In [1]:
```
using Pkg
Pkg.activate(".")

using BenchmarkTools
using Profile
using ProfileView

using Plots

using QuadGK
using SpecialFunctions
```

```
  Activating environment at `~/Documents/schoolwork-codes/physics-215-julia/session-2/Project.toml`
```

Benchmarking and profiling user-defined complete elliptic integral of the 1st kind $K(m)$ against the Julia-defined `SpecialFunctions` version

For this exercise, we will be benchmarking the performance of two versions of the complete elliptic integral of the 1st kind. We will define our own version of it first, then compare its performance to the existing version defined in the `SpecialFunctions.jl` module.

The complete elliptic integral of the first kind is defined as

$$K(m) = \int_0^{\pi/2} \frac{d\theta}{\sqrt{1 - m\sin^2\theta}}, \quad m \in [0, 1), \tag{1}$$

where m is a parameter related to the elliptic modulus k by $k^2 = m$. This integral function often appears as a solution to problems in mechanics involving nonlinear modifications to periodic motion, such as the restricted three-body problem (see Dvorak and Lhotka 2014 for a summary).

To define our own complete elliptic integral of the first kind, we will need a way to perform integration. For our purposes, we will be using the `QuadGK.jl` module to do the integration for us. Note that series and polynomial representations of $K(m)$ also exist, and we will implement one such series later on in this notebook. (Interestingly enough, the `SpecialFunctions` implementation of $K(m)$ does use a polynomial approximation, as cited in its documentation.)

User-defined $K(m)$ (integral version)

We code the integral as follows:

In [2]:
```
"""
elliptic1(m)

    Input: m ∈ [0, 1) (no explicit type, but ideally it is an array of floats)
        (Note that the SpecialFunctions implementation has m ∈ (-∞, 1]. We will not b
    Output: Vector{Tuple{Float64, Float64}},
        where the first tuple element is the value of the function, and the second is
"""

function elliptic1(m)
    invdn = θ -> 1.0/ sqrt(1 - m .* (sin(θ))^2)
    return quadgk(invdn, 0, pi/2, rtol=1e-8)
end
```

Out[2]: `elliptic1 (generic function with 1 method)`

As a quick sanity check, we can compare the results of our function `elliptic1(m)` and the `SpecialFunctions` version `ellipk(m)` for a certain range of values. In our case, we let m have 20 values from 0 to 0.9 (deliberately avoiding 1 as `elliptic1(m)` cannot handle values of `Inf`).

In [3]:
```
mrange = LinRange(0, 0.9, 20);

k₁ = ellipk.(mrange);

integ = elliptic1.(mrange);
k₂ = first.(integ);
error = last.(integ);

δ = abs.(k₁ - k₂);
```

In [4]:
```
["ellipk(m)" "elliptic1(m)" "Absolute deviation" "Integration error"
k₁ k₂ δ error]
```

Out[4]:
```
21×4 Matrix{Any}:
  "ellipk(m)"   "elliptic1(m)"   "Absolute deviation"   "Integration error"
 1.5708        1.5708           0.0                    0.0
 1.58991       1.58991          2.22045e-16            1.17018e-13
 1.61012       1.61012          2.22045e-16            2.05984e-11
 1.63156       1.63156          0.0                    1.23465e-10
 1.65436       1.65436          2.22045e-16            3.91055e-10
 1.67869       1.67869          0.0                    8.38641e-10
 1.70477       1.70477          2.22045e-16            1.2039e-9
 1.73284       1.73284          0.0                    5.329e-10
 1.76319       1.76319          4.44089e-16            3.52386e-9
 1.79622       1.79622          0.0                    1.5721e-8
 1.8324        1.8324           2.22045e-16            1.82069e-11
 1.87233       1.87233          0.0                    5.77602e-11
 1.91684       1.91684          2.22045e-16            1.60814e-10
 1.967         1.967            2.22045e-16            3.97142e-10
 2.02437       2.02437          0.0                    8.22103e-10
 2.09118       2.09118          4.44089e-16            1.0389e-9
 2.17091       2.17091          4.44089e-16            2.31036e-9
 2.26933       2.26933          4.44089e-16            1.03879e-11
 2.39719       2.39719          1.33227e-15            1.09064e-10
 2.57809       2.57809          4.44089e-16            8.53201e-10
```

In [5]:
```
eps(Float64)
```

Out[5]: `2.220446049250313e-16`

Recall that the 64-bit epsilon is given by the value of `eps(Float64)`. Given the absolute deviation values above, we can conclude that our defined function comes close to the predefined function up to a reasonable degree of accuracy.

User-defined $K(m)$ (series version)

We can also define $K(m)$ as the following sum:

$$K(m) = \frac{\pi}{2}\sum_{n=0}^{\infty}\left(\frac{(2n)!}{2^{2n}(n!)^2}\right)^2 m^n \tag{2}$$

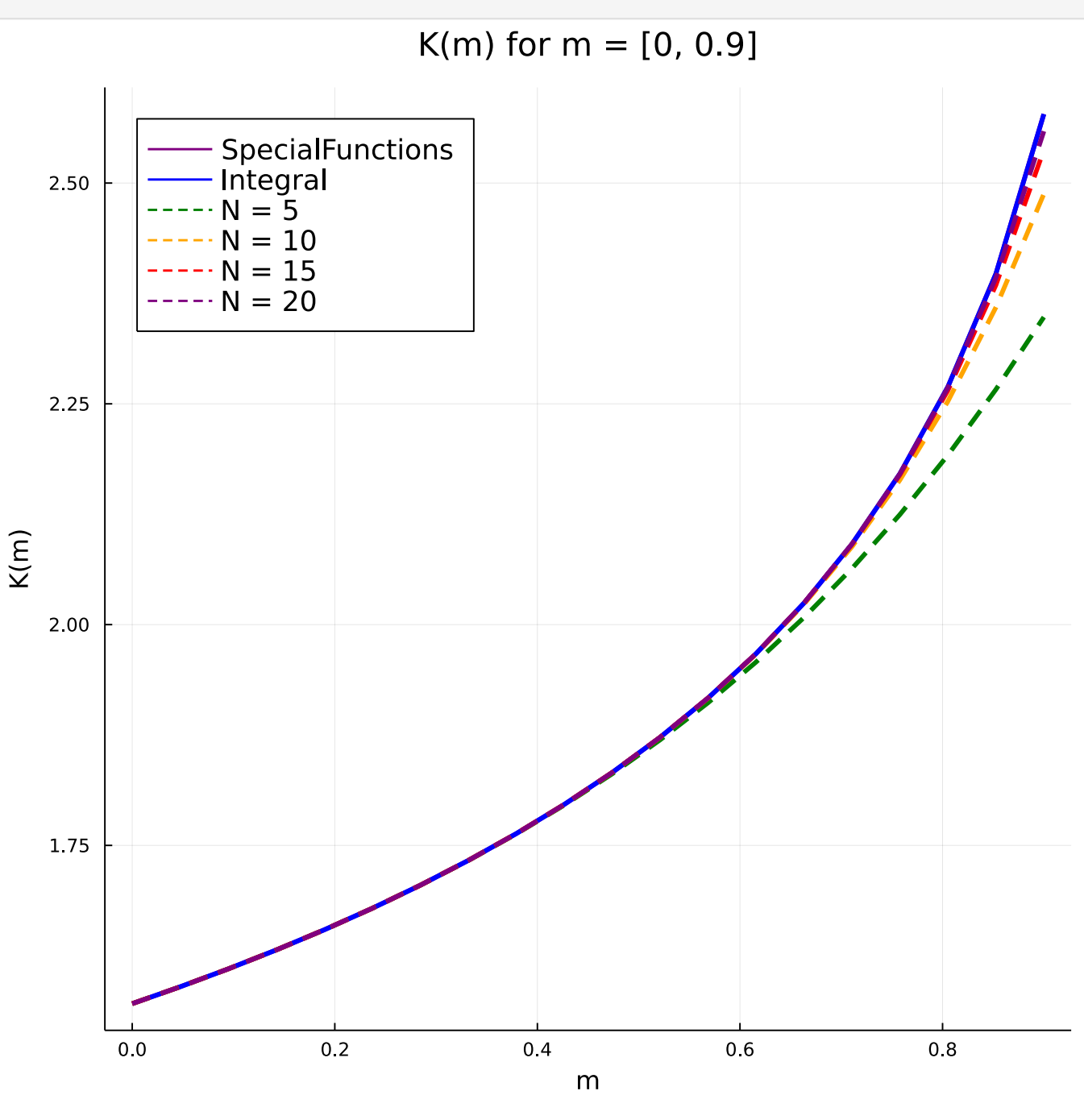
We can calculate a truncated version of this sum up to some finite order N using the following code.

In [6]:
```
"""
sum_elliptic1(m; N::Int)

    Input:
        m ∈ [0, 1) (no explicit type, but ideally it is an array of floats)
        N::Int (N > 30 gives out NaN and Inf values. Do not go past it.)
        (Note that the SpecialFunctions implementation has m ∈ (-∞, 1]. We will not b
    Output: Vector{BigFloat}, showing all the values of the function for every value o
"""

function sum_elliptic1(m; N::Int)
    sum = zeros(length(m))
    for i in 0:N
        aᵢ = (factorial(big(2i)) / ((2^(2i))*(factorial(big(i))^2)))^2
        sum += aᵢ.* m.^i
    end
    return (pi/2).*sum
end
```

Out[6]: `sum_elliptic1 (generic function with 1 method)`

To compare the accuracy of this series representation against the previous implementations, we plot `sum_elliptic1` at different orders of N and compare it with `ellipk` and `elliptic1`.

In [7]:
```
gr()

p = plot(mrange, [k₁, k₂], label = ["SpecialFunctions" "Integral"], ylabel = "K(m)", x
title = "K(m) for m = [0, 0.9]", lw = 3,
    size = (700,700), legendfontsize = 12, palette=:rainbow)

Nrange = [5, 10, 15, 20];
for n in Nrange
    k₃ = sum_elliptic1(mrange; N = n)
    plot!(p, mrange, k₃, label = "N = $(n)", lw = 3, ls =:dash)
end

display(p)
```



We can see here that `sum_elliptic1` deviates from `ellipk` and `elliptic1` for $m \gtrsim 0.6$ at low orders of N. As we increase N however, the deviations start to decrease. In particular, for $N = 20$, we have the following absolute deviation (compared to `ellipk(m)`).

In [8]:
```
k₃ = sum_elliptic1(mrange; N = 20);
δ₂ = abs.(k₁ .- k₃);
```

In [9]:
```
["ellipk(m)" "sum_elliptic1(m)" "Absolute deviation"
k₁ k₃ δ₂]
```

Out[9]:
```
21×3 Matrix{Any}:
  "ellipk(m)"   "sum_elliptic1(m)"   "Absolute deviation"
 1.5708        1.5708               0.0
 1.58991       1.58991              1.06001e-16
 1.61012       1.61012              6.7278e-17
 1.63156       1.63156              9.80373e-18
 1.65436       1.65436              4.51604e-17
 1.67869       1.67869              2.24367e-15
 1.70477       1.70477              1.08629e-13
 1.73284       1.73284              2.94776e-12
 1.76319       1.76319              5.21356e-11
 1.79622       1.79622              6.6588e-10
 1.8324        1.8324               6.59096e-09
 1.87233       1.87233              5.31964e-08
 1.91684       1.91684              3.63718e-07
 1.967         1.967                2.17027e-06
 2.02437       2.02436              1.15782e-05
 2.09118       2.09112              5.63836e-05
 2.17091       2.17065              0.000255493
 2.26933       2.26823              0.00109883
 2.39719       2.3926               0.0045961
 2.57809       2.55866              0.0194328
```

For our purposes, we'll consider $N = 20$ for benchmarking the series representation of $K(m)$.

Timing and benchmarking $K(m)$

Now that we have established our functions above, we can start timing and benchmarking each of them. There are multiple `@time` macros at our disposal, such as `@time`, `@timev`, `@timed`, and `@elapsed`. We can determine the best macro for our purposes by checking the type each macro outputs.

In [10]:
```
time_k = @time ellipk.(mrange)
typeof(time_k)
```

```
  0.000021 seconds (4 allocations: 384 bytes)
```

Out[10]: `Vector{Float64} (alias for Array{Float64, 1})`

In [11]:
```
timev_k = @timev ellipk.(mrange)
typeof(timev_k)
```

```
  0.000020 seconds (4 allocations: 384 bytes)
elapsed time (ns): 19541
bytes allocated:   384
pool allocs:       4
```

Out[11]: `Vector{Float64} (alias for Array{Float64, 1})`

In [12]:
```
timed_k = @timed ellipk.(mrange)
println(timed_k[:time])

typeof(timed_k)
```

```
1.8828e-5
```

Out[12]: `NamedTuple{(:value, :time, :bytes, :gctime, :gcstats), Tuple{Vector{Float64}, Float64, Int64, Float64, Base.GC_Diff}}`

In [13]:
```
elapsed_k = @elapsed ellipk.(mrange)
println(elapsed_k)
typeof(elapsed_k)
```

```
1.8178e-5
```

Out[13]: `Float64`

Note that among all of these macros, only `@elapsed` provides the direct `Float64` output of the time. Thus for our purposes, we will be using `@elapsed` in our initial (naive) timing comparisons. (Recall that in Session 1, I have discussed that `@elapsed` is sufficient to obtain the proper runtime of the function or routine that it calls.)

In [14]:
```
t₁ = @elapsed ellipk.(mrange);
t₂ = @elapsed elliptic1.(mrange);
t₃ = @elapsed sum_elliptic1(mrange; N = 20);
```

In [15]:
```
#all time outputs are in seconds

["ellipk(m) (t₁)" "elliptic1(m) (t₂)" "sum_elliptic1(m) (t₃)"
t₁ t₂ t₃]
```

Out[15]:
```
2×3 Matrix{Any}:
  "ellipk(m) (t₁)"    "elliptic1(m) (t₂)"    "sum_elliptic1(m) (t₃)"
 1.9772e-5           6.999e-5              0.00018934
```

In [16]:
```
["t₂/t₁" "t₃/t₁" "t₃/t₂"
t₂/t₁ t₃/t₁ t₃/t₂]
```

Out[16]:
```
2×3 Matrix{Any}:
  "t₂/t₁"    "t₃/t₁"    "t₃/t₂"
 3.53985    9.57617    2.70524
```

From here, we can naively consider that the series function `sum_elliptic1(m; N)` is the slowest of the three functions, as it takes about 2-3x as long as the integral function `elliptic1(m)` and about 9-10x longer than `ellipk(m)`. To obtain a much more statistically robust benchmarking, however, we will have to use the `@benchmark` macro.

In [17]:
```
bench₁ = @benchmark ellipk.($mrange) samples = 1_000_000 seconds = 300
```

Out[17]:
```
BenchmarkTools.Trial: 1000000 samples with 252 evaluations.
 Range (min … max):  266.127 ns …  58.538 μs  ┊ GC (min … max): 0.00% … 95.08%
 Time  (median):     446.782 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   511.581 ns ± 547.857 ns  ┊ GC (mean ± σ):  2.96% ±  3.10%

 266 ns          Histogram: frequency by time         1.31 μs <

 Memory estimate: 240 bytes, allocs estimate: 1.
```

In [18]:
```
bench₂ = @benchmark elliptic1.($mrange) samples = 1_000_000 seconds = 300
```

Out[18]:
```
BenchmarkTools.Trial: 1000000 samples with 1 evaluation.
 Range (min … max):  23.898 μs … 15.253 ms  ┊ GC (min … max): 0.00% … 98.94%
 Time  (median):     43.173 μs              ┊ GC (median):    0.00%
 Time  (mean ± σ):   42.783 μs ± 90.805 μs  ┊ GC (mean ± σ):  3.01% ±  1.45%

 23.9 μs         Histogram: frequency by time        88.3 μs <

 Memory estimate: 13.86 KiB, allocs estimate: 601.
```

In [19]:
```
bench₃ = @benchmark sum_elliptic1($mrange; N = $20) samples = 1_000_000 seconds = 300
```

Out[19]:
```
BenchmarkTools.Trial: 1000000 samples with 1 evaluation.
 Range (min … max):  128.434 μs …  25.264 ms  ┊ GC (min … max): 0.00% … 67.66%
 Time  (median):     173.798 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   213.865 μs ± 577.677 μs  ┊ GC (mean ± σ):  7.33% ±  2.70%

 128 μs          Histogram: frequency by time         362 μs <

 Memory estimate: 118.92 KiB, allocs estimate: 2231.
```

Above we have taken simple benchmarks for each of the three functions, attempting to obtain 1,000,000 samples each within a span of 300 seconds. Note that not all of these benchmarks may finish with 1,000,000 samples; given the variable runtime of each function, it is possible that the benchmarking process takes a whole 300 seconds before it obtains the requisite number of samples. We can still, however, use these benchmarks to obtain time ratio estimates to compare the functions.

In [20]:
```
["t₂/t₁" "t₃/t₁" "t₃/t₂"
round(median(bench₂.times)/median(bench₁.times)) round(median(bench₃.times)/median(benc
```

Out[20]:
```
2×3 Matrix{Any}:
   "t₂/t₁"     "t₃/t₁"    "t₃/t₂"
 97.0       389.0       4.0
```

Thus we can see here that the second function (the user defined `elliptic1` integral form) is over an order of magnitude slower than the `SpecialFuntions` version of the complete elliptic integral. In contrast, `sum_elliptic1` is over two orders of magnitude slower than `ellipk` and a bit slower than `elliptic1`.

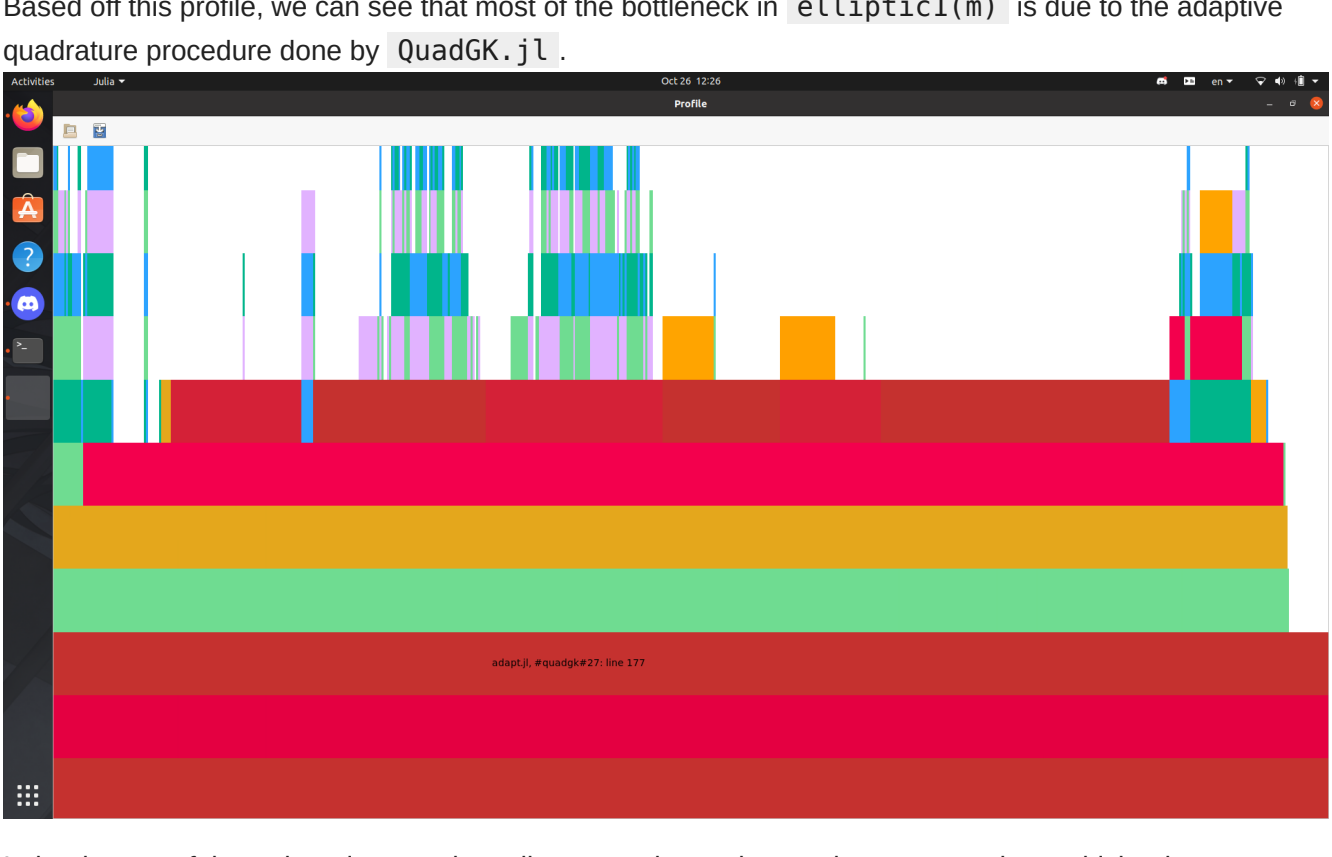
Profiling `elliptic1(m)` and `sum_elliptic1(m)`

Now that we have an idea how fast our functions are running compared to the base case, we can begin to profile them to see where our code bottlenecks. To do this we simply run the `Profile` and `ProfileView` modules as follows.

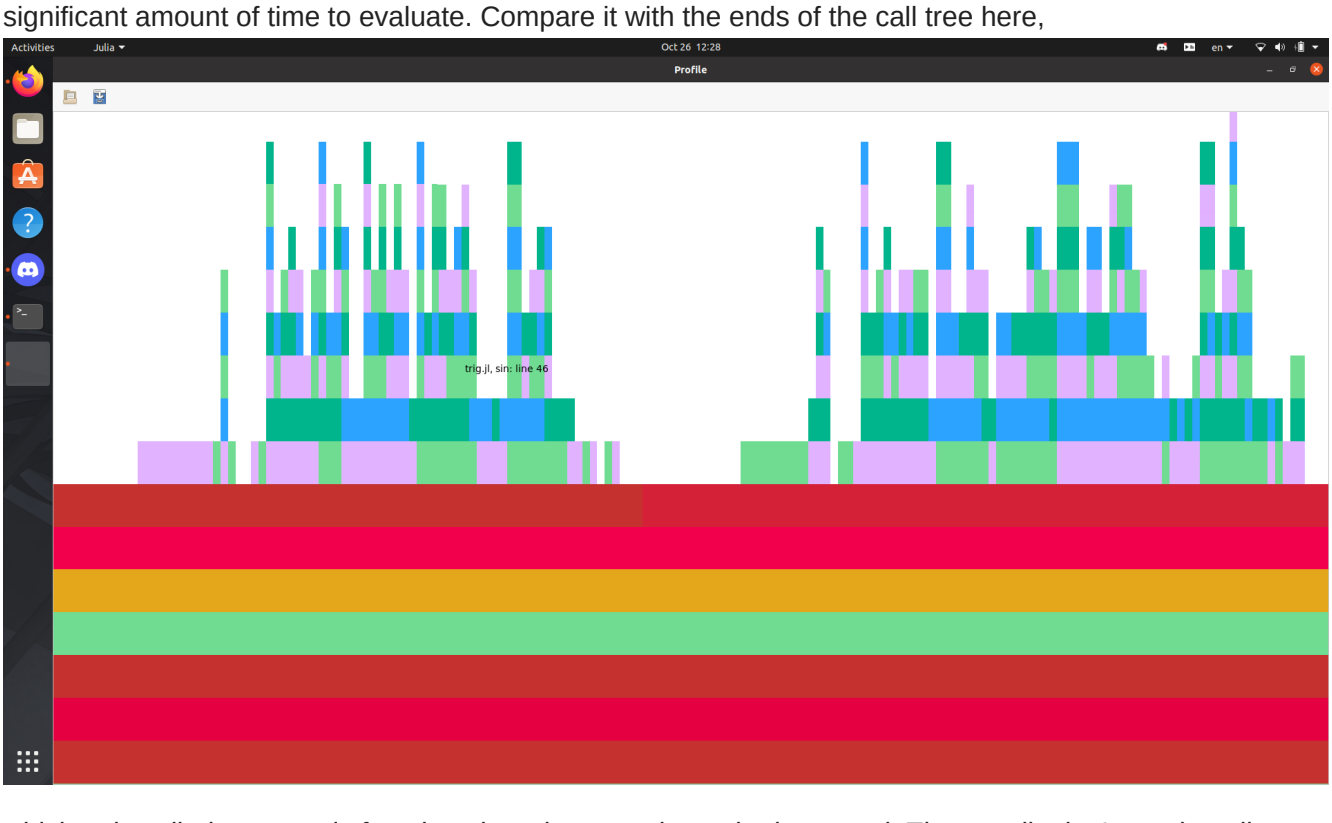
In [23]:
```
#first we profile elliptic1(m) with 100,000 samples

@profview for _ in 1:100_000 elliptic1.(mrange) end
#profile opens in another window
```

Out[23]:
```
Gtk.GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visible=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-default=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0, no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=NULL, window, opacity=1.000000, double-buffered, halign=GTK_ALIGN_FILL, valign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-end=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALSE, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, border-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Profile", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WIN_POS_NONE, default-width=800, default-height=600, destroy-with-parent=FALSE, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, type-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-hint=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, decorated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-for, attached-to, has-resize-grip, resize-grip-visible, application, is-active=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE, focus-visible=FALSE, is-maximized=FALSE)
/home/gadizon/.julia/packages/QuadGK/ENhXl/src/adapt.jl:113, MethodInstance for handle_infinities(::QuadGK.var"#28#29"{Nothing, Float64, Int64, Int64, typeof(LinearAlgebra.norm)}, ::var"#1#2"{Float64}, ::Tuple{Float64, Float64})
/home/gadizon/.julia/packages/QuadGK/ENhXl/src/adapt.jl:177, MethodInstance for var"#quadgk#27"(::Nothing, ::Float64, ::Int64, ::Int64, ::Function, ::typeof(quadgk), ::Function, ::Float64, ::Float64)
/home/gadizon/.julia/packages/QuadGK/ENhXl/src/adapt.jl:177, MethodInstance for var"#quadgk#27"(::Nothing, ::Float64, ::Int64, ::Int64, ::Function, ::typeof(quadgk), ::Function, ::Float64, ::Float64)
/home/gadizon/.julia/packages/QuadGK/ENhXl/src/adapt.jl:177, MethodInstance for var"#quadgk#27"(::Nothing, ::Float64, ::Int64, ::Int64, ::Function, ::typeof(quadgk), ::Function, ::Float64, ::Float64)
/home/gadizon/.julia/packages/QuadGK/ENhXl/src/adapt.jl:177, MethodInstance for var"#quadgk#27"(::Nothing, ::Float64, ::Int64, ::Int64, ::Function, ::typeof(quadgk), ::Function, ::Float64, ::Float64)
./special/trig.jl:38, MethodInstance for sin(::Float64)
/home/gadizon/.julia/packages/QuadGK/ENhXl/src/adapt.jl:46, MethodInstance for adapt(::Function, ::Vector{QuadGK.Segment{Float64, Float64, Float64}}, ::Float64, ::Float64, ::Int64, ::Vector{Float64}, ::Vector{Float64}, ::Vector{Float64}, ::Int64, ::Float64, ::Float64, ::Int64, ::typeof(LinearAlgebra.norm))
/home/gadizon/.julia/packages/QuadGK/ENhXl/src/adapt.jl:46, MethodInstance for adapt(::Function, ::Vector{QuadGK.Segment{Float64, Float64, Float64}}, ::Float64, ::Float64, ::Int64, ::Vector{Float64}, ::Vector{Float64}, ::Vector{Float64}, ::Int64, ::Float64, ::Float64, ::Int64, ::typeof(LinearAlgebra.norm))
/home/gadizon/.julia/packages/QuadGK/ENhXl/src/adapt.jl:46, MethodInstance for adapt(::Function, ::Vector{QuadGK.Segment{Float64, Float64, Float64}}, ::Float64, ::Float64, ::Int64, ::Vector{Float64}, ::Vector{Float64}, ::Vector{Float64}, ::Int64, ::Float64, ::Float64, ::Int64, ::typeof(LinearAlgebra.norm))
```

Based off this profile, we can see that most of the bottleneck in `elliptic1(m)` is due to the adaptive quadrature procedure done by `QuadGK.jl`.



Indeed, most of the red portions up the call tree are due to the quadrature procedure, which takes a significant amount of time to evaluate. Compare it with the ends of the call tree here,



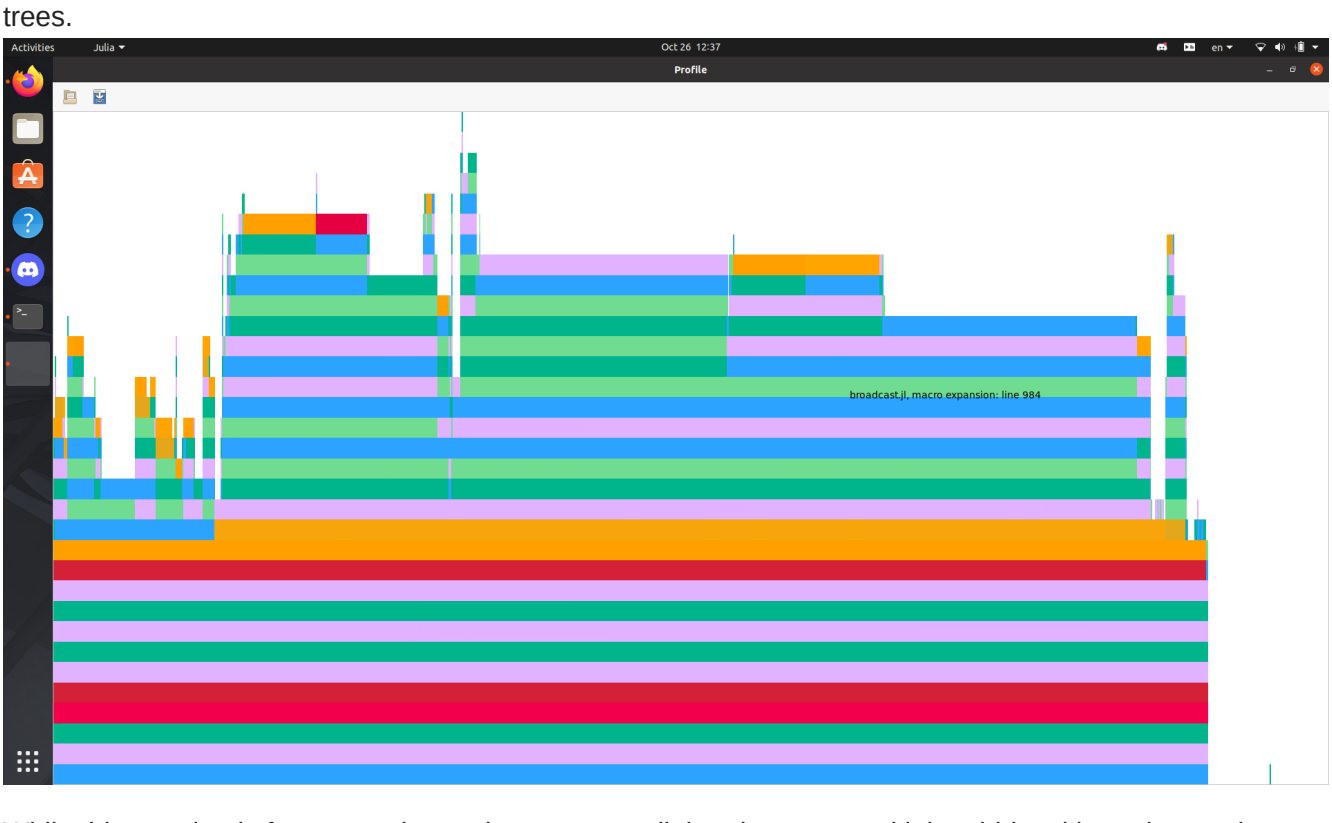
which only call trigonometric functions in order to evaluate the integrand. These calls don't need to allocate as much time compared to the quadrature procedure itself.

Now we let us profile `sum_elliptic1(m)` and see where its bottleneck lies.

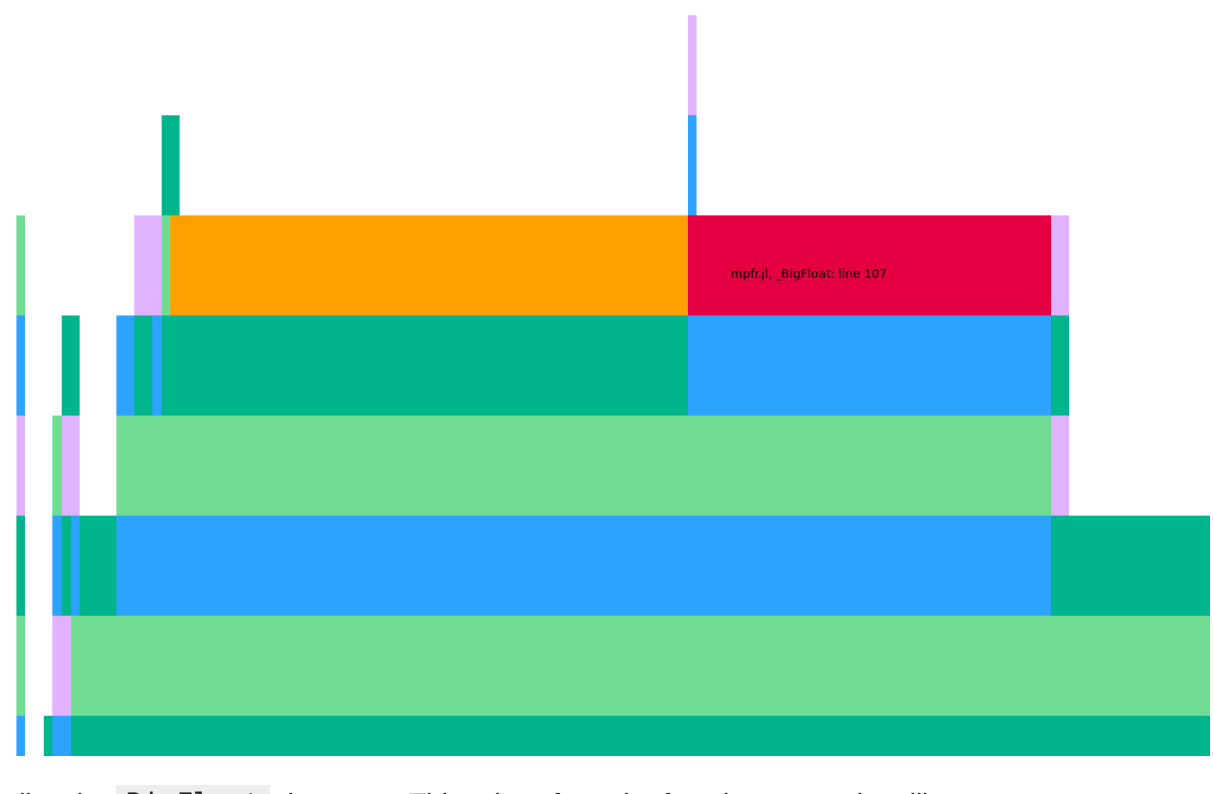
In [25]:
```
@profview for _ in 1:100_000 sum_elliptic1(mrange; N = 20) end
#profile opens in another window
```

Out[25]:
```
Gtk.GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visible=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-default=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0, no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=NULL, window, opacity=1.000000, double-buffered, halign=GTK_ALIGN_FILL, valign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-end=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALSE, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, border-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Profile", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WIN_POS_NONE, default-width=800, default-height=600, destroy-with-parent=FALSE, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, type-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-hint=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, decorated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-for, attached-to, has-resize-grip, resize-grip-visible, application, is-active=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE, focus-visible=FALSE, is-maximized=FALSE)
```

Compared to `elliptic1(m)`, the red bar bottlenecks are situated more towardsthe roots of the call trees.



While this may imply faster-running code, note as well that there are multiple mid-level branches to the execution, which mean that there are more processes overall to call as the code runs. We should also note that there is this leaf-level block of red in the call tree



which handles the `BigFloat` data type. This arises from the fact that our code utilizes

```
    aᵢ = (factorial(big(2i)) / ((2^(2i))*(factorial(big(i))^2)))^2
```

the function `big(x)` to ensure that the `factorial(x)` function is able to take in values greater than 20. We have the option to remove this bottleneck and restrict ourselves to factorial arguments less than 20, as throughout the notebook we did not in fact go over $N = 20$ orders in evaluating our series representation for $K(m)$. We are unsure, however, if this will lead to a significant speed-up for our code.