Mathematical diseases in climate models and how to cure them

Ali Ramadhan
Valentin Churavy

CLiMA
CLIMATE MODELING ALLIANCE

EAPS
Earth, Atmospheric and Planetary Sciences

MIT

julia

CSAIL
Sea ice formation

2 km
Lots of parameterizations = lots of parameters!

- There are 20 parameters in this ocean vertical mixing parameterization!
- Many parameterizations so we end up with 100~1000 tunable parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{RI}^b$</td>
<td>0.3</td>
<td>Bulk Richardson number criterion</td>
</tr>
<tr>
<td>$C_{SL}^b$</td>
<td>0.1</td>
<td>Surface layer fraction</td>
</tr>
<tr>
<td>$C_E^b$</td>
<td>3.19</td>
<td>Unresolved kinetic energy constant</td>
</tr>
<tr>
<td>$C_{NL}^b$</td>
<td>6.33</td>
<td>Non-local flux proportionality constant</td>
</tr>
<tr>
<td>$C_T^b$</td>
<td>0.4</td>
<td>Wind mixing constant / von Karman parameter</td>
</tr>
<tr>
<td>$C_{stab}^a$</td>
<td>2.0</td>
<td>Proportionality constant for effect of stable buoyancy forcing on wind mixing</td>
</tr>
<tr>
<td>$C_{nst}^a$</td>
<td>1.0</td>
<td>Exponent for effect of stable buoyancy forcing on wind mixing</td>
</tr>
<tr>
<td>$C_{nst}^c$</td>
<td>6.4</td>
<td>Proportionality constant for effect of unstable buoyancy forcing on wind mixing</td>
</tr>
<tr>
<td>$C_{U}^{T,T}$</td>
<td>0.25</td>
<td>Exponent for effect of unstable buoyancy forcing on wind mixing of momentum</td>
</tr>
<tr>
<td>$C_{T}^{T,T}$</td>
<td>0.5</td>
<td>Exponent for effect of unstable buoyancy forcing on wind mixing of momentum</td>
</tr>
<tr>
<td>$C_U^b$</td>
<td>0.599</td>
<td>Convective mixing constant for momentum</td>
</tr>
<tr>
<td>$C_T^b$</td>
<td>1.36</td>
<td>Convective mixing constant for scalars</td>
</tr>
<tr>
<td>$C_{U}^d$</td>
<td>0.5</td>
<td>Transitional normalized depth for unstable mixing of momentum</td>
</tr>
<tr>
<td>$C_{T}^d$</td>
<td>2.5</td>
<td>Transitional normalized depth for unstable mixing of scalars</td>
</tr>
<tr>
<td>$C_{U}^{mb}$</td>
<td>0.33</td>
<td>Exponent for effect of wind on convective mixing of momentum</td>
</tr>
<tr>
<td>$C_T^{mb}$</td>
<td>0.33</td>
<td>Exponent for effect of wind on convective mixing of scalars</td>
</tr>
<tr>
<td>$K_{u0}$</td>
<td>$10^{-5}$</td>
<td>Interior/background turbulent diffusivity for momentum</td>
</tr>
<tr>
<td>$K_{T0}$</td>
<td>$10^{-5}$</td>
<td>Interior/background turbulent diffusivity for temperature</td>
</tr>
<tr>
<td>$K_{S0}$</td>
<td>$10^{-5}$</td>
<td>Interior/background turbulent diffusivity for salinity</td>
</tr>
</tbody>
</table>

Source: Greg Wagner, OceanTurb.jl documentation
Lots of parameterizations = lots of parameters!

- Climate models are tuned to reproduce 20th century then run forward to 2300.
- This is not very scientific...

Source: IPCC AR5
To truly simulate the atmosphere and ocean...

- Direct numerical simulation requires grid spacing of ~1 mm.
- Ocean volume is 1.35 billion km$^3$. Atmosphere is >5 billion km$^3$.
- **We need ~10$^{28}$ grid points.**
- Not enough compute power or storage space in the world!
To truly simulate the atmosphere and ocean...

- Climate models use \( \sim 10^8 \) grid points to be fast.
- Climate models are expensive to run:
  - Need to run for 1,000~10,000 years.
  - Need to run \( \sim 100 \) simulations to calculate statistics.
- For now, parameterizations are the way to go.
Idea 1: Optimize parameters with physics and observations

- Parameterizations should at least agree with basic physics and observations.
- Use high-resolution simulations to train and build parameterizations.
Oceananigans.jl

A fast and friendly incompressible fluid flow solver in Julia that can be run in 1-3 dimensions on CPUs and GPUs.

gpu  climate  julia  ocean  fluid-dynamics  climate-change

Julia  MIT  31  189  90 (5 issues need help)  9  Updated 3 days ago
Idea 2: Neural differential equations for climate parameterizations

- Use a neural network $\mathbb{NN}$ in a differential equation for physics we don’t know.

- Equation climate model needs to solve:

$$
\partial_t \overline{T} + \partial_z \overline{wT} = \partial_z (\kappa \partial_z \overline{T}) + \frac{Q}{\rho c_p} + ???
$$

- Possible parameterizations

$$(1) \quad \partial_t \overline{T} = \mathbb{NN}(\overline{T}, \ldots)$$

$$(2) \quad \partial_t \overline{T} = \partial_z \left[ \mathbb{NN}(\overline{T}, \partial_z \overline{T}, \ldots) \right] + \frac{Q}{\rho c_p}$$
Still a work-in-progress!

\[
\partial_t \overline{T} = \text{NN}(\overline{T}, \ldots) \quad \partial_t \overline{T} = \partial_z \left[ \text{NN}(\overline{T}, \partial_z \overline{T}, \ldots) \right] + \frac{Q}{\rho c_p}
\]
Why I like Julia as a climate modeler

- User interface and model backend all in one language.
- Our Julia model is as fast as our legacy Fortran model.
- Native GPU compiler: single code base compiles to both CPU and GPU!
- More productive development and more powerful user API.
- Multiple dispatch makes our software easy to extend/hack.
- Sizable Julia community interested in scientific computing.
Climate modeling: Why so much uncertainty?

- Most uncertainty in climate predictions is due to humans.
- Huge model uncertainty is due to *missing physics*.
- Cannot resolve every cloud and every ocean wave so we must parameterize these things.
- We can try to use most of the computing power to make sure parameterizations reproduce basic physics and observations.
- Will this lead to better climate predictions?
  - Maybe, maybe not. But hopefully we can get rid of “model tuning” and make software development for climate modeling easier.
Looking to buy/rent a bicycle!
How can we help?

http://worrydream.com/ClimateChange/
Tools for scientists & engineers

Languages for technical computing

R and Matlab are both forty years old, weighed down with forty years of cruft and bad design decisions. Scientists and engineers use them because they are the vernacular, and there are no better alternatives. [...] it’s only slightly an unfair generalization to say that almost every programming language researcher is working on:

1. languages and methods for software developers
2. languages for novices or end-users,
3. implementation of compilers or runtimes, or
4. theoretical considerations, often of type systems.

Equation

\[
\begin{align*}
\text{SE A-level pressure filter on } P \\
\text{Source: } \text{http://worrydream.com/ClimateChange/}
\end{align*}
\]
“The limits of my language are the limits of my world”
(Wittgenstein)
My port of our research code from CPU-based C++ to GPU-accelerated JuliaLang is so much faster (1 week -> 1 hour walltimes) and so much easier to add stuff to... it's a nice holiday gift to myself 🌬️ 🌟.
Yet another high-level language?

- Dynamically typed, high-level syntax
- Open-source, permissive license
- Built-in package manager
- Interactive development

```julia
function mandel(z)
    c = z
    maxiter = 80
    for n = 1: maxiter
        if abs(z) > 2
            return n - 1
        end
        z = z^2 + c
    end
    return maxiter
end

julia> mandel(complex(.3, -.6))
14
```
### Yet another high-level language?

<table>
<thead>
<tr>
<th>Typical features</th>
<th>Unusual features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamically typed, high-level syntax</td>
<td>Great performance!</td>
</tr>
<tr>
<td>Open-source, permissive license</td>
<td>JIT AOT-style compilation</td>
</tr>
<tr>
<td>Built-in package manager</td>
<td>Most of Julia is written in Julia</td>
</tr>
<tr>
<td>Interactive development</td>
<td>Reflection and metaprogramming</td>
</tr>
<tr>
<td></td>
<td>GPU code-generation support</td>
</tr>
</tbody>
</table>
Problem:
“We have to duplicate our code for GPUs and CPUs.”
Status quo

function cpu_code(A)
    for I in eachindex(A)
        A[I] = # Stencil operation
    end
end

function gpu_code(A)
    I = threadIdx().x + # ...
    A[I] = # Stencil operation
end

My answer: Why don’t you just?

1. Separate each kernel (for-loop body) into a new function
2. Add a general way to call GPU kernels
3. Profit!
Kernel!

```plaintext
My answer: Why don't you just?

kernel!(A, I, args...) = # Stencil op

function launch(::CPU, f, A, args...) for I in eachindex(A)
    f(A, I, args...)
end

function launch(::GPU, f, A, args...) for I in eachindex(A)
    f(A, I, args...)
end

N = length(A)
threads = min(N, 128)
blocks = ceil(Int, N / threads)
@cuda threads=threads, blocks=blocks kernelf(f, A, args...)
end

function kernelf(f::F, A, args...) where F
    I = threadIdx().x + # ...
    I > length(A) && return nothing
    f(A, I, args...)
    return nothing
end
```
This didn’t work

- Kernel fusion.
  - Reduce number of \textit{global} memory loads and stores.

- GPU functionality & low-level control.
  - \textit{Shared} memory.
  - Register shuffling.
  - Reasoning about \textit{warp} and \textit{thread} level data access.
  - More aggressive inlining on the GPU.
My answer could have been

Ideal solution:

- Let’s write a bespoke language and compiler.
- Domain-Specific-Language for climate simulations.
  - Finite Volume.
  - Discontinuous Galerkin.
- 2+ years development time ;}
Real solution: A botch

- Minimal effort, quick delivery.
- Needs to be extensible and hackable.
- Get the job done now.

- Julia is good at these kinds of hacks.
- And you can let them grow into bespoke solutions.

To botch: to put together in a makeshift way.
CLIMA: GPUifyLoops.jl

- Macro based, kernel language for code that runs on CPU and GPU.
- OCCA/OpenACC-esque.
- Very minimal; primary goal: fast GPU code.

```julia
function kernel(data)
    @loop for i in (1:length(data); threadIdx().x)
        # work
    end
end
@launch(GPU(), threads=(length(data),), blocks=(1,)) kernel()
```

https://github.com/vchuravy/GPUifyLoops.jl
Implementation of @loop

macro loop(expr)
    induction = expr.args[1]
    body = expr.args[2]
    index = induction.args[2]
    
    cpuidx = index.args[1]
    # index.args[2] is a linenode
    gpuidx = index.args[3]
    # Switch between CPU and GPU index
    index = Expr(:if, (!$isdevice()), cpuidx, gpuidx)
    induction.args[2] = index

    # use cpuidx to boundscheck on GPU.
    bounds_chk = quote
        if $isdevice() && !($gpuidx in $cpuidx)
            continue
        end
    end
    pushfirst!(body.args, bounds_chk)
    
    expr = Expr(:for, induction, body)
    return esc(expr)
end
Why can Julia run on the GPU at all

- Support for staged programming:
  User generates code for a specific call-signature.
- Introspection & Reflection.
- Build upon LLVM.
  LLVM.jl allows you to generate your own LLVM module and inject it back into Julia.
- Dynamic language that tries to avoid runtime uncertainties and provides tools to understand the behaviour of code.
Avoid runtime uncertainty

1. Sophisticated type system
2. Type inference
3. Multiple dispatch
4. Specialization
5. JIT compilation

Julia: Dynamism and Performance
Reconciled by Design (doi:10.1145/3276490)
Introspection and staged metaprogramming

- String macros
- Macros
- Generated functions
  Cassette.jl passes
- LLVM IR
  LLVMcall
  LLVM.jl
- Assembly

@edit
@which
@code_lowered
@code_warntype
@code_typed optimize=false
@code_typed
@code_llvm optimize=false
@code_llvm
@code_native
Dynamic semantics + Static analysis

julia> function mandel(z)
    c = z
    maxiter = 80
    for n = 1:maxiter
        if abs(z) > 2
            return n-1
        end
    end
    z = z^2 + c
    return maxiter
end
julia> mandel(UInt32(1))
2

julia> methods(abs)
# 13 methods for generic function "abs":
[1] abs(x::Float64) in Base at float.jl:522
[2] abs(x::Float32) in Base at float.jl:521
[3] abs(x::Float16) in Base at float.jl:520
...
[13] abs(z::Complex) in Base at complex.jl:260

Everything is a virtual function call?
What happens on a call

\[
\text{sin}(x) \quad \text{methods}(	ext{sin})
\]
# 12 methods for generic function "sin":

1. \text{sin}(x::\text{BigFloat}) \text{ in Base.MPFR at mpfr.jl:743}
2. \text{sin}(::\text{Missing}) \text{ in Base.Math at math.jl:1072}
3. \text{sin}(a::\text{Complex}\{\text{Float16}\}) \text{ in Base.Math at math.jl:1020}
4. \text{sin}(a::\text{Float16}) \text{ in Base.Math at math.jl:1019}
5. \text{sin}(z::\text{Complex}\{T\}) \text{ where } T \text{ in Base at complex.jl:796}
6. \text{sin}(x::T) \text{ where } T<:\text{Union}\{\text{Float32}, \text{Float64}\} \text{ in Base.Math at special/trig.jl:30}
7. \text{sin}(x::\text{Real}) \text{ in Base.Math at special/trig.jl:53}

The right method is chosen using dispatch and then a method specialization is compiled for the signature.
Multiple dispatch

Rule: Call most specific method

\[
\begin{align*}
  f(x, y::\text{Int}) &= 0 \\
  f(x::\text{Int}, y) &= 1 \\
  f(x, y::\text{Float64}) &= 2 \\
\end{align*}
\]

julia> f(1, "hello")
1

julia> f("hello", 1.0)
2

julia> f(1, 1.0)
ERROR: MethodError: 
  f(::\text{Int64}, ::\text{Float64}) is ambiguous.
Candidates:
  f(x, y::\text{Float64}) in Main at REPL[2]:1
  f(x::\text{Int64}, y) in Main at REPL[3]:1
Possible fix, define
  f(::\text{Int64}, ::\text{Float64})
Method specialization

```
julia> ml = methods(sin);
julia> m = ml.ms[6]
sin(x::T) where T<:Union{Float32, Float64} in Base.Math at special/trig.jl:30
julia> m.specializations

julia> sin(1.0);
julia> m.specializations
TypeMapEntry(..., Tuple{typeof(sin), Float64}, ..., MethodInstance for sin(::Float64), ...)
```

Julia specializes and compiles methods on concrete call signatures.
**Dynamic semantics + Static analysis**

```
julia> function mandel(z)
    c = z
    maxiter = 80
    for n = 1:maxiter
        if abs(z) > 2
            return n-1
        end
        z = z^2 + c
    end
    return maxiter
end
julia> mandel(UInt32(1))
2
```
Dynamic semantics + Static analysis

julia> function mandel(z)
    c = z
    maxiter = 80
    for n = 1:maxiter
        if abs(z) > 2
            return n-1
        end
        z = z^2 + c
    end
    return maxiter
end

julia> mandel(UInt32(1))
2
Dynamic semantics + Static analysis

```
julia> function mandel(z)
    c = z
    maxiter = 80
    for n = 1:maxiter
        if abs(z) > 2
            return n-1
        end
        z = z^2 + c
    end
    return maxiter
end
```

```
julia> mandel(UInt32(1))
2
```

```
julia> @code_typed optimize=false mandel(UInt32(1))
1 ─ (z@_7 = z@_2)::UInt32
    (c = z@_7)::UInt32
    (maxiter = 80)::Compiler.Const(80, false)
    %4 = (1:maxiter)::Compiler.Const(1:80, false)
    (%5 = Base.iterate(%4))::Compiler.Const((1, 1), false)
    %6 = (%5 === nothing)::Compiler.Const(false, false)
    %7 = Base.not_int(%6)::Compiler.Const(true, false)
    goto #6 if not %7
2 ─ %9 = _5::Tuple(Int64,Int64)::Tuple(Int64,Int64)
    (n = Core.getfield(%9, 1))::Int64
    %11 = Core.getfield(%9, 2)::Int64
    %12 = Main.abs(z@_7)::UInt32
    %13 = (%12 > 2)::Bool
    goto #4 if not %13
3 ─ %15 = (n - 1)::Int64
    return %15
4 ─ %17 = z@_7::UInt32
    %18 = Core.apply_type(Base.Val, 2)::Compiler.Const(Val{2}, false)
    %19 = (%18())::Compiler.Const(Val{2}(), false)
    %20 = Base.literal_pow(Main.^, %17, %19)::UInt32
    (z@_7 = %20 + c)::UInt32
    (%5 = Base.iterate(%4, %17)::Union{Nothing, Tuple{Int64,Int64}})
    %23 = (%5 === nothing)::Bool
    %24 = Base.not_int(%23)::Bool
    goto #6 if not %24
5 ─ goto #2
6 ─ return maxiter::Core.Compiler.Const(80, false)
```
Dynamic semantics + Static analysis

julia> function mandel(z)
    c = z
    maxiter = 80
    for n = 1:maxiter
        if abs(z) > 2
            return n-1
        end
        z = z^2 + c
    end
    return maxiter
end

julia> mandel(UInt32(1))
2
Julia static analysis

“Julia is a dynamic language and follows dynamic semantics — Never forget”

Type-inference as an optimization to find static (or near static) subprograms

- Aggressive de-virtualization
- Inlining
- Constant propagation

Raises problem of cache invalidation.
Julia secrets — Cache invalidation

“Julia is a dynamic language and follows dynamic semantics — Never forget”

Julia 0.3

```
 julia> f() = 1
 julia> g() = f()
 julia> g()
 1

 julia> f() = 2
 julia> g()
 1
```

Julia 1.0

```
 julia> f() = 1
 julia> g() = f()
 julia> g()
 1

 julia> f() = 2
 julia> g()
 2
```

https://github.com/JuliaLang/julia/issues/265
Julia secrets—Constant propagation

“Julia is a dynamic language and follows dynamic semantics — Never forget”

```julia
julia> f() = sin(2.0)
f (generic function with 1 method)

julia> @code_typed f()
CodeInfo(1
  1 - return 0.9092974268256817
) => Float64
```
Julia secrets
“Julia is a dynamic language and follows dynamic semantics — Never forget”

```julia
julia> f(t) = ntuple(length(t)) do i
        sin(t[i])
    end
f (generic function with 1 method)

julia> @code_typed f((1.0, 2.0f0, 3+1im))
CodeInfo(1 ─ %1 = %new(var"#5#6"{Tuple{Float64,Float32,Complex{Int64}}}, t)::var"#5#6"{Tuple{Float64,Float32,Complex{Int64}}}
   │ %2 = invoke Main.ntuple(%1::var"#5#6"{Tuple{Float64,Float32,Complex{Int64}}}, 3::Int64)::Tuple
   └── return %2)
) => Tuple

=> Heuristic decided not to specialize
```
Julia secrets— Force specialization
“Julia is a dynamic language and follows dynamic semantics – Never forget”

julia> f(t) = ntuple(Val(length(t))) do i
    Base.@_inline_meta
    sin(t[i])
end
f (generic function with 1 method)

julia> @code_typed f((1.0, 2.0f0, 3+1im))
CodeInfo(
  1 ─ %1 = Base.getfield(t, 1, true)::Float64
  |  %2 = invoke Main.sin(%1::Float64)::Float64
  |  %3 = Base.getfield(t, 2, true)::Float32
  |  %4 = invoke Main.sin(%3::Float32)::Float32
  |  %5 = Base.getfield(t, 3, true)::Complex{Int64}
  |  %6 = invoke Main.sin(%5::Complex{Int64})::Complex{Float64}
  |  %7 = Core.tuple(%2, %4, %6)::Tuple{Float64,Float32,Complex{Float64}}
         return %7
) => Tuple{Float64,Float32,Complex{Float64}}
Julia secrets

“Julia is a dynamic language and follows dynamic semantics — Never forget”

Concrete types are not extendable \(\text{Int64 <: Number <: Any}\)

- Dynamic semantics implies no closed-world semantics
- Enables more aggressive de-virtualization
- Data can be stored inline/consecutively in memory

You can’t inherit from \(\text{Int64}\), but you can subtype \(\text{Signed}\)

Julia uses multiple-dispatch and for de-virtualization we need \textbf{final} call signatures.
Why Julia?
Walks like Python, talks like Lisp, runs like Fortran

- Hackable & extendable language
  - Metaprogramming & staged programming
- Built upon LLVM
  - My “favourite LLVM” frontend
- Users in scientific computing
- Open development & MIT license

**Personal goal:** Enable scientists/engineers and CS/HPC experts to collaborate efficiently

https://www.nature.com/articles/d41586-019-02310-3
Valentin Churavy (@vchuravy)
Ali Ramadhan (@ali-ramadhan)

Thanks to: Tim Besard, Chris Hill, Jarret Revels, Jean-Michel Campin, Jameson Nash, Greg Wagner, Nathan Daly, John Marshall, Jane Herriman, Andre Souza, Jeff Bezanson, Raffaele Ferrari, Alan Edelman, Lucas Wilcox, Keno Fischer, Simon Bryne, Kiran Pamnany, and many others!

vchuravy@csail.mit.edu
alir@mit.edu
JuliaCon: Yearly user and developer meetup

2019: Baltimore, MD ~360 attendees
2020: 27th - 31st of July, 2020, Lisbon, Come join us!