Tips & Tricks for Speed in Evolutionary Computation

Aleksander Jekic
Xavier F. C. Sánchez Díaz

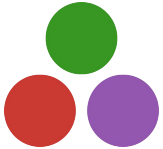
Overview

- Compilation & Profiling
- Search space, fitness function and representation
- Use a fast language
- Use the appropriate data structs
- Keep evaluation to a minimum
- Surrogate Models
- Parallelism
- Vectorization and Broadcasting
- Type Declarations
- Code Optimization

Writing high-performance code

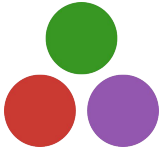
Introduction

When should you optimize code?



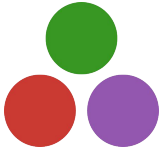
- Industry rule of thumb: Optimize only if you have to.
- Donald Knuth: "Premature optimization is the root of all evil (or at least most of it) in programming."
- Focus on readability and correctness in the first version.
  - Hard to think about optimization at the same time.
  - Hard to know what is worth optimizing in advance.
- **In Project 2, it should be more effective to make the search space smaller than to optimize code.**
  - Think algorithmically, like in the main lectures.
  - Better to do less work, than more work at a faster pace.

Why is not all code the same speed?



- Code -> Assembly -> Binary.
- The less precise your code is, the more is lost in translation. The compiler is not able to write the most efficient assembly code for your problem.
- On a low level, adding floats and integers is very different operations.
- Python 🐍: “I will make your vague code work somehow, do not worry.”
  - Mostly used as an interface for C libraries like **NumPy**.
- C++/C/Java/Rust 🚀: “If you specify everything, I will make it really fast.”
- Julia 👩‍🔬: “I see what you are trying to do. Let me try to make it fast.”
  - Julia can be thankless to optimize because often the compiler was already doing what you spent extra time spelling out.

How to know what to improve?

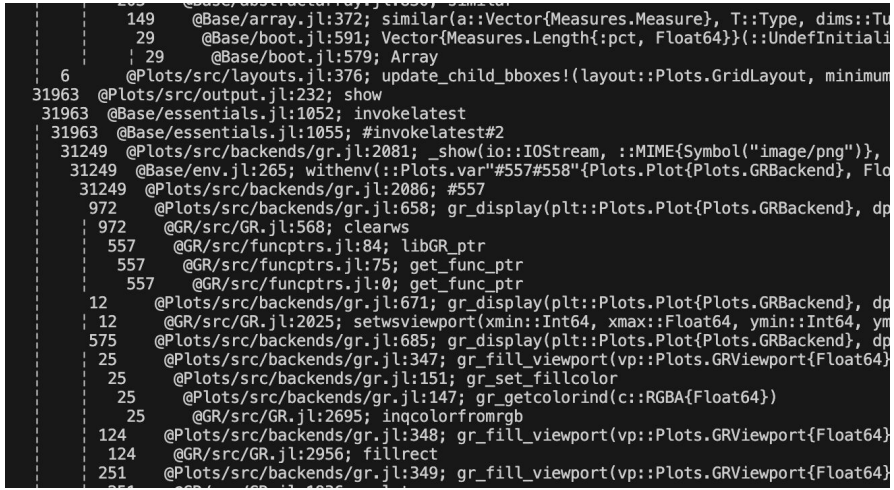


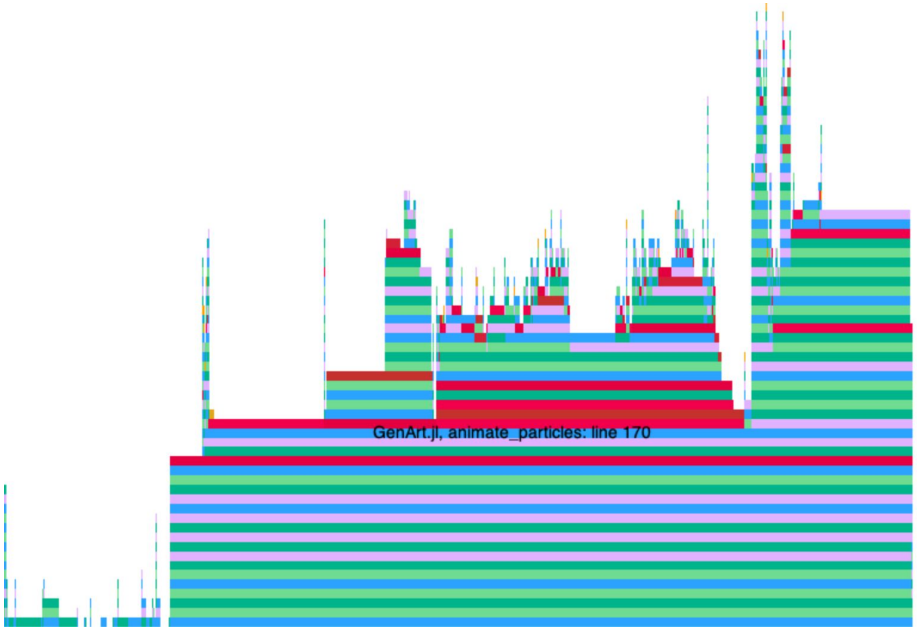
- Test your intuition quickly by commenting out what you think is a bottleneck.
- With Benchmark.jl, you can add @btime to get concrete information about performance.
- With Profile.jl, you can use @profile to see what part of your code dominates the run.
- Run your code once without @profile to avoid the compilation being included.

```
using BenchmarkTools.jl

@btime library, library_size = generate_functions()
#312.768 ns (2 allocations: 304 bytes)
```

```
using Profile
using ProfileView

#Run once to compile it
my_function()

#Profile the function
@profile my_function()

#Print profiling tree
Profile.print()

#Display flamegraph
ProfileView.view()
```

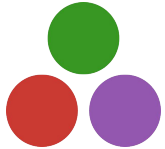
Profiling trees and flame graphs





NTNU

What is Julia👩‍🔬 actually doing?



- In Julia your tricks often will not work because the compiler is already doing it for you.
- The game in Julia is to find where the compiler failed to optimize for you.
- @code_native and @code_llvm lets you see how Julia compiles your code.
- You can learn as you go or ask ChatGPT for help to interpret the output. Note that the assembly language depends on your hardware.

```
function add_numbers(a::Int, b::Int)
    return a + b
end

@code_llvm add_numbers(3, 5)
```

```
; @ int.jl:87 within `+`
 %0 = add i64 %"b::Int64", %"a::Int64"
; 
 ret i64 %0
```

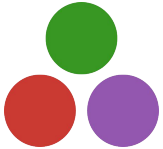
```
function add_numbers(a, b)
    return a + b
end

code_llvm(add_numbers, (Any, Any))
```

```
%jlcallframe1 = alloca [2 x ptr], align 8
 @ /Users/aleksandrajekic/Documents/Programmering/tests.jl:8 within `add_numbers`
store ptr %"a::Any", ptr %jlcallframe1, align 8
%0 = getelementptr inbounds ptr, ptr %jlcallframe1, i64 1
store ptr %"b::Any", ptr %0, align 8
%1 = call nonnull ptr @ijl_apply_generic(ptr nonnull @"jl_global#9219.jit", ptr nonnull %jlcallframe1, i32 2)
ret ptr %1
```

Modelling and Evaluation

Theoretical tips

Reduce the search space



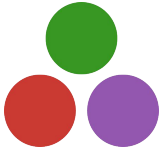
The **search space** depends both on **fitness function** and **representation!**

- **Difficulty:** depends
  - Creating a good objective function can be **incredibly difficult** from a real-world problem description
  - Choosing the right representation is **usually straight-forward** from a defined objective function but not always

Example problem:

Assign ambulances to service points (hospitals, response centres, etc.) What is the optimal assignment such that response time is minimised, given a simulation of emergency incidents?

The ambulance problem I



- X: the set of variables to be set. Each station is a variable.
- D: their domains. Each station i has a capacity w_i, and can have between 0 and w_i units.
- We have one constraint: $\sum_i x\_i \leq u$

where u is the total number of units (ambulances) available.

We have m stations, and u ambulances.

The ambulance problem II

You can represent an **individual** in two ways

A. Ambulances per stations (with check)
   a. $x=[3, 2, 1, 1, 2, 1, \ldots]$: each station has some ambulances
   b. Extra (cheap) operation: check that $\text{sum}(x) \leq u$.
B. Station id for each ambulance (no check)
   a. $x=[1, 7, 29, 12, 15, 20, ...]$: ambulance 1 goes in station 1, ambulance 2 goes in station 7…

With $m=19$ and $u=49$

- Case A has $(u-1)\text{Choose}(m-1) \cong 7.30 \times 10^{12}$ combinations
- Case B has $m^u \cong 4.55 \times 10^{62}$ combinations

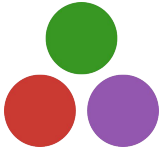
Use a fast language

Scientific computing is usually done in C/C++/Fortran.

Julia is a *newcomer* since 2017. Python is very slow.

- **Difficulty:** medium
  - You *may* have to learn a new language, but that's fun!
  - You are always looking stuff up, even when writing Python, so just jump into it.

Use the appropriate data structs



A **case study** on Ant Colony Optimisation:

- TSP on a fully connected graph with n=90 vertices
- 1000 ants, 100 iterations, random lengths (floats)
- Algorithm implementation from Kochenderfer and Wheeler, 2019 (Algorithms 9.9 & 9.10)
- Case A: using a **dictionary of lengths** (as in the book)
- Case B: using a **matrix** instead
- Same seed, same hyper-parameters

Use the appropriate data structs

Dictionary:

```
BenchmarkTools.Trial: 1 sample with 1 evaluation per sample.
 Single result which took 49.953 s (10.18% GC) to evaluate,
 with a memory estimate of 35.58 GiB, over 123869747 allocations.
```

```
@benchmark ACO(g, lengths)
```

~50 seconds
~123.8M allocations

Use the appropriate data structs

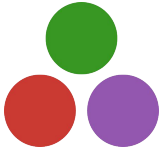
Matrix:

```
BenchmarkTools.Trial: 1 sample with 1 evaluation per sample.
 Single result which took 35.844 s (13.87% GC) to evaluate,
 with a memory estimate of 35.06 GiB, over 102500309 allocations.
```

```
@benchmark ACO(g, lengths)
```

~36 seconds
~102.5M allocations

Keep evaluation to a minimum I



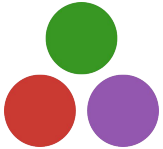
Evaluations are often very expensive.

Save fitness evaluations **per generation**: use an **array of fitnesses** or a field/attribute in a struct/object.

Do not recalculate the fitness of an individual every time you need it.

Difficulty: easy

Keep evaluation to a minimum II

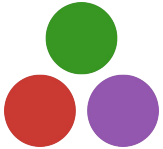


Save fitness evaluations **across generations**: use an **archive/hashmap of fitnesses** so you do not have to evaluate again.

This will save you calculations every time an individual is repeated. Since we are often exploiting as well as exploring, the same individual may occur quite often.

Difficulty: medium

Use a surrogate model



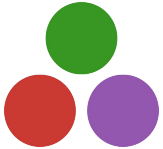
If fitness **evaluation** is **expensive**, make a simpler, faster model that serves as an *approximator*, e.g.:

1. Run multiple simulations (get a good sample)
2. Create a table
3. Train an ML model to approximate the table
4. Use the ML model as the fitness evaluation

Difficulty: hard (and time consuming), unless you have experience with the particular ML model

Example: Evaluating a trained small neural network is actually just a matrix calculation.

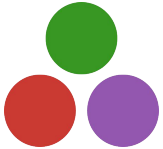
Make it Parallel



Your laptop has **multiple cores**. Use all of them!

- **Difficulty** depends on communication complexity:
  - Multithreading: easy (and can easily halve the running time of many functions)
  - Distributed: medium (for example combining the powers of your computers and trying an island model)
  - Message Passing: hard (can create a lot of new problems)

Use NTNU's HPC-cluster — IDUN



If you have a supervisor (i. e. for your master thesis), you can get permission to access NTNU’s supercomputer.
https://www.hpc.ntnu.no/idun/how-to-get-access-to-idun/

It is not really hard, but very time-consuming (like fixing your parents’s printer) if you have not done it before. Here are some guides
https://www.hpc.ntnu.no/idun/documentation/ Basic Linux experience is very helpful.

You can access Idun both with SSL and in a GUI https://apps.hpc.ntnu.no/ However, you need to be at NTNU or be using NTNU’s VPN.

Difficulty: hard

Code and Implementation

Practical tips

Make it Parallel: Multithreading

In Julia, adding `@threads` (from `Base.Threads`) allows for parallel `for` loops

C/C++ have similar pragma directives
(compiler dependent)

```
duration = @elapsed begin
    @threads for i in axes(records, 1)
        X_sub = get_columns(X, dec2ind(i; pad=m))
        res = get_fitness(model, X_sub, y; rng=myrng)
        records[i, 1] = res[1]  # accuracy
        records[i, 2] = res[2]  # time in sec
    end
end
```

Multithreading example

```julia
using BenchmarkTools
using Base.Threads

random_numbers = rand(1000000)
thread_sums = zeros(Float64, nthreads())

function threaded()

    @threads for i in 1:length(arr)
        thread_sums[threadid()] += random_numbers[i]^2
    end

    return sum(thread_sums)

end
```

```julia
function unthreaded()

    sum_of_random_squares = 0

    for i in 1:length(random_numbers)
        sum_of_random_squares += random_numbers[i]^2
    end

    return sum_of_random_squares

end


@btime threaded()
@btime unthreaded()
```

```
23.221 ms (4999526 allocations: 76.29 MiB)
63.220 ms (5998980 allocations: 106.80 MiB)
```

Using 6 cores.

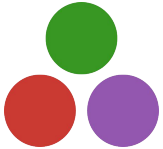
Make it Parallel: Distributed

In Julia, you can use the `Distributed` module, along with `@sync`, `@pmap` and `@everywhere`

Useful for multi-core (e.g. your laptop or an HPC cluster)

```julia
using Distributed
addprocs(4)

# @distributed is asynchronous
# use @sync to force syncing
@sync @distributed for i in 1:5
    a[i] = i
end

foo = 2  # locally defined
# Tell all procs about foo
@everywhere bar = $foo
```

Make it Parallel: Message Passing



With MPI you can *literally* decide how cores talk to each other.

You get finer detail, but it is difficult to set up.

HPC-cluster ready.

```julia
# examples/01-hello.jl
using MPI
MPI.Init()

comm = MPI.COMM_WORLD
println("Hello world, I am $(MPI.Comm_rank(comm)) of $(MPI.Comm_size(comm))")
MPI.Barrier(comm)
```

```
> mpiexecjl -n 4 julia examples/01-hello.jl
Hello world, I am rank 1 of 4
Hello world, I am rank 2 of 4
Hello world, I am rank 0 of 4
Hello world, I am rank 3 of 4
```

Some tips for Julia

- Help the compiler out:
  - Put everything into functions.
  - Smaller, individual functions are usually better.
  - Don’t change a data type mid-function.
- Julia stores whole columns in order. Accessing matrices in column-order may save you significant time. Row-order requires jumping back and forth.
- Preallocating data structures is faster than growing them dynamically.
- Dot operators are much cheaper on vectors/matrices.
  - Write .=, .+, .-, ./ instead of =, +, -, /, etc. to vectorize/broadcast.

Vectorization/Broadcasting

```julia
using BenchmarkTools

A = rand(10000, 10000)
B = rand(10000, 10000)
C = zeros(10000, 10000)

function meaningless_matrix(A, B, C)

    C = A + B

    return C
end

function dot_meaningless_broadcast(A, B, C)

    C .= A .+ B

    return C
end

@btime meaningless_matrix(A, B, C)
@btime dot_meaningless_broadcast(A, B, C)
```

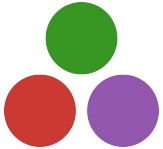
```
  59.840 ms (3 allocations: 762.94 MiB)
  18.683 ms (0 allocations: 0 bytes)
```

Use type declarations

- Declaring data types is the bread and butter for performance in most programming languages.
- However, in Julia the compiler very often does quite well even without your help.
- Always declare global variables.
- Avoid abstract types like Real. Specific is better, like Float64.
- @code_warntype in front of a function can be used to see how Julia handles your typing.

```
A = rand(Float64, 10000, 10000)
B = rand(Float64, 10000, 10000)
C = zeros(Float64, 10000, 10000)

function dot_meaningless_broadcast(A::Matrix{Float64},
    B::Matrix{Float64}, C::Matrix{Float64})::Matrix{Float64}
    C .= A .+ B
    return C
end
```

Some extra fun, nerdy ones

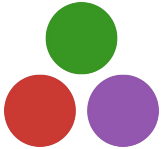


- @inbounds, does not check bounds of loop
- @fastmath, do you really need your floats that accurate?
  - If you only need a few decimals, this can save you a lot.
- @inline, remove function call overhead in assembly code.
- @simd, processor-level parallelization of loop.
- Aside from @fastmath, the compiler usually automatically applies these when they are relevant, so typically they do not help.
- You can create your own data types.

Resources for writing high-performance code

Want to learn more?

Helpful resources



- Official Julia performance tips: https://docs.julialang.org/en/v1/manual/performance-tips/
- Intro to high-performance coding: https://viralinstruction.com/posts/hardware/
- More on writing high-performance Julia: https://viralinstruction.com/posts/optimise/
- Book/slides on assembly code: https://rayseyfarth.com/asm/
- Julia practice: https://ucidatascienceinitiative.github.io/IntroToJulia/
- Great course if you are interested in scientific computing and Julia: https://book.sciml.ai

How to actually get good

- (Do this in your spare time for fun, if you want to.)
- **Difficulty:** Extremely hard
- Implement a famous, simple algorithm from scratch.
  - Try to beat libraries on speed and memory.
- Some suggestions: Operators for evolutionary algorithms, Dormand-Prince method, neural network with backpropagation.
- Try to get commits accepted for libraries on Github. Don't give up.
- One last time: **In Project 2, it should be more effective to make the search space smaller than to optimize code.**
  - In other words**, think more like in the main lectures!**