How composable software tools help developing multi-physics codes in julia

Boris Kaus[1],
Nicolas Berlie[1], Valentin Churavy[2], Matias Cosarinsky[3],
Thibault Duretz[4], Daniel Kiss[1], Jeremy Kozdon[5], Albert de Montserrat[6],
Lucas Moser[1], Nils Medinger[1], Samuel Omlin[7], Ludovic Raess[6],
Patrick Sanan[6], Arne Spang[1], Marcel Thielmann[8], Ivan Utkin[6]

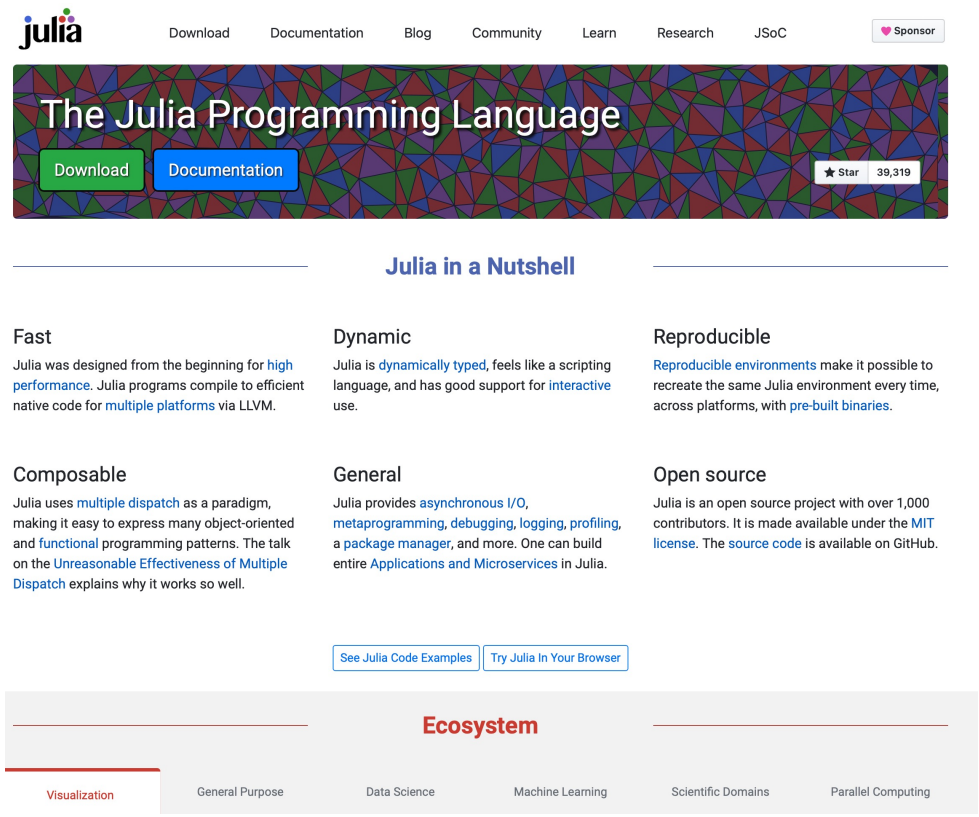
[1]Johannes-Gutenberg University, Mainz, [2]MIT JuliaLab, [3]University of Buenos Aires,
[4]University of Frankfurt, [5]Naval Postgraduate School, Monterey, USA
[6]ETH Zürich, Switzerland, [7]CSCS (Swiss Supercomputing Center), [8]University of Bayreuth

EGU 2022, GD 9.1

JOHANNES GUTENBERG UNIVERSITÄT MAINZ
MAGMA
Consolidator Grant Project
European Research Council
Established by the European Commission

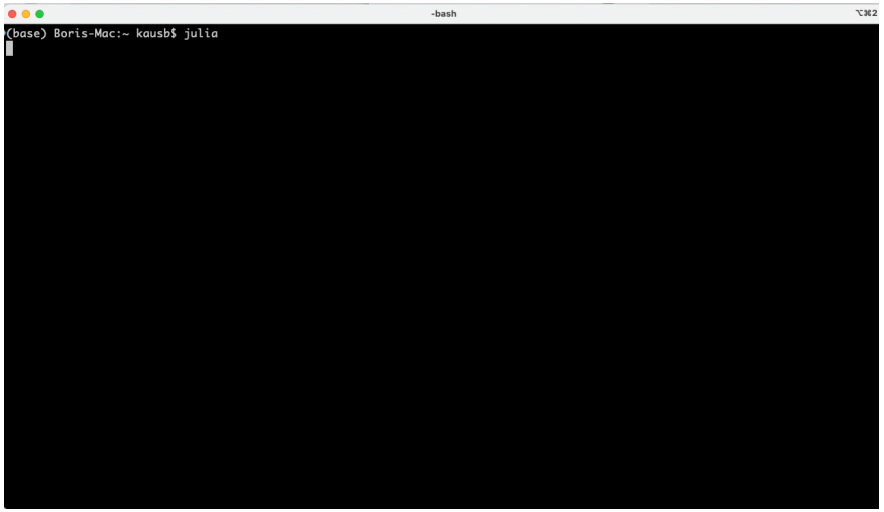
GPU4GEO
Platform for Advanced Scientific Computing

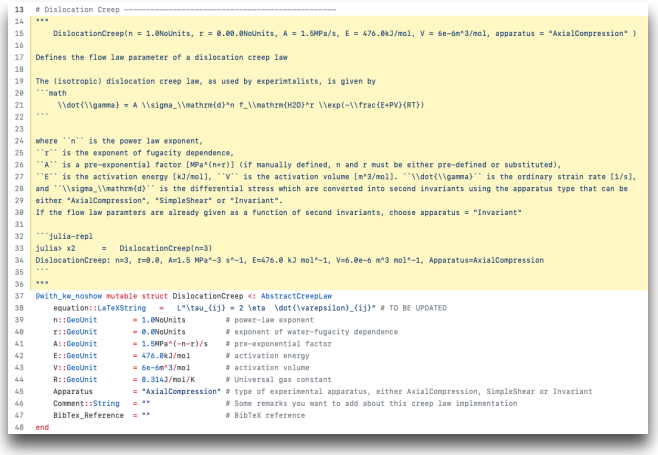
- Julia:
  - Modern, high-level, compiled language
  - As easy to program as MATLAB but (almost) as fast as compiled C/Fortran codes
- Easy to develop, document, distribute packages
- Lots of packages already exist
  - machine learning, plotting, optimisation
- Runs on large (GPU) HPC systems
- Packages (mostly) work together seamlessly



Cool features

- Build-in package manager
- Build-in testing framework
- Automatically install dependencies (e.g., PETSc)
- Easy to share packages
- Automatically generate (online) documentation





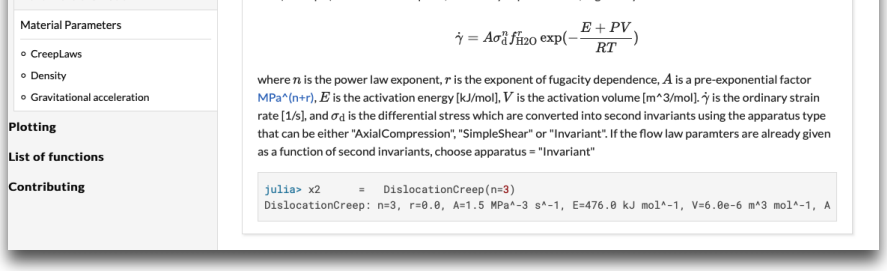
```
# Dislocation Creep ------------------------------------------------
"""
    DislocationCreep(n = 1.0NoUnits, r = 0.00.0NoUnits, A = 1.5MPa/s, E = 476.0kJ/mol, V = 6e-6m^3/mol, apparatus = "AxialCompression" )

Defines the flow law parameter of a dislocation creep law

The (isotropic) dislocation creep law, as used by experimtalists, is given by
```math
    \\dot{\\gamma} = A \\sigma_\\mathrm{d}^n f_\\mathrm{H2O}^r \\exp(-\\frac{E+PV}{RT})
```

where ``n`` is the power law exponent,
``r`` is the exponent of fugacity dependence,
``A`` is a pre-exponential factor [MPa^(n+r)] (if manually defined, n and r must be either pre-defined or substituted),
``E`` is the activation energy [kJ/mol], ``V`` is the activation volume [m^3/mol]. ``\\dot{\\gamma}`` is the ordinary strain rate [1/s],
and ``\\sigma_\\mathrm{d}`` is the differential stress which are converted into second invariants using the apparatus type that can be
either "AxialCompression", "SimpleShear" or "Invariant".
If the flow law paramters are already given as a function of second invariants, choose apparatus = "Invariant"

```julia-repl
julia> x2      =   DislocationCreep(n=3)
DislocationCreep: n=3, r=0.0, A=1.5 MPa^-3 s^-1, E=476.0 kJ mol^-1, V=6.0e-6 m^3 mol^-1, Apparatus=AxialCompression
```
"""
@with_kw_noshow mutable struct DislocationCreep <: AbstractCreepLaw
    equation::LaTeXString   =   L"\tau_{ij} = 2 \eta \dot{\varepsilon}_{ij}" # TO BE UPDATED
    n::GeoUnit   = 1.0NoUnits   # power-law exponent
    r::GeoUnit   = 0.0NoUnits   # exponent of water-fugacity dependence
    A::GeoUnit   = 1.5MPa^(-n-r)/s   # pre-exponential factor
    E::GeoUnit   = 476.0kJ/mol   # activation energy
    V::GeoUnit   = 6e-6m^3/mol   # activation volume
    R::GeoUnit   = 8.314J/mol/K   # Universal gas constant
    Apparatus   = "AxialCompression" # type of experimental apparatus, either AxialCompression, SimpleShear or Invariant
    Comment::String   = ""   # Some remarks you want to add about this creep law implementation
    BibTex_Reference = ""   # BibTeX reference
end
```



Ok cool – how do I start?

- Quick starting tutorial made for geoscientists used to MATLAB:

http://www.staff.uni-bayreuth.de/~bt303651/courses/gmg_tutorial/

Composable software

$$\dot{\varepsilon} = A\tau^n \exp\left(-\frac{E}{RT}\right)$$

```
julia> ε(τ, T, n, E, R, A) = A * τ^n * exp(- E/(R*T))
ε (generic function with 1 method)
```

```
julia> using GeoParams

julia> n=3.5; A = 2.5e-17Pa^(-n)/s; E=532kJ/mol; R=8.3145J/mol/K;

julia> ε(10e6Pa, 1273K, n, E, R, A)
1.1723642823160781e-14 s⁻¹·⁰
```

Material parameters are uncertain

```
julia> using Measurements

julia> E=(532 ± 5)kJ/mol
532.0 ± 5.0 kJ mol⁻¹·⁰

julia> ε(10e6Pa, 1273K, n, E, R, A)
1.17e-14 ± 5.5e-15 s⁻¹·⁰
```

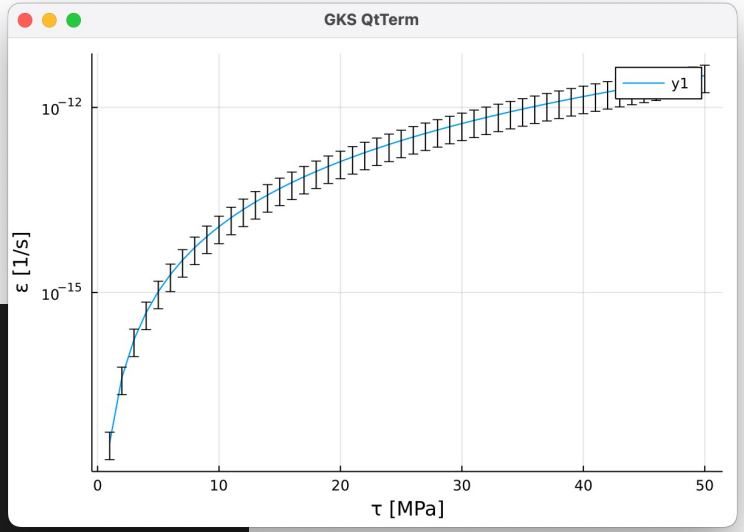
Plot

```
julia> τ=(1:50)*1e6Pa;

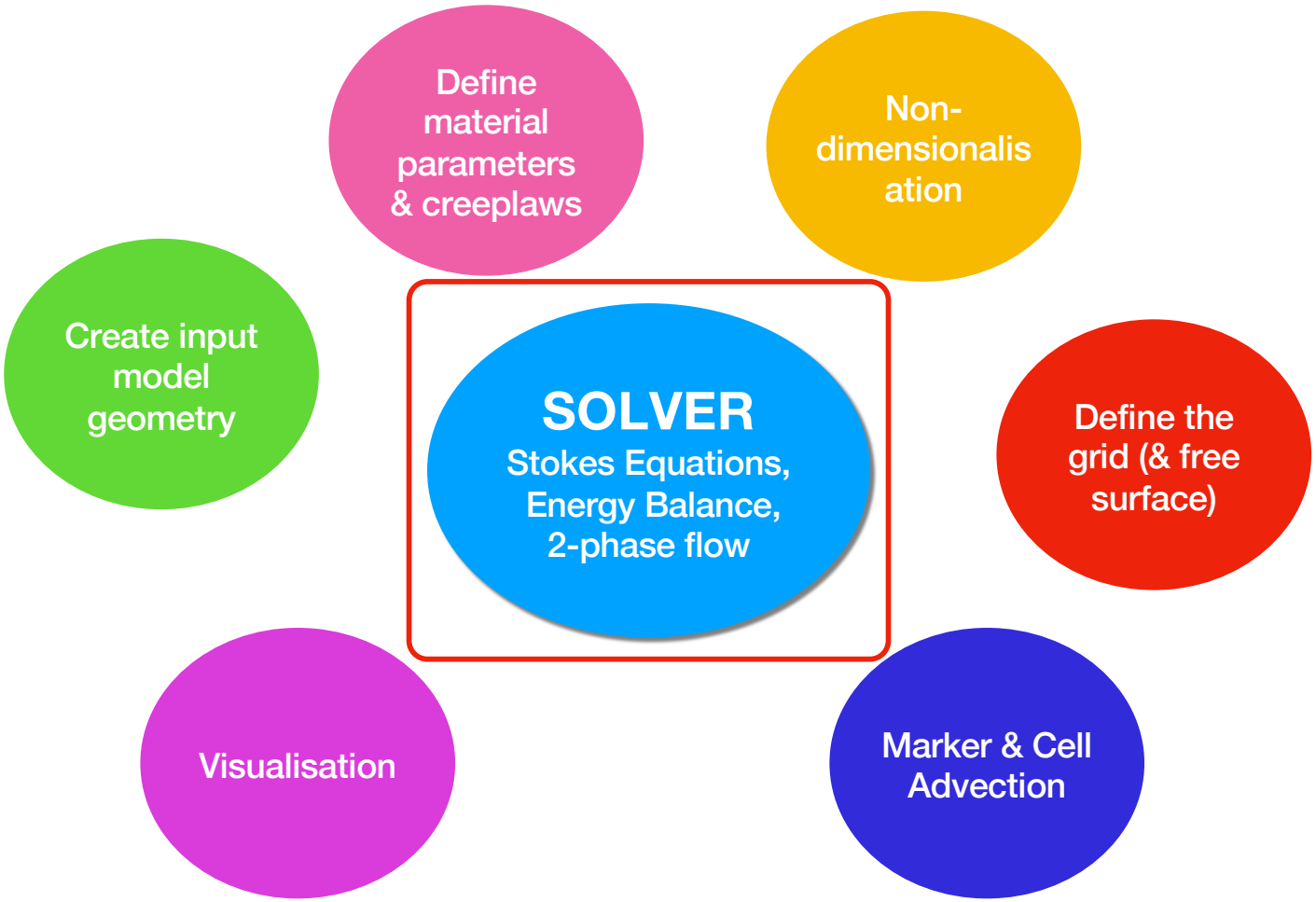
julia> ε_vec= ε.(τ, 1273K, n, E, R, A);

julia> using Plots;

julia> plot(ustrip.(τ)/1e6,ustrip.(ε_vec),
            yaxis=:log, xlabel="τ [MPa]", ylabel="ε [1/s]")
```



Building blocks in geodynamic software



Solvers

Two main approaches

Pseudotransient codes GPU's

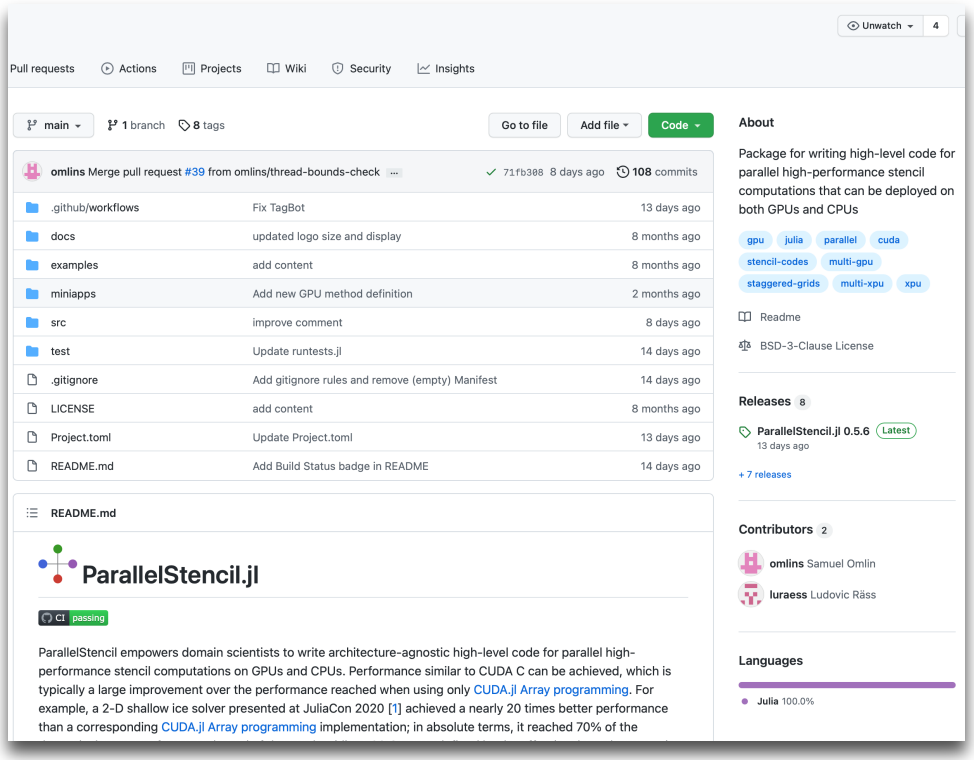
ParallelStencil.jl
ImplicitGlobalGrid.jl

Implicit codes CPU's

PETSc.jl

ParallelStencil.jl

- Allows evaluating (explicit) stencils very fast
- Works on CPUs & GPUs
- Optimized for GPU's (93% parallel efficiency on 5120 GPU's)
- Very simple, intuitive, codes
- Requires equations that have a time-derivative



3D diffusion example on 1 CPU/GPU

```
using ParallelStencil
using ParallelStencil.FiniteDifferences3D
@static if USE_GPU
    @init_parallel_stencil(CUDA, Float64, 3);
else
    @init_parallel_stencil(Threads, Float64, 3);
end

@parallel function diffusion3D_step!(T2, T, Ci, lam, dt, dx, dy, dz)
    @inn(T2) = @inn(T) + dt*(lam*@inn(Ci)*(@d2_xi(T)/dx^2 + @d2_yi(T)/dy^2 + @d2_zi(T)/dz^2));
    return
end

function diffusion3D()
# Physics
lam        = 1.0;                                       # Thermal conductivity
cp_min     = 1.0;                                       # Minimal heat capacity
lx, ly, lz = 10.0, 10.0, 10.0;                          # Length of domain in dimensions x, y and

# Numerics
nx, ny, nz = 256, 256, 256;                             # Number of gridpoints dimensions x, y an
nt         = 100;                                       # Number of time steps
dx         = lx/(nx-1);                                 # Space step in x-dimension
dy         = ly/(ny-1);                                 # Space step in y-dimension
dz         = lz/(nz-1);                                 # Space step in z-dimension

# Array initializations
T   = @zeros(nx, ny, nz);
T2  = @zeros(nx, ny, nz);
Ci  = @zeros(nx, ny, nz);

# Initial conditions (heat capacity and temperature with two Gaussian anomalies each)
Ci .= 1.0./( cp_min .+ Data.Array([5*exp(-(((ix-1)*dx-lx/1.5))^2-(((iy-1)*dy-ly/2))^2-(((iz-1)*dz-
                                   5*exp(-(((ix-1)*dx-lx/3.0))^2-(((iy-1)*dy-ly/2))^2-(((iz-1)*dz-
T  .= Data.Array([100*exp(-(((ix-1)*dx-lx/2)/2)^2-(((iy-1)*dy-ly/2)/2)^2-(((iz-1)*dz-lz/3.0)/2)^2)
                  50*exp(-(((ix-1)*dx-lx/2)/2)^2-(((iy-1)*dy-ly/2)/2)^2-(((iz-1)*dz-lz/1.5)/2)^2)
T2 .= T;                                                # Assign also T2 to get correct boundary

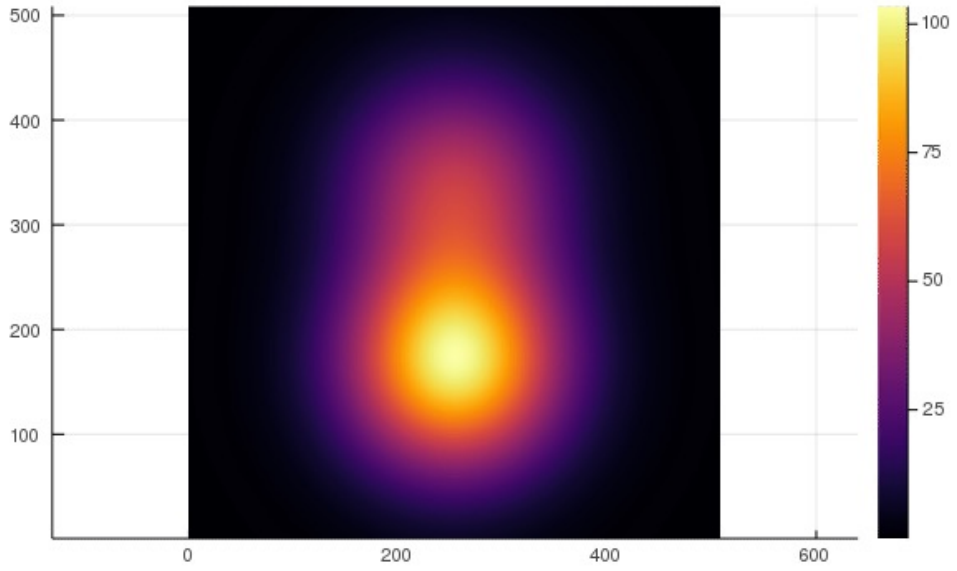
# Time loop
dt = min(dx^2,dy^2,dz^2)*cp_min/lam/8.1;                # Time step for the 3D Heat diffusion
for it = 1:nt
    @parallel diffusion3D_step!(T2, T, Ci, lam, dt, dx, dy, dz);
    T, T2 = T2, T;
end

end

diffusion3D()
```

$$Cp\frac{\partial T}{\partial t} = \lambda\left(\frac{\partial T}{\partial x^2} + \frac{\partial T}{\partial y^2} + \frac{\partial T}{\partial z^2}\right)$$

$$T^{new} = T + dt\,\frac{\lambda}{Cp}\left(\frac{\partial T}{\partial x^2} + \frac{\partial T}{\partial y^2} + \frac{\partial T}{\partial z^2}\right)$$



Making it work in parallel

use ImplicitGlobalGrid.jl & add a few lines

```
#(...)
using ImplicitGlobalGrid
#(...)
function diffusion3D()
# Physics
#(...)
# Numerics
#(...)
init_global_grid(nx, ny, nz);
dx  = lx/(nx_g()-1);                                    # Space step in x-dimension
dy  = ly/(ny_g()-1);                                    # Space step in y-dimension
dz  = lz/(nz_g()-1);                                    # Space step in z-dimension

# Array initializations
#(...)

# Initial conditions (heat capacity and temperature with two Gaussian anomalies each)
Ci .= 1.0./( cp_min .+ Data.Array([5*exp(-((x_g(ix,dx,Ci)-lx/1.5))^2-((y_g(iy,dy,Ci)-ly/2))^2-((z
                                  5*exp(-((x_g(ix,dx,Ci)-lx/3.0))^2-((y_g(iy,dy,Ci)-ly/2))^2-((z
T  .= Data.Array([100*exp(-((x_g(ix,dx,T)-lx/2)/2)^2-((y_g(iy,dy,T)-ly/2)/2)^2-((z_g(iz,dz,T)-lz/
                 50*exp(-((x_g(ix,dx,T)-lx/2)/2)^2-((y_g(iy,dy,T)-ly/2)/2)^2-((z_g(iz,dz,T)-lz/

# Time loop
#(...)
for it = 1:nt
    #(...)
    update_halo!(T2);
    #(...)
end

finalize_global_grid();
end

diffusion3D()
```

Stokes example

Stokes equations:

$$\frac{\partial v_i}{\partial x_i} = 0$$

$$\frac{\partial P}{\partial x_i} + \frac{\partial \tau_{ij}}{\partial x_j} + \rho g_i = 0$$

Pseudo-transient form:

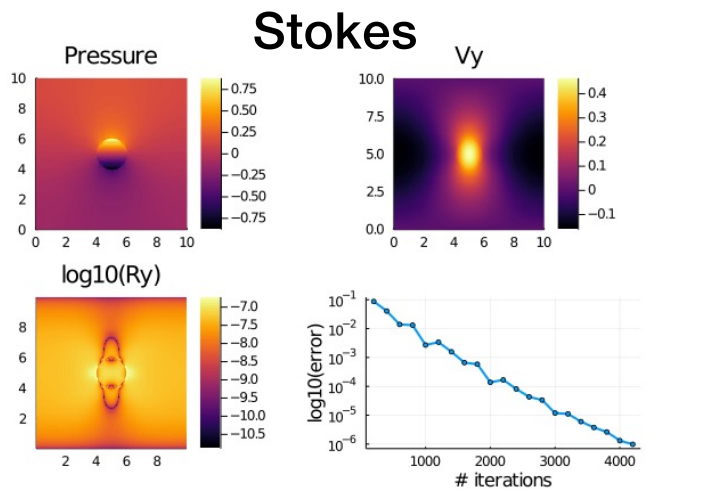
$$\frac{\partial v_i}{\partial x_i} = -\frac{\partial P}{\partial \tau_p}$$

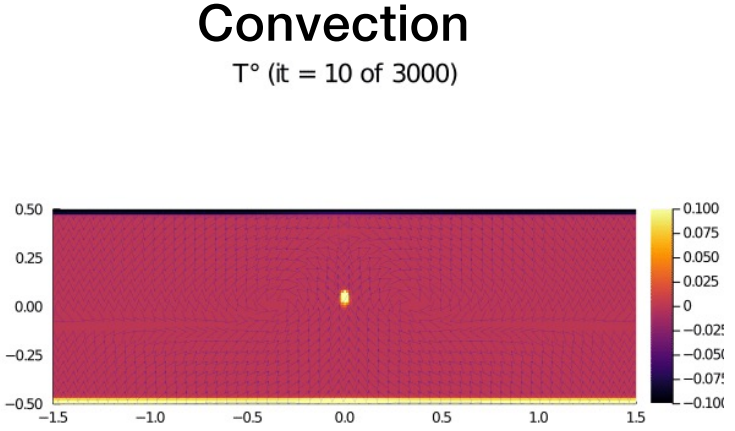
$$\frac{\partial P}{\partial x_i} + \frac{\partial \tau_{ij}}{\partial x_j} + \rho g_i = \frac{\partial v_i}{\partial \tau_{v_i}}$$

Add time-derivative

```
    @parallel compute_timesteps!(dτVx, dτVy, dτPt, Mus, Vsc, Ptsc, min_dxy2, max_nxy)
    err=2*ε; iter=1; niter=0; err_evo1=[]; err_evo2=[]
    while err > ε && iter <= iterMax
        if (iter==11)  global wtime0 = Base.time()  end
        @parallel compute_P!(∇V, Pt, Vx, Vy, dτPt, dx, dy)
        @parallel compute_τ!(∇V, τxx, τyy, τxy, Vx, Vy, Mus, dx, dy)
        @parallel compute_dV!(Rx, Ry, dVxdτ, dVydτ, Pt, Rog, τxx, τyy, τxy, dampX, dampY, dx, dy)
        @parallel compute_V!(Vx, Vy, dVxdτ, dVydτ, dτVx, dτVy)
        @parallel (1:size(Vx,1), 1:size(Vx,2)) bc_y!(Vx)
        @parallel (1:size(Vy,1), 1:size(Vy,2)) bc_x!(Vy)
        if mod(iter,nout)==0
            global mean_Rx, mean_Ry, mean_∇V
            mean_Rx = mean(abs.(Rx)); mean_Ry = mean(abs.(Ry)); mean_∇V = mean(abs.(∇V))
            err = maximum([mean_Rx, mean_Ry, mean_∇V])
            push!(err_evo1, maximum([mean_Rx, mean_Ry, mean_∇V])); push!(err_evo2,iter)
            @printf("Total steps = %d, err = %1.3e [mean_Rx=%1.3e, mean_Ry=%1.3e, mean_∇V=%1.3e] \n", iter, err, mean_Rx, mean_Ry, mean_∇V)
        end
        iter+=1; niter+=1
    end
    # Performance
    wtime    = Base.time() - wtime0
    A_eff    = (3*2)/1e9*nx*ny*sizeof(Data.Number)  # Effective main memory access per iteration [GB] (Lower bound of required memory access: Te has to be r
    wtime_it = wtime/(niter-10)                     # Execution time per iteration [s]
    T_eff    = A_eff/wtime_it                       # Effective memory throughput [GB/s]
    @printf("Total steps = %d, err = %1.3e, time = %1.3e sec (@ T_eff = %1.2f GB/s) \n", niter, err, wtime, round(T_eff, sigdigits=2))
    # Visualisation
    p1 = heatmap(X,  Y, Array(Pt)', aspect_ratio=1, xlims=(X[1],X[end]), ylims=(Y[1],Y[end]), c=:inferno, title="Pressure")
    p2 = heatmap(X, Yv, Array(Vy)', aspect_ratio=1, xlims=(X[1],X[end]), ylims=(Yv[1],Yv[end]), c=:inferno, title="Vy")
    p4 = heatmap(X[2:end-1], Yv[2:end-1], log10.(abs.(Array(Ry)')), aspect_ratio=1, xlims=(X[2],X[end-1]), ylims=(Yv[2],Yv[end-1]), c=:inferno, title="log10
    p5 = plot(err_evo2,err_evo1, legend=false, xlabel="# iterations", ylabel="log10(error)", linewidth=2, markershape=:circle, markersize=3, labels="max(err
    # display(plot(p1, p2, p4, p5))
    plot(p1, p2, p4, p5); frame(anim)
    gif(anim, "Stokes2D.gif", fps = 15)
    return
end

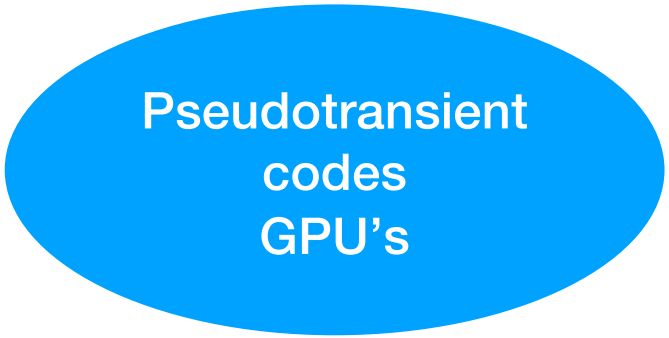
Stokes2D()
```



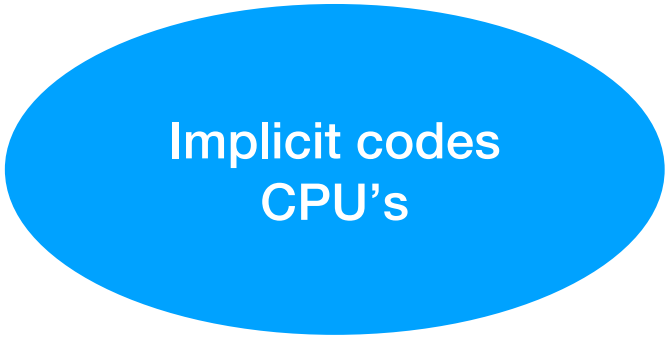


Solvers

Two main approaches



ParallelStencil.jl
ImplicitGlobalGrid.jl

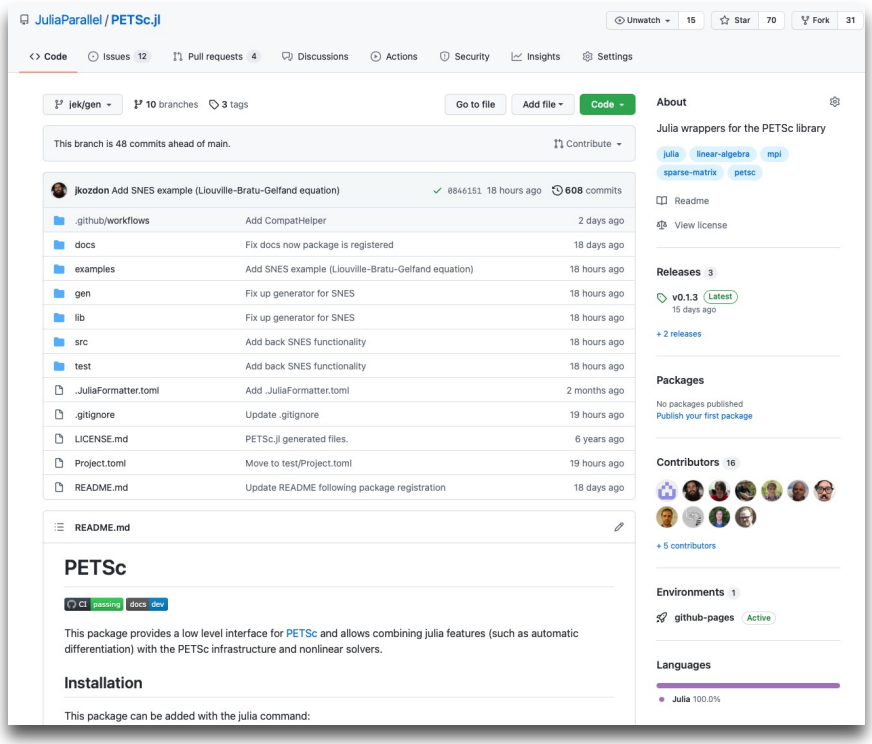


PETSc.jl

Implicit codes: PETSc.jl

- Julia interface to the **PETSc** package
- Automatic installation of parallel PETSc
  - On windows, mac & linux
- Wrappers to the **full** PETSc package (huge)
- Work in progress to make it Julia-friendly:
  - Adding tests, examples for regular & staggered grids and nonlinear solvers

```
(@v1.6) pkg> add PETSc#jek/gen
```



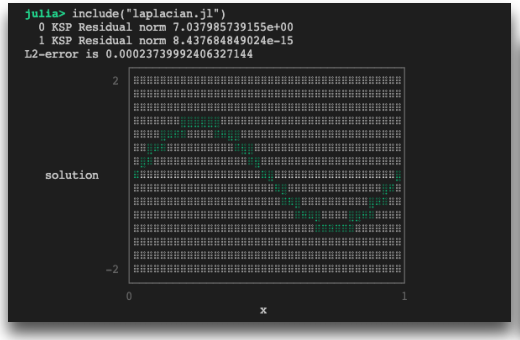
Simple 1D example

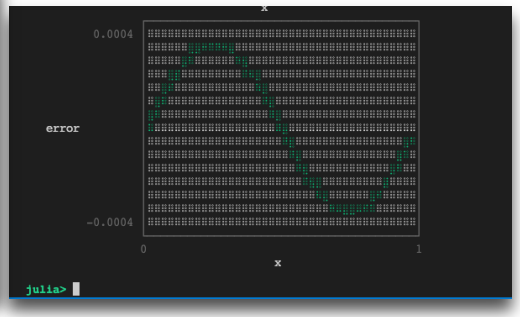
laplacian.jl

```julia
# # Finite Difference Laplacian
#
# In this example a simple finite difference, 1-D laplacian with Dirichlet
# boundary conditions is solved first constructing the operator as Julia sparse
# matrix then solving using PETsc

using PETSc
using SparseArrays: spdiagm
using UnicodePlots: lineplot, lineplot!

# Initialize PETSc
PETSc.initialize()

# Set our PETSc Scalar Type
PetscScalar = Float64

# Set the total number of grid points
Nq = 100

# number of interior points
n = Nq - 2

# create the interior grid and get the grid spacing
x = range(PetscScalar(0), length = Nq, stop = 1)[2:(end - 1)]
Δx = PetscScalar(x.step)

# Create the finite difference operator with zero Dirichlet boundary conditions
A =
    spdiagm(
        -1 => -ones(PetscScalar, n - 1),
        0 => 2ones(PetscScalar, n),
        1 => -ones(PetscScalar, n - 1),
    ) / Δx^2

```

```julia
# Setup the exact solution and the forcing function
κ = 2PetscScalar(π)
u(x) = sin(κ * x)
f(x) = κ^2 * sin(κ * x)

# Set up the PETSc solver
ksp = PETSc.KSP(A; ksp_monitor = true);

# Solve the problem
v = ksp \ f.(x);

# Compute the L2-error
ϵ = sqrt(sum((v - u.(x)) .^ 2 * Δx))
println("L2-error is $ϵ")

# Plot the solution and error
display(lineplot(x, v, xlabel = "x", ylabel = "solution"))
display(lineplot(x, v - u.(x), xlabel = "x", ylabel = "error"))
```

Run:





2D parallel, multigrid, example

dmda_laplacian.jl

Options (multigrid):

```
27  opts = if !isinteractive()
28      PETSc.parse_options(ARGS)
29  else
30      (ksp_monitor = true, ksp_view = true, pc_type = "mg", pc_mg_levels = 2)
31  end
```

Setup regular grid with 1 dof/node:

```
45  # Set the total number of grid points in each direction
46  Nq = (11, 13)
47
48  # Set up the problem by using an exact solution and then using this to set the
49  # boundary data and forcing
50  exact(x, y) = cos(π * x) * sin(π * y)
51  forcing(x, y) =
52      derivative(x -> exact(x, y), x) * derivative(x -> exact(x, y), x) +
53      derivative(x -> exact(x, y), y) * derivative(x -> exact(x, y), y)
54
55  # Create the PETSC dmda object
56  da = PETSc.DMDA(
57      petsclib,
58      comm,
59      (PETSc.DM_BOUNDARY_NONE, PETSc.DM_BOUNDARY_NONE), # Use no ghost nodes
60      Nq,                                               # Global grid size
61      1,                                                # Number of DOF per node
62      1,                                                # Stencil width
63      PETSc.DMDA_STENCIL_STAR;                          # Stencil type
64      opts...,
65  )
```

Create stiffness matrix **A**:

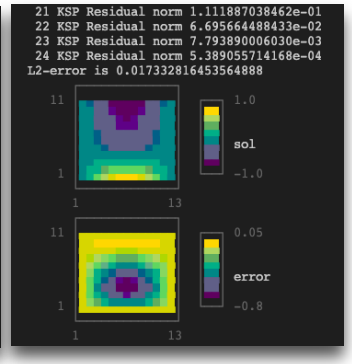
```
122 #
123 # Set the left-hand side operators of the ksp
124 #
125 PETSc.setcomputeoperators!(ksp) do A, _, ksp
126     # Get the distributed array from the ksp.
127     da = PETSc.getDMDA(ksp)
128
129     # Get the corners of the box that this processor is responsible for
130     corners = PETSc.getcorners(da)
131
132     # get the global grid dimension
133     Nq = PETSc.getinfo(da).global_size[1:2]
134
135     # Get the grid spacing
136     Δx = PetscScalar(range(PetscScalar(0), length = Nq[1] + 2, stop = 1).step)
137     Δy = PetscScalar(range(PetscScalar(0), length = Nq[2] + 2, stop = 1).step)
138
139     # Interior Points
140     interior =
141         (CartesianIndex(2, 2, 1)):(CartesianIndex(Nq[1] - 1, Nq[2] - 1, 1))
142
143     # Computational stencil for the interior points
144     sten = (
145         CartesianIndex(-1, 0, 0),
146         CartesianIndex(1, 0, 0),
147         CartesianIndex(0, -1, 0),
148         CartesianIndex(0, 1, 0),
149         CartesianIndex(0, 0, 0),
150     )
151     vals = (1 / Δx^2, 1 / Δx^2, 1 / Δy^2, 1 / Δy^2, -2 / Δx^2 - 2 / Δy^2)
152
153     # Loop over all the points on the processory
154     for i in ((corners.lower):(corners.upper))
155         # for the interior points set the stencil otherwise just set to identity
156         if i ∈ interior
157             for (s, v) in zip(sten, vals)
158                 A[i, i + s] = v
159             end
160         else
161             A[i, i] = 1
162         end
163     end
164
165     # Assemble the matrix after inserting values
166     PETSc.assemble!(A)
167
168     # 0 is the PETSc success value
169     return 0
170 end
```

Solve:

```
172 #
173 # Solve the problem
174 #
175 PETSc.solve!(ksp)
```

Run in serial:

```
julia> include("dmda_laplacian.jl")
  0 KSP Residual norm 1.736255511358e+02
  1 KSP Residual norm 4.414497154237e+01
  2 KSP Residual norm 2.520389300836e+01
  3 KSP Residual norm 1.611549119992e+01
  4 KSP Residual norm 1.500098304141e+01
  5 KSP Residual norm 9.593741137468e+00
  6 KSP Residual norm 6.391590461969e+00
  7 KSP Residual norm 6.374391995519e+00
  8 KSP Residual norm 6.371697258309e+00
  9 KSP Residual norm 5.001872512229e+00
 10 KSP Residual norm 3.144695517123e+00
 11 KSP Residual norm 2.790923454586e+00
 12 KSP Residual norm 2.789657467422e+00
 13 KSP Residual norm 2.763641656848e+00
 14 KSP Residual norm 2.255220414309e+00
 15 KSP Residual norm 2.053970582263e+00
 16 KSP Residual norm 9.783512062737e-01
 17 KSP Residual norm 4.788546926136e-01
 18 KSP Residual norm 4.788427595860e-01
 19 KSP Residual norm 1.535867712153e-01
 20 KSP Residual norm 1.121598410917e-01
 21 KSP Residual norm 1.111887038462e-01
 22 KSP Residual norm 6.695664488433e-02
 23 KSP Residual norm 7.793890006030e-03
 24 KSP Residual norm 5.389055714168e-04
L2-error is 0.017332816453564888
```



Run in parallel:

```
(base) Boris-Mac:examples kausb$ mpiexec -n 4 julia --project dmda_laplacian.jl -ksp_monitor
  0 KSP Residual norm 3.781639864438e+00
  1 KSP Residual norm 8.737831247486e-01
  2 KSP Residual norm 4.027424761990e-01
  3 KSP Residual norm 2.372197584674e-01
  4 KSP Residual norm 1.606579735858e-01
  5 KSP Residual norm 1.057495674226e-01
  6 KSP Residual norm 5.329788503159e-02
  7 KSP Residual norm 2.046898867049e-02
  8 KSP Residual norm 7.472342057572e-03
  9 KSP Residual norm 2.981940100341e-03
 10 KSP Residual norm 1.121832242932e-03
 11 KSP Residual norm 3.478855469908e-04
 12 KSP Residual norm 1.500748194096e-04
 13 KSP Residual norm 6.058873266575e-05
 14 KSP Residual norm 1.341063210186e-05
L2-error is 0.03232681962721379
```

Nonlinear example & automatic differentiation

Porosity_Waves.jl (1D/2D/3D; currently in #bjpk/dmda_multiple_dof)

$$\frac{\partial \phi}{\partial t} = -De\frac{\partial P_e}{\partial t} - \phi^m P_e$$

$$De\frac{\partial P_e}{\partial t} = \nabla\left(\phi^n (P_e + e_x)\right) - \phi^m P_e$$

Viscoelastic porosity waves

Compute residual (1D/2D/3D):

```
223                 res_Phi[i] =
224                     (Phi[i] - Phi_old[i]) / dt +
225                     De * (Pe[i] - Pe_old[i]) / dt +
226                     (Phi[i]^m) * Pe[i]
227
228                 Phi_c_p1 = (Phi[i + ix_p1] + Phi[i]
229                 Phi_c_m1 = (Phi[i + ix_m1] + Phi[i]
230
231                 res_Pe[i] =
232                     De * (Pe[i] - Pe_old[i]) / dt -
233                     (
234                         (Phi_c_p1^n) * ((Pe[i + ix_p1] - Pe[i]) / Δx + 1) -
235                         (Phi_c_m1^n) * ((Pe[i] - Pe[i + ix_m1]) / Δx + 1)
236                     ) / Δx + (Phi[i]^m) * Pe[i]
```

Automatically generate Jacobian:

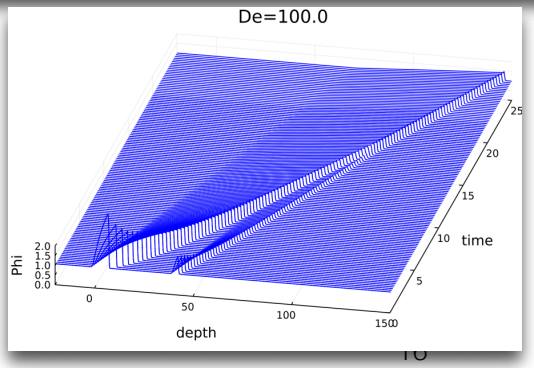
```
333         S = forwarddiff_color_jacobian(
334             ForwardDiff_res,
335             x_julia,
336             colorvec = colors,
337             sparsity = sparsity_pattern,
338             jac_prototype = jac,
339         )
```

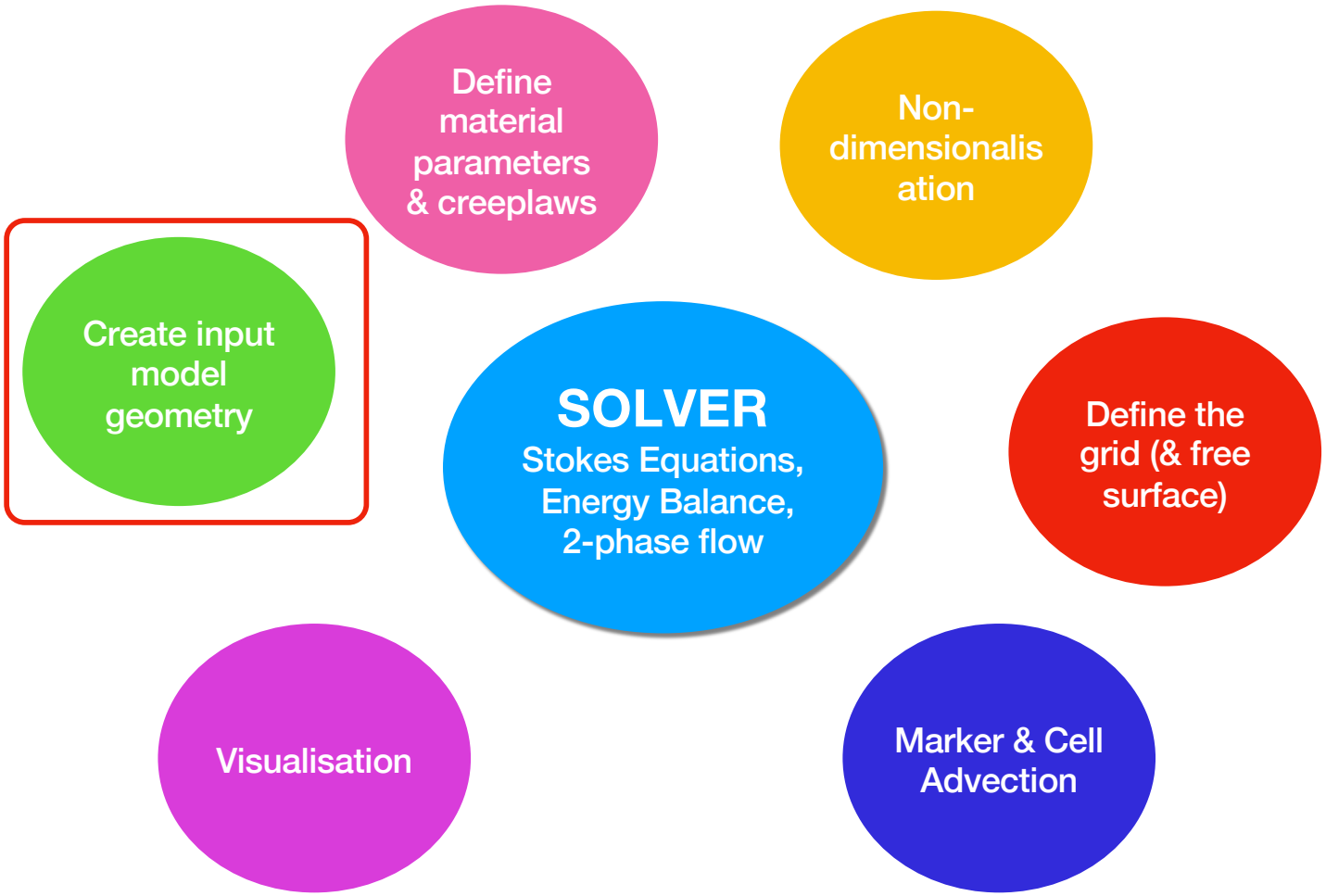
Solve:

```
358 # Timestep loop
359 time, it = PetscScalar(0), 1;
360 while (it < max_it && time < max_time)
361     global time, it, x_old, g_xold, g_x
362
363     # Solve nonlinear system of equations
364     PETSc.solve!(g_x, snes)
```

Run in parallel:

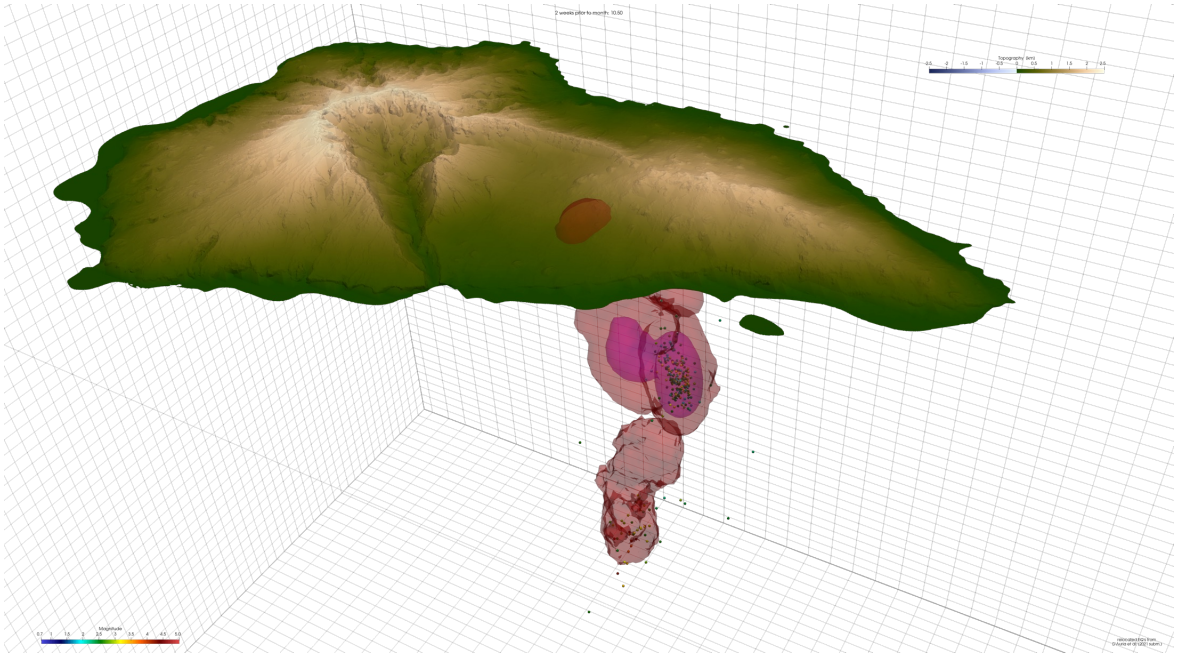
```
) Boris-Mac:examples kausb$ mpiexec -n 4 julia --project Porosity_Waves.jl -dim 1 -snes_monitor
NES Function norm 7.897081957802e+02
NES Function norm 1.277174473868e+02
NES Function norm 3.580768713408e+00
NES Function norm 1.165561906092e-03
NES Function norm 6.243276281152e-10
 6.243276281151642e-10
tep 2, time=0.01
NES Function norm 4.451447560129e+02
NES Function norm 5.391002477765e+01
NES Function norm 4.305783977553e-01
NES Function norm 3.939487645282e-05
NES Function norm 2.597884347782e-13
 2.597884347782e-13
tep 3, time=0.02
NES Function norm 3.268038812632e+02
NES Function norm 6.646007778133e+01
NES Function norm 4.868988699169e-01
NES Function norm 5.785868447956e-05
NES Function norm 2.208284440367e-13
 2.208284440367e-13
Timestep 4, time=0.03
```




Define material parameters & creeplaws
Non-dimensionalisation
Create input model geometry
SOLVER
Stokes Equations, Energy Balance, 2-phase flow
Define the grid (& free surface)
Visualisation
Marker & Cell Advection

GeophysicalModelGenerator.jl

- Import geophysical data
  - Seismic tomography
  - GPS velocity
  - topography
  - EQ locations
  - Screenshots from published papers.
- Create Paraview files from it
- Many tutorials
- Generate geodynamic input models



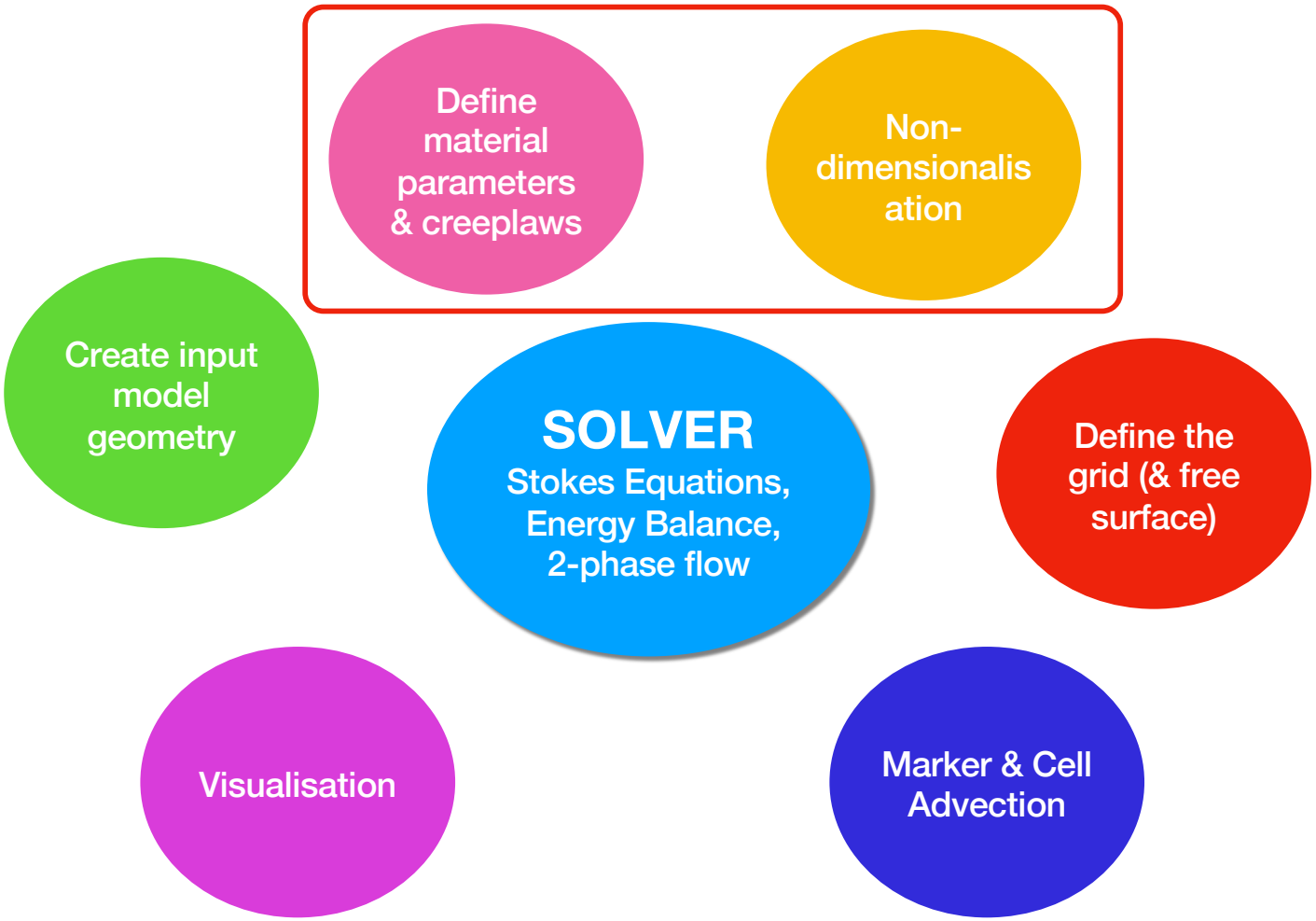
2021 LaPalma eruption
(seismicity isosurfaces & topography)

Example:

```
julia> using GMT
julia> G = gmtread("@earth_relief_01m.grd", limits=[4,20,37,49]);
```

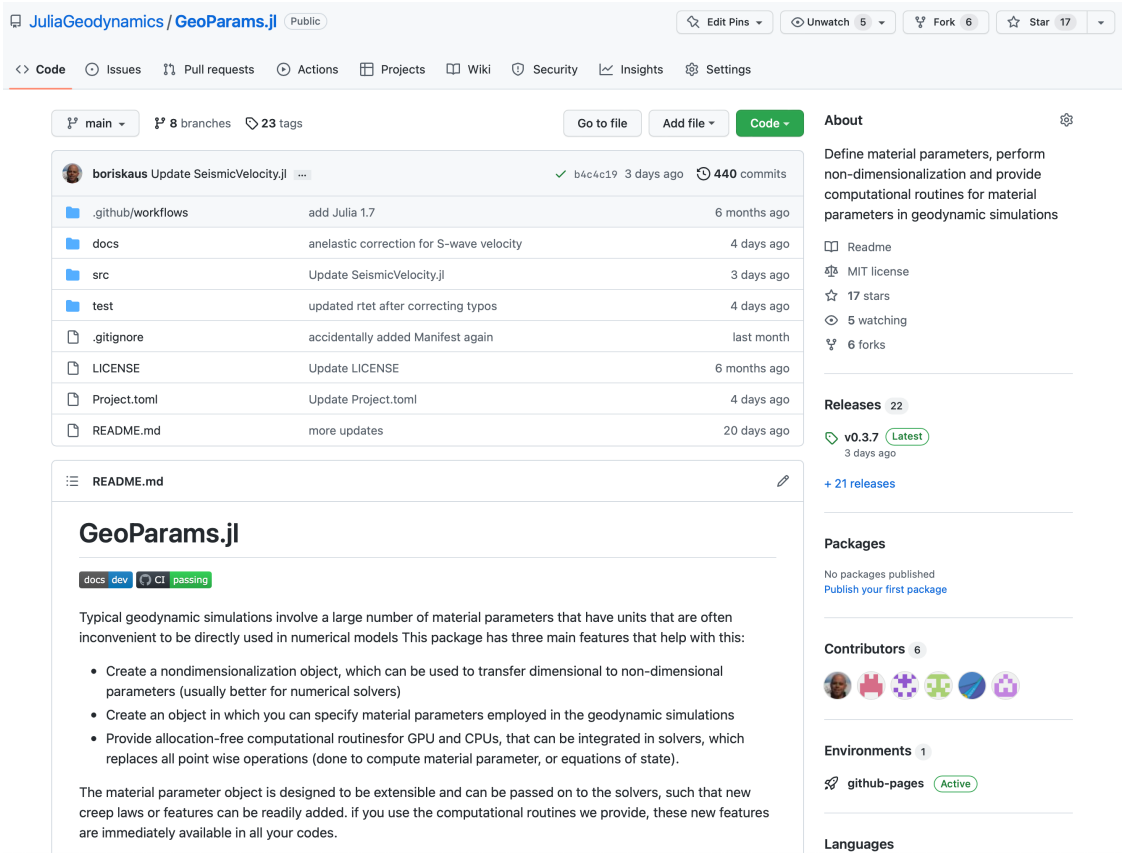
```
julia> Lon,Lat,Depth    =   LonLatDepthGrid(G.x[1:end-1],G.y[1:end-1],0);
julia> Depth[:,:,1]     =   1e-3*G.z';
julia> data_Topo        =   GeoData(Lon, Lat, Depth, (Topography=Depth*km,))
julia> Write_Paraview(data_Topo, "Topography_Alps")
```

See tutorial on how to create a input model for the 3D geodynamic code LaMEM from this


Define material parameters & creeplaws
Non-dimensionalisation
Create input model geometry
SOLVER
Stokes Equations, Energy Balance, 2-phase flow
Define the grid (& free surface)
Visualisation
Marker & Cell Advection

GeoParams.jl

- Define material parameters
  - Density
  - Elasticity
  - Creep laws
  - Plasticity
  - …
- Nondimensionalize (any) input data
- Provide a single computational interface to retrieve parameters
  - To be used within the solvers
  - Allocation-free
  - Works on GPU and CPU



Non-dimensionalisation

- Define characteristic units

```
julia> using GeoParams
julia> CharDim = GEO_units(length=1000km, temperature=1000C, stress=10MPa, viscosity=1e20Pas)
Employing GEO units
Characteristic values:
         length:      1000 km
         time:        0.3169 Myrs
         stress:      10 MPa
         temperature: 1000.0 °C
```

- Automatically non-dimensionalize (any) material parameter using this.

```
julia> A    =   6.3e-2MPa^-3.05*s^-1
0.063 MPa^-3.05 s^-1
julia> A_ND =   Nondimensionalize(A, CharDim);
```

- Convert into different units

```
julia> uconvert(Pa^-3.05*s^-1, A)
3.157479571851836e-20 Pa^-3.05 s^-1
```

Define material parameters

- Define properties for a single phase, using dimensional units:

```
julia> MatParam = SetMaterialParams(Name="Viscous Matrix", Phase=2,
                                    Density   = ConstantDensity(),
                                    CreepLaws = LinearViscous(η=1e23Pa*s))
Phase 2 : Viscous Matrix
        | [dimensional units]
        |
        |-- Density           : Constant density: ρ=2900 kg m^-3
        |-- Gravity           : Gravitational acceleration: g=9.81 m s^-2
        |-- CreepLaws         : Linear viscosity: η=1.0e23 Pa s
```

- Same but with non-dimensionalisation:

```
julia> CharDim = GEO_units(length=1000km, temperature=1000C, stress=10MPa, viscosity=1e20Pas
julia> MatParam = SetMaterialParams(Name="Viscous Matrix", Phase=2,
                                    Density   = ConstantDensity(),
                                    CreepLaws = LinearViscous(η=1e23Pa*s), CharDim=CharDim)
Phase 2 : Viscous Matrix
        | [non-dimensional units]
        |
        |-- Density           : Constant density: ρ=2.8999999999999996e-18
        |-- Gravity           : Gravitational acceleration: g=9.81e20
        |-- CreepLaws         : Linear viscosity: η=999.9999999999998
```

- Using this in your code:

```
args= (;T=Arrays.T_K, P=Arrays.P)
compute_density!(Rho, MatParam, Phases, args)
```

  - Constant density:

```
MatParam = (SetMaterialParams(Name="Crust", Phase=0,
                    Density   = ConstantDensity(ρ=2900kg/m^3)),)
```

  - Variable density (no need to change your solver!):

```
MatParam = (SetMaterialParams(Name="Mantle", Phase=0,
                Density   = PerpleX_LaMEM_Diagram("test_data/Peridotite.in"), );
```

Constitutive relationships

Work in Progress

$$\dot{\varepsilon}_{ij} = \dot{\varepsilon}_{ij}^{DislocationCreep} + \dot{\varepsilon}_{ij}^{DiffusionCreep} + \dot{\varepsilon}_{ij}^{PeierlsCreep} + \dot{\varepsilon}_{ij}^{elastic} + \dot{\varepsilon}_{ij}^{plastic} + \ldots$$

- Same for any geodynamic code
- Cause for many typos/mistakes
- Often nonlinear; usually requires pointwise iterations
- This automatically deals with nonlinearitis
- Allows adding arbitrary number of rheological elements
- New rheologies added will automatically work with all codes that employ GeoParams.jl

`GeoParams.MaterialParameters.CreepLaw.ComputeCreepLaw_EpsII` — Function

```
ComputeCreepLaw_EpsII(TauII, s:<AbstractCreepLaw, p::CreepLawVariables)
```

Returns the strainrate invariant $\dot{\varepsilon}_{II}$ for a given deviatoric stress invariant τ_{II} for any of the viscous creep laws implemented. p is a struct that can hold various parameters (such as P, T) that the creep law may need for the calculations

$$\dot{\varepsilon}_{II} = f(\tau_{II})$$

source

`GeoParams.MaterialParameters.CreepLaw.ComputeCreepLaw_TauII` — Function

```
ComputeCreepLaw_TauII(EpsII, s:<AbstractCreepLaw, p::CreepLawVariables)
```

Returns the deviatoric stress invariant τ_{II} for a given strain rate invariant $\dot{\varepsilon}_{II}$ for any of the viscous creep laws implemented. p is a struct that can hold various parameters (such as P, T) that the creep law may need for the calculations

$$\tau_{II} = f(\dot{\varepsilon}_{II})$$

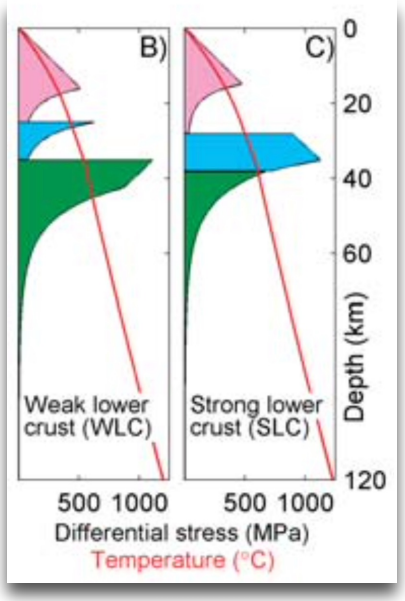
GeoParams.jl: nice to have

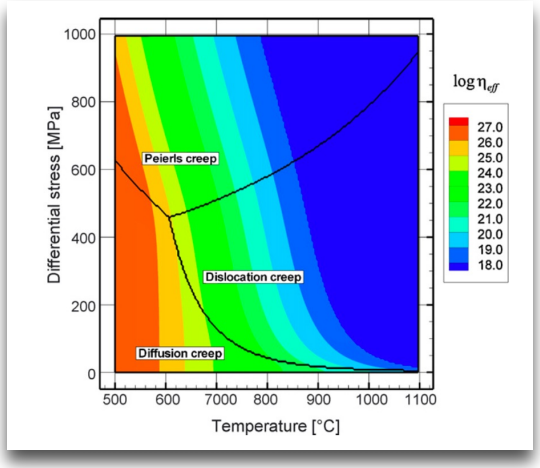
- *Automatically* generate LaTeX data tables of a simulation (+ references)
  - Minimize potential for mistakes …
- Additional functionality
  - More creep laws
  - Phase diagrams
  - Elasticity, plasticity, thermal parameters, softening
  - …
- Plotting routines
  - 1D strength envelops
  - Deformation maps
  - Stress-strain rate curves

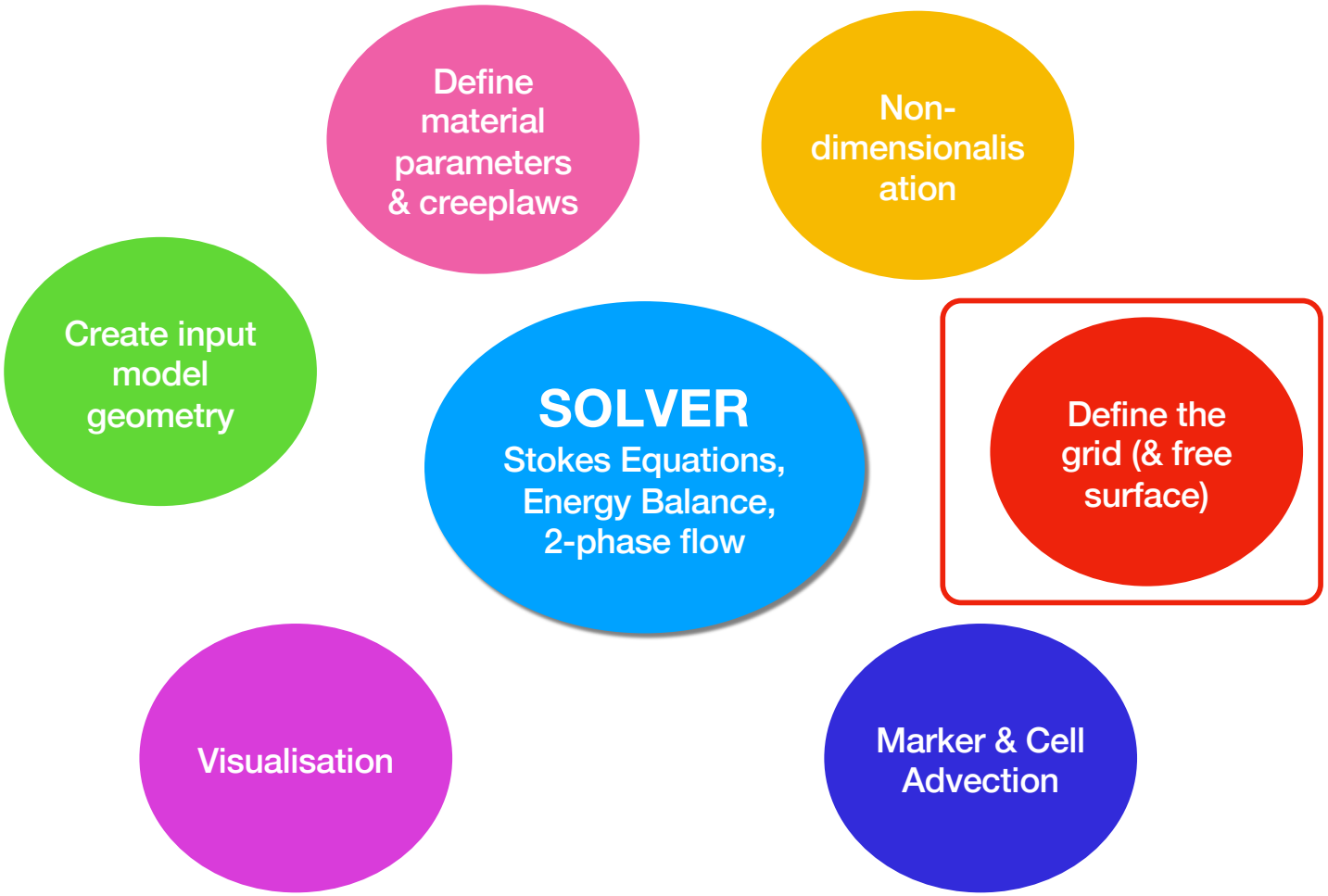
Table 1
Material properties.

	A ($Pa^{-n}s^{-1}$)	n	Q (kJ/mol)	G (GPa)	b (MPa)	θ	α (K^{-1})	H_R (W/m^3)	k (W/m/K)	ρ (kg/m^3)	c (J/kg/K)	f
Upper crust	5.07×10^{-18}	2.3	154	10	10	30	3.2×10^{-5}	0.5×10^{-6}	2.5	2700	1050	1
Lower crust incl.-model	3.20×10^{-20}	3.0	276	10	10	30	3.2×10^{-5}	0	2.5	2850	1050	1
Lower crust	5.05×10^{-28}	4.7	485	10	10	30	3.2×10^{-5}	0.2×10^{-6}	2.2	2850	1050	0.57
Mantle	2.50×10^{-17}	3.5	532	10	Pe*	Pe*	3.2×10^{-5}	0	3.0	3300	1050	1
Weak inclusion	5.05×10^{-28}	4.7	485	10	10	30	3.2×10^{-5}	0.2×10^{-6}	2.2	2850	1050	0.1
Strong incl. upper crust	5.07×10^{-18}	2.3	154	10	10	30	3.2×10^{-5}	0.2×10^{-6}	2.2	2850	1050	50
Strong incl. lower crust	2.50×10^{-17}	3.5	532	10	10	30	3.2×10^{-5}	0	3.0	3300	1050	1

Creep parameters A, n and Q are from Afonso and Ranalli (2004). Upper crust corresponds to granite, lower crust to diabase and mantle to olivine. See text for explanation of symbols. Pe*: The stresses in the mantle are limited by a Peierls creep law with parameters from Goetze and Evans (1979); see also Schmalholz et al. (2009). For the lower crust in the inclusion model (Fig. 2A) another flow law was used (Maryland Diabase from Carter and Tsenn, 1987) than for the lower crust in the lithospheric model (Fig. 2B).

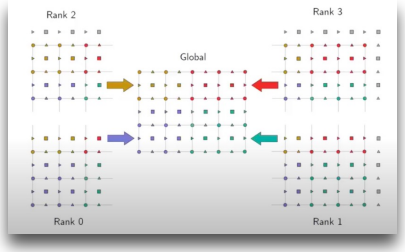





Define material parameters & creeplaws
Non-dimensionalisation
Create input model geometry
SOLVER
Stokes Equations, Energy Balance, 2-phase flow
Define the grid (& free surface)
Visualisation
Marker & Cell Advection

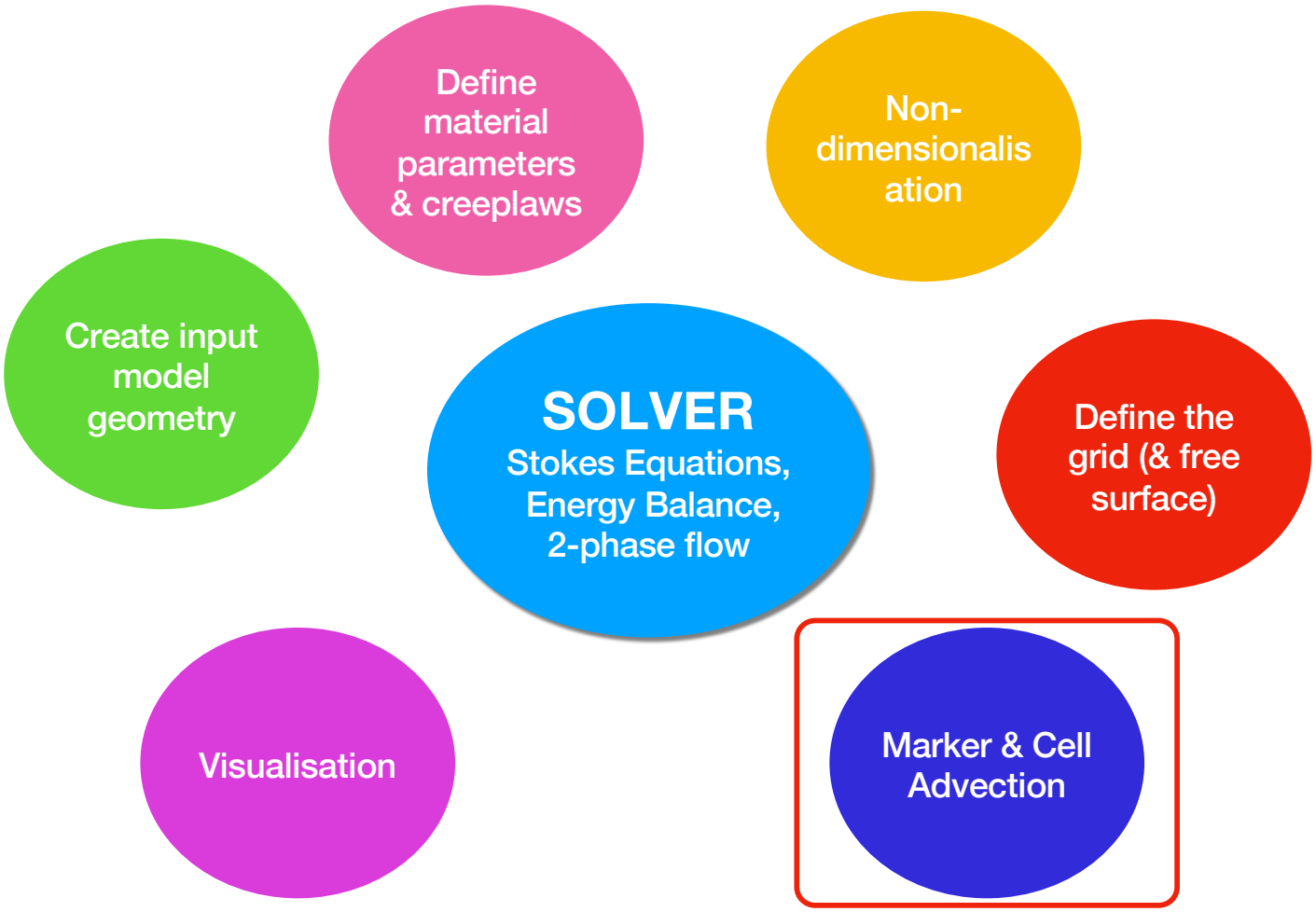
CompGrids.jl [*in progress*]

- Unified interface to create grids
  - Works with ParallelStencil.jl and with PETSc.jl
  - Collocated or staggered finite difference grids
  - CPU or GPU
  - MPI parallel or on a single core
- Create initial conditions & output/visualisation independent of the solver
- Try different solvers for the same problem



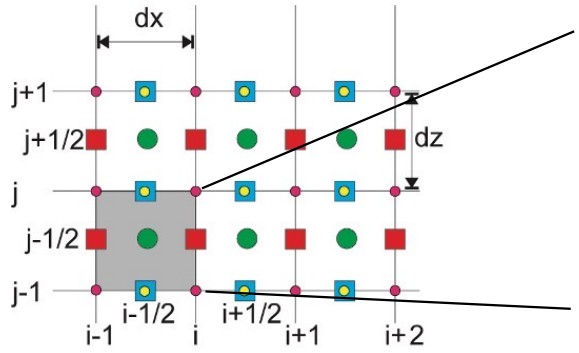

```
julia> @init_backend(PETSc, Threads, true);

julia> grid = RegularRectilinearCollocatedGrid(
                            size=(10,20),
                            extent=(100.0,110),
                            fields=(T=0,P=11.22))
RegularRectilinearCollocatedGrid{Float64, 2, Backend{BackendPETSc, Float64}}
        Backend: PETSc ( Threads | MPI )
       gridtype: Collocated
         domain: x ∈ [-50.0, 50.0], y ∈ [-55.0, 55.0]
       topology: (Bounded, Bounded)
     resolution: (10, 20) (global, no halo)
                 (10, 20) (local,  no halo)
 grid spacing Δ: (11.11111111111111, 5.7894736842105265)
         fields: (:T, :P)
```

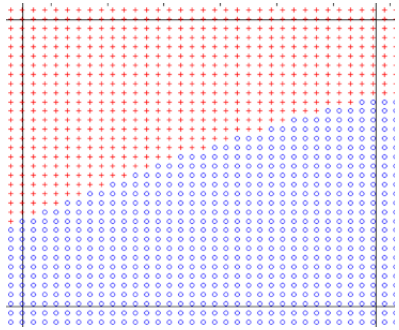
```
julia> @init_backend(ParallelStencil, Threads, true);

julia> grid = RegularRectilinearCollocatedGrid(
                          size=(10,20, 30),
                             x=(10,20),y=(0,30), z=(-10,0),
                          fields=(T=0,P=11.22))
Global grid: 12x22x32 (nprocs: 1, dims: 1x1x1)
RegularRectilinearCollocatedGrid{Float64, 3, Backend{BackendParallelStencil, Float64}}
        Backend: ParallelStencil ( Threads | MPI )
       gridtype: Collocated
         domain: x ∈ [10.0, 20.0], y ∈ [0.0, 30.0], z ∈ [-10.0, 0.0]
       topology: (Bounded, Bounded, Bounded)
     resolution: (10, 20, 30) (global, no halo)
                 (12, 22, 32) (local  + halo)
                 (12, 22, 32) (global + halo)
 grid spacing Δ: (1.1111111111111112, 1.5789473684210527, 0.3448275862068966)
         fields: (:T, :P)
```


Define material parameters & creeplaws
Non-dimensionalisation
Create input model geometry
SOLVER
Stokes Equations, Energy Balance, 2-phase flow
Define the grid (& free surface)
Visualisation
Marker & Cell Advection

MACadvect.jl [*planned*]

- Marker & Cell advection
  - Works with PETSc or ParallelStencil
  - On CPU or GPU
  - On a single core or on a MPI-parallel machine
- Automatically interpolate parameters from markers to grid (and back)
- Particle Insertion/Deletion routines
- Works for an arbitrary number of fields carried on the markers
- Crucial to have for many geodynamics codes!





```
struct Marker{T} <: AbstractMarker
    coord  :: Vector{T}     # Vector of coordinates
    phase  :: Integer       # The phase type of the particle
    props  :: NamedTuple    # Various properties that the marker can carry
end
```

```
julia> m = Marker([1.0],0,(T=800, P=1000, τ=[10.0 20.0; 30.0 40.0] ))
Marker{Float64}([1.0], 0, (T = 800, P = 1000, τ = [10.0 20.0; 30.0 40.0]))

julia> m.props.τ
2×2 Matrix{Float64}:
 10.0  20.0
 30.0  40.0

julia> m.phase
0
```

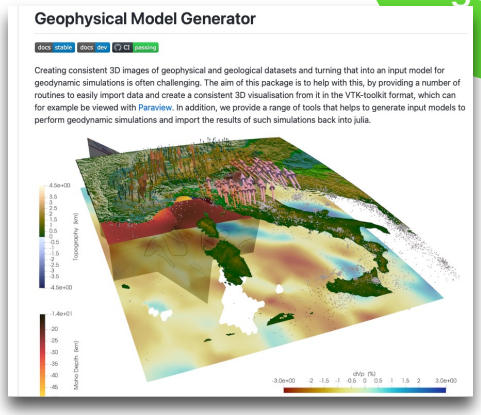
New (composable) tools to develop new software packages in julia

Define material parameters & creeplaws

GeoParams.jl

Non-dimensionalisation

Create input model geometry



GeophysicalModelGenerator.jl

SOLVER
Stokes Equations, Energy Balance, 2-phase flow

PETSc.jl
ParallelStencil.jl
ImplicitGlobalGrid.jl
JustRelax.jl

```
julia> @init_backend(ParallelStencil, Threads, true);

julia> grid = RegularRectilinearCollocatedGrid(
                          size=(10,20, 30),
                             x=(10,20),y=(0,30), z=(-10,0),
                        fields=(T=0,P=11.22))
Global grid: 12x22x32 (nprocs: 1, dims: 1x1x1)
RegularRectilinearCollocatedGrid{Float64, 3, Backend{BackendParallelStencil, Float64}}
        Backend: ParallelStencil ( Threads | MPI )
       gridtype: Collocated
         domain: x ∈ [10.0, 20.0], y ∈ [0.0, 30.0], z ∈ [-10.0, 0.0]
       topology: (Bounded, Bounded, Bounded)
     resolution: (10, 20, 30) (global, no halo)
                 (12, 22, 32) (local  + halo)
                 (12, 22, 32) (global + halo)
 grid spacing Δ: (1.1111111111111112, 1.5789473684210527, 0.3448275862068966)
         fields: (:T, :P)
```

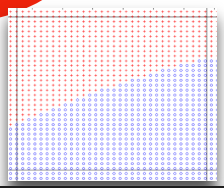
```
julia> @init_backend(PETSc, Threads, true);
```

CompGrids.jl

Define the grid (& free surface)

Visualisation

Marker & Cell Advection



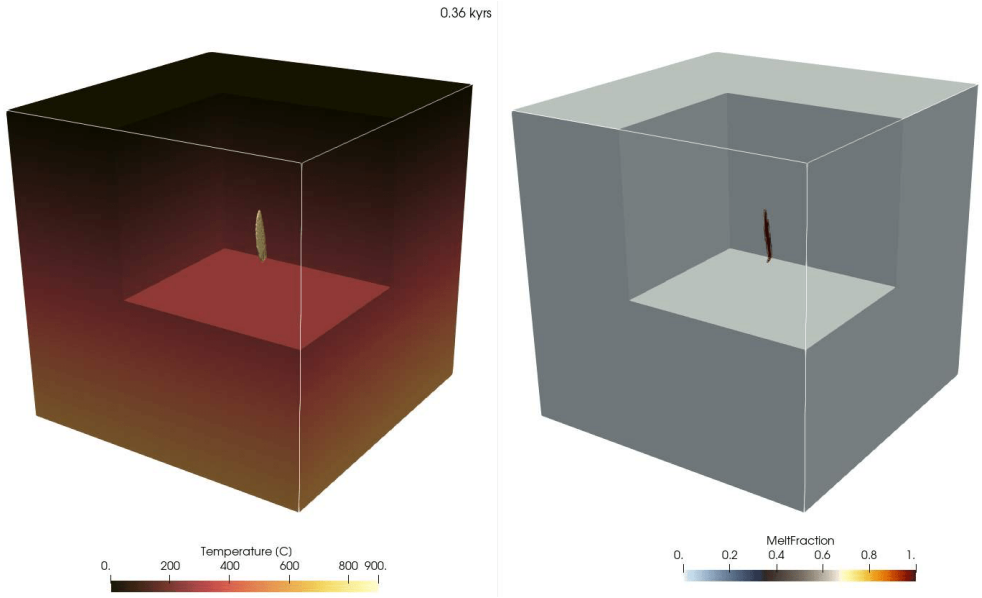
```
julia> m = Marker([1.0],0,(T=800, P=1000, τ=[10.0 20.0; 30.0 40.0] ))
Marker{Float64}([1.0], 0, (T = 800, P = 1000, τ = [10.0 20.0; 30.0 40.0]))

julia> m.props.τ
2×2 Matrix{Float64}:
 10.0  20.0
 30.0  40.0

julia> m.phase
0
```

MACadvect.jl

https://github.com/JuliaGeodynamics

Case study: MagmaThermoKinematics.jl

- Thermal evolution of intrusion of magma in the crust due to dikes
- 3D code < 100 lines
- Works on GPU & (parallel) CPU
  - thanks to ParallelStencil.jl
- Employs GeoParams.jl
  - Easy to change material parameters



https://github.com/boriskaus/MagmaThermoKinematics.jl

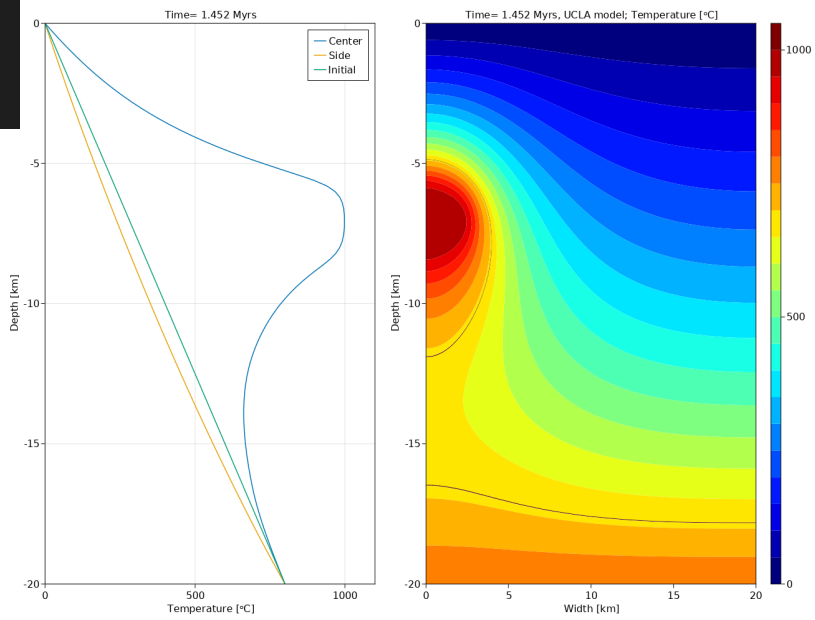
https://github.com/boriskaus/MagmaThermoKinematics.jl

/examples/Example2D_paper.jl

```
MatParam     = (SetMaterialParams(Name="Rock & partial melt", Phase=1,
                          Density    = ConstantDensity(ρ=2700/m^3),
                    EnergySourceTerms = ConstantLatentHeat(Q_L=3.13e5J/kg),
                    #     Conductivity = ConstantConductivity(k=3.3Watt/K/m),     # in case we use constant k
                         Conductivity = T_Conductivity_Whittington_parameterised(),    # T-dependent k
                         #Conductivity = T_Conductivity_Whittington(),                  # T-dependent k
                         HeatCapacity = ConstantHeatCapacity(cp=1000J/kg/K),
                             Melting = MeltingParam_4thOrder()),                      # Marxer & Ulmer data
            # add more parameters here, in case you have >1 phase in the model
            )
```

```
args1  = (;T=Arrays.T_K, P=Arrays.P)
args2  = (;z=-Arrays.Z)
```

```
@parallel (1:Nx, 1:Nz) compute_meltfraction_ps!(Arrays.ϕ, Mat_tup, Phases, args1)
@parallel (1:Nx, 1:Nz) compute_dϕdT_ps!(Arrays.dϕdT, Mat_tup, Phases, args1)
@parallel (1:Nx, 1:Nz) compute_density_ps!(Arrays.Rho, Mat_tup, Phases, args1)
@parallel (1:Nx, 1:Nz) compute_heatcapacity_ps!(Arrays.Cp, Mat_tup, Phases, args1 )
@parallel (1:Nx, 1:Nz) compute_conductivity_ps!(Arrays.Kc, Mat_tup, Phases, args1 )
@parallel (1:Nx, 1:Nz) compute_radioactive_heat_ps!(Arrays.Hr, Mat_tup, Phases, args2)
@parallel (1:Nx, 1:Nz) compute_latent_heat_ps!(Arrays.Hl, Mat_tup, Phases, args1)
```



Summary

1. Create parallel CPU/GPU solvers:
   - ParallelStencil.jl/ImplicitGlobalGrid.jl: for (parallel) pseudo-transient GPU-based codes
   - *PETSc.jl*: implicit, MPI-parallel codes. Support for staggered grids, multigrid, nonlinear solvers, automatic differentiation
2. From solver to production code:
   - *GeoParams.jl*: Define material parameters & computational routines
   - *CompGrids.jl*: Create a computational grid but switch solvers in a simple way
   - [*MACadvect.jl*]: Parallel marker & cell advection
3. Generate (realistic) input models:
   - GeophysicalModelGenerator.jl: Look at 'real' data & generate input models

- Simplifies writing parallel solvers & extending that to a production code