Profiling and Optimizing R code

Andrew Redd, PhD. VINCI R Expert

- Tools:
  - Rstudio
  - `profvis`
- Not what to do but **how to do it**

What is Profiling?

In software engineering, profiling ("program profiling", "software profiling") is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization, and more specifically, performance engineering.

Wikipedia Definition

Profiling in R

The `Rprof()` function is the basic function for profiling in R. It measures every 0.02 seconds (can be changed):

- Memory (if specified)
- Time spent
- Function calls

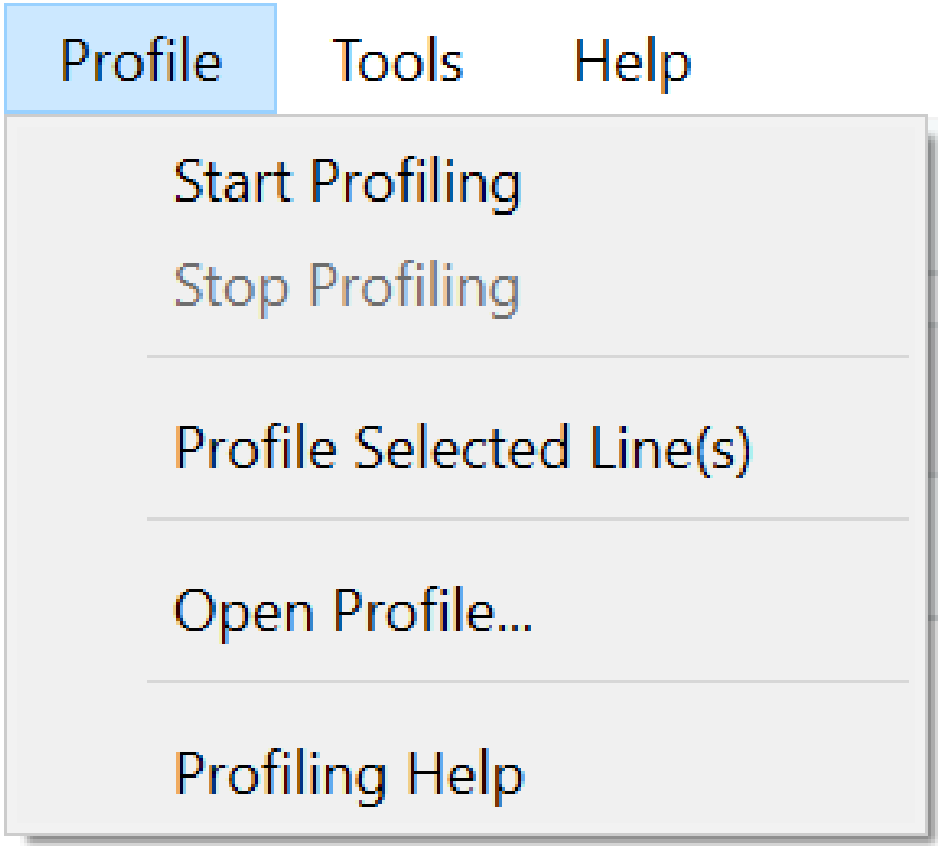
and writes to a profile log file (`Rprof.out`). This file must subsequently be processed and summarized (`summaryRprof()`).

Don't worry you rarely will need to call these directly.

Caveats

- Time
  - Time is measured slightly differently beteween Windows and Unix like systems.
- Lazy Evaluation
  - Lazy evaluation can at times muddy the picture of when certain expressions are evaluated.

Profiling in RStudio



Rstudio Profile Menu

Profile Menu Details

- **Start Profiling/Stop Profiling**
  - Give you greatest control but can complicate interpretation.
- **Profile Selected Line(s)**
  - IMO best option for getting to the root of problems. To use highlight the lines to run and select this option.
- **Open Profile…**
  - To look at a saved profile file.
- **Profileing Help**
  - Opens https://rstudio.github.io/profvis/ in a web browser.

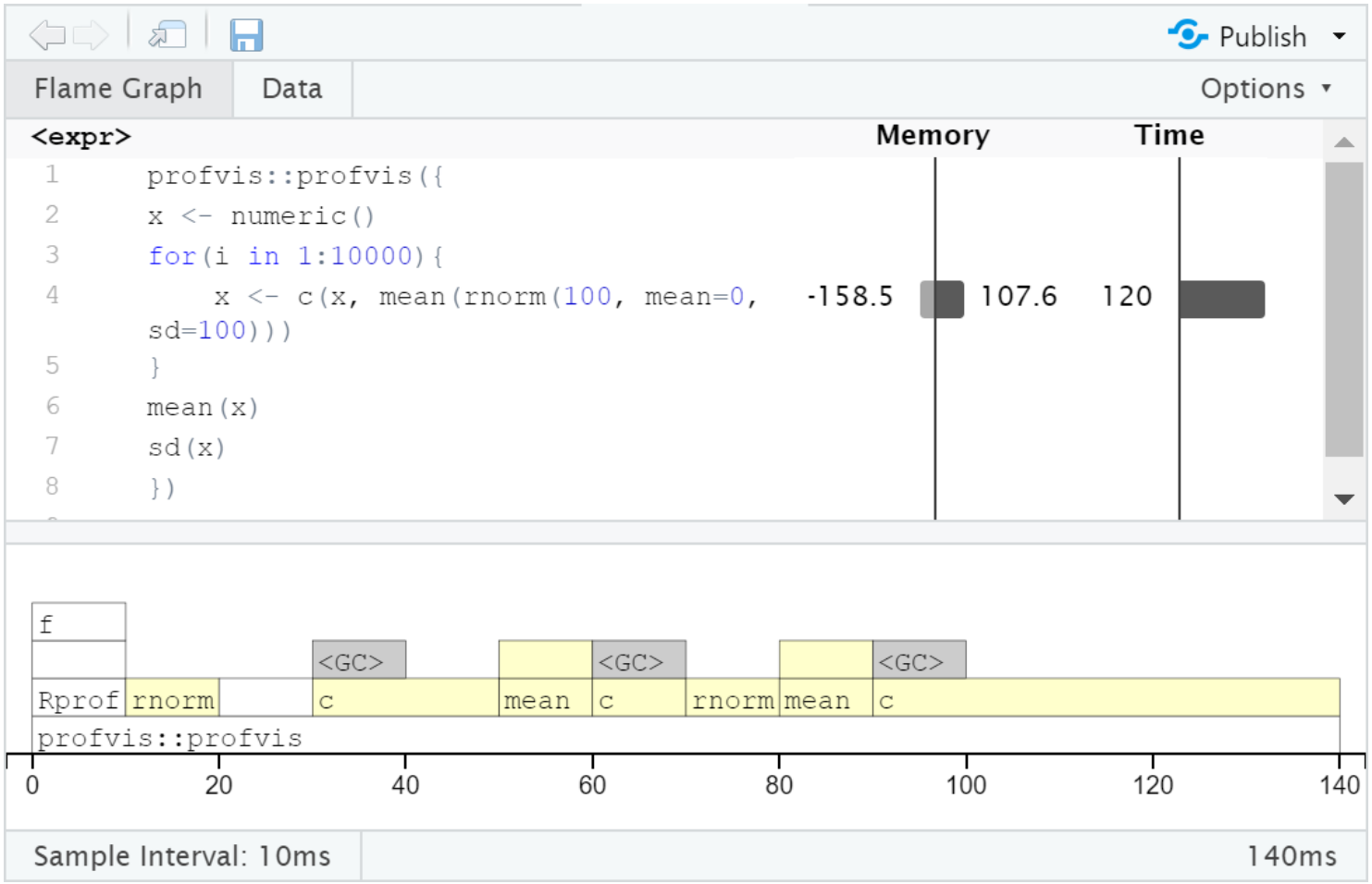
Example 1

We will start with an example, something that we know to be a terrible idea (growing a vector) to illustrate.

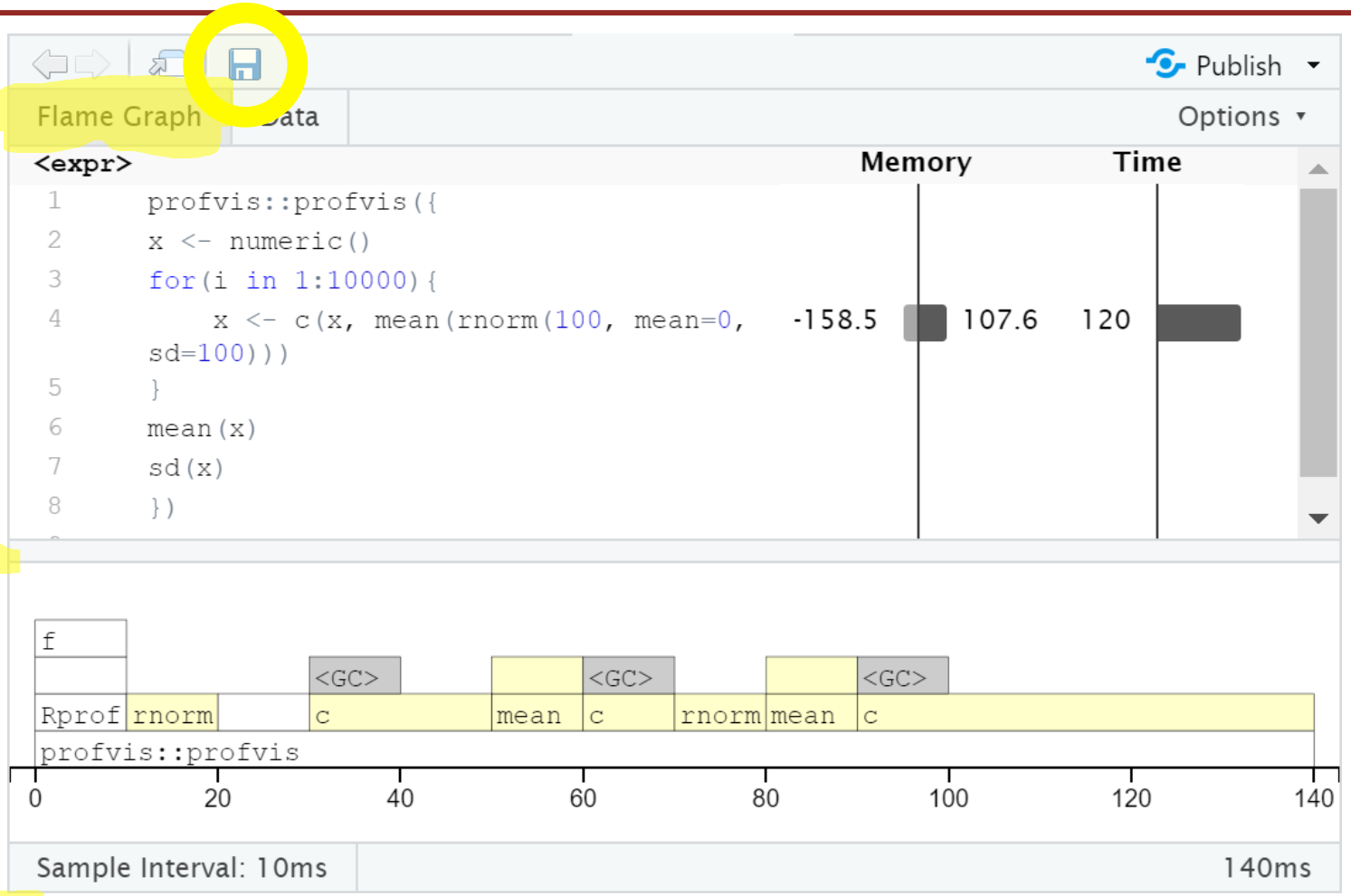
```
x <- numeric()
for(i in 1:10000){
    x <- c(x, mean(rnorm(100, mean=0, sd=100)))
}
mean(x)
sd(x)
```

Each time profiling is performed there are minor variations.

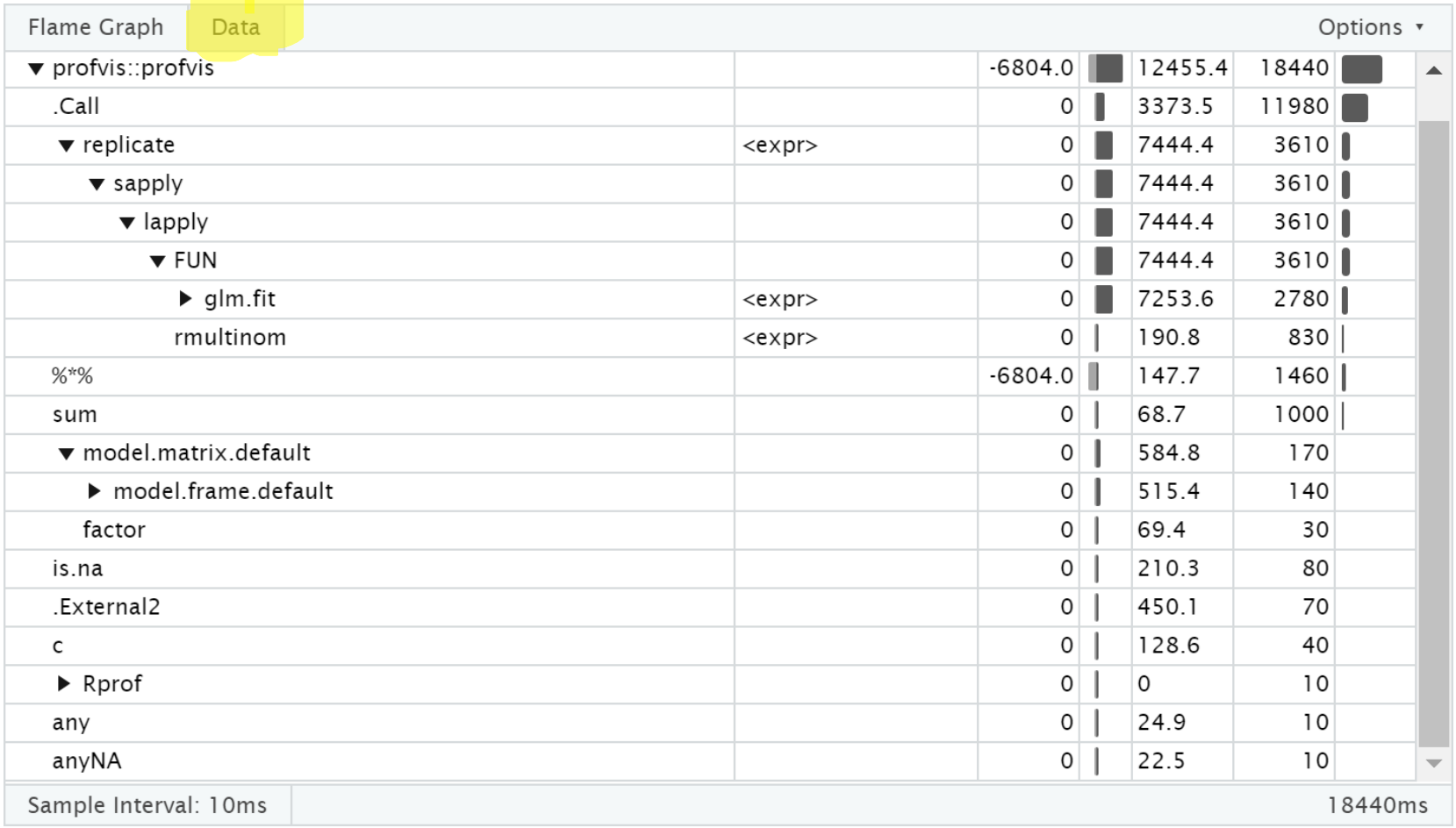
Example 1 - Profile



Sidebar – A walk around profvis



Sidebar – A walk around profvis



Flame Graph | Data | Options

▼ profvis::profvis		-6804.0		12455.4	18440
.Call		0		3373.5	11980
▼ replicate	<expr>	0		7444.4	3610
▼ sapply		0		7444.4	3610
▼ lapply		0		7444.4	3610
▼ FUN		0		7444.4	3610
▶ glm.fit	<expr>	0		7253.6	2780
rmultinom	<expr>	0		190.8	830
%*%		-6804.0		147.7	1460
sum		0		68.7	1000
▼ model.matrix.default		0		584.8	170
▶ model.frame.default		0		515.4	140
factor		0		69.4	30
is.na		0		210.3	80
.External2		0		450.1	70
c		0		128.6	40
▶ Rprof		0		0	10
any		0		24.9	10
anyNA		0		22.5	10

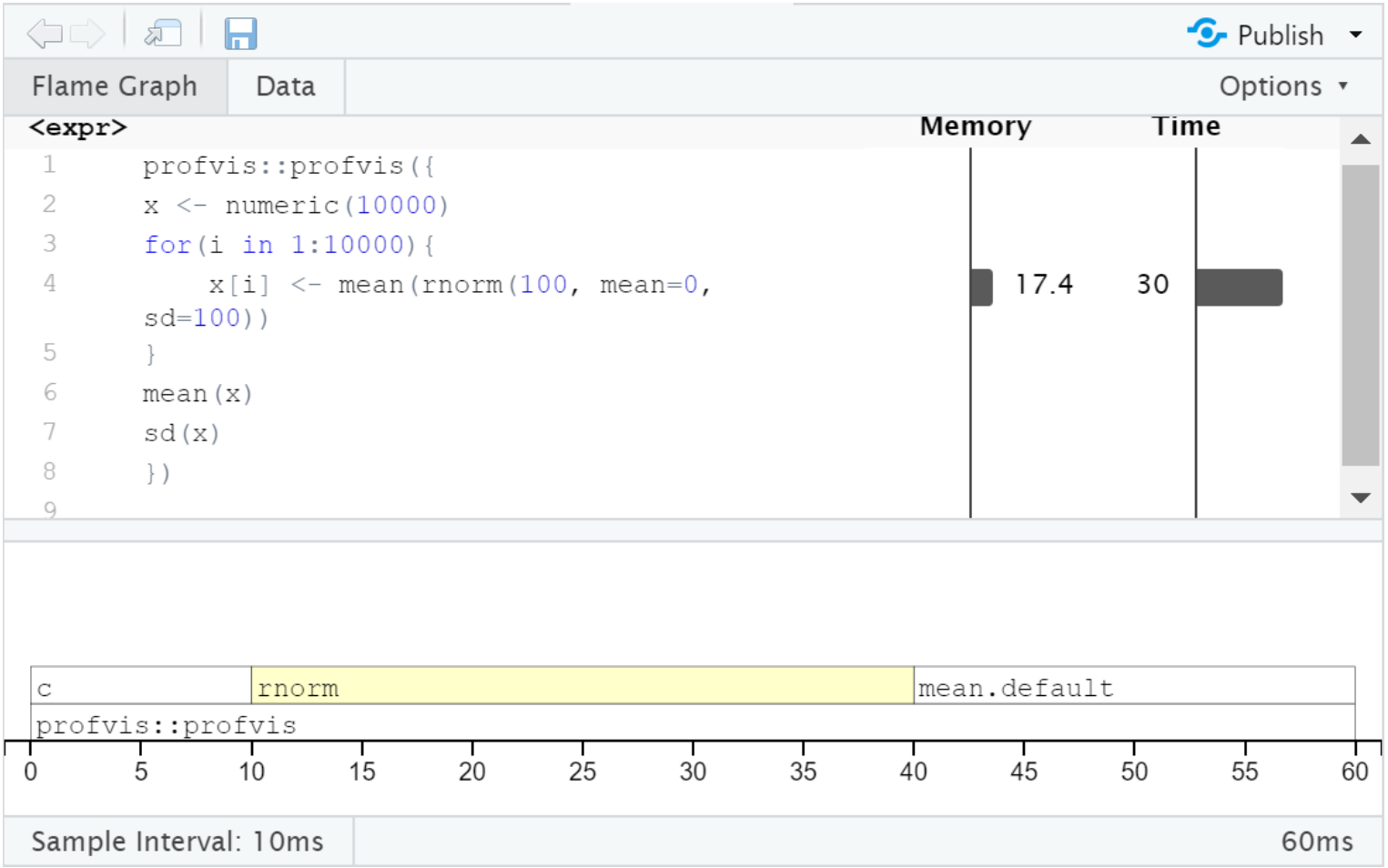
Sample Interval: 10ms | 18440ms

Example 1 - Problems

We identify that the use of `c()` is particularly problematic. Let's fix with a pre- allocated vector.

```
x <- numeric(10000)
for(i in 1:10000){
    x[i] <- mean(rnorm(100, mean=0, sd=100))
}
mean(x)
sd(x)
```

Example 1 - Optimized



Example 1 - skip the `for` loop

```
x <- colMeans(
     matrix(
          rnorm(100*10000, mean=0, sd=100),
          nrow=100
     )
)
mean(x)
sd(x)
```

Results take approx. 30ms which is too short for profiling to properly measure.

Example 2- Something more complex

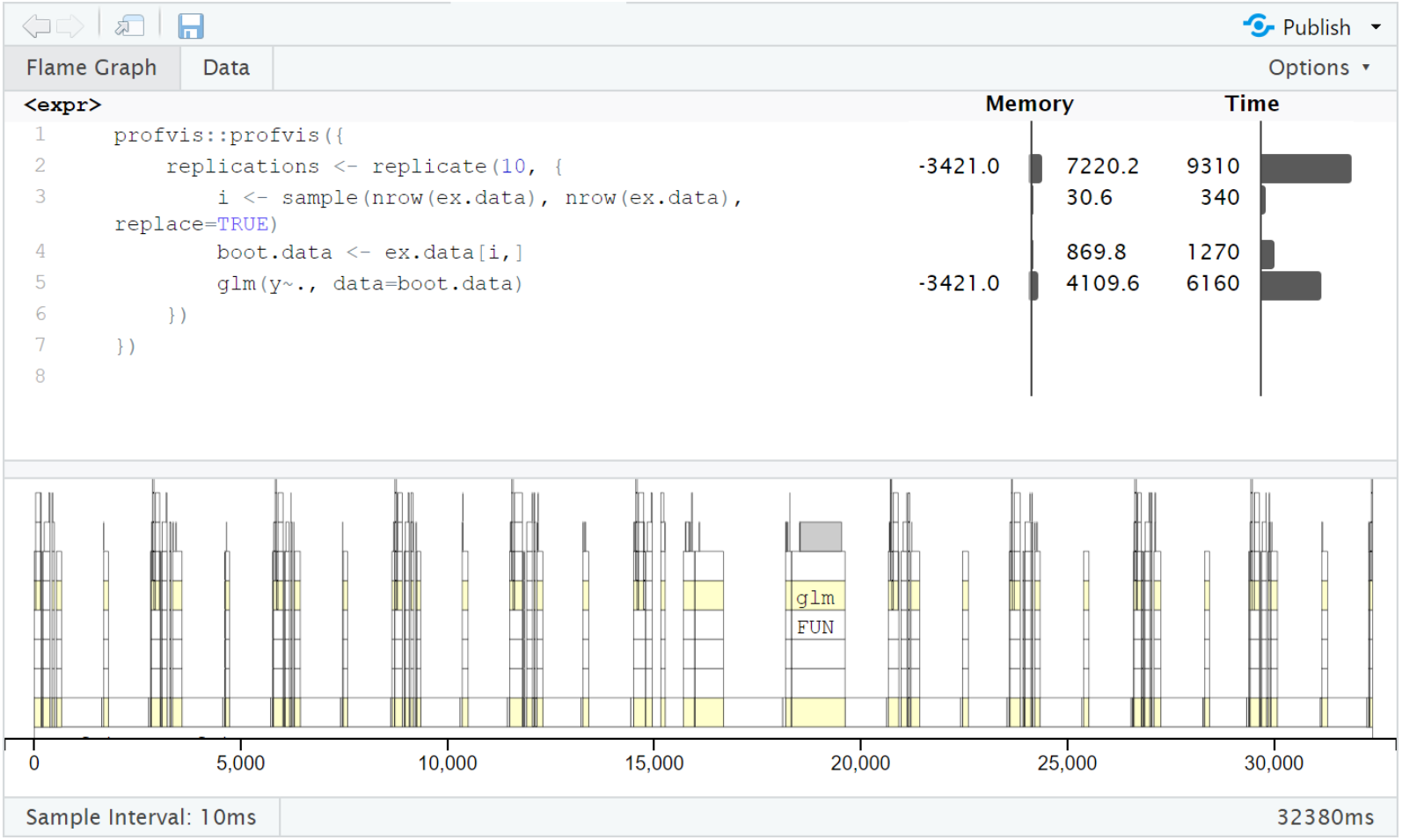
This example is inspired by recent requests to the VINCI help desk.

- Bootstrap 500+ times
- GLM model
- 1M+ rows

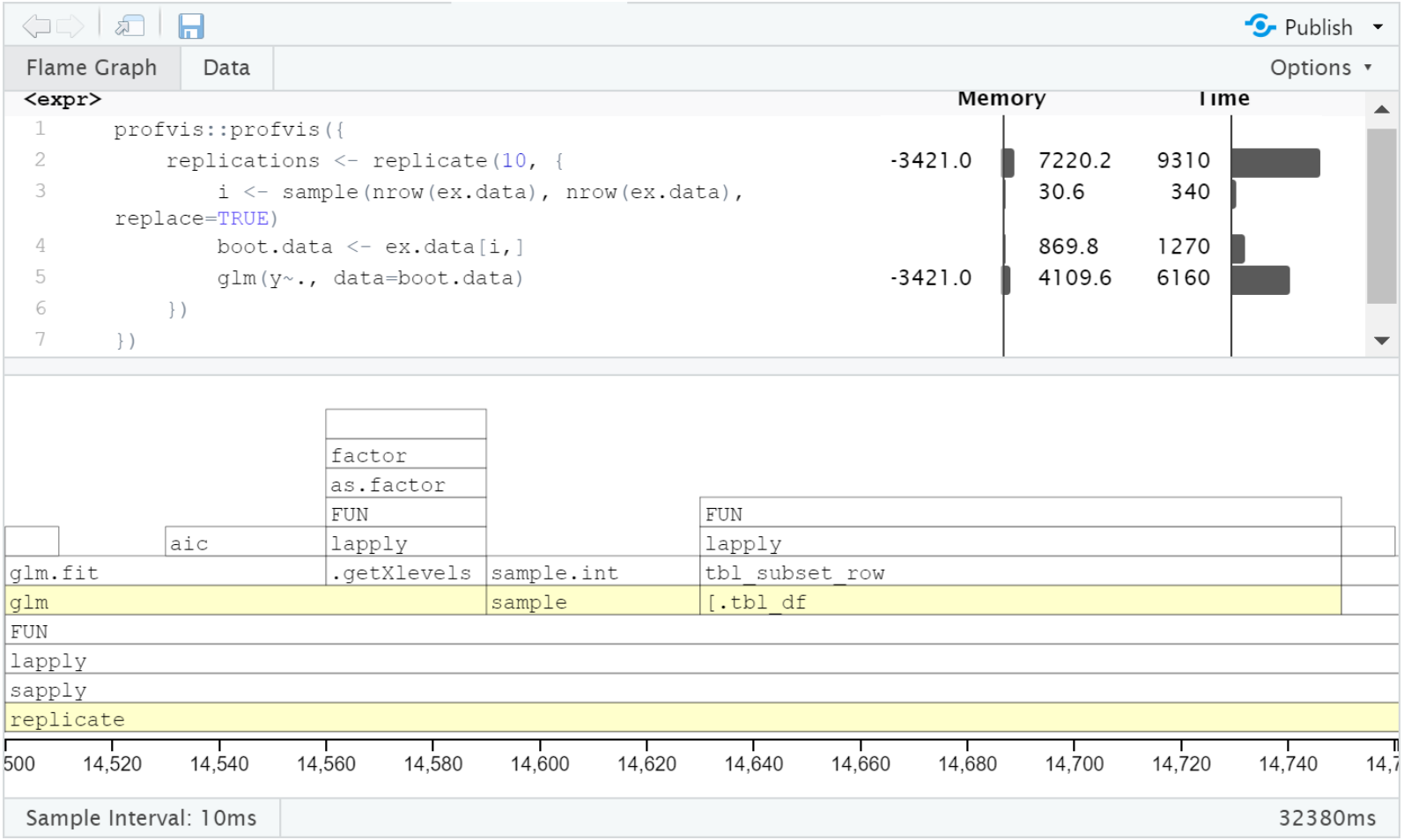
Example 2 - Code

```
profvis::profvis({
    replications <- replicate(10, {
        i <- sample(nrow(ex.data), nrow(ex.data), replace=TRUE)
        boot.data <- ex.data[i,]
        glm(y~., data=boot.data)
    })
})
```

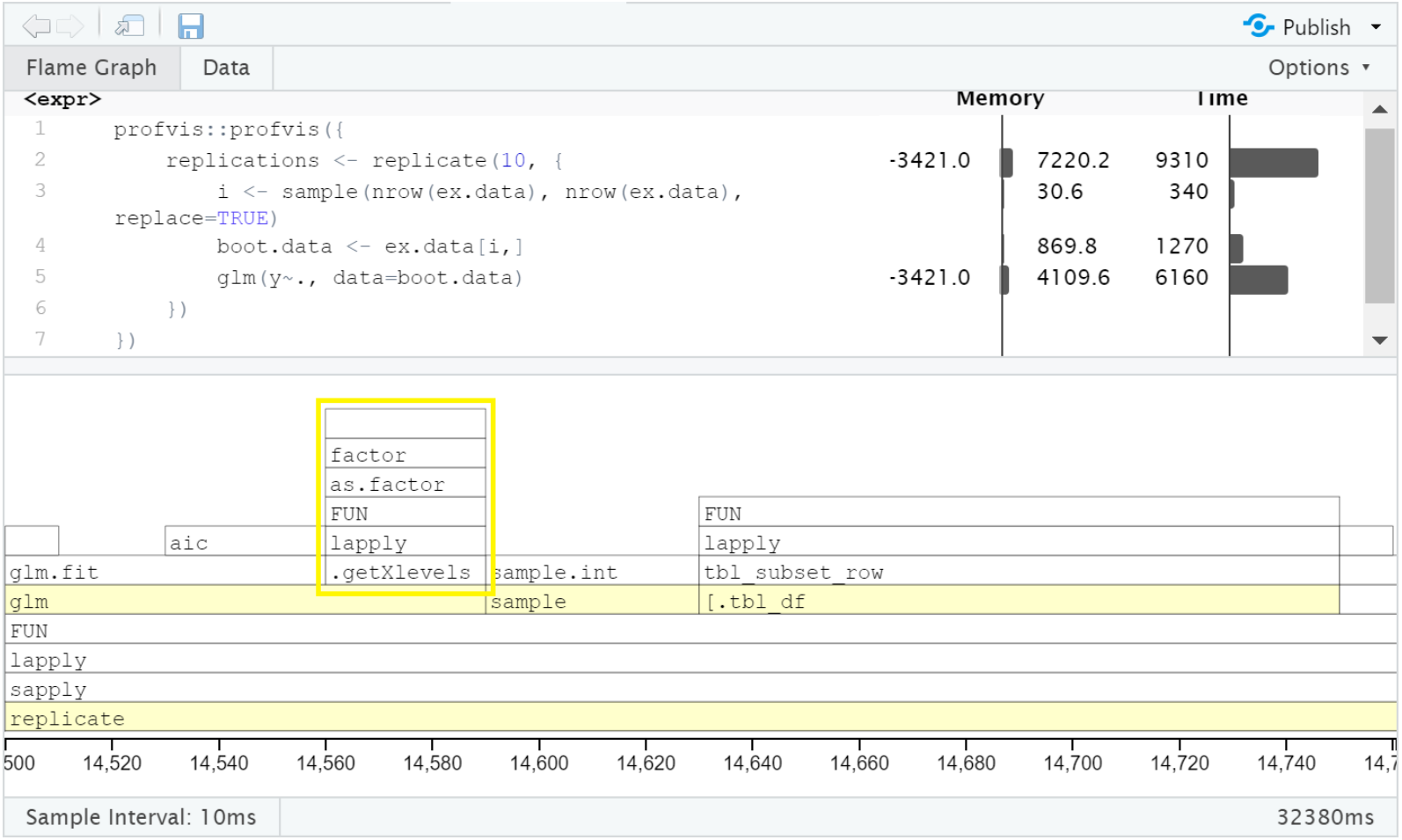
Example 2 - Profile 1



Example 2 - Zoomed in



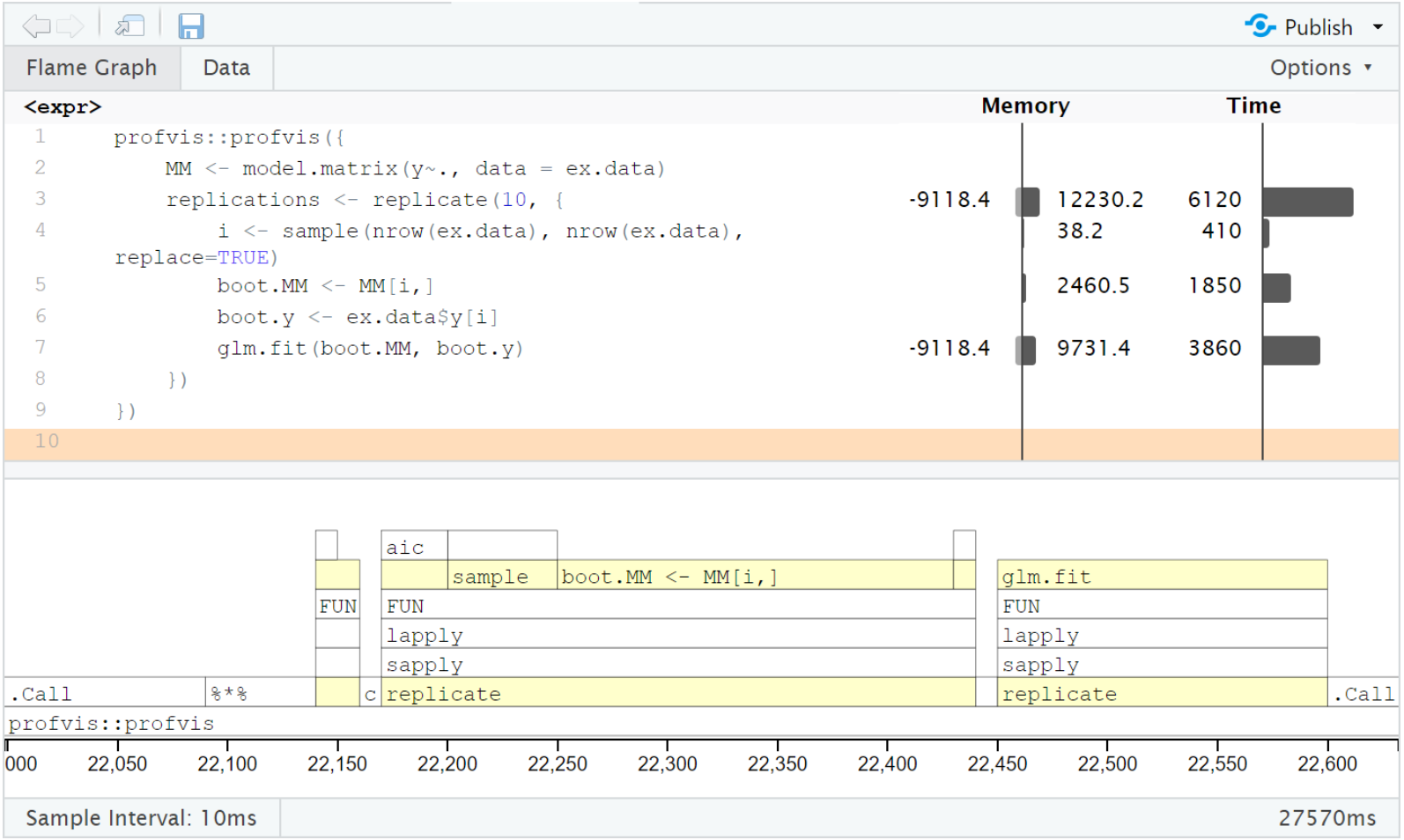
Example 2 - Zoomed in



Example 2 - Fixed as.factor

```
profvis::profvis({
    MM <- model.matrix(y~., data = ex.data)
    replications <- replicate(10, {
        i <- sample(nrow(ex.data), nrow(ex.data), replace=TRUE)
        boot.MM <- MM[i,]
        boot.y <- ex.data$y[i]
        glm.fit(boot.MM, boot.y)
    })
})
```

Example 2 - Profile 2



Example 2 - Fixed indexing

```
profvis::profvis({
    MM <- model.matrix(y~., data = ex.data)
    replications <- replicate(10, {
        weights <- rmultinom(1, nrow(MM), rep(1,nrow(MM)))
        glm.fit(MM, ex.data$y, weights)
    })
})
```

Example 2 - Profile 3

Example 2 - Summary

What we did

- `glm()` → `glm.fit()`
- tibble → matrix
- weights not rows

Results

- 32,380 ms → 18,440 ms (57% of original)
- Memory usage approximately the same.

More info

- VINCI R Academy
  - https://vincicentral.med.va.gov/SitePages/VINCI_University-R_Academy.aspx
- https://rstudio.github.io/profvis/