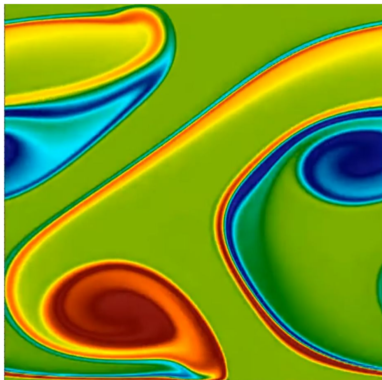
Oceananigans: A Fast, Friendly and Flexible Library to Simulate Ocean Dynamics

Francis J. Poulin, University of Waterloo
Gregory Wagner
Navid Constantinou
Simone Silvestri
And many others

Scientific Areas of Interest:

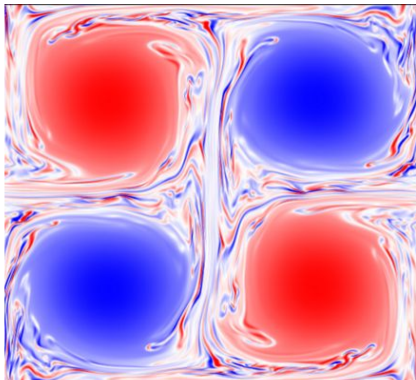
Driving my work in scientific computing

Ocean Dynamics



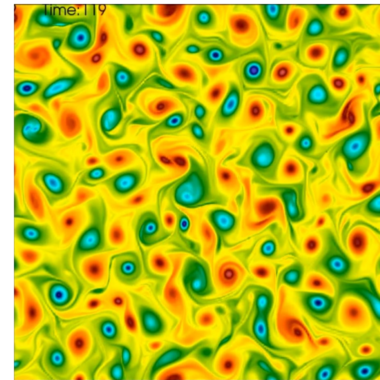
- Ocean circulation, vortices and jets
- Air-sea-ice interactions
- Biogeochemical systems (plankton, carbon)

Astrophysical Flows



- Solar tachocline dynamics (vortices, jets)
- Planet formation from protoplanetary disks
- Oceans on icy moons

Computing and Software



- Physics informed machine learning
- Open-source scientific computing
- Software for education

Unifying theme: multiscale nonlinear fluid dynamics

My Early Experiences with Computing:

How I learned (the hard way)

❌ Avoiding Computation:

Undergrad and Masters

- Minimal computing
- Learned lots of theory
- Avoided computing

⚠️ Turning Point:

PhD at MIT

- Right Whale migration in Cape Cod Bay
- At a crossroads:

Give up?
or
Learn to compute?

✅ Learned Quickly:

PhD

- Numerical methods (Linear Algebra, PDEs)
- Scientific programming (C, Matlab)
- First simulations

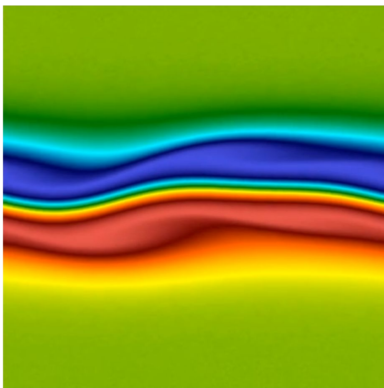
This experience shaped how I approach scientific computing and software

My first Shallow Water solver:

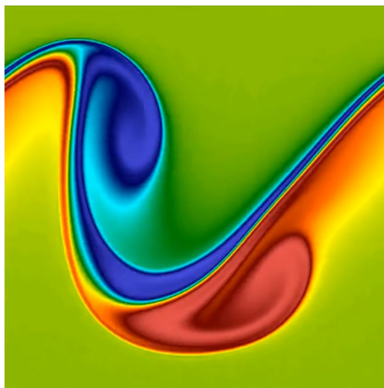
But we can do better, so much better

Francis Poulin

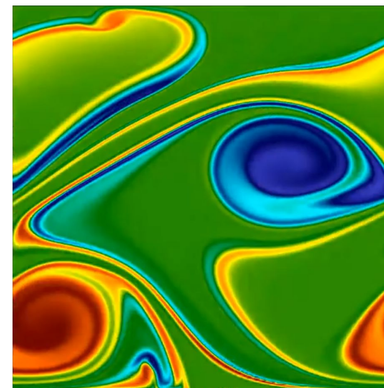
Prototyping		Efficiency		Workflow
Matlab		C		Shell scripts



- Finite difference method
- Wrote code in Matlab
- Generated plots



- Rewrote in C
- Slow development
- Data handling in binary



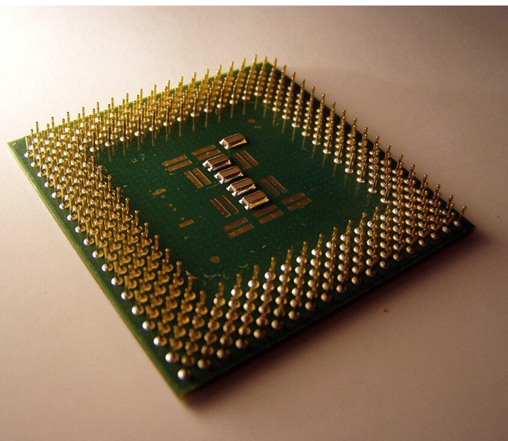
- Shell scripts
- Calculate in C
- Visualize in Matlab

Three languages. One model. Too much work.

Hardware available today

CPU

- Many libraries available
- Not accelerating as fast as Moore's law

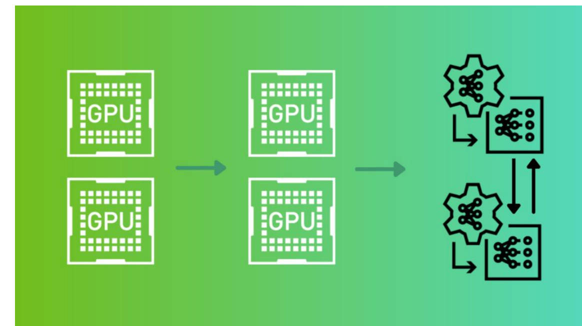


GPU

- Can be x100 faster
- Needs different libraries

Multi-GPU

- MPI + GPU is powerful
- Harder to set up



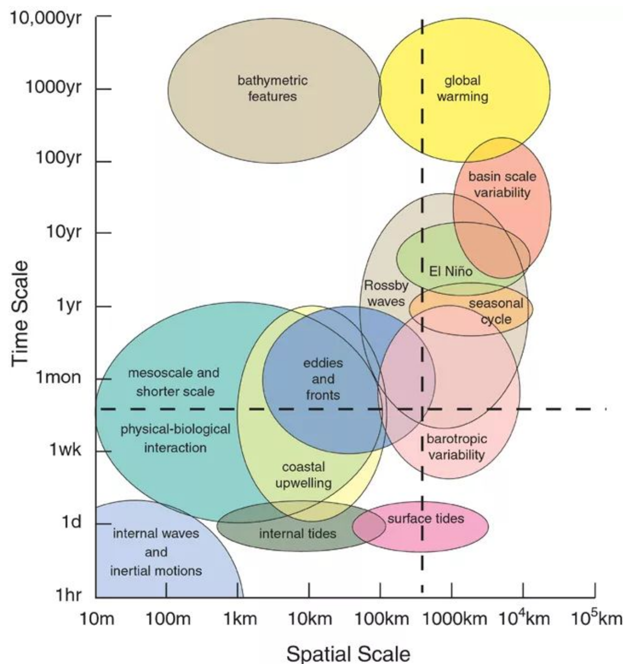
Too soon to add quantum computing but it's coming!

Sources of Error in Ocean Models

Francis Poulin

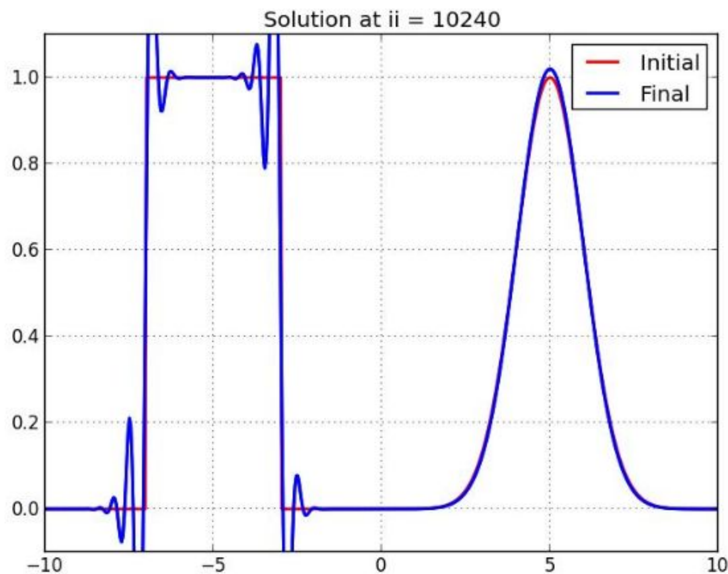
Physical Model Equations

- Models approximate the real world
- Parameterize subgrid scale processes



Numerical Methods

- Spatial and temporal errors
- Data assimilation errors



Levels of abstraction in computing:

Scientific computing spans a spectrum of abstraction

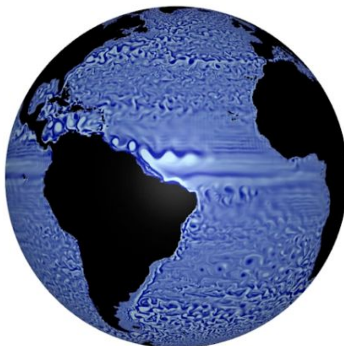
Francis Poulin

Low-level numerical

```
46:   PetscCall(MatSetValues(*B, 1, &n, n, indices, vv, INSERT_VALUES));
47:   PetscCall(MatSetValues(*B, n, indices, 1, &n, vv, INSERT_VALUES));
48:   PetscCall(MatSetValues(*B, 1, &n, 1, &n, &c, INSERT_VALUES));

50:   PetscCall(MatAssemblyBegin(*B, MAT_FINAL_ASSEMBLY));
51:   PetscCall(MatAssemblyEnd(*B, MAT_FINAL_ASSEMBLY));
52:   PetscCall(VecRestoreArrayRead(v, &vv));
53:   PetscCall(PetscFree(cnt));
54:   PetscCall(PetscFree(indices));
55:   PetscFunctionReturn(PETSC_SUCCESS);
56: }

58: int main(int argc, char **args)
59: {
60:   Mat          A, B;
61:   PetscViewer fd;                        /* viewer */
62:   char         file[PETSC_MAX_PATH_LEN]; /* input file name */
63:   PetscBool    flg;
64:   Vec          v;

66:   PetscFunctionBeginUser;
67:   PetscCall(PetscInitialize(&argc, &args, NULL, help));
68:   /*
69:      Determine files from which we read the two linear systems
70:      (matrix and right-hand-side vector).
71:   */
72:   PetscCall(PetscOptionsGetString(NULL, NULL, "-f0", file, sizeof(file), &flg));
```

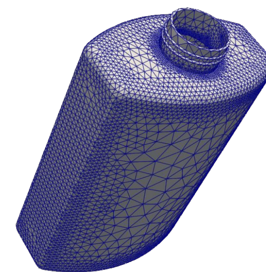
- Linear solvers (PETSc)
- Optimized, scalable
- Flexible, you develop

Legacy models



- Mature general circulation models
- Very rigid in structure

Programmable



- Equations in code
- High-level abstractions
- Flexible and performant

Low-level Mid-level High-level

Goal: fast, friendly and flexible

Why Oceananigans?

Hello, ocean!

MIT General Circulation Model (MITgcm, 1997)

- Foundation of modern ocean modelling
- CPU-based HPC
- Difficult to adapt

Climate Modeling Alliance (CliMA, 2017)

- Need modern Earth system models
- Flexible and scalable
- Rethinking modeling

Oceananigans (2020)

- MITgcm-like but for modern computing
- Programmable, GPU
- From idealized to global

MITgcm → CliMA → Oceananigans

From legacy models → programmable frameworks

Go20 Conference 2026

Oceananigans: Programmable fluid dynamics on modern hardware

Francis Poulin

Friendly

- Readable, equation based code
- Minimal setup
- Julia language (2009)

```julia
using Oceananigans

grid = RectilinearGrid(size = (128, 128),
                       x = (0, 2π),
                       y = (0, 2π),
                       topology = (Periodic, Periodic, Flat))

model = NonhydrostaticModel(grid; advection=WENO())

ϵ(x, y) = 2rand() - 1
set!(model, u=ϵ, v=ϵ)

simulation = Simulation(model; Δt=0.01, stop_iteration=100)
run!(simulation)
```

Flexible

- Modular model design
- Adaptable physics
- Same code runs on CPU, GPU, and clusters

Laptop (CPU)

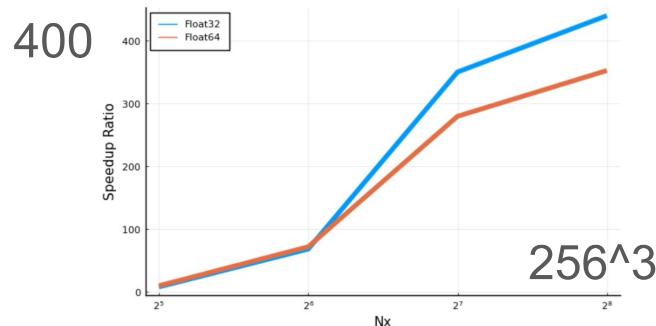
GPU node

Cluster

Fast

- GPU-acceleration
- Efficient parallelism
- JIT compiled kernels

Speed up: GPU vs CPU

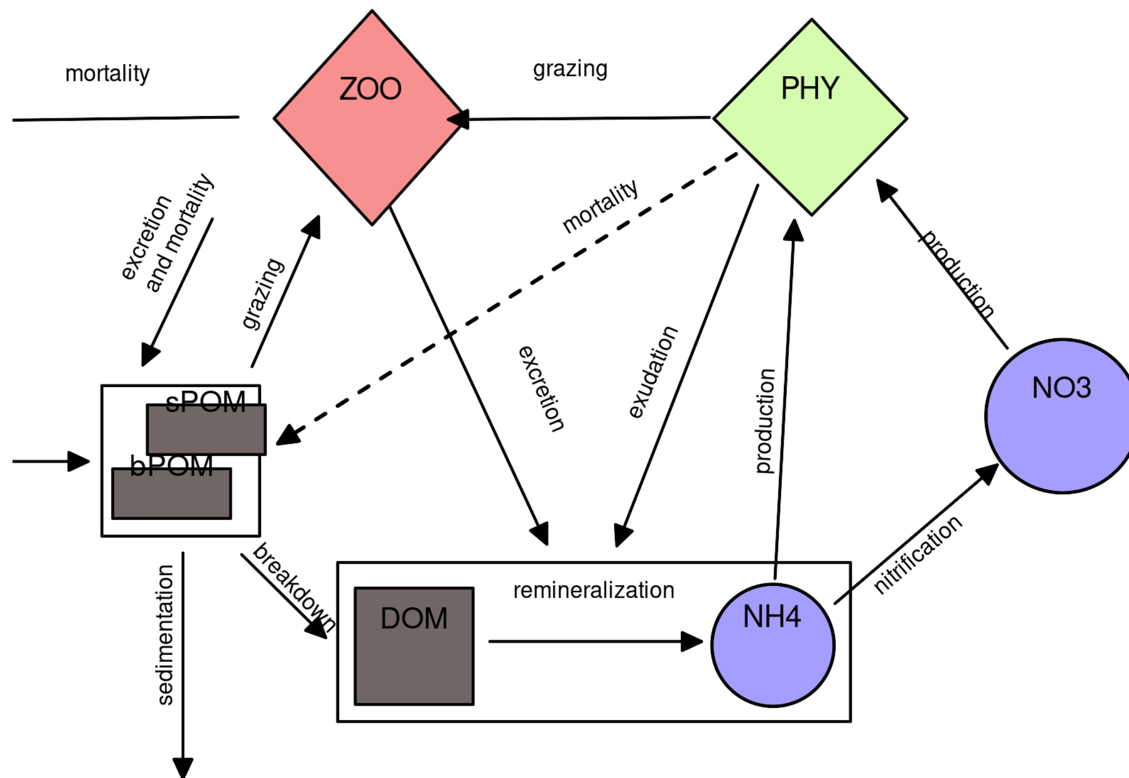


One language. One model. From laptop to supercomputer

Oceananigans vs MITgcm

Compare results with biogeochemistry

- OceanBioMe adds biogeochemistry
- MITgcm: 3 months
- Oceananigans
  - 3 hours of coding
  - 1 week
- Gained a factor of 10!

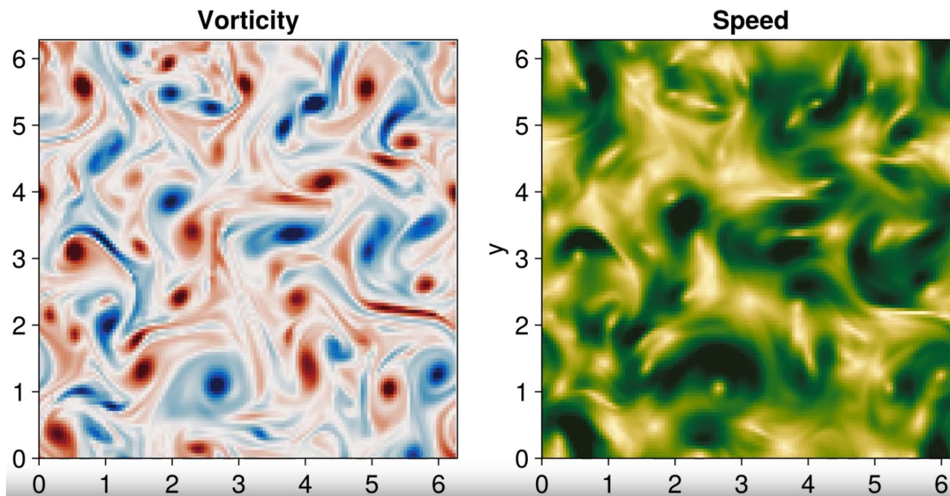


Faster code means faster science

A complete simulation in one script

One script
=
Setup
+
Run
+
Analysis
+
visualization

```julia
1  using Oceananigans, CUDA # using CUDA allows us to use an Nvidia GPU
2
3  # The third dimension is "flattened" to reduce the domain from three to two dimensions.
4  topology = (Periodic, Periodic, Flat)
5  architecture = GPU() # CPU() works just fine too for this small example.
6  x = y = (0, 2π)
7  grid = RectilinearGrid(architecture; size=(256, 256), x, y, topology)
8
9  model = NonhydrostaticModel(grid; advection=WENO(order=9))
10
11 ϵ(x, y) = 2rand() - 1 # Uniformly-distributed random numbers between [-1, 1).
12 set!(model, u=ϵ, v=ϵ)
13
14 simulation = Simulation(model; Δt=0.01, stop_time=10)
15 run!(simulation)
16
17 u, v, w = model.velocities
18 ζ = ∂x(v) - ∂y(u)
19
20 using CairoMakie
21 heatmap(ζ, colormap=:balance, axis=(; aspect=1))
```



Programmable Interface

Traditional

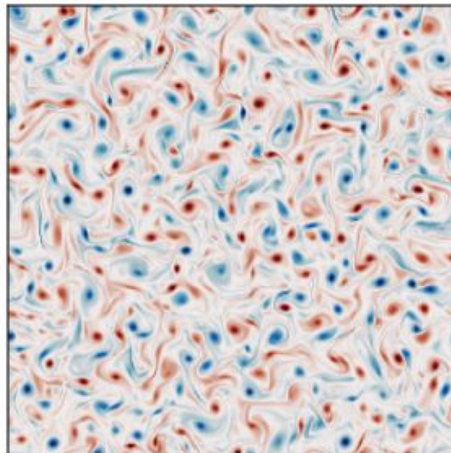
Code → config → run → post process → plot

Oceananigans

One script to do it all

```
1 function circling_source(x, y, t)
2     δ, ω, r = 0.1, 2π/3, 2
3     dx = x + r * cos(ω * t)
4     dy = y + r * sin(ω * t)
5     return exp(-(dx^2 + dy^2) / 2δ^2)
6 end
7
8 forcing = (; c = circling_source)
9 model = NonhydrostaticModel(; grid, advection=WENO(order=9), tracers=:c, forcing)
```

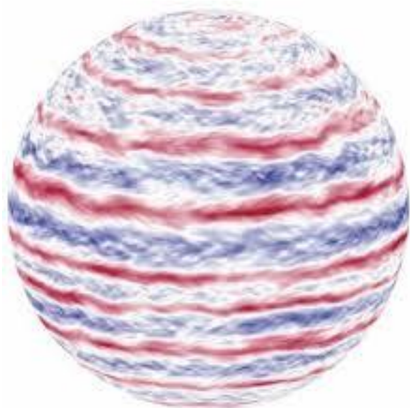
User code is part of the model



From Simple to Complex

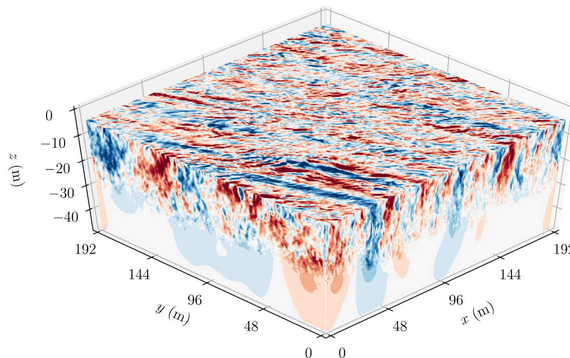
Francis Poulin

2D Turbulence



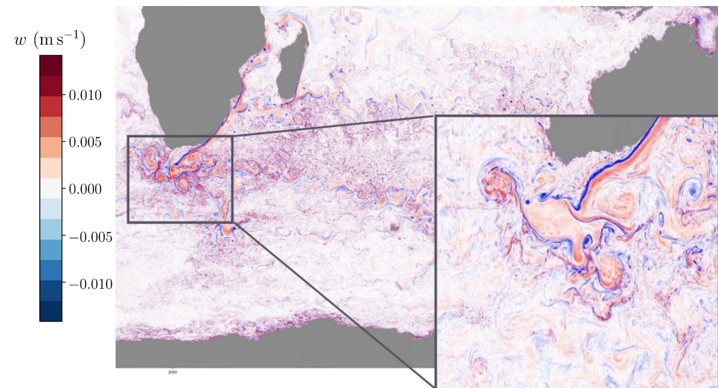
- Grid
- Initial conditions
- Forcing

3D Turbulence



- Number of dimensions
- Tracers
- Diagnostics

Ocean Turbulence



- Geometry
- Domain
- Winds

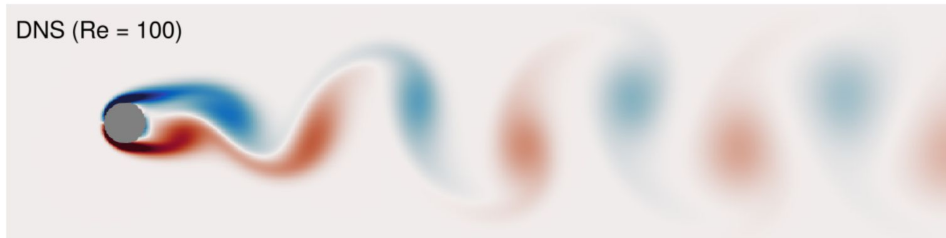
Small changes → Medium changes → Big changes

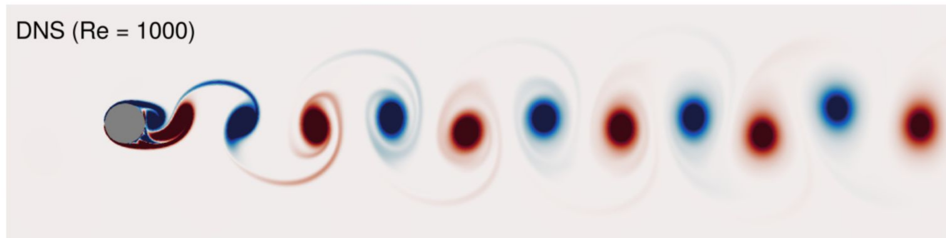
Same base code → increasing complexity

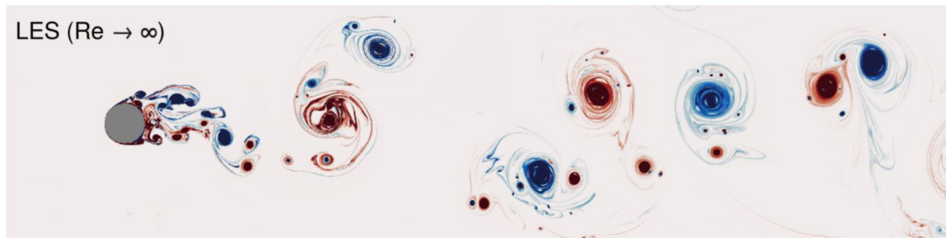
Small Scale Capability: O(1 m)

DNS → resolves all scales
LES → resolves large scales
Re is the Reynolds number

```
1  r, U, Re, Ny = 1/2, 1, 1000, 2048
2
3  grid = RectilinearGrid(GPU(), size=(2Ny, Ny), x=(-3, 21), y=(-6, 6),
4                         topology=(Periodic, Bounded, Flat))
5
6  cylinder(x, y) = (x^2 + y^2) ≤ r^2
7  grid = ImmersedBoundaryGrid(grid, GridFittedBoundary(cylinder))
8
9  closure = ScalarDiffusivity(ν=1/Re)
10
11 no_slip = FieldBoundaryConditions(immersed=ValueBoundaryCondition(0))
12 boundary_conditions = (u=no_slip, v=no_slip)
13
14 # Implement a sponge layer on the right side of the domain that
15 # relaxes v → 0 and u → U over a region of thickness δ
16 @inline mask(x, y, δ=3, x₀=21) = max(zero(x), (x - x₀ + δ) / δ)
17 Fu = Relaxation(target=U; mask, rate=1)
18 Fv = Relaxation(target=0; mask, rate=1)
19
20 model = NonhydrostaticModel(; grid, closure, boundary_conditions, forcing=(u=Fu, v=Fv))
```







Mesoscale Capability: O(100 km)

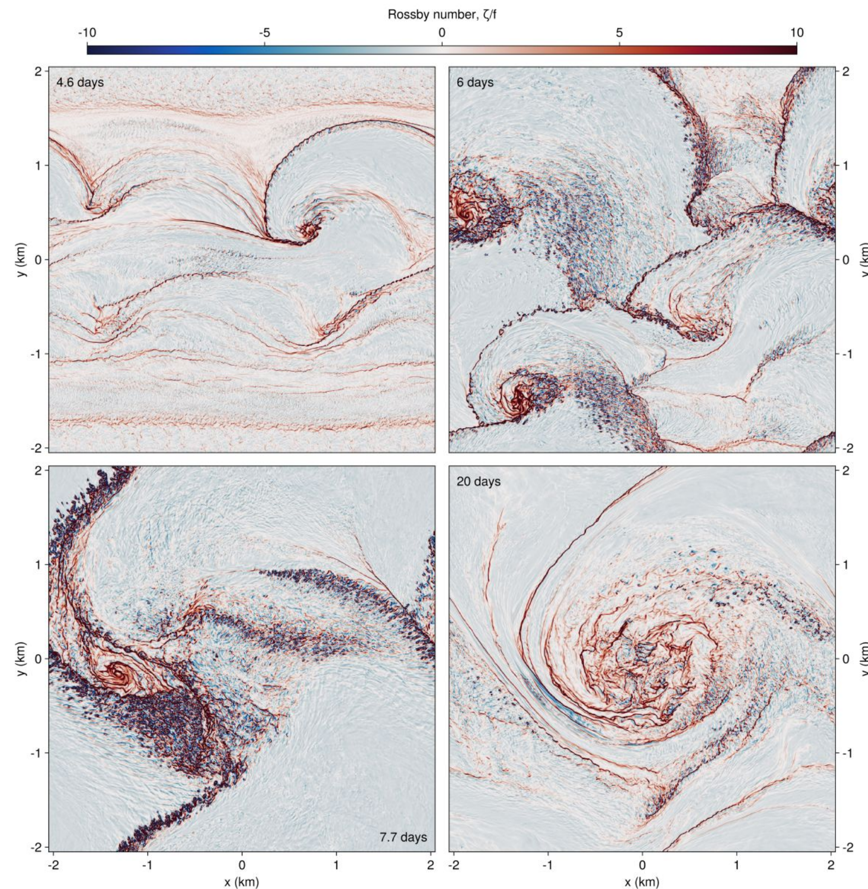
Eady turbulence is well resolved

Resolves baroclinic instability

Minimal numerical dissipation is needed

High order advection on a GPU

```
1  grid = RectilinearGrid(GPU(); size = (1024, 1024, 64),
2                               x = (0, 4096), y = (0, 4096), z = (0, 128),
3                               topology = (Periodic, Periodic, Bounded))
4
5  f, N², Ri = 1e-4, 1e-7, 1
6  parameters = (f=f, Λ=sqrt(N²/Ri)) # U = Λz, so Ri = N² / ∂z(U) = N² / Λ and Λ = N / √Ri.
7
8  @inline U(x, y, z, t, p) = + p.Λ * z
9  @inline B(x, y, z, t, p) = - p.f * p.Λ * y
10
11 background_fields = (u = BackgroundField(U; parameters),
12                      b = BackgroundField(B; parameters))
13
14 model = NonhydrostaticModel(; grid, background_fields,
15                             advection = WENO(order=9), coriolis = FPlane(; f),
16                             tracers = :b, buoyancy = BuoyancyTracer())
17
18 Δz = minimum_zspacing(grid)
19 bᵢ(x, y, z) = N² * z + 1e-2 * N² * Δz * (2rand() - 1)
20 set!(model, b=bᵢ)
```



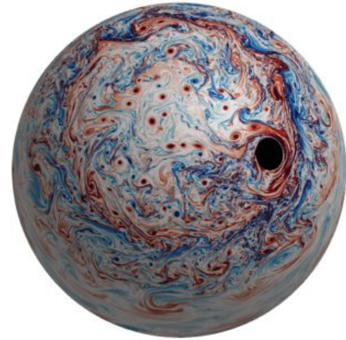
Planetary Scale Capability: O(1000 km)

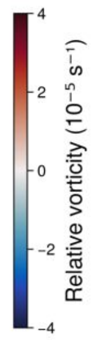
Francis Poulin

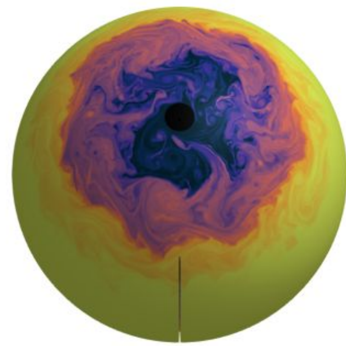
High resolution global simulations

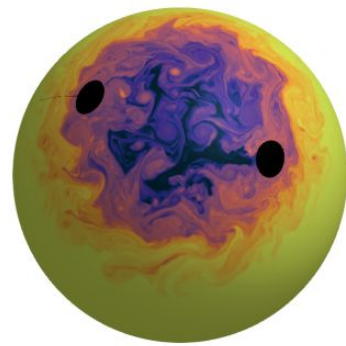
Multiple spherical grids

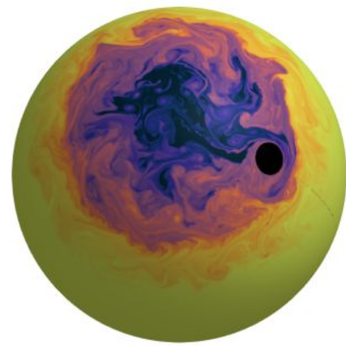
Resolves baroclinic instability

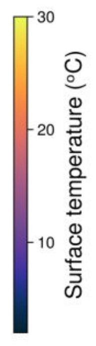












Same framework as small-scale simulations

Coupled Ocean–Sea Ice Modelling

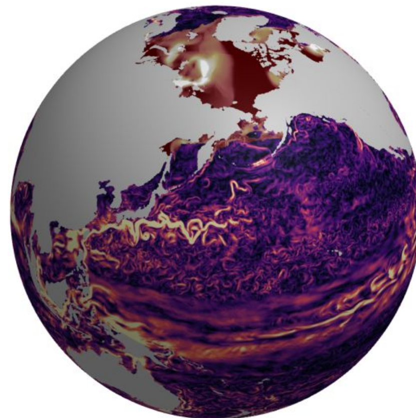
Francis Poulin

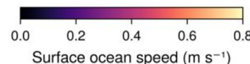
```
1  arch = GPU()
2  Nx, Ny, Nz = 2160, 1080, 60 # 1/6th degree
3  z = ExponentialDiscretization(Nz, -6000, 0, mutable=true) # z* vertical coord
4  underlying_grid = TripolarGrid(arch; z, size=(Nx, Ny, Nz), halo=(7, 7, 7))
5
6  # Smoothed ETOPO1 bathymetry; minimum_depth removes shallow cells whose
7  # gravity-wave CFL would force an impractically small time step.
8  bottom_height = NumericalEarth.regrid_bathymetry(underlying_grid;
9      minimum_depth = 10meters, interpolation_passes = 5, major_basins = 3)
10 grid = ImmersedBoundaryGrid(underlying_grid, GridFittedBottom(bottom_height);
11                             active_cells_map = true)
12
13 # Build the ocean simulation. Substeps are explicit so the gravity-wave CFL
14 # (≈ Δx_min / sqrt(g H) ≈ 14 s on subpolar 1/6° cells) is satisfied at our
15 # production Δt: 100 substeps × 9 s = Δt = 15 min covers it with margin.
16 free_surface = SplitExplicitFreeSurface(grid; substeps = 100)
17 ocean = NumericalEarth.ocean_simulation(grid;
18     momentum_advection = WENOVectorInvariant(order=9),
19     tracer_advection   = WENO(order=7),
20     equation_of_state  = TEOS10EquationOfState(),
21     free_surface)
22
23 date = CFTime.DateTimeProlepticGregorian(1993, 1, 1)
24 set!(ocean.model, T = NumericalEarth.Metadatum(:temperature, dataset=ECCO4Monthly()),
25                   S = NumericalEarth.Metadatum(:salinity,    dataset=ECCO4Monthly()))
26
27 sea_ice = NumericalEarth.sea_ice_simulation(grid, ocean; advection=WENO(order=5))
28 set!(sea_ice.model, h = NumericalEarth.Metadatum(:sea_ice_thickness,     dataset=
       ECCO4Monthly()),
29                       = NumericalEarth.Metadatum(:sea_ice_concentration, dataset=
       ECCO4Monthly()))
30
31 # Coupled with JRA55 prescribed atmosphere and a Radiation model that
32 # computes net radiative fluxes at the air-sea and air-ice interfaces.
33 atmosphere    = NumericalEarth.JRA55PrescribedAtmosphere(arch)
34 radiation     = NumericalEarth.Radiation(arch)
35 coupled_model = NumericalEarth.OceanSeaIceModel(ocean, sea_ice; atmosphere, radiation)
```

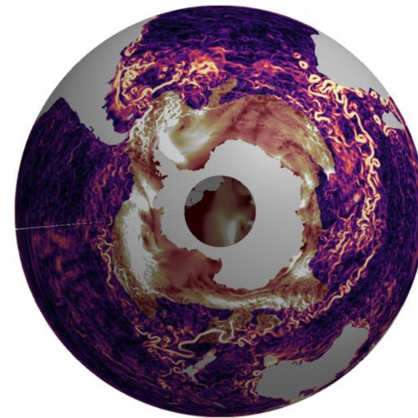
Coupled ocean-sea ice simulations

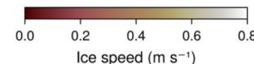
Ready-to-use climate datasets

Easily configurable experiments









From idealized turbulence → Earth system simulations

Go20 Conference 2026

Abstractions for Mathematics

Francis Poulin

```
 1 u, v, w = model.velocities
 2
 3 # "Lazy" expression trees and reductions representing computations:
 4 ζ = ∂x(v) - ∂y(u)
 5 s = √(u^2 + v^2)
 6 Z = Integral(ζ^2, dims=1)
 7
 8 # Building and computing a Field that instantiates an AbstractOperation:
 9 ζ_field = Field(∂x(v) - ∂y(u))
10 compute!(ζ_field)
11
12 # KernelFunctionOperations: user-defined stencil computations
13 @inline φ²(i, j, k, grid, φ, args...) = φ(i, j, k, grid, args...)^2
14
15 @inline function Ri(i, j, k, grid, velocities, buoyancy, tracers)
16     ∂z_u² = ℑxᶜᵃᵃ(i, j, k, grid, φ², ∂zᶠᶜᶠ, velocities.u)
17     ∂z_v² = ℑyᵃᶜᵃ(i, j, k, grid, φ², ∂zᶜᶠᶠ, velocities.v)
18     S² = ∂z_u² + ∂z_v²
19     N² = ∂z_b(i, j, k, grid, buoyancy, tracers)
20     Ri = N² / S²
21     return ifelse(S² == 0, zero(grid), Ri)
22 end
23
24 Ri = KernelFunctionOperation{Center, Center, Face}(Ri, grid,
25          model.velocities, model.buoyancy, model.tracers)
26
27 # Save diagnostics periodically after computing online on GPU
28 output_writer = JLD2Writer(model, (; ζ, Ri), schedule=TimeInterval(1hour))
```

Code looks like regular math

$\omega = \partial x(v) - \partial y(u)$

Easy to compute diagnostics

KernelFunctionOperation is needed for complex equations

Code looks like math and calculates any function

Nonhydrostatic Equations

Mass Conservation

$$\nabla \cdot \mathbf{u} = 0$$

Momentum Conservation (Newton's Second Law)

$$\partial_t \mathbf{u} = -\nabla p - \underbrace{(\mathbf{u} \cdot \nabla)\,\mathbf{u}}_{\text{advection}} - \underbrace{\mathbf{f} \times \mathbf{u}}_{\text{Coriolis}} - \underbrace{b\,\hat{\mathbf{g}}}_{\text{buoyancy}} - \underbrace{\nabla \cdot \tau}_{\text{closure}} + \underbrace{\mathbf{F}_u}_{\text{forcing}}$$

Tracer Conservation

$$\partial_t c = -\underbrace{(\mathbf{u} \cdot \nabla)\,c}_{\text{advection}} - \underbrace{\nabla \cdot \mathbf{J}_c}_{\text{closure}} + \underbrace{S_c}_{\text{biogeochemistry}} + \underbrace{F_c}_{\text{forcing}}$$

Hydrostatic model replaced the vertical equation with hydrostatic balance

u → velocity
p → pressure
b → buoyancy (like density)
c → tracers (T, S, etc.)

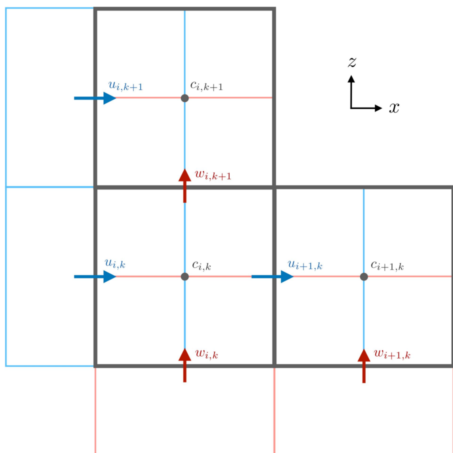
Finite-volume discretization on a staggered C grid

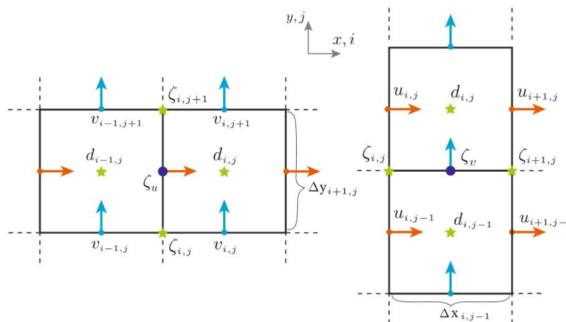
Finite Volume Framework

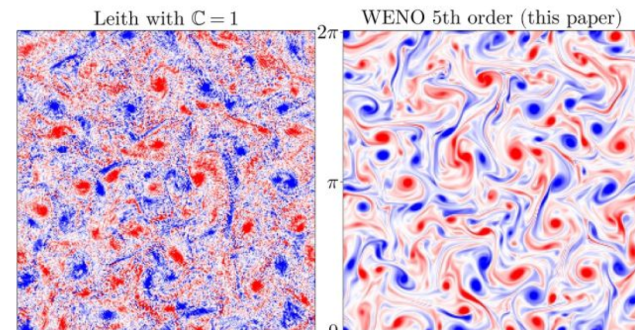
Staggered C grid (efficient)

Finite volume formulation (Conservative)

Minimal dissipation (High order methods)

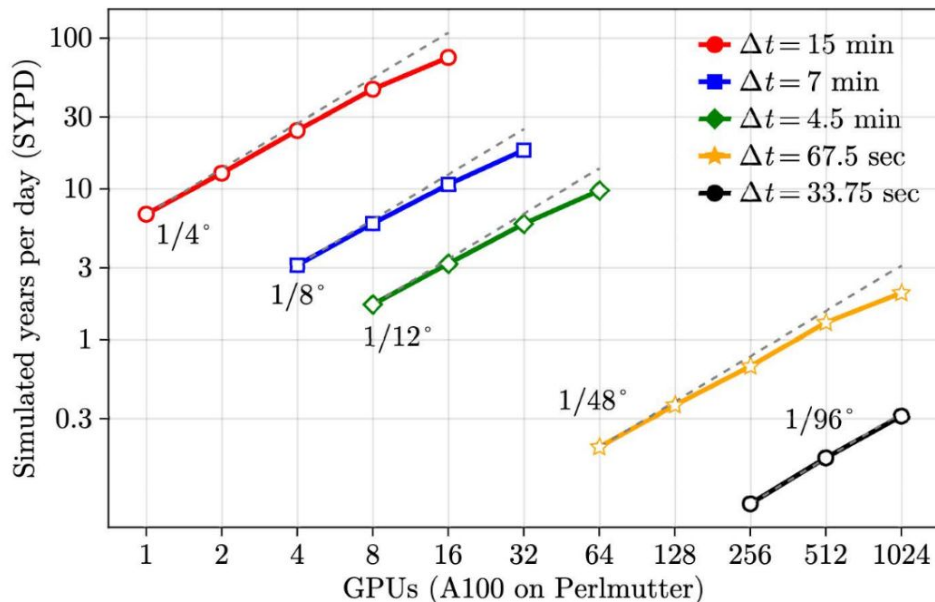






Simple numerics + high resolution = accuracy

Performance (GPU payoff)



1/4th degree: **> 25 SYPD** on **4 GPUs**

1/12th degree: **10 SYPD** on **64 GPUs**

1/48th degree: **> 1 SYPD** on **512 GPUs**

- GPU-native implementation
- Single-GPU global simulations
- High effective resolution

GPUs allow for high resolution simulations

Design philosophy

Single framework

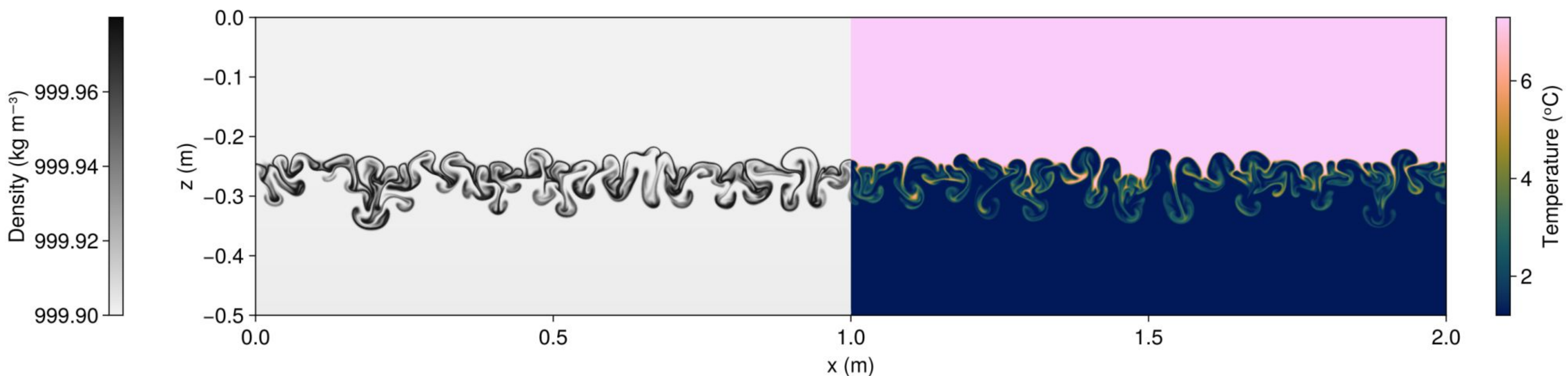
- From simple models to global ocean
- One model for all scales
- Many tools available

GPU-first

- Modern hardware
- Serial or parallel
- New methods must scale well on GPUs

Programmable

- Single script-based workflows
- User code = model code
- Rapid prototyping

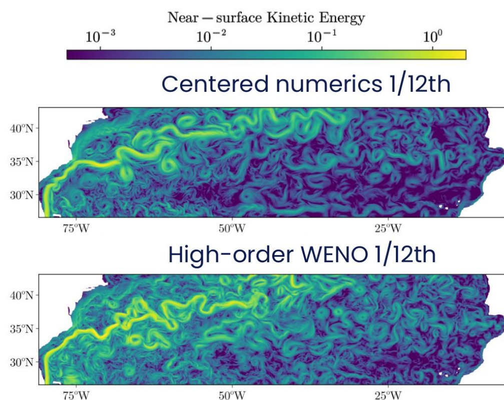


Design is based both on performance and usability

Why this matters for science

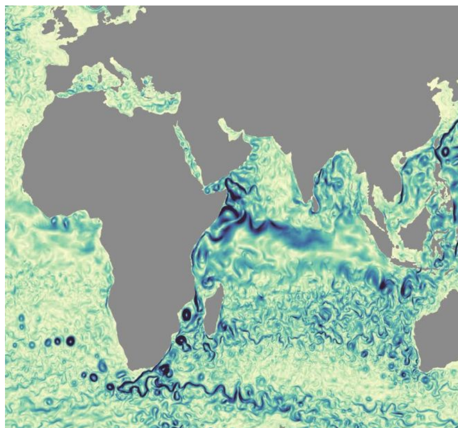
Faster experimentation

- Prototype and modify models quickly
- Explore ideas without writing new code



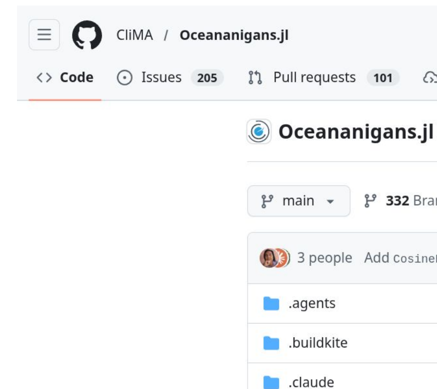
Higher resolution

- Leverage GPU and high order methods
- Resolve smaller scales, parameterize less



Reproducibility

- Script based works flows and easily shared
- Sharing with community is easy on Github



Scientific discoveries will happen faster

Takeaways

- Scientific computing is evolving
- Programmable frameworks enable flexible, high performance modelling
- Oceananigans spans small to global sales

manuscript submitted to *JAMES*

High-level, high-resolution ocean modeling at all scales with Oceananigans

Gregory L. Wagner[1,2], Simone Silvestri[1,3], Navid C. Constantinou[4,5], Ali Ramadhan[6], Jean-Michel Campin[1], Chris Hill[1], Tomás Chor[6,7], Jago Strong-Wright[8], Xin Kai Lee[1], Francis Poulin[9], Andre Souza[1], Keaton J. Burns[1,10], Siddhartha Bishnu[1,8], John Marshall[1], and Raffaele Ferrari[1]