In [1]:

```
using Pkg
Pkg.activate(".")

using BenchmarkTools
using DataFrames
```

```
  Activating environment at `~/Documents/schoolwork-codes/physics-215-julia/session-4/
Project.toml`
```

Session 4: Fast function calls

KR1: Variable implementations

For this session, we will examine the impact of calling global variables on the speed of code. We use a relatively simple gravitational central force code, where `G` is Newton's gravitational constant, `M` is the mass of the Sun, and `rand_pos` is a random position of a test particle along the x-y plane.

In [2]:

```
G = 6.67e-11 #in SI units
M = 1.989e30 #in kg

function centralforce(r::Vector)
    squaredr = r[1]^2.0 + r[2]^2.0
    return -G*M/squaredr
end
```

Out[2]:

```
centralforce (generic function with 1 method)
```

In [4]:

```
rand_pos = rand(2);
```

We can benchmark the performance of this code and get the following times.

In [5]:

```
mark0 = @benchmark centralforce($rand_pos)
```

Out[5]:

```
BenchmarkTools.Trial: 10000 samples with 985 evaluations.
 Range (min … max):  54.961 ns …  1.102 μs  ┊ GC (min … max): 0.00% … 91.90%
 Time  (median):     68.420 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   71.533 ns ± 34.957 ns  ┊ GC (mean ± σ):  1.72% ±  3.34%

  55 ns        Histogram: log(frequency) by time      94.8 ns <

 Memory estimate: 64 bytes, allocs estimate: 4.
```

In [6]:

```
@code_warntype centralforce(rand_pos)
```

```
Variables
  #self#::Core.Const(centralforce)
  r::Vector{Float64}
  squaredr::Float64

Body::Any
1 ─ %1 = Base.getindex(r, 1)::Float64
│   %2 = (%1 ^ 2.0)::Float64
│   %3 = Base.getindex(r, 2)::Float64
│   %4 = (%3 ^ 2.0)::Float64
│        (squaredr = %2 + %4)
│   %6 = -Main.G::Any
│   %7 = (%6 * Main.M)::Any
│   %8 = (%7 / squaredr)::Any
└──      return %8
```

As we can see with `@code_warntype`, leaving `G` and `M` as plain global variables sets their type as `Any`, which slows down code operations as Julia has to narrow down the concrete variable type each time the code runs. We can alleviate this problem by setting all global constants as `const`s, fixing both their value and variable typing.

In [7]:

```
const G_const = 6.67e-11 #in SI units
const M_const = 1.989e30 #mass of sun in kg

function centralforce_const(r::Vector)
    squaredr = r[1]^2.0 + r[2]^2.0
    return -G_const*M_const/squaredr
end
```

Out[7]:

```
centralforce_const (generic function with 1 method)
```

If we benchmark this new function `centralforce_const` with the same `rand_pos`, we get a significantly improved runtime on the code.

In [8]:

```
mark1 = @benchmark centralforce_const($rand_pos)
```

Out[8]:

```
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range (min … max):  1.977 ns … 23.678 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     2.418 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   2.406 ns ±  0.448 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%

  1.98 ns        Histogram: frequency by time        2.59 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

In [9]:

```
speedup1 = median(mark0.times) / median(mark1.times)
table = DataFrame("Method"=>["Global","Constant"],"Speedup" => [1.0, speedup1]);

print(table)
```

```
2×2 DataFrame
 Row │ Method    Speedup
     │ String    Float64
─────┼───────────────────
   1 │ Global      1.0
   2 │ Constant   28.296
```

Furthermore, `@code_warntype` shows us that there is no type ambiguity in the resulting code, with both inputs and outputs being `Float64`.

In [10]:

```
@code_warntype centralforce_const(rand_pos)
```

```
Variables
  #self#::Core.Const(centralforce_const)
  r::Vector{Float64}
  squaredr::Float64

Body::Float64
1 ─ %1 = Base.getindex(r, 1)::Float64
│   %2 = (%1 ^ 2.0)::Float64
│   %3 = Base.getindex(r, 2)::Float64
│   %4 = (%3 ^ 2.0)::Float64
│        (squaredr = %2 + %4)
│   %6 = -Main.G_const::Core.Const(-6.67e-11)
│   %7 = (%6 * Main.M_const)::Core.Const(-1.326663e20)
│   %8 = (%7 / squaredr)::Float64
└──      return %8
```

We can also test the difference in speed if one of the variables, in this case `mass`, is parameterized into the function instead of being declared as a global variable.

In [11]:

```
function centralforce_param(r::Vector; mass = 1.989e30)
    squaredr = r[1]^2.0 + r[2]^2.0
    return -G_const*mass/squaredr
end
```

Out[11]:

```
centralforce_param (generic function with 1 method)
```

From here we can compare different parameterization approaches. The first benchmark has `mass` be parameterized with the global variable `M`. The second benchmark has `mass` parameterized with the *constant* global variable `M_const`. The third benchmark has the value of `mass` directly specified into the function call.

In [12]:

```
mark2a = @benchmark centralforce_param($rand_pos, mass = M)
```

Out[12]:

```
BenchmarkTools.Trial: 10000 samples with 800 evaluations.
 Range (min … max):  152.213 ns …  1.810 μs  ┊ GC (min … max): 0.00% … 88.97%
 Time  (median):     181.864 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   185.224 ns ± 45.199 ns  ┊ GC (mean ± σ):  0.68% ±  2.53%

  152 ns        Histogram: log(frequency) by time       210 ns <

 Memory estimate: 48 bytes, allocs estimate: 3.
```

In [13]:

```
mark2b = @benchmark centralforce_param($rand_pos, mass = M_const)
```

Out[13]:

```
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range (min … max):  2.610 ns … 39.743 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     3.064 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   3.076 ns ±  0.553 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%

  2.61 ns      Histogram: log(frequency) by time     3.74 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

In [14]:

```
mark2c = @benchmark centralforce_param($rand_pos, mass = 1.989e30)
```

Out[14]:

```
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range (min … max):  2.611 ns … 17.038 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     3.066 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   3.086 ns ±  0.389 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%

  2.61 ns      Histogram: log(frequency) by time     3.73 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

In [15]:

```
speedup2a = median(mark0.times) / median(mark2a.times)
speedup2b = median(mark0.times) / median(mark2b.times)
speedup2c = median(mark0.times) / median(mark2c.times)

push!(table, ["Parametrized", speedup2a]);
push!(table, ["Parametrized (implicit)", speedup2b]);
push!(table, ["Parametrized (explicit)", speedup2c]);

print(table)
```

```
5×2 DataFrame
 Row │ Method                   Speedup
     │ String                   Float64
─────┼────────────────────────────────────
   1 │ Global                    1.0
   2 │ Constant                 28.296
   3 │ Parametrized              0.376213
   4 │ Parametrized (implicit)  22.3302
   5 │ Parametrized (explicit)  22.3157
```

As we can see in the above table, parameterizing `mass` with the non-constant global variable actually slows down the code by an order of magnitude compared to just leaving it as a global constant to input into the code. The constant variable parameterizations have a speed-up comparable to that of the pure constant global variable input, although there is not much difference in their relative performance.

KR2-5: Speeding up function calls via numerical and Julia macro techniques

For this section, we will show how implementing efficient numerical algorithms can shorten code run time, and how turning these algorithms into Julia macros can speed up the code even further.

Consider the following polynomial expansion of some function $p(x)$ up to some finite order n:

$$p(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n. \tag{1}$$

We can evaluate $p(x)$ at any x for some set of coefficients $\{a_i\}$ naively through the following implementation:

In [16]:

```
function poly_naive(x, a::Vector)
    p = zero(x)
    for i in eachindex(a)
        p = p + (a[i] * x^(i-1))
    end
    return p
end
```

Out[16]:

```
poly_naive (generic function with 1 method)
```

For our purposes, we generate a constant set of 100 random coefficients `coeffs` for our polynomial expansion.

In [17]:

```
const coeffs = rand(100);
```

This allows us to write our naive polynomial evaluation as `f_naive(x)`.

In [18]:

```
f_naive(x) = poly_naive(x, coeffs)
```

Out[18]:

```
f_naive (generic function with 1 method)
```

Suppose we wish to evaluate our given polynomial at $x = 3.5$. We can benchmark the performance of the naive implementation to get the following runtimes.

In [19]:

```
x = 3.5
```

Out[19]:

```
3.5
```

In [21]:

```
poly_mark0 = @benchmark f_naive($x)
```

Out[21]:

```
BenchmarkTools.Trial: 10000 samples with 4 evaluations.
 Range (min … max):  7.852 μs …  12.969 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     8.241 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   8.281 μs ± 295.227 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%

  7.85 μs      Histogram: log(frequency) by time      9.01 μs <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

It is possible to improve on this runtime by evaluating the polynomial via Horner's method. Horner's method is a recursive evaluation algorithm which effectively "linearizes" the higher polynomial orders at each step, reducing the complexity of the computational operations. The schema of the algorithm goes as

$$b_n = a_n, \tag{2}$$

$$b_{n-1} = a_{n-1} + b_n x, \tag{3}$$

$$b_{n-2} = a_{n-2} + b_{n-1} x, \tag{4}$$

$$\vdots \tag{5}$$

$$b_0 = a_0 + b_1 x, \tag{6}$$

where $p(x) = b_0$.

We can implement the said algorithm with the following `poly_horner` implementation.

In [22]:

```
function poly_horner(x, a::Vector)
    b = zero(x)
    for i in reverse(eachindex(a))
        b = a[i] + b*x
    end
    return b
end
```

Out[22]:

```
poly_horner (generic function with 1 method)
```

In [23]:

```
f_horner(x) = poly_horner(x, coeffs)
```

Out[23]:

```
f_horner (generic function with 1 method)
```

Benchmarking this Horner implementation at $x = 3.5$ shows an order of magnitude speedup compared to the naive implementation.

In [24]:

```
poly_mark1 = @benchmark f_horner($x)
```

Out[24]:

```
BenchmarkTools.Trial: 10000 samples with 681 evaluations.
 Range (min … max):  179.107 ns … 312.634 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     189.200 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   189.812 ns ±   8.858 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%

  179 ns        Histogram: log(frequency) by time        239 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

In [25]:

```
poly_speedup1 = median(poly_mark0.times) / median(poly_mark1.times)

table2 = DataFrame("Method"=>["Naive","Horner"],"Speedup" => [1.0, poly_speedup1]);

print(table2)
```

```
2×2 DataFrame
 Row │ Method  Speedup
     │ String  Float64
─────┼─────────────────
   1 │ Naive    1.0
   2 │ Horner  43.5545
```

We can improve on the Horner implementation further by noting that the generation of the $\{b_i\}$ coefficients is a recursive process that can be iterated by the function `muladd(x, y, z)`, which performs the operation `x * y + z`. This allows us to write the Horner method as the following Julia macro:

In [26]:

```
macro horner(x, a...)
    ex = esc(a[end])
    for i in length(a)-1:-1:1
        ex = :(muladd(t,$(ex), $(esc(a[i]))))
    end
    Expr(:block, :(t=$(esc(x))), ex)
end
```

Out[26]:

```
@horner (macro with 1 method)
```

In [27]:

```
f_horner_macro(x) = @horner(x, coeffs)
```

Out[27]:

```
f_horner_macro (generic function with 1 method)
```

Benchmarking its performance for $x = 3.5$ gives us a significant three orders of magnitude faster code compared to the naive implementation. It is also two orders of magnitude faster compared to the function implementation of the Horner method.

In [28]:

```
poly_mark2 = @benchmark f_horner_macro($x)
```

Out[28]:

```
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range (min … max):  1.374 ns … 15.141 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     1.643 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   1.624 ns ±  0.314 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%

  1.37 ns        Histogram: frequency by time        1.72 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

In [29]:

```
poly_speedup2 = median(poly_mark0.times) / median(poly_mark2.times)

push!(table2, ["Macro", poly_speedup2]);

print(table2)
```

```
3×2 DataFrame
 Row │ Method  Speedup
     │ String  Float64
─────┼─────────────────
   1 │ Naive      1.0
   2 │ Horner    43.5545
   3 │ Macro   5015.52
```

To put this to scale, suppose we let the naive implementation run for a full 24 hours/1440 minutes. The equivalent function-based Horner implementation will be able to execute the code in just around 30 minutes of runtime, while the equivalent macro-based implementation will finish the execution in just half a minute. This clearly demonstrates the advantages of implementing already efficient chunks of code as Julia macros.

In [30]:

```
for r in eachrow(table2)
    println("$(24*60/r.Speedup) mins for $(r.Method) method.")
end
```

```
1440.0 mins for Naive method.
33.062020155635096 mins for Horner method.
0.2871087919422365 mins for Macro method.
```

In [31]:

```
transform!(table2,:Speedup=>ByRow(x->24*60/x)=>:"Time(mins)")
print(table2)
```

```
3×3 DataFrame
 Row │ Method  Speedup    Time(mins)
     │ String  Float64    Float64
─────┼──────────────────────────────
   1 │ Naive      1.0     1440.0
   2 │ Horner    43.5545    33.062
   3 │ Macro   5015.52       0.287109
```