

## **4.1. Генерация псевдослучайной выборки для заданного распределения. Моделирование данных с заданными статистическими характеристиками.**

### **Задание**

***Выполнить все команды и упражнения из разделов 4.1.1-4.1.4.***

#### ***4.1.1. Базовый пример***

Если Вы хотите получать одинаковые результаты от генератора случайных чисел, необходимо указать стартовую позицию. Это важно, чтобы можно было повторить результаты имитации и для отладки кода.

Попробуйте выполнить команды:

```
>vecpoisson=rpois(100,5)
>mean(vecpoisson)
## [1] 4.74
```

Если это делать много раз, Вы получите разные значения. Если рассмотреть выборку полученных значений, то Вы получите выборку распределения средних значений 100 случайных переменных для распределения Пуассона.

Чтобы выдавался один и тот же набор случайных чисел используйте функцию `set.seed()`.

```
>set.seed(198911)
>vecpoisson=rpois(100,5)
>mean(vecpoisson)
## [1] 4.8
>set.seed(198911)
>vecpoisson=rpois(100,5)
>mean(vecpoisson)
## [1] 4.8
```

#### ***4.1.2. Применение функции `apply()`***

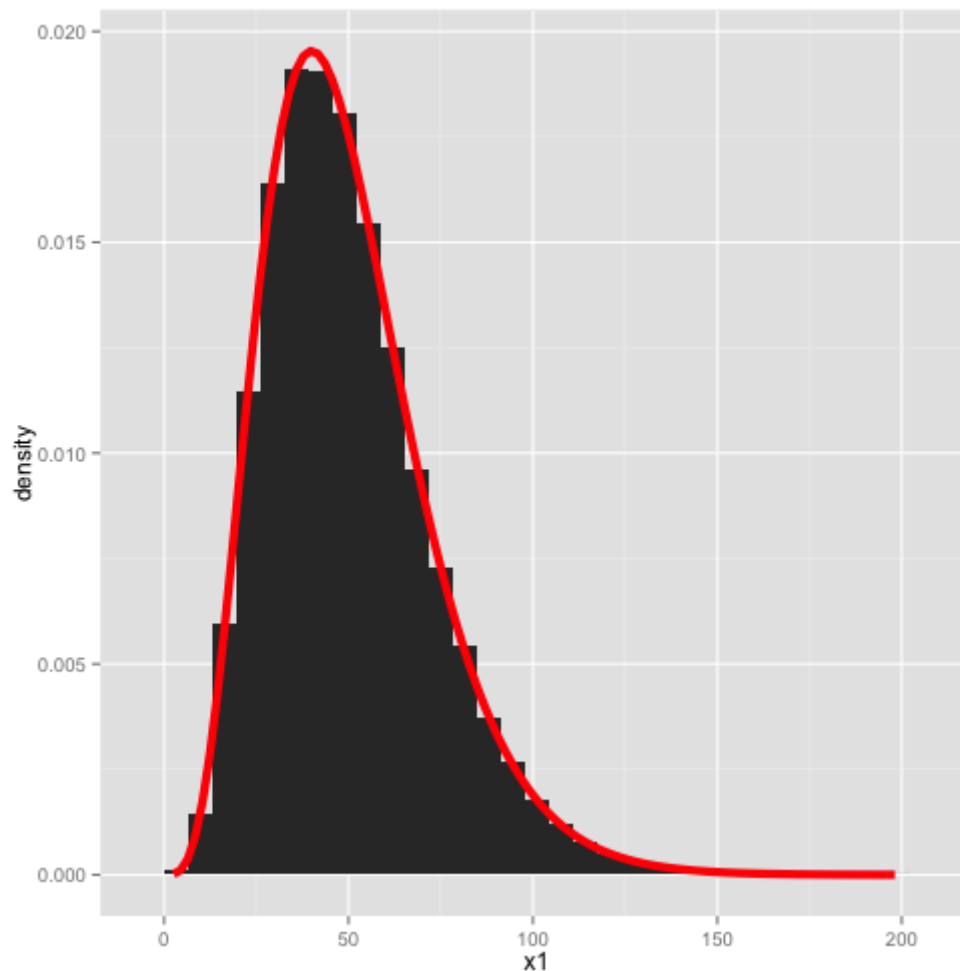
Чтобы не использовать медленные циклы в R для имитационного моделирования лучше использовать такие функции как `apply()`.

Сгенерируем 5 случайных значений экспоненциального распределения с параметром 0.1 и сложим их вместе: `sum(rexp(n=nexps, rate=rate))`. Известно, что сумма экспоненциально распределенных случайных величин дает гамма-распределение. Функция `replicate()` позволяет выполнять одни и те же действия, не увеличивая слишком код. Ниже выполним 10000 раз наборов из 5 чисел.

```
reps <- 50000
nexps <- 5
rate <- 0.1
set.seed(0)
system.time(
  x1 <- replicate(reps, sum(rexp(n=nexps, rate=rate)))
) # replicate
##      user  system elapsed
##  0.304    0.003    0.307
head(x1)
## [1] 38.01 42.64 85.13 58.21 16.67 81.43
```

Чтобы проверить получение гамма-распределения, построим гистограмму распределения `qqplot`.

```
require(ggplot2)
ggplot(data.frame(x1), aes(x1)) +
  geom_histogram(aes(y=..density..)) +
  stat_function(fun=function(x) dgamma(x, shape=nexps, scale=1/rate),
               color="red", size=2)
plot of chunk unnamed-chunk-6
```



Можно сделать проверку гамма-распределения можно использовать другие команды, откройте справку `help(replicate)` и Вы увидите, что есть такие функции как `sapply()`, `lapply()` и `vapply()`. Они связаны в свою очередь с функциями `apply()` и `tapply()`.

Код с функцией `sapply()` будет следующим:

```
set.seed(0)

system.time(x1 <- sapply(1:reps, function(i){sum(rexp(n=nexps, rate=rate))}))
# simple apply

##      user  system elapsed
## 0.277    0.017    0.295

head(x1)

## [1] 38.01 42.64 85.13 58.21 16.67 81.43
```

`lapply()`:

```
set.seed(0)

system.time(x1 <- lapply(1:reps, function(i){sum(rexp(n=nexps, rate=rate))}))
# list apply

##      user  system elapsed
```

```
##    0.231    0.001    0.232
head(x1)
## [[1]]
## [1] 38.01
##
## [[2]]
## [1] 42.64
##
## [[3]]
## [1] 85.13
##
## [[4]]
## [1] 58.21
##
## [[5]]
## [1] 16.67
##
## [[6]]
## [1] 81.43
```

Если мы применяем простую функцию как `sum()`, как правило, лучше сделать матрицу со всеми данными имитации и применить данную функцию к соответствующим частям матрицы. Функции `rowSums()` и `colSums()` демонстрируют особую эффективность в этом случае.

```
set.seed(0)
system.time(x1 <- apply(matrix(rexp(n=nexps*reps, rate=rate), nrow=nexps), 2, sum))
# apply on a matrix
##      user  system elapsed
##    0.114    0.003    0.117
head(x1)
## [1] 38.01 42.64 85.13 58.21 16.67 81.43
set.seed(0)
system.time(x1 <- colSums(matrix(rexp(n=nexps*reps, rate=rate), nrow=nexps)))
# using colSums
##      user  system elapsed
##    0.018    0.001    0.018
head(x1)
## [1] 38.01 42.64 85.13 58.21 16.67 81.43
```

Если у Вас многоядерный процессор, то скорость обработки можно повысить посредством параллельной обработки данных. В пакете `parallel` есть функция `mclapply()`, которая является параллельным вариантом `lapply()`. По умолчанию используется один процессор, для изменения числа процессор измените `mc.cores`.

```
require(parallel)

set.seed(0)

system.time(x1 <- mclapply(1:reps, function(i){sum(rexp(n=nexps,
rate=rate))})) # multi-cluster apply

##      user  system elapsed
##    0.385    0.095    0.250

head(x1)

## [[1]]
## [1] 28.92
##
## [[2]]
## [1] 67.89
##
## [[3]]
## [1] 26.47
##
## [[4]]
## [1] 37.11
##
## [[5]]
## [1] 31.21
##
## [[6]]
## [1] 24.9
```

### ***4.1.3 Генерация нормально-распределенных случайных величин***

Пример получения 1000 случайных значений нормального распределения  $N(0,1)$ :

```
samples = rnorm(1000, 0, 1)
```

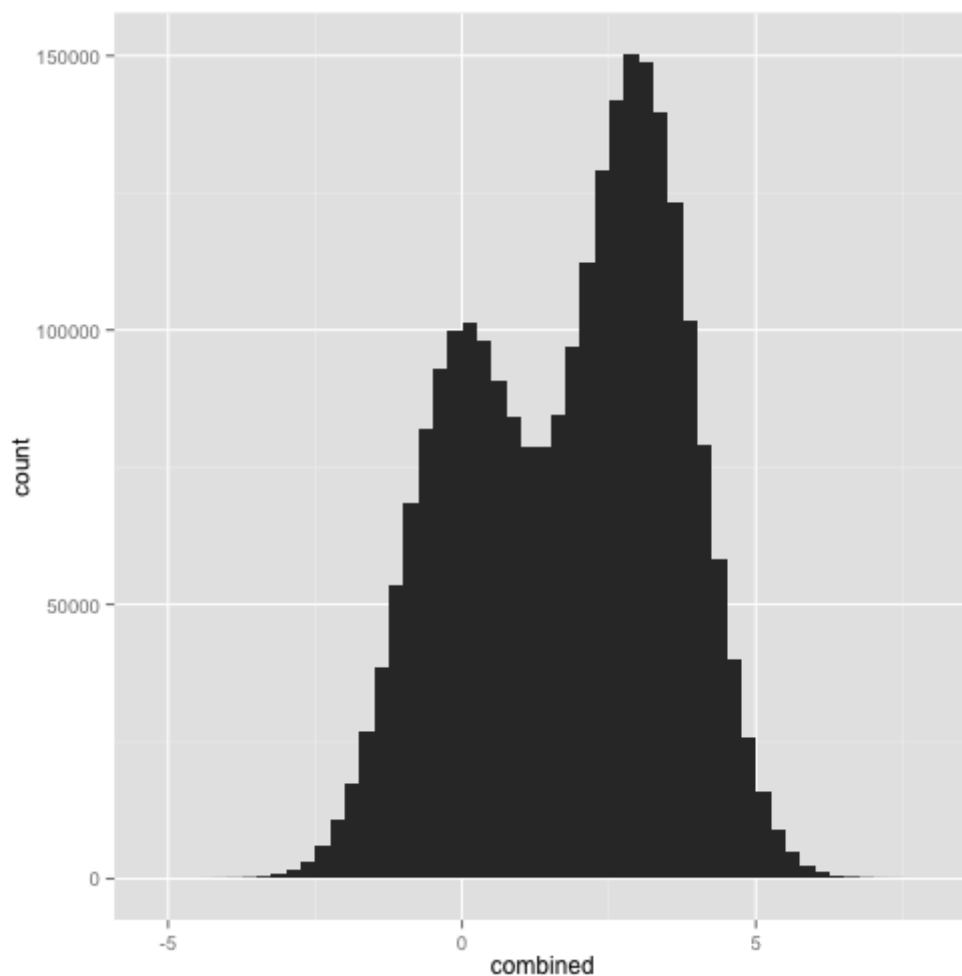
Возьмем случайную выборку из 1,000,000 значений для нормального распределения с математическим ожиданием 0 и стандартным

отклонением 1  $N(0,1)$  и выборку размером 1,500,000  $N(3,1)$ . Построим гистограмму с шириной столбца 0.25:

```
require(ggplot2)
sampa=rnorm(1000000,0,1)
sampb=rnorm(1500000,3,1)
combined = c(sampa, sampb)

plt = ggplot(data.frame(combined), aes(x=combined)) +
  stat_bin(binwidth=0.25, position="identity")

plt
```



Упражнение. Ответьте на вопросы

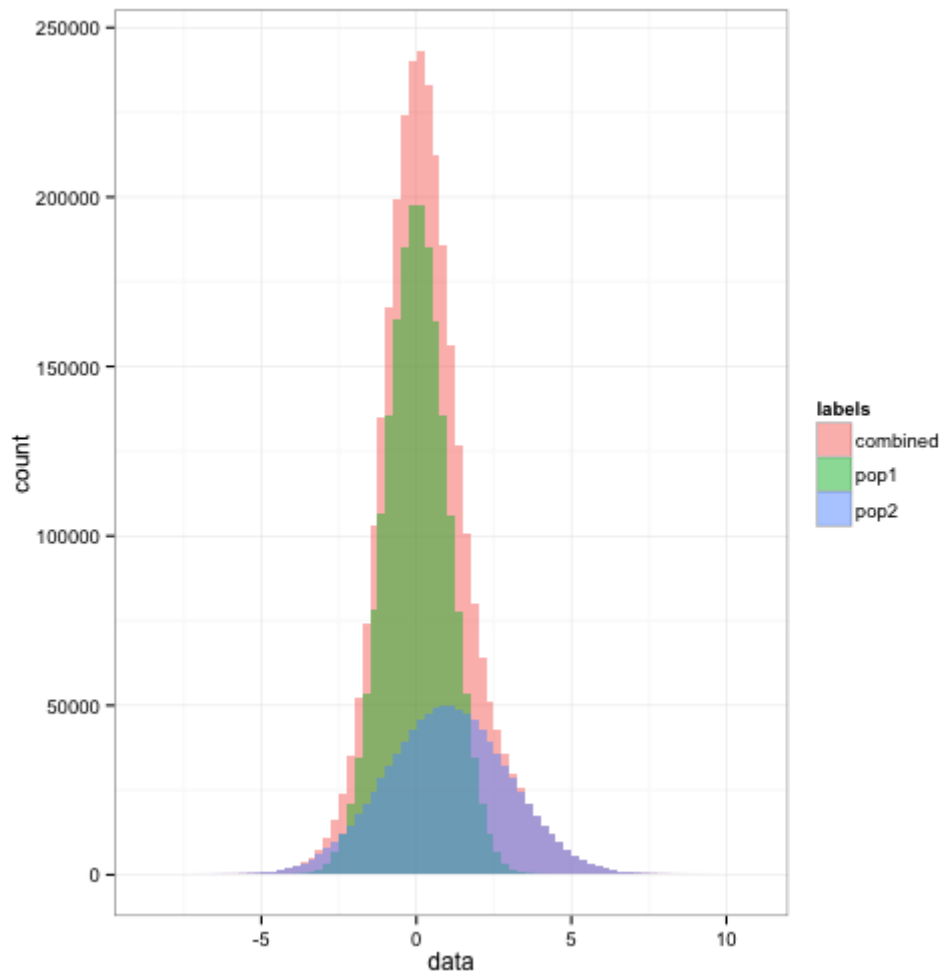
- Сколько у данного распределения мод. Почему так произошло?
- Нарисуйте график распределения частот, используя `geom_density()`.

Теперь построим 3 000 000 точек из смеси двух случайных выборок для нормального распределения.

```
pop1=rnorm(2000000)
pop2=rnorm(1000000, 1, 2)
combined = c(pop1, pop2)

plt= ggplot(data.frame(data=c(combined, pop1, pop2), labels=rep(c("combined", "pop1",
"pop2"), c(3e6, 2e6, 1e6))), aes(x=data)) + stat_bin(aes(fill=labels),
position="identity", binwidth=0.25, alpha=0.5) + theme_bw()

plt
```



Упражнение. Ответьте на вопросы

3) Какое отличие от предыдущей гистограммы.

4) Постройте диаграмму частот для pop1 и pop2 на одном графике.

Сгенерируем два многомерных нормальных кластера из 100 000 точек каждый:

```
require(MASS)
```

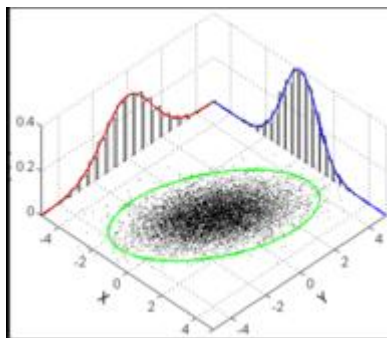
```

Sigma=matrix(c(5,3,3,2),2,2)
ex1=mvrnorm(100000,rep(0,2),Sigma)
Sigma=matrix(c(9,-5,-1,5),2,2)
ex2=mvrnorm(n=100000, rep(3, 2), Sigma)

```

### Упражнение

5) Постройте функции частот распределения.



Для этого используйте `geom_point()` и `geom_density2d()`.

#### **4.1.4. Метод Монте-Карло**

Мы покажем, как вычислить вероятность простых событий с помощью имитационного моделирования методом Монте-Карло.

Предположим, мы бросили два кубика. Какова вероятность, что их сумма не меньше 7? Мы подойдем к этому, моделируя множество бросков двух честных игральных костей, а затем вычисляя долю тех испытаний, сумма которых равна не менее 7. Будет удобно написать функцию, которая имитирует испытания и возвращает TRUE, если сумма не менее 7 (мы называем это событием), в противном случае – FALSE (функция `sample()` возвращает случайную выборку).

```

isEvent = function(numDice, numSides, targetValue, numTrials){
  apply(matrix(sample(1:numSides, numDice*numTrials, replace=TRUE), nrow=numDice),
+2, sum) >= targetValue
}

```

Мы разработали функцию, будем применять её для метода Монте-Карло для 5 попыток.

```
set.seed(0)
```



```
#try 5 trials
outcomes = isEvent(2, 6, 7, 5)
mean(outcomes)
## [1] 1
```

Теоретически ответ должен быть  $28/36$ , т.е. около 0.6; наш ответ 1 слишком далек. Попробуем сделать 10 000 испытаний:

```
set.seed(0)
outcomes = isEvent(2, 6, 7, 10000)
mean(outcomes)
## [1] 0.5843
```

Как видим результат приблизился к теоретическому. В методе Монте-Карло используются простые вычисления, но их требуется достаточно много, поэтому лучше их распараллелить с помощью пакета `parallel`.

```
require(parallel)
```

Используем функцию `pvec()`, для этого сделаем пользовательскую функцию `isEventPar()`.

```
isEventPar = function(numDice, numSides, targetValue, trialIndices){
  sapply(1:length(trialIndices), function(x) sum(sample(1:numSides, numDice,
replace=TRUE)) >= targetValue)
}
```

Вызовем `pvec()` :

```
set.seed(0)
outcomes = pvec(1:10000, function(x) isEventPar(2, 6, 7, x))
mean(outcomes)
## [1] 0.5931
```

Результаты параллельного и обычного алгоритмов практически не отличаются за счет использования `set.seed(0)`.

## 4.2. Повышение эффективности работы алгоритмов

### Задание

**Выполнить все команды и упражнения из разделов 4.2.1-4.2.3.**

#### 4.2.1. Использование утилиты Rprof

Мы будем использовать библиотеки `profvis` и `bench`.

```
library(profvis)
```

```
library(bench)
```

Рассмотрим следующую функцию `f()`

```
f <- function() {  
  pause(0.1)  
  g()  
  h()  
}
```

```
g <- function() {  
  pause(0.1)  
  h()  
}
```

```
h <- function() {  
  pause(0.1)  
}
```

Использована функция `pause` из библиотеки `profvis`.

Если мы будем профилировать код для `f()`, останавливаясь каждые 0.1 сек, то мы увидели бы следующую выдачу

```
"pause" "f"  
"pause" "g" "f"  
"pause" "h" "g" "f"  
"pause" "h" "f"
```

Из этой выдачи было бы понятно, что код тратит 0.1 с на непосредственное выполнение `f()`, 0.2 с на `g()` и 0.1 с на `h()`.

Если использовать утилиту `Rprof()`, то мы получим следующее:

```
tmp <- tempfile()
Rprof(tmp, interval = 0.1)
f()
Rprof(NULL)
writeLines(readLines(tmp))
#> sample.interval=100000
#> "pause" "g" "f"
#> "pause" "h" "g" "f"
#> "pause" "h" "f"
```

При работе профилировщик замедляет выполнение кода, поэтому при профилировании необходимо обеспечить баланс между точностью выдачи профилировщика (а это требует вмешательство в работу программы) и замедлением из-за вмешательства. Поэтому каждый раз результаты профилирования немного отличаются друг от друга.

Профилировщик `Rprof()` создает достаточно большой объем выборки (больше 100 запусков) для функции, которая работает пару секунд. В то же время не хватает визуализации, которую обеспечивает библиотека `profvis`.

#### ***4.2.2. Визуализация процесса профилирования***

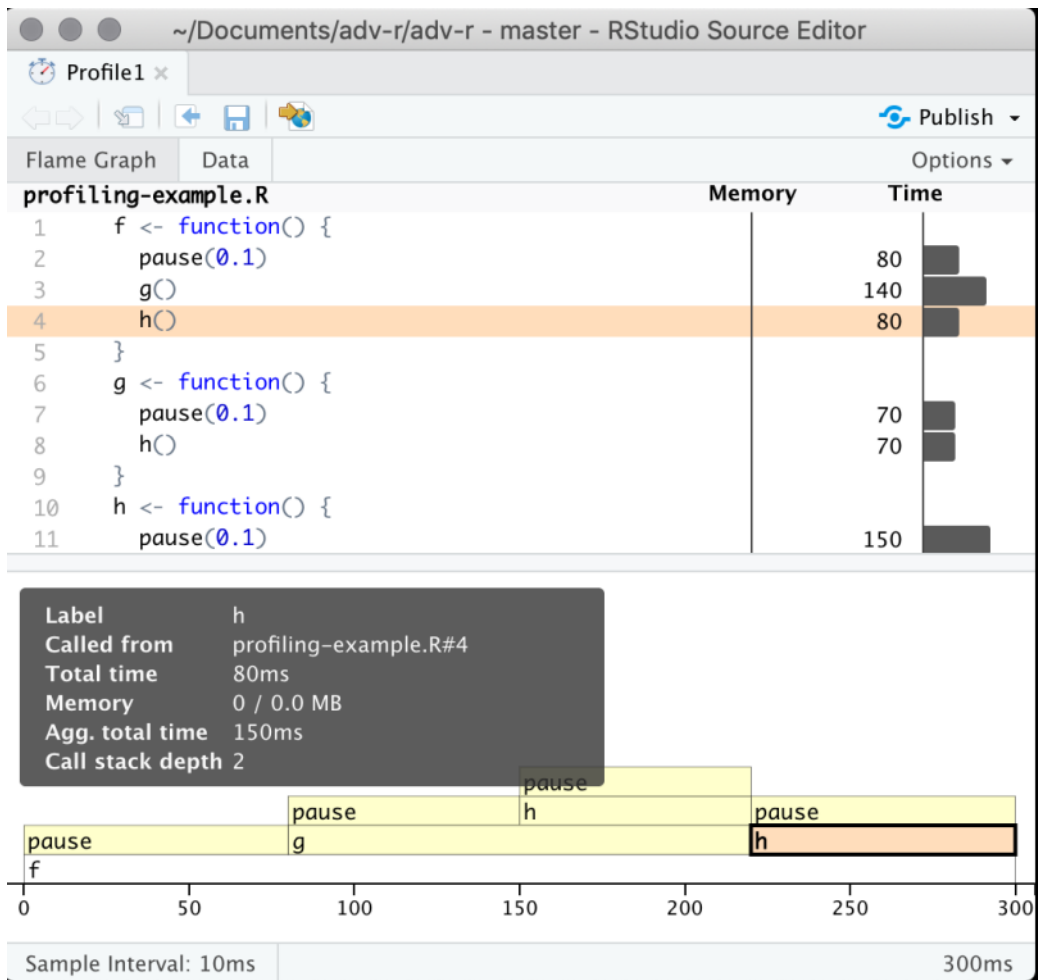
Запустить `profvis` можно 2 способами:

1. Из меню Profile в RStudio.
2. Подключить функцию `profvis()` из библиотеки `profvis`. Лучше код программы сохранить в файле `.r`, а затем сослаться на этот файл при профилировании.

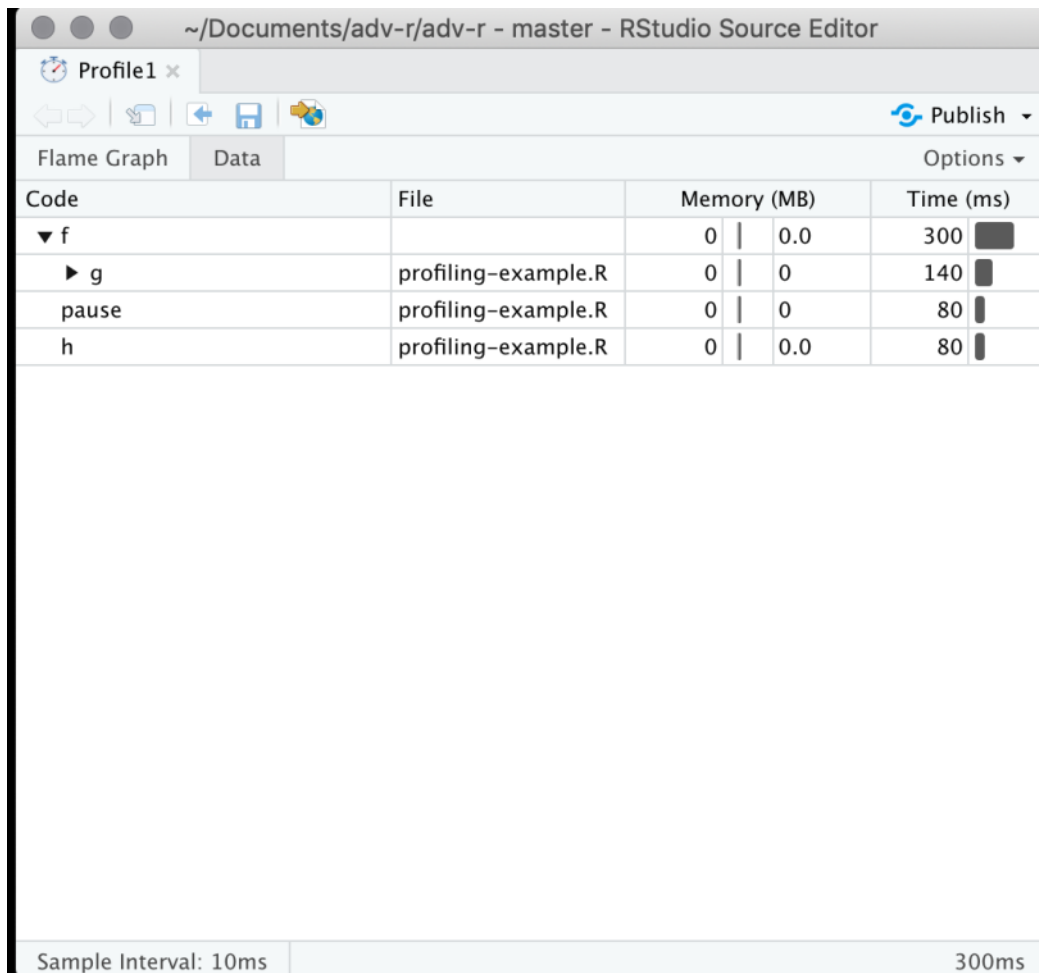
```
source("profiling-example.R")
profvis(f())
```

После профилирования `profvis` откроет интерактивный html-документ, где можно изучить результаты профилирования. Первая активная вкладка - это Flame Graph.





Также есть вкладка о данных Data. Это по сути дело другое представление данных из Flame Graph для больших кодов.

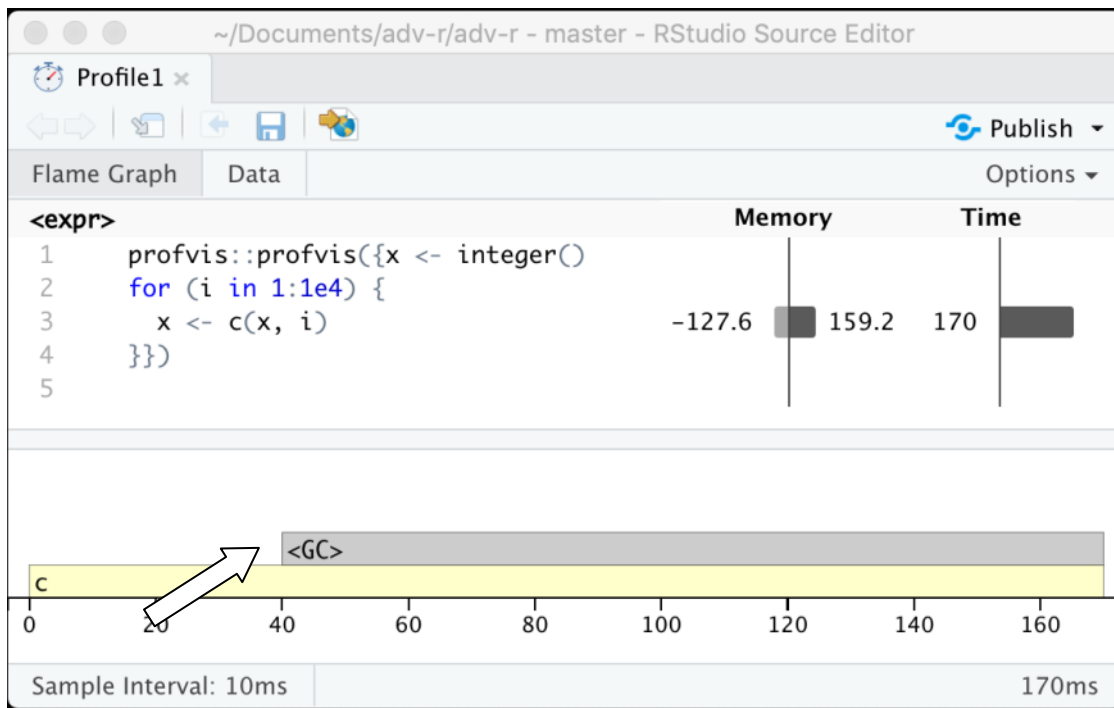


Во вкладке flame graph внизу Вы можете увидеть особый вид вызова <GC>, это Garbage Collector – сборщик мусора. Если <GC> отнимет много времени работы программы, то у Вас много копирования мало живущих объектов данных.

Создайте код.

```
x <- integer()
for (i in 1:1e4) {
  x <- c(x, i)
}
```

Выполните его профилизацию и Вы увидите, что большая часть времени ушла на сбор мусора.



Можно увидеть, что программы использовала еще также много памяти, которая то заполнялась (до 159.2), а потом освобождалась (127.6). Проблем в том, что использовался цикл `for`, который создавал каждый раз новый объект для `x`.

### Упражнение

Выполните профилизацию данного кода

```
f <- function(n = 1e5) {
  x <- rep(1, n)
  rm(x)
}
```

### **4.2.3. Микротесты (Microbenchmarking)**

Микротесты оценивают эффективность маленького кусочка кода, на работу которого от наносекунд (ns) до микро (µs) или миллисекунд (ms). Для этого можно использовать пакет (библиотеку) `bench`.

```
x <- runif(100)
(lb <- bench::mark(
  sqrt(x),
  x ^ 0.5
))
#> # 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)      865ns   1.05µs   679021.    848B         0
#> 2 x^0.5        3.78µs   4.17µs   203205.    848B         0
```

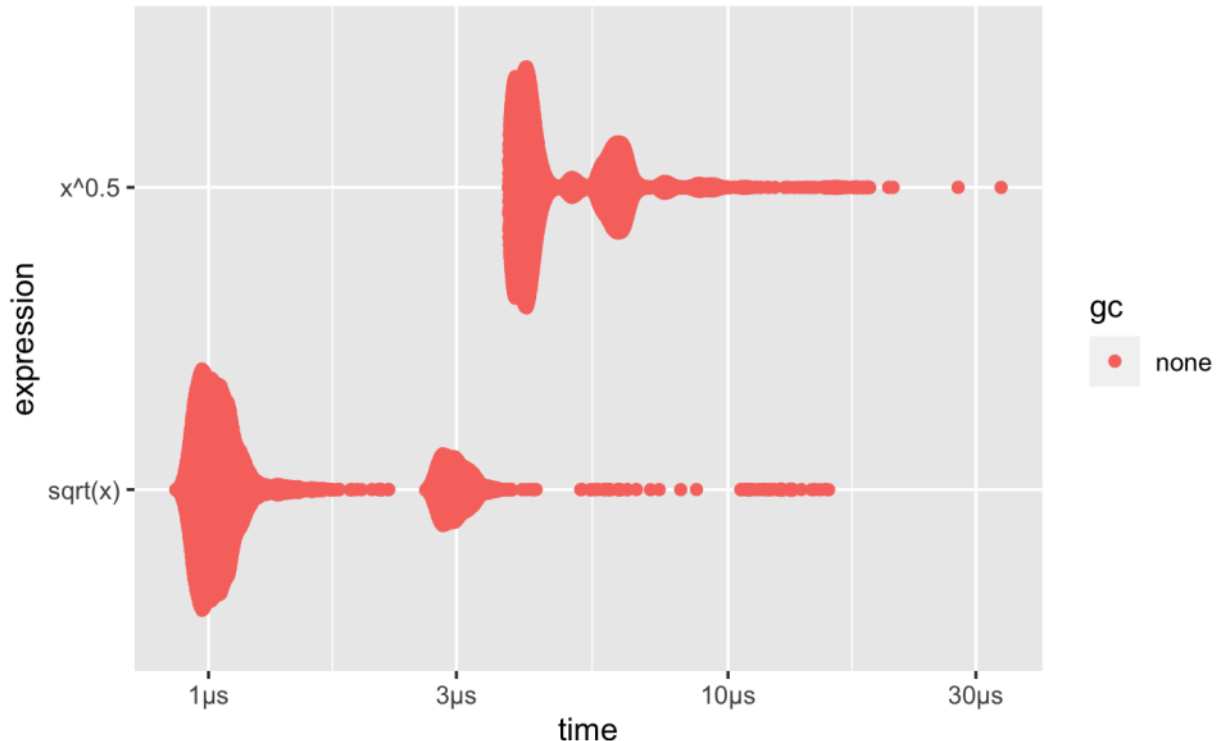
По умолчанию `bench::mark()` выполняет каждое выражение минимум 1 раз (`min_iterations = 1`), столько итераций сколько можно выполнить за 0.5 секунд (`min_time = 0.5`). Функция `mark()` проверяет, чтобы каждый запуск программы возвращал одно и то же значение. Если необходимо, сравнивать скорость работы при разных возвращаемых значениях установите `check = FALSE`.

Функция `bench::mark()` предоставляет результаты в виде фрейма данных формата `Tibble` с данными о минимальном, медианном времени выполнения, число итераций в секунду, данные об использовании памяти и сборщика мусора, а также другие значения.

Визуализируйте результирующие данные `bench::mark()` с помощью `plot()`.

```
plot(lb)
```

```
#> Loading required namespace: tidyrr
```



Можно увидеть мультимодальность распределения времени выполнения и смещенность вправо. Часто такой график связан с работой других программ на компьютере.



### Упражнение

Проведите microbenchmarking для двух функций:

$$x^{1/2}$$

$$\exp(\log(x)/2)$$