UNIVERSITY OF
COPENHAGEN

Measuring and Improving Performance

Computational Statistics

Johan Larsson

Department of Mathematical Sciences, University of Copenhagen

March 31, 2026

Today's Agenda

Profiling
Identifying bottlenecks in code

Benchmarking
Comparing performance of implementations

Improving Performance
Writing efficient code

Two Types of Performance

Speed
How long does it take to run?

Memory
How much memory space does it use?

At Odds
Often, improving one worsens the other.

In this Course
We will focus on **speed**, not **memory**.

Should You Optimize?

What is Your Use Case?
Before thinking about improving performance, consider if you even need to. Maybe your code is already fast enough?

If you're only doing it for yourself, then is the **investment** worth it?

Premature Optimization: The Root of All Evil?

> *We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*
>
> *—Donald Knuth*

Readability
Improving performance often comes at the cost of code readability.

Profiling

But if your code is too slow, then it's time to profile it.

A profiler quantifies how much time each part of your code takes to run.

Two Types of Profiling

Instrumentation-based profiling inserts code to measure time taken by functions.

Sampling-based profiling periodically samples the call stack to see what functions are running.

This is what R's profilers do.

Outputs of Profilers

Profilers record both time and memory usage.

In this course we focus mostly on time.

Profiling in R

The R package profvis provides tools to visualize profiling output.[1].

```
library(profvis)
```

Simple to Use

1. Source the code you want to profile using `source()`.
2. Either run `profvis()` on the expression you want to profile or in RStudio, `Profile -> Start Profiling`. Run your code, and then `Profile ->`Stop Profiling‘.

The result is an interactive webpage (tab in RStudio).

```
source("<your-file>.R")
profvis(your_function())
```

[1] Can be activated directly in RSTudio (but not Positron)

Let's profile the implementation of the kernel density estimator from the last lecture.

```r
# In file R/kernel.R
kern_dens <- function(x, h, m = 512) {
  rg <- range(x)
  xx <- seq(rg[1] - 3 * h, rg[2] + 3 * h, length.out = m)
  y <- numeric(m)

  for (i in seq_along(xx)) {
    for (j in seq_along(x)) {
      y[i] <- y[i] + exp(-(xx[i] - x[j])^2 / (2 * h^2))
    }
  }

  y <- y / (sqrt(2 * pi) * h * length(x))

  list(x = xx, y = y)
}
```

Example

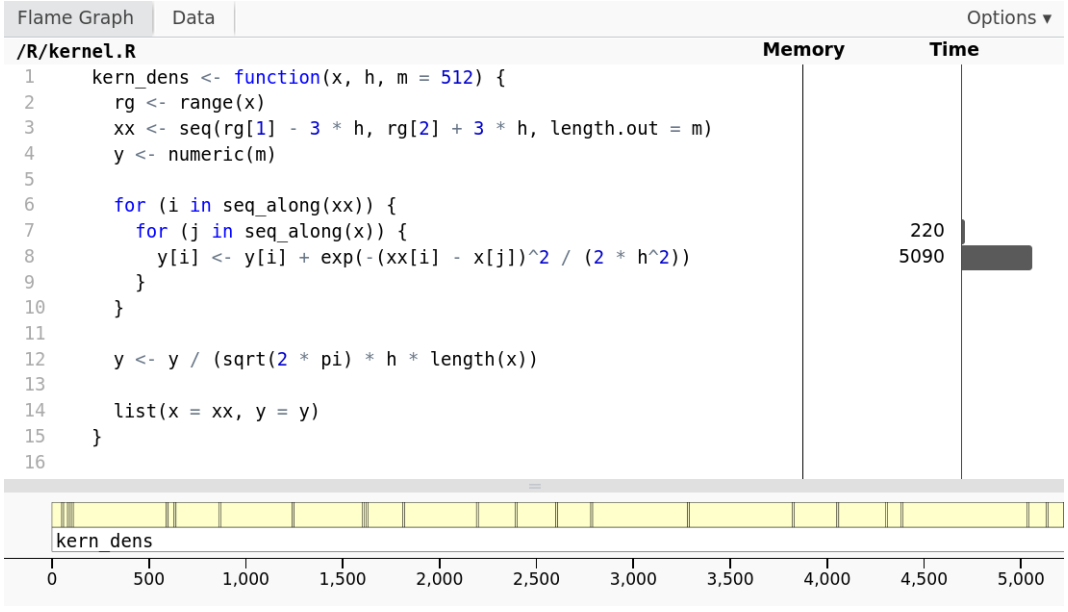
Now let's run the profiler.

```
source(here::here("R", "kernel.R"))

x <- rnorm(1e5)

profvis(kern_dens(x, 0.2))
```

We could have also just placed the code inside the same source file, but this way we can keep the code clean.


Flame Graph
Data
Options ▾
/R/kernel.R
Memory
Time
1 kern_dens <- function(x, h, m = 512) {
2 rg <- range(x)
3 xx <- seq(rg[1] - 3 * h, rg[2] + 3 * h, length.out = m)
4 y <- numeric(m)
5
6 for (i in seq_along(xx)) {
7 for (j in seq_along(x)) {
8 y[i] <- y[i] + exp(-(xx[i] - x[j])^2 / (2 * h^2))
9 }
10 }
11
12 y <- y / (sqrt(2 * pi) * h * length(x))
13
14 list(x = xx, y = y)
15 }
16
220
5090
kern_dens
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
4,500
5,000

A First Optimization

The bottleneck is the nested loop, which imposes considerable overhead in R.

Fixing this is easy! We can vectorize the inner loop.

```
# Old (Nested Loop)
kern_dens_vec <- function(x, h, m = 512) {
  # ...
  for (i in seq_along(xx)) {
    for (j in seq_along(x)) {
      y[i] <- y[i] + exp(-(xx[i] - x[j])^2 / (2 * h^2))
    }
  }
  # ...
}
```

A First Optimization

The bottleneck is the nested loop, which imposes considerable overhead in R.

Fixing this is easy! We can vectorize the inner loop.

```
# New (vectorized)
kern_dens_vec <- function(x, h, m = 512) {
  # ...

  for (i in seq_along(xx)) {
    y[i] <- sum(exp(-(xx[i] - x)^2 / (2 * h^2)))
  }

  # ...
}
```

How much faster is this? We'll benchmark this soon.

Common Pitfalls

Nested Calls
With multiple function calls on the same line, you won't directly see where the time is spent.

Anonymous Functions
Complicates profiling since you will only see `<Anonymous>` in the output.

Resolution
Resolution (sampling frequency) may too low for small pieces of code—but do you really need to profile them then?

Quarto and R Markdown
Running profvis inside Quarto/RMarkdown often does not work well. But, if you do, using `keep.source = TRUE` in `source()` might help.

Benchmarking

Benchmarking

The purpose of benchmarking is to compare the performance of different implementations of the same task.

The bench Package

We use the R package bench for benchmarking.

```
library(bench)
```

`bench::mark()`

The main function of the package: a high-precision timer that adaptively chooses the number of iterations to get accurate results.

Also checks results for correctness, unless `check = FALSE`.

A First, Simple Benchmark

```
x <- runif(100)

sqrt_bench <- bench::mark(
  sqrt(x),
  x^0.5
)

sqrt_bench
```

```
# A tibble: 2 x 6
  expression      min   median `itr/sec` mem_alloc `gc/sec`
  <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
1 sqrt(x)    360.07ns 391.04ns  1349338.        NA        0
2 x^0.5        1.88us   1.97us   417534.        NA        0
```

plot.bench_mark()

```
plot(sqrt_bench)
```

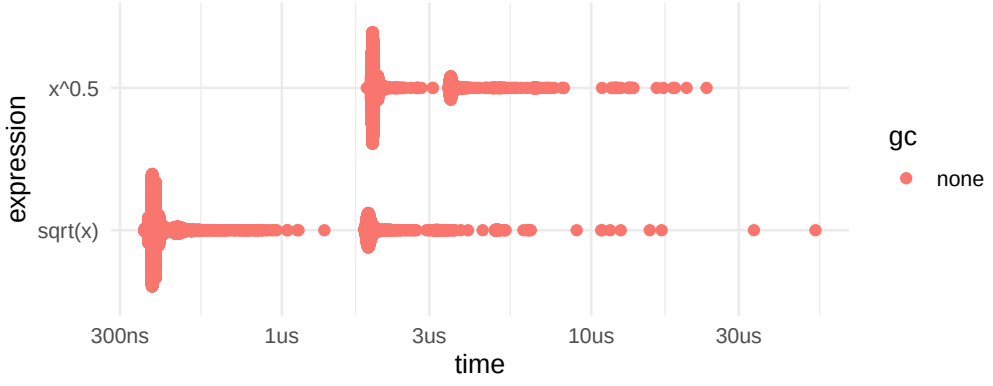


Figure 1: A beeswarm plot of the benchmark results.

The ggbeeswarm package is necessary for default plot behavior.

Parameterized Benchmarking

Achieve parameterized benchmarking with `bench::press()`, that runs functions across outer product of its initial arguments.

```
dens_bench <- bench::press(
  n = 2^(6:9),
  {
    x <- rnorm(n)
    bench::mark(
      base = density(x, 0.2),
      loop = kern_dens(x, 0.2),
      vectorized = kern_dens_vec(x, 0.2),
      check = FALSE # Why? Recall last lecture!
    )
  }
)
```

Bench Press Results

```
head(dens_bench, 6)
```

```
# A tibble: 6 x 7
  expression     n      min   median `itr/sec` mem_alloc
  <bch:expr> <dbl> <bch:tm> <bch:tm>     <dbl> <bch:byt>
1 base          64 178.05us 185.38us    5244.         NA
2 loop          64   5.03ms   5.07ms     191.         NA
3 vectorized    64 624.87us 644.48us    1488.         NA
4 base         128  182.4us 194.06us    5022.         NA
5 loop         128   9.96ms  10.04ms      99.1        NA
6 vectorized   128 969.31us 988.07us     998.         NA
# i 1 more variable: `gc/sec` <dbl>
```

```
plot(dens_bench)
```

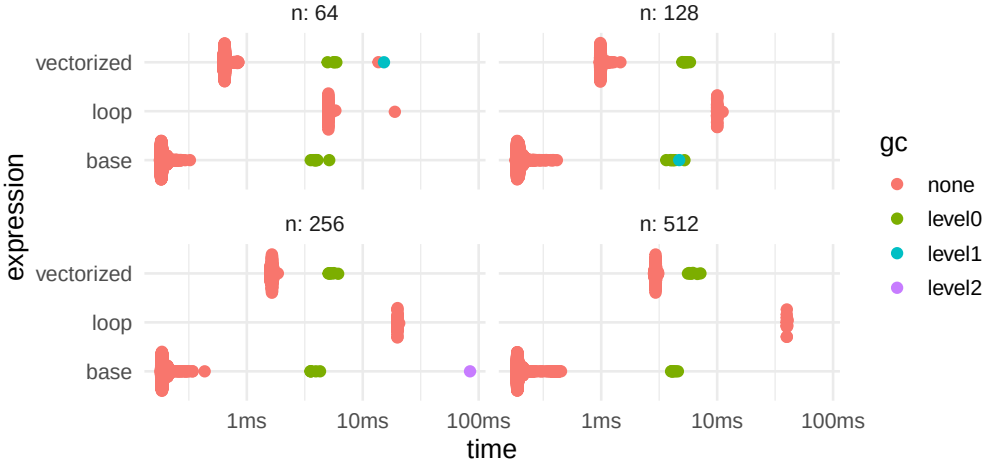


Figure 2: Default plot method for `bench::press()` results, here showing the performance of three different implementations of a kernel density estimator.

```r
mutate(dens_bench, method = as.character(expression)) |>
  ggplot(aes(n, median, color = method)) +
  geom_point() +
  geom_line() +
  labs(y = "time (µs)")
```

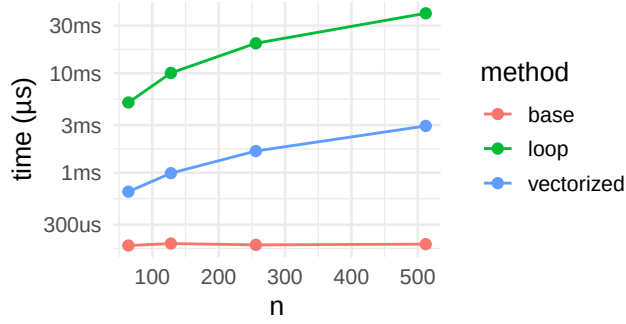


Figure 3: Custom plot of benchmark results for the three different implementations.

Notes About Benchmarking

Don't Relativize Too Much
Look at absolute values too: does the difference matter (for your use case)?

Background Tasks
Your computer may be doing something else at the same time.

Hardware
Hardware matters: different computers may give different results.

Improving Performance

Improving Performance

We could divide the strategies for optimizing code into multiple categories.

General Strategies

Strategies that apply to **every** computer language, such as

- avoiding copies,
- storing data wisely,and
- parallelizing.

R-Specific Tricks

Strategies that apply selectively to the R language, such as calling

- vectorized (compiled) functions, or
- specialized functions.

Context-Specific Tricks

Strategies that are specific to the problem at hand, such as **binning** (later this lecture).

R Is Slow ...

... when used like a low-level language.

R is an **interpreted** (as opposed to *compiled*) language. You have the convenience of never having to wait around for something to compile, but pay the price when you need a piece of code to be fast.

It was written mainly for specifying statistical models (not for developing new numerical methods).

It is suitable for high-level programming where most low-level computations are implemented in a compiled language (e.g. `lm()` and `qr()`.)

R Is Fast ...

... when most computations are carried out by calls to compiled code.

```
x <- rnorm(1e4)
bench_res <- bench::mark(
  loop = {
    y <- numeric(length(x))
    for (i in seq_along(x)) {
      y[i] <- 10 * x[i]
    }
    y
  },
  vectorized = 10 * x
)
```

```
plot(bench_res)
```

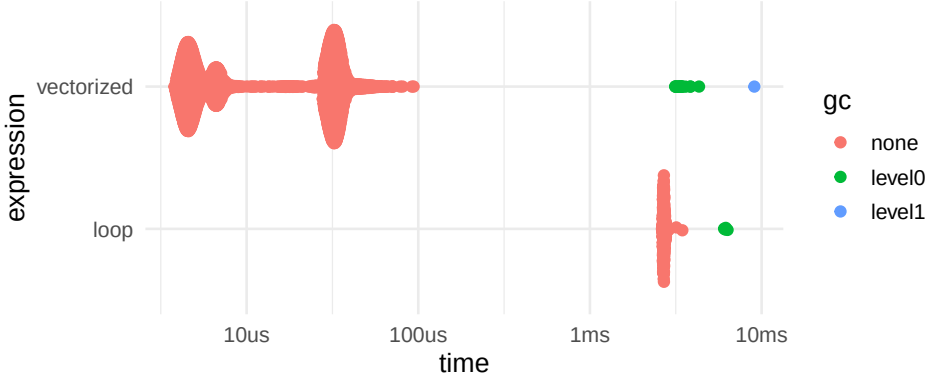


Figure 4: Benchmark results for a loop vs. a vectorized computation

Vectorization

Vectorization means rewriting code to operate on entire vectors at once, instead of looping over elements. (E.g. `mean()` instead of a `for` loop.)

The term is misleading. Most "vectorized" code also involves a loop, but one that is typically written in C, C++, or Fortran.

Beware of Loops in Disguise

Just because you ran a vectorized function, it doesn't make it fast.

```
loop_bench <- bench::mark(
  sapply = sapply(x, function(x_i) 10 * x_i),
  loop = {
    y <- numeric(length(x))

    for (i in seq_along(x)) {
      y[i] <- 10 * x[i]
    }

    y
  }
)
```

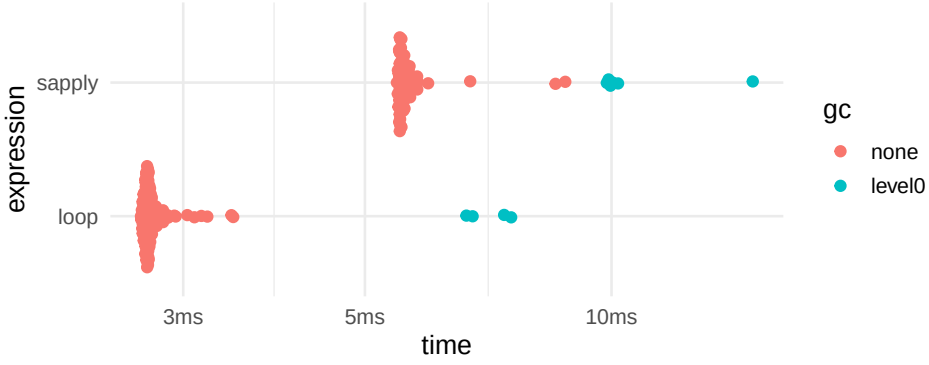


Figure 5: The loop is slightly faster than the vectorized `sapply()`.

Binning in Density Estimation

The line profiler revealed that most time is spent on the kernel evaluation, but we cannot optimize the kernel further.

But we can use binning: create B bins and evaluate the kernel and replace

$$\hat{f}(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

with

$$\hat{f}(x) = \frac{1}{nh} \sum_{j=1}^{B} n_j K\left(\frac{x - c_j}{h}\right)$$

where c_j is the center of the jth bin and n_j is the number of data points in the jth bin.

Turns complexity from $O(nm)$ to $O(n) + O(mB)$.

Binning Procedure

1. Determine the range of the data and divide it into equal-sized bins.
2. Count the number of data points in each bin.
3. For each bin, compute the kernel density estimate using the bin count and the kernel function.
4. For a new data point, find the bin it belongs to and use the pre-computed kernel density estimate for that bin.

It's an example of context-specific optimization.

Caution
If $n < B$, then binning will not help. (But do you need to optimize for this scenario?)

Implementation

A key to the implementation is this function, which loops over the data, for each point checking which bin center is closest.

```
kern_bin <- function(x, l, u, B) {
  w <- numeric(B)
  delta <- (u - l) / (B - 1)

  for (j in seq_along(x)) {
    i <- floor((x[j] - l) / delta + 0.5) + 1
    w[i] <- w[i] + 1
  }

  w / sum(w)
}
```

Full Binning Implementation

```
kern_dens_bin <- function(x, h, m = 512) {
  rg <- range(x) + c(-3 * h, 3 * h)
  xx <- seq(rg[1], rg[2], length.out = m)

  w <- kern_bin(x, rg[1], rg[2], m)

  kerneval <- exp(-(xx - xx[1])^2 / (2 * h^2)) / (sqrt(2 * pi) * h)
  kerndif <- toeplitz(kerneval)
  y <- colSums(w * kerndif)

  list(x = xx, y = y, h = h)
}
```

The `kern_dens_bin()` function computes bin weights using `kern_bin()` with grid points as bin centers.

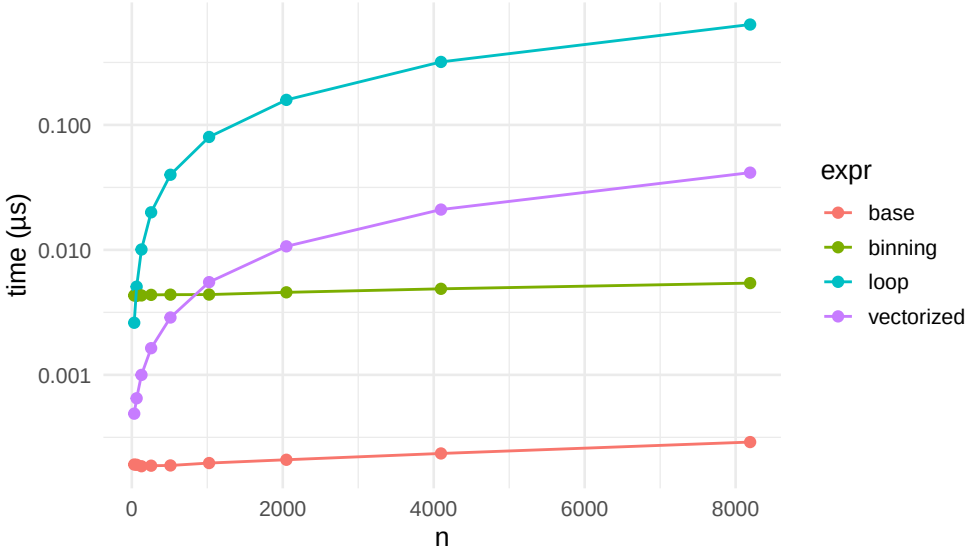


Figure 6: The relative benefit of binning increases with the size of the data.

Testing

As always, we should test our code to ensure it works as expected.

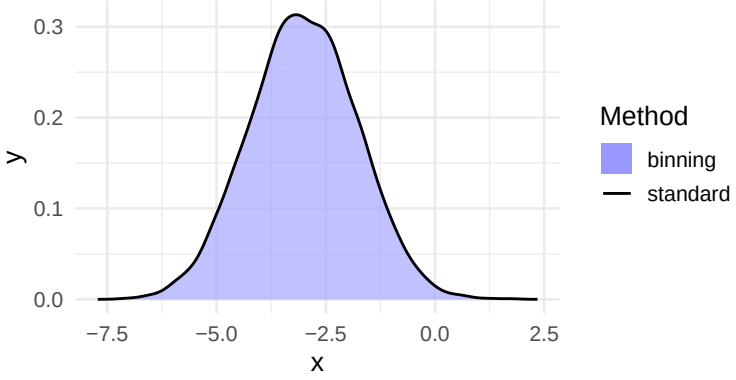


Figure 7: The standard density estimator as a black line and the binning estimate as a filled area.

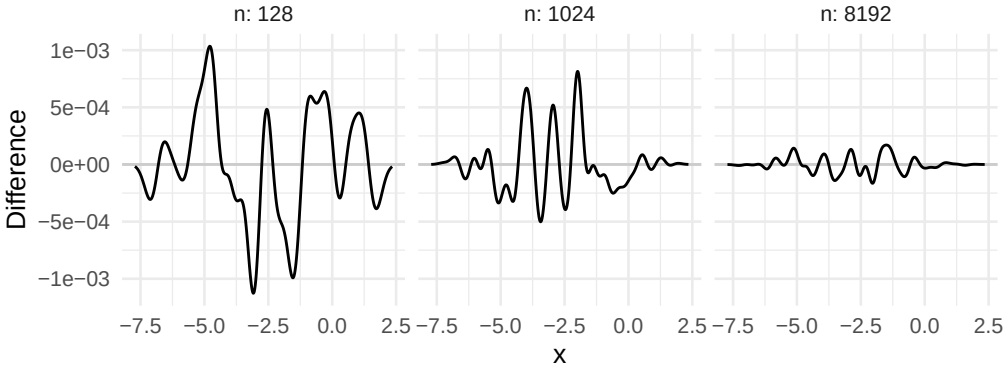


Figure 8: Absolute errors due to bining decrease with increasing length of data sequence.

Line Profiling

The `kern_dens_bin()` function is so much faster for long sequences that to get good profiling results we use a 512 times longer data sequence.

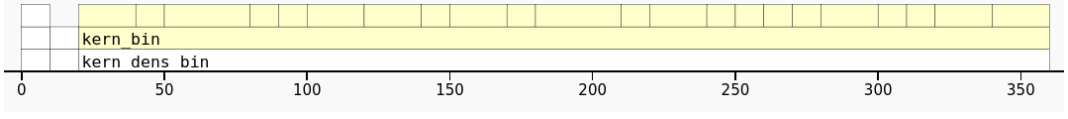
```
x <- rnorm(2^22)
profvis(kern_dens_bin(x, 0.2))
```

```r
kern_bin <- function(x, l, u, B) {
  w <- numeric(B)
  delta <- (u - l) / (B - 1)
  for (j in seq_along(x)) {
    i <- floor((x[j] - l) / delta + 0.5) + 1
    w[i] <- w[i] + 1
  }
  w / sum(w)
}

kern_dens_bin <- function(x, h, m = 512) {
  rg <- range(x) + c(-3 * h, 3 * h)
  xx <- seq(rg[1], rg[2], length.out = m)
  weights <- kern_bin(x, rg[1], rg[2], m)
  kerneval <- exp(-(xx - xx[1])^2 / (2 * h^2)) / (sqrt(2 * pi) *
h)
  kerndif <- toeplitz(kerneval)
  y <- colSums(weights * kerndif)
  list(x = xx, y = y, h = h)
}

```

10
190
140
340



Copy-on-Write (Modify)

Recall: R passes objects by reference, but when an object is modified a copy is created.

Examples

- Accessing a column of a matrix.
- Growing vectors (`c()`) and matrices (`rbind()`, `cbind()`).

```
x <- rnorm(100)
x <- c(x, 4) # 101 values are allocated
```

Memory

Memory in R

In R, everything is typically loaded into memory.

Garbage Collection

R includes a garbage collector, which intermittently releases unused blocks in memory.

Trade-Offs

Storing intermediate objects that are used multiple times will boost performance at the cost of additional memory storage.

Storage Mode

R stores matrices in column-major order.

When you access a column of a matrix, you are accessing a contiguous block of memory.

Some operations are faster with column-major order and others are faster with row-major order.

Computing Matrix-Vector Products

Let's say we want to compute the product of a matrix `x` and a vector `y`.

Strategy 1

We access rows, then transpose.

```
f1 <- function(x, y) {
  n <- NROW(x)
  z <- double(n)

  for (i in seq_len(n)) {
    z[i] <- t(x[i, ]) %*% y
  }

  z
}
```

Strategy 2

Transpose first, then access columns.

```
f2 <- function(x, y) {
  n <- NROW(x)
  x_t <- t(x)
  z <- double(n)

  for (i in seq_len(n)) {
    z[i] <- x_t[, i] %*% y
  }

  z
}
```

Strategy 3

Access columns directly, update sums together.

```
f3 <- function(x, y) {
  n <- NROW(x)
  p <- NCOL(x)
  z <- double(n)

  for (i in seq_len(p)) {
    z <- z + x[, i] * y[i]
  }

  z
}
```

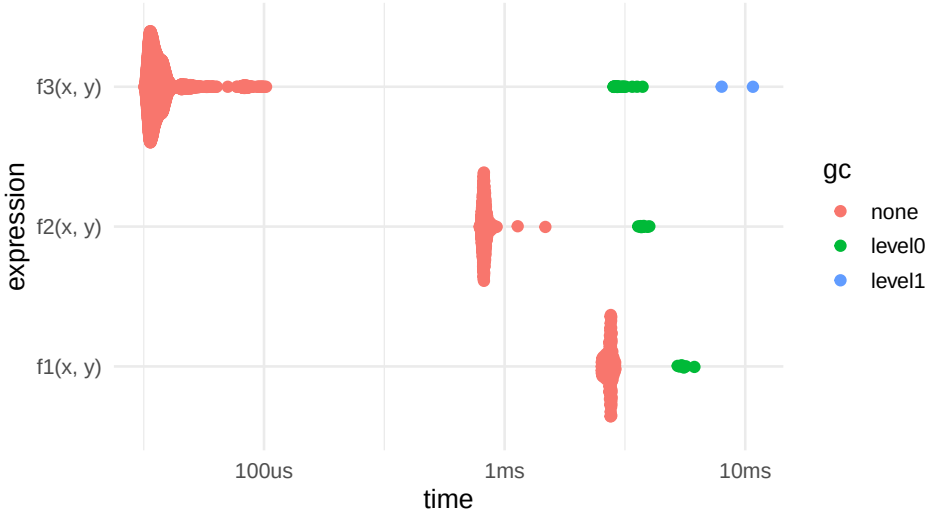


Figure 9: Benchmark results for three different implementations of matrix-vector multiplication.

Exercises

Exercises

Exercise 1
Benchmark `gauss()` against `dnorm()`; plot and compare the results. Before starting: which do you think will be faster?

```
gauss <- function(x, h = 1) {
  exp(-x^2 / (2 * h^2)) / (h * sqrt(2 * pi))
}
```

Exercise 2
Profile `gauss()` by breaking the function apart. Is there a bottleneck? Can you improve the performance?

Summary

First find bottlenecks through profiling.

Benchmark different implementations and use existing solutions.

Vectorize and use domain-specific knowledge to improve performance.

Keep in Mind

Beware of the **root of all evil**: profile first, optimize second.

Keep readability in mind: performant code can be hard to read.

Write tests: performance improvements can introduce bugs.