.182  11825.186    10
  pre.alloc   2837.942   3054.548   3111.4351   3105.1235   3165.486   3398.774    10
     append 138246.516 142391.590 143718.6829 144297.3290 145392.458 146827.529    10
```

Unless the runtime of an approach is nondeterministic for a reason, or unless you did really a lot of "times", it may be better to inspect 'median' and 'uq' (upper quartile) than mean. These are more robust to outliers caused e.g. by a different application on your computer needing ressources randomly at some point.

Profiling and Benchmarking

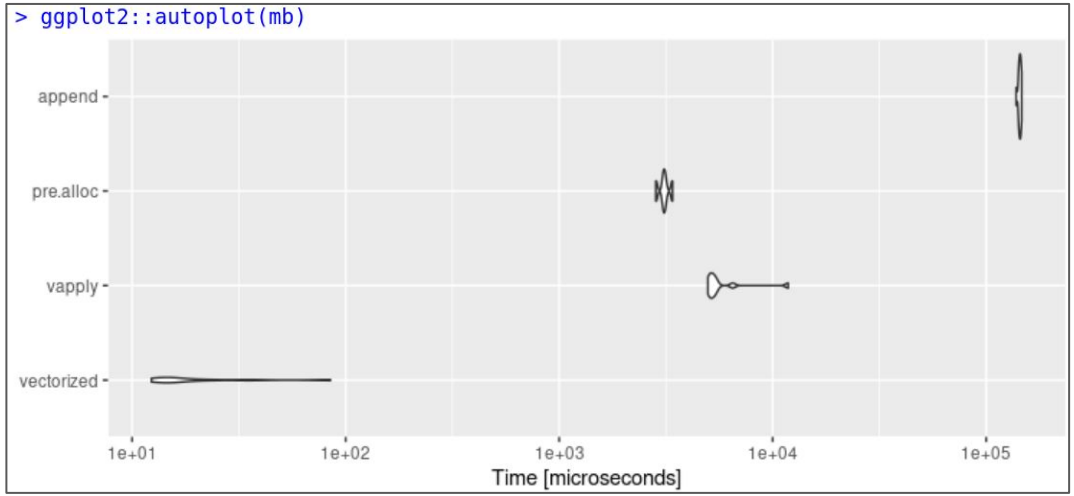
Benchmarking

`microbenchmark` package example: compare different ways of looping over vectors:

```
xs = rnorm(10000)
mb <- microbenchmark::microbenchmark(
  vectorized = xs^2,
  vapply = vapply(xs, function(x) x^2, 0),
  pre.alloc = {
    res <- numeric(length(xs))
    for (i in seq_along(xs)) {
      res[[i]] <- xs[[i]]^2
    }
    res
  },
  append = {
    res <- numeric(0)
    for (x in xs) {
      res <- c(res, x^2)
    }
    res
  },
  times = 10
)
```

Plot the result! (note log scale here)

```
> ggplot2::autoplot(mb)
```



Profiling and Benchmarking

Benchmarking -- Notes:

- Use system.time if you want to know the time spent on a function that takes considerable time
- *don't* do `system.time(for (i in seq_len(100)) myFun(x))`. This is what the `microbenchmark` package is for.
- Limits of benchmarks:
  - *Different* microbenchmark runs give different numbers, sometimes off by a factor of 2 or more. This is because conditions change. Even things like the temperature of your CPU can influence runtimes (through "turbo boost")! Therefore only compare runtimes *within* a call of microbenchmark(), unless the difference is very large.
  - There may be interactions between parts of your code that microbenchmarks can miss (e.g. how the GC is used, CPU caching).
  - What is faster or slower could change between small and large datasets, keep that in mind.
  - (If you are at the point where you worry about this a lot, you should probably consider Rcpp...)

What We Expect You to Know

- Profiling
  - Know how to use `profvis` to find hotspots of your code
  - Be aware of lazy evaluation of function arguments
- Benchmarking
  - Know how to use `system.time()`
  - Know how to use `microbenchmark`
    - call it
    - interpret results
- Writing fast code
  - Avoid writing unnecessarily slow code
  - Don't sacrifice maintainability, code clarity, or your time for speed unless you know it gives relevant benefits
    - the code in question is actually a hot spot
    - the gain in speed is worth it