Chapter 1

Optimizing performance

Abstract

In this chapter we explain how to make your photobiology calculations execute as fast as possible. The code has been profiled and the performance bottlenecks removed in most cases the implementing some functions in C++. Furthermore copying of spectra is minimized by using package `data.table` as the base class of all objects where spectral data is stored. However, it is possible to improve performance even more by changing some defaults and writing efficient user code. This is what is discussed in the present chapter, and should not be of concern unless several thousands of spectra need to be processed.

1.1 Packages used in this chapter

For executing the examples listed in this chapter you need first to load the following packages from the library:

```
library(photobiology)

## Loading required package:  lubridate

library(photobiologyWavebands)
library(microbenchmark)
```

Although not a recommended practice, just to keep the examples shorter, we `attach` a data set for the solar spectrum:

1.2 Introduction

When developing the current version of photobiology quite a lot of effort was spent in optimizing performance, as in one of our experiments, we need to process several hundreds of thousands of measured spectra. The defaults should provide good performance in most cases, however, some further improvements are achievable, when a series of different calculations are done on the same spectrum, or when a series of spectra measured at exactly the same wavelengths are used for calculating weighted irradiances or exposures.

There is also a lot you can achieve by carefully writing the code in your own scripts. The packages themselves are fairly well optimized for speed. In your

own code try to avoid unnecessary copying of big objects. The r4photobiology suite makes extensive use of the `data.table` package, using it also in your own code could help. Try to avoid use of explicit loops by replacing them with vectorized operations, and when sequentially building vectors in a loop, preallocate an object big enough before entering the loop.

Being R an interpreted language, there is rather little automatic code optimization taking place, so you may find that even simple things like moving invariant calculations out of loops, and avoiding repeated calculations of the same value by storing the value in a variable can improve performance.

This type of 'good style' optimizations have been done throughout the suite's code, and more specific problem identified by profiling and and dealt with case by case. Of course, to achieve maximum overall performance, to should follow the same approach with your own code.

1.3 Task: avoiding repeated validation

In the case of doing calculations repeatedly on the same spectrum, a small improvement in performance can be achieved by setting the parameter `check.spectrum=FALSE` for all but the first call to `irradiance()`, or `photon_irradiance()`, or `energy_irradiance()`, or the equivalent functions for ratios. It is also possible to set this parameter to FALSE in all calls, and do the check beforehand by explicitly calling `check_spectrum()`.

1.4 Task: caching of multipliers

In the case of calculating weighted irradiances on many spectra having exactly the same wavelength values, then a significant improvement in the performance can be achieved by setting `use.cached.mult=TRUE`, as this reuses the multipliers calculated during successive calls based on the same waveband. However, to achieve this increase in performance, the tests to ensure that the wavelength values have not changed, have to be kept to the minimum. Currently only the length of the wavelength array is checked, and the cached values discarded and recalculated if the length changes. For this reason, this is not the default, and when using caching the user is responsible for making sure that the array of wavelengths has not changed between calls.

1.5 Task: benchmarking

You can use the package `microbenchmark` to time the code and find the parts that slow it down. I have used it, and also I have used profiling to optimize the code for speed. The examples below show how choosing different values from the defaults can speed up calculations when the same calculations are done repeatedly on spectra measured at exactly the same wavelengths, something which is usual when analyzing spectra measured with the same instrument. The choice of defaults is based on what is best when processing a moderate number of spectra, say less than a few hundreds, as opposed to many thousands.

```
attach(sun.spct)
```

Convenience functions

The convenience functions are slightly slower than the generic `irradiance` function.

```
res001 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, PAR(), use.cache=TRUE),
  times=100L, control=list(warmup = 10L))
res002 <- microbenchmark(
  irradiance(w.length, s.e.irrad, PAR(), unit.out="photon",
             use.cache=TRUE),
  times=100L, control=list(warmup = 10L))
```

Using the generic reduces the median execution time from 0.104 ms to 0.0969 ms, by 6.9% if using the cache.

Using cached multipliers

Using the cache when repeatedly applying the same waveband has a large impact on the execution time.

```
res011 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, PAR()),
  times=100L, control=list(warmup = 10L))
res012 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, PAR(), use.cache=TRUE),
  times=100L, control=list(warmup = 10L))
```

When using an unweighted waveband the cache reduces the median execution time from 0.162 ms to 0.1 ms, by 38%.

When using BSWFs the speed up by use of the cache is more important, and dependent on the complexity of the equation used in the calculation.

```
res021 <- microbenchmark(
  energy_irradiance(w.length, s.e.irrad, CIE()),
  times=100L, control=list(warmup = 10L))
res022 <- microbenchmark(
  energy_irradiance(w.length, s.e.irrad, CIE(), use.cache=TRUE),
  times=100L, control=list(warmup = 10L))
```

When using a weighted waveband, in this example, `CIE()`, the cache reduces the median execution time from 0.294 ms to 0.109 ms, by 63%.

Disabling checks

Disabling the checking of the spectrum halves once again the execution time for unweighted wavebands.

```
res031 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, PAR(), use.cache=TRUE),
  times=100L, control=list(warmup = 10L))
res032 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, PAR(), use.cache=TRUE,
                    check.spectrum=FALSE),
  times=100L, control=list(warmup = 10L))
```

When using an unweighted waveband, in this example, `PAR()`, the disabling the data validation checking reduces the median execution time from 0.1 ms to 0.0715 ms, by 29%.

Using stored wavebands

Saving a waveband object and reusing it, can give an additional speed up when all other optimizations are also used.

```
myPAR <- PAR()
res041 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, PAR(), use.cache=TRUE,
                    check.spectrum=FALSE),
  times=100L, control=list(warmup = 10L))
res042 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, myPAR, use.cache=TRUE,
                    check.spectrum=FALSE),
  times=100L, control=list(warmup = 10L))
```

When using an unweighted waveband, in this example, `PAR()`, using a saved waveband object reduces the median execution time from 0.0715 ms to 0.0627 ms, by 12%.

Saving a waveband object that uses weighting and reusing it, gives an additional speed up when all other optimizations are also used.

```
myCIE <- CIE()
res051 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, CIE(), use.cache=TRUE,
                    check.spectrum=FALSE),
  times=100L, control=list(warmup = 10L))
res052 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, myCIE, use.cache=TRUE,
                    check.spectrum=FALSE),
  times=100L, control=list(warmup = 10L))
```

When using a weighted waveband, in this example, `CIE()`, using a saved waveband object reduces the median execution time from 0.0791 ms to 0.0635 ms, by 20%.

Inserting hinges

Inserting 'hinges' to reduce integration errors slows down the computations considerably. If the spectral data is measured with a small wavelength step, the errors are rather small. By default the use of 'hinges' is automatically decided based on the average wavelength step in the spectral data. The 'cost' of using hinges depends on the waveband definition, as BSWFs with discontinuities in the slope require several hinges, while unweighted one requires at most two, one at each boundary.

```
res061 <- microbenchmark(
  energy_irradiance(w.length, s.e.irrad, PAR(), use.cache=TRUE),
  times=100L, control=list(warmup = 10L))
res062 <- microbenchmark(
  energy_irradiance(w.length, s.e.irrad, PAR(), use.cache=TRUE,
                    use.hinges=TRUE),
  times=100L, control=list(warmup = 10L))
```

When using an uweighted waveband, in this example, `PAR()`, enabling use of hinges increases the median execution time from 0.0996 ms to 0.0635 ms, by a factor of 0.6374.

Inserting 'hinges' to reduce integration errors slows down the computations a lot. If the spectral data is measured with a small wavelength step, the errors are rather small. By default the use of 'hinges' is automatically decided based on the average wavelength step in the spectral data.

```
res071 <- microbenchmark(
  energy_irradiance(w.length, s.e.irrad, CIE(), use.cache=TRUE),
  times=100L, control=list(warmup = 10L))
res072 <- microbenchmark(
  energy_irradiance(w.length, s.e.irrad, CIE(), use.cache=TRUE,
                    use.hinges=TRUE),
  times=100L, control=list(warmup = 10L))
```

When using an weighted waveband, in this example, `CIE()`, enabling use of hinges increases the median execution time from 0.108 ms to 0.394 ms, by a factor of 3.6397.

1.6 Overall speed-up achievable

`GEN.G`

If we consider a slow computation, using a BSWF with a complex equation like `GEN.G`, we can check the best case improvement in throughput that can be —on a given hardware and software system.

```
# slowest
res081 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, GEN.G(),
                    use.cache=FALSE,
                    use.hinges=TRUE,
                    check.spectrum=TRUE),
                      times=100L, control=list(warmup = 10L))
# default
res082 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, GEN.G()),
                       times=100L, control=list(warmup = 10L))

# fastest
gen.g <- GEN.G()
res083 <- microbenchmark(
  irradiance(w.length, s.e.irrad, gen.g,
             use.cache=TRUE,
             use.hinges=FALSE,
             check.spectrum=FALSE,
             unit.out="photon"),
                       times=100L, control=list(warmup = 10L))
```

When using a weighted waveband, in this example, `GEN.G()`, enabling all checks and optimizations for precision, and disabling all optimizations for speed yields a median execution time of 0.546 ms, accepting all defaults yields a median execution time 0.263 ms, and disabling all checks, optimizations for precision and enabling all optimizations for speed yields a median execution

time of 0.0599, in relation to the slowest one, execution times are 100, 48, and 11%.

Finally we compare the returned values for the irradiance, to see the impact on them of optimizing for speed.

```
# slowest
photon_irradiance(w.length, s.e.irrad, GEN.G(),
                  use.cache=FALSE,
                  use.hinges=TRUE,
                  check.spectrum=TRUE)

## GEN.G.300
## 2.579e-07

# default
photon_irradiance(w.length, s.e.irrad, GEN.G())

## GEN.G.300
## 2.592e-07

# fastest
gen.g <- GEN.G()
irradiance(w.length, s.e.irrad, gen.g,
           use.cache=TRUE,
           use.hinges=FALSE,
           check.spectrum=FALSE,
           unit.out="photon")

## GEN.G.300
## 2.592e-07
```

These results are based on spectral data at 1 nm interval, for more densely measured data the effect of not using hinges becomes even smaller. In contrast, with data measured at wider wavelength steps, the errors will be larger. They also depend on the specific BSWF being used.

CIE

If we consider a slow computation, using a BSWF with a complex equation like `CIE`, we can check the best case improvement in throughput that can be —on a given hardware and software system.

```
# slowest
res101 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, CIE(),
                    use.cache=FALSE,
                    use.hinges=TRUE,
                    check.spectrum=TRUE),
                      times=100L, control=list(warmup = 10L))
# default
res102 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, CIE()),
                       times=100L, control=list(warmup = 10L))

# fastest
cie <- CIE()
res103 <- microbenchmark(
  irradiance(w.length, s.e.irrad, cie,
```

```
            use.cache=TRUE,
            use.hinges=FALSE,
            check.spectrum=FALSE,
            unit.out="photon"),
                    times=100L, control=list(warmup = 10L))
```

When using a weighted waveband, in this example, `CIE()`, enabling all checks and optimizations for precision, and disabling all optimizations for speed yields a median execution time of 0.594 ms, accepting all defaults yields a median execution time 0.306 ms, and disabling all checks, optimizations for precision and enabling all optimizations for speed yields a median execution time of 0.0601, in relation to the slowest one, execution times are 100, 52, and 10%.

Finally we compare the returned values for the irradiance, to see the impact on them of optimizing for speed.

```
# slowest
photon_irradiance(w.length, s.e.irrad, CIE(),
                  use.cache=FALSE,
                  use.hinges=TRUE,
                  check.spectrum=TRUE)

## CIE98.298
## 2.038e-07

# default
photon_irradiance(w.length, s.e.irrad, CIE())

## CIE98.298
## 2.037e-07

# fastest
CIE <- CIE()
irradiance(w.length, s.e.irrad, CIE,
           use.cache=TRUE,
           use.hinges=FALSE,
           check.spectrum=FALSE,
           unit.out="photon")

## CIE98.298
## 2.037e-07
```

These results are based on spectral data at 1 nm interval, for more densely measured data the effect of not using hinges becomes even smaller. In contrast, with data measured at wider wavelength steps, the errors will be larger. They also depend on the specific BSWF being used.

Using `split_irradiance`

Using the cache also helps with `split_irradiance`.

```
res111 <- microbenchmark(
  split_photon_irradiance(w.length, s.e.irrad,
                          c(400, 500, 600, 700)),
  times=100L, control=list(warmup = 10L))
res112 <- microbenchmark(
  split_photon_irradiance(w.length, s.e.irrad,
```

```
                          c(400, 500, 600, 700),
                          use.cache=TRUE),
  times=100L, control=list(warmup = 10L))
```

When using `split_irradiance`, the cache reduces the median execution time from 0.514 ms to 0.33 ms, by 36%.

Using hinges slows down calculations:

```
res121 <- microbenchmark(
  split_photon_irradiance(w.length, s.e.irrad,
                          c(400, 500, 600, 700),
                          use.cache=TRUE),
  times=100L, control=list(warmup = 10L))
res122 <- microbenchmark(
  split_photon_irradiance(w.length, s.e.irrad,
                          c(400, 500, 600, 700),
                          use.cache=TRUE,
                          use.hinges=TRUE),
  times=100L, control=list(warmup = 10L))
```

When using `split_irradiance`, enabling use of hinges increases the median execution time from 0.329 ms to 0.636 ms, by a factor of 1.9313. There is less overhead than if calculating the same three wavebands separately, as all hinges are inserted in a single operation.

Disabling checking of the spectrum reduces the execution time, but proportionally not as much as for the `irradiance` functions, as the spectrum is checked only once independently of the number of bands into which it is split.

```
res131 <- microbenchmark(
  split_photon_irradiance(w.length, s.e.irrad,
                          c(400, 500, 600, 700),
                          use.cache=TRUE),
  times=100L, control=list(warmup = 10L))
res132 <- microbenchmark(
  split_photon_irradiance(w.length, s.e.irrad,
                          c(400, 500, 600, 700),
                          use.cache=TRUE,
                          check.spectrum=FALSE),
  times=100L, control=list(warmup = 10L))
```

When using `split_irradiance`, disabling the data validation check reduces the median execution time from 0.328 ms to 0.299 ms, by 8.9%.

As all the execution times are in milliseconds, all the optimizations discussed above are totally irrelevant unless you are planning to repeat similar calculations on thousands of spectra. They apply only to the machine, OS and version of R and packages used when building this typeset output.

```
detach(sun.spct)
```

1.7 Preliminary tests of spectral objects

```
res211 <- microbenchmark(
  q_irrad(sun.spct, PAR(), use.cached.mult=TRUE),
  times=100L, control=list(warmup = 10L))
res212 <- microbenchmark(
  q_irrad(sun.spct, list(UVC(), UVB(), UVA(), PAR()), use.cached.mult=TRUE),
  times=100L, control=list(warmup = 10L))
```

When using `q_irrad_spct` with one waveband the time is 0.191 ms and it increases to 0.473 ms, by 2.48 times when with four wavebands.

```
res221 <- microbenchmark(
  q_irrad(sun.spct),
  times=100L, control=list(warmup = 10L))
```

When using `q_irrad_spct` time is 0.28 ms.

```
res231 <- microbenchmark(
  q_irrad(sun.spct, PAR(), use.cached.mult=TRUE),
  times=100L, control=list(warmup = 10L))
res232 <- microbenchmark(
  q_irrad(sun.spct, PAR(), use.cached.mult=FALSE),
  times=100L, control=list(warmup = 10L))
```

When using `q_irrad` with cache enabled is 0.195 ms and it increases to 0.24 ms, by 1.23 times when with cacheing disabled.

```
res241 <- microbenchmark(
  q_irrad(sun.spct, CIE(), use.cached.mult=TRUE),
  times=100L, control=list(warmup = 10L))
res242 <- microbenchmark(
  q_irrad(sun.spct, CIE(), use.cached.mult=FALSE),
  times=100L, control=list(warmup = 10L))
```

When using `q_irrad` with cache enabled is 0.205 ms and it increases to 0.392 ms, by 1.91 times when with cacheing disabled.

```
res251 <- microbenchmark(
  q_irrad(sun.spct, CIE(), use.cached.mult=TRUE),
  times=100L, control=list(warmup = 10L))
res252 <- microbenchmark(
  q_irrad(sun.spct, CIE(), use.cached.mult=TRUE, use.hinges=TRUE),
  times=100L, control=list(warmup = 10L))
```

`q_irrad` without hinges enabled (the default when the wavelength step $\Delta\lambda < 1.1\,\text{nm}$) takes 0.205 ms but the execution time increases to 2.2 ms, by 9.74 times with use of hinges enabled.

```
q_irrad(sun.spct, CIE(), use.cached.mult=TRUE)

## CIE98.298
## 2.037e-07
## attr(,"time.unit")
## [1] "second"

q_irrad(sun.spct, CIE(), use.cached.mult=FALSE)
```

```
## CIE98.298
## 2.037e-07
## attr(,"time.unit")
## [1] "second"

q_irrad(sun.spct, CIE(), use.cached.mult=TRUE, use.hinges=TRUE)

## CIE98.298
## 2.038e-07
## attr(,"time.unit")
## [1] "second"
```

The differnce in the returned value is rather small.

```
cp_sun.spct <- copy(sun.spct)
res261 <- microbenchmark(
  sun_out.spct <- cp_sun.spct * 2 + cp_sun.spct,
  times=100L, control=list(warmup = 10L))
res262 <- microbenchmark(
  sun_out.spct <- with(sun.data, s.e.irrad * 2 + s.e.irrad),
  times=100L, control=list(warmup = 10L))
res263 <- microbenchmark(
  sun_out.spct <- with(sun.dt, s.e.irrad * 2 + s.e.irrad),
  times=100L, control=list(warmup = 10L))
res264 <- microbenchmark(
  sun_out.spct <- with(cp_sun.spct, s.e.irrad * 2 + s.e.irrad),
  times=100L, control=list(warmup = 10L))
res265 <- microbenchmark(
  sun_out.spct <- cp_sun.spct[ , s.e.irrad := s.e.irrad * 2 + s.e.irrad],
  times=100L, control=list(warmup = 10L))
```

When using operators with spectral objects execution time is 3.19 ms and it decreases to 0.0095 ms, to 0.298 % when using operators on vectors in a data frame, to 0.0106 ms, to 0.334 % when using operators on vectors in a data table, to 0.0106 ms, to 0.334 % when using operators on vectors in a `source.spct`, and to 0.452 ms, to 14.2 % when using data table syntax in a `source.spct`.

1.8 Profiling

Profiling is basically fine-grained benchmarking. It provides information about in which part of your code the program spends most time when executing. Once you know this, you can try to just make those critical sections execute faster. Speed-ups can be obtained either by rewriting these parts in a compiled language like C or C++, or by use of a more efficient calculation algorithm. A detailed discussion is outside the scope of this handbook, so only a brief example will be shown here.

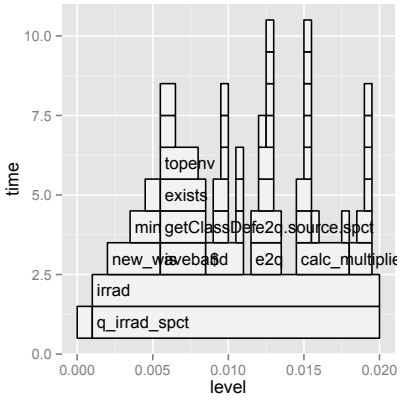
```
library(ggplot2)
library(profr)
```

```
profr.df <- profr({q_irrad_spct(sun.spct)},
                  interval = 0.0005, quiet = TRUE)

## Warning:  incomplete final line found on 'C:\Users\aphalo\AppData\Local\Temp\Rtmp8Q1JdY\file5d14670095
```

```
head(profr.df)
```

```
##    level g_id t_id               f  start    end
## 37     1    1    1 lazyLoadDBfetch 0.0000 0.0010
## 38     1    2    1    q_irrad_spct 0.0010 0.0200
## 39     2    1    1           irrad 0.0010 0.0200
## 40     3    1    1    new_waveband 0.0020 0.0055
## 41     3    2    1              is 0.0055 0.0085
## 42     3    3    1               $ 0.0085 0.0110
##    n  leaf   time       source
## 37 1  TRUE 0.0010         base
## 38 1 FALSE 0.0190 photobiology
## 39 1  TRUE 0.0190 photobiology
## 40 1  TRUE 0.0035 photobiology
## 41 1 FALSE 0.0030      methods
## 42 1  TRUE 0.0025         base
```

```
ggplot(profr.df)
```



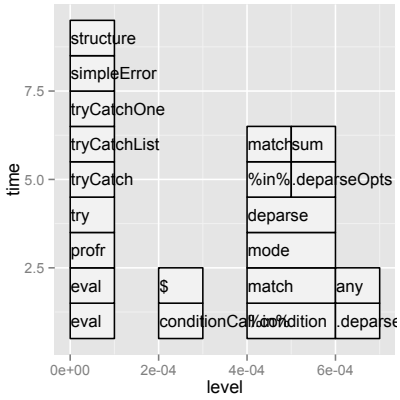

```
profr.df <- profr({q_irrad_spct(sun.spct, my_PAR, use.hinges=TRUE)},
                  interval = 0.0001, quiet = TRUE)
```

```
## Warning:  incomplete final line found on 'C:\Users\aphalo\AppData\Local\Temp\Rtmp8Q1JdY\file5d14412586e'
```

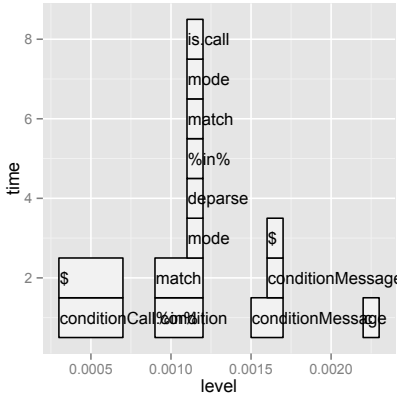
```
head(profr.df)
```

```
##    level g_id t_id                      f start
## 46     1    1    1                   eval 0e+00
## 47     1    2    2 conditionCall.condition 2e-04
## 48     1    3    3                   %in% 4e-04
## 49     1    4    3            .deparseOpts 6e-04
## 50     2    1    1                   eval 0e+00
## 51     2    2    2                      $ 2e-04
##      end n  leaf  time source
## 46 1e-04 1 FALSE 1e-04   base
## 47 3e-04 1 FALSE 1e-04   base
## 48 6e-04 1 FALSE 2e-04   base
## 49 7e-04 1 FALSE 1e-04   <NA>
## 50 1e-04 1 FALSE 1e-04   base
## 51 3e-04 1  TRUE 1e-04   base
```

```
ggplot(profr.df)
```

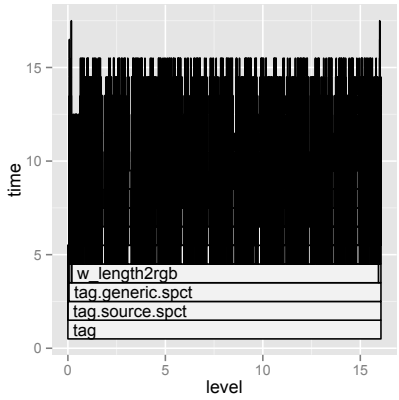



```
my_sun.spct <- copy(sun.spct)
profr.df <- profr({q_irrad_spct(my_sun.spct, my_PAR, use.hinges=FALSE)},
                  interval = 0.0001, quiet = TRUE)
head(profr.df)

##    level g_id t_id                      f  start
## 32     1    1    1 conditionCall.condition 0.0003
## 33     1    2    2                   %in% 0.0009
## 34     1    3    3        conditionMessage 0.0015
## 35     1    4    4                      c 0.0022
## 36     2    1    1                      $ 0.0003
## 37     2    2    2                  match 0.0009
##       end n  leaf  time source
## 32 0.0007 1 FALSE 4e-04   base
## 33 0.0012 1 FALSE 3e-04   base
## 34 0.0017 1  TRUE 2e-04   base
## 35 0.0023 1  TRUE 1e-04   base
## 36 0.0007 1  TRUE 4e-04   base
## 37 0.0012 1  TRUE 3e-04   base

ggplot(profr.df)
```




```
my_sun.spct <- copy(sun.spct)
profr.df <- profr({tag(my_sun.spct)},
                  interval = 0.001, quiet = TRUE)
head(profr.df)
```

```
##    level g_id t_id              f start    end n
## 29     1    1    1            tag 0.000 16.059 1
## 30     2    1    1 lazyLoadDBfetch 0.000  0.011 1
## 31     2    2    1 tag.source.spct 0.011 16.059 1
## 32     3    1    1  ..getNamespace 0.003  0.004 1
## 33     3    2    1     <Anonymous> 0.004  0.005 1
## 34     3    2    2     <Anonymous> 0.006  0.010 1
##     leaf   time       source
## 29 FALSE 16.059 photobiology
## 30  TRUE  0.011         base
## 31 FALSE 16.048 photobiology
## 32 FALSE  0.001         <NA>
## 33 FALSE  0.001         <NA>
## 34 FALSE  0.004         <NA>

ggplot(profr.df)
```




```
my_sun.spct <- copy(sun.spct)
Rprof("profile1.out", line.profiling=TRUE, interval = 0.002)
tag(my_sun.spct)
Rprof(NULL)
summaryRprof("profile1.out", lines = "show")[["by.line"]]

##                    self.time self.pct total.time
## s.e.irrad2rgb.r#36     0.022     9.48      0.022
## tag.spct.r#8           0.000     0.00      0.232
## tag.spct.r#155         0.210    90.52      0.232
##                    total.pct
## s.e.irrad2rgb.r#36      9.48
## tag.spct.r#8          100.00
## tag.spct.r#155        100.00

# profr.df <- parse_rprof("profile1.out")
# head(profr.df)
# ggplot(profr.df)
```

```
try(detach(package:profr))
try(detach(package:photobiologyWavebands))
try(detach(package:photobiology))
try(detach(package:microbenchmark))
try(detach(package:ggplot2))
```