Using deal.II together with the Julia programming language

Code: https://github.com/fredrikekre/Deal.II_with_Julia

Fredrik Ekre - `f.ekre@tu-braunschweig.de`
Kristoffer Carlsson - `kristoffer.carlsson@juliahub.com`

What is Julia?

julia - Introduction

- Created at MIT, first public release in 2013. 1.0 released in 2018 (no more breaking changes), latest release as of now is 1.9.3. (C 1972, C++ 1985, Python 1992)
- MIT licensed (free, open source)
- **General purpose** programming language with some features that make it especially suitable for scientific computing.
- "JIT (Just In Time) compiled" (compilation happens just before a function runs)
  - Packages can cache compiled code for reuse in future Julia sessions
- Garbage collected, no explicit memory management
- Dynamically typed
- Performance achieved through specialization on function argument types and type inference (devirtualization)

julia - Basic syntax

```julia
julia> struct Circle
           r::Float64
       end

julia> area(c::Circle) = c.r^2 * π;

julia> struct Rectangle
           width::Float64
           height::Float64
       end

julia> area(r::Rectangle) = r.width * r.height;

julia> shapes = [Rectangle(2, 3),
                 Circle(2),
                 Rectangle(5, 2),];

julia> area.(shapes)
3-element Vector{Float64}:
  6.0
 12.566370614359172
 10.0
```

julia - Genericness

```
julia> f(x) = x^2 + x;

julia> f(2)
6

julia> f(3.0)
12.0

julia> f(2 + im)
5 + 5im

julia> f([1 2; 2 3])
2×2 Matrix{Int64}:
  6  10
 10  16

julia> using ForwardDiff: Dual

julia> f(Dual(3, 1))
Dual(12,7)
```

julia - Specialization

```
julia> @code_native f(2)
        imulq   %rdi, %rax
        addq    %rdi, %rax
...

julia> @code_native f(3.0)
        vmulsd  %xmm0, %xmm0, %xmm1
        vaddsd  %xmm0, %xmm1, %xmm0
...

julia> @code_native f(2 + im)
...
        imulq   %rcx, %rsi
        movq    %rdx, %rdi
        imulq   %rcx, %rdi
        leaq    (%rdx,%rdi,2), %rdi
        imulq   %rdx, %rdx
        addq    %rcx, %rsi
        subq    %rdx, %rsi
...

julia> @code_native f(Dual(2,3))
...
        imulq   %rcx, %rsi
        addq    %rcx, %rsi
        imulq   %rdx, %rcx
...
```

Why use Julia with deal.ii?

Why not only C++?

- Julia can in some cases feel more productive than C++
  - Easy to add and use packages
  - Easy to inspect generated code
  - Hot reload code without restarting
  - No build scripts, no header files, no linking, …
- Julia arguably easier to get started with

```
(@v1.9) pkg> add Tensors # add package
 ...

julia> using Tensors # load package

# use package
julia> σ = rand(SymmetricTensor{2,2})
2×2 SymmetricTensor{2, 2, Float64, 3}:
 0.219063  0.217709
 0.217709  0.916764

julia> σ_vm = √(3/2 * dev(σ) ⊡ dev(σ))
0.7840853502179173
```

```
julia> f_vm(σ) = √(3/2 * dev(σ) ⊡ dev(σ));

julia> @code_llvm debuginfo=:none f_vm(σ)
define double @julia_f_vm_835([1 x [3 x double]]* ... {
  %1 = getelementptr inbounds [1 x [3 x double]], [1 x [3 x double]]* %0
...
  %2 = getelementptr inbounds [1 x [3 x double]], [1 x [3 x double]]*
%0,...
  %3 = load double, double* %1, align 8
  %4 = load double, double* %2, align 8
  %5 = fadd double %3, %4
...

julia> @code_native f_vm(σ)

...
    movabsq    $.LCPI0_2, %rax
    vmulpd    %ymm1, %ymm0, %ymm1 # SIMD instructions
    vmulpd    %ymm0, %ymm1, %ymm0
    vmulpd    (%rax), %ymm0, %ymm0
    movabsq    $.LCPI0_3, %rax
    vbroadcastsd    (%rax), %ymm1
...
```

Why not only Julia?

- A full-fledged FEM library is a huge project.
- Some FEM packages exist in Julia:
  - https://github.com/Ferrite-FEM/Ferrite.jl (spiritually like deal.ii)
  - https://github.com/gridap/Gridap.jl (spiritually like Fenics)
  - …
- But nothing as robust, comprehensive, battle tested, documented etc. as deal.ii exist in the Julia world (yet).

Goal

- Offload the “weak form evaluation” to Julia.
  This includes:
  - Evaluating the “material routine”
  - Assembling the residual and tangent stiffness
- Use deal.ii for everything else:
  - Grids
  - DoF distribution
  - Parallelization
  - Adaptive mesh refinement
  - Shape function + function evaluation
  - Solving
  - Export results
  - …

How?

Call Julia from C (embedding)

- Julia has a C API
- Can expose Julia functions to C (via C ABI)
- No overhead compared to a normal C function call.
- Memory can be shared between C++ and Julia via pointers
- (More modern C++ API available externally, https://github.com/Clemapfel/jluna)

```c
#include <julia.h>

int main(int argc, char *argv[])
{
    /* required: setup the Julia context */
    jl_init();

    /* run Julia commands */
    jl_eval_string("print(sqrt(2.0))");

    /* get a C function pointer */
    double (*sqrt_jl)(double) =
jl_unbox_voidpointer(
        jl_eval_string("@cfunction(sqrt, Float64,
(Float64,))"));
    double ret = sqrt_jl(2.0);

    jl_atexit_hook(0);
    return 0;
}
```

CMakeLists.txt

- FindJulia.cmake : https://github.com/barche/embedding-julia
- CMakeLists.txt:

```
# rpaths
set(CMAKE_MACOSX_RPATH 1)
set(CMAKE_SKIP_BUILD_RPATH FALSE)
set(CMAKE_BUILD_WITH_INSTALL_RPATH TRUE)
set(CMAKE_INSTALL_RPATH_USE_LINK_PATH TRUE)

# find julia
set(CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR})
find_package(Julia REQUIRED)

# include + link
target_include_directories(${TARGET} PUBLIC
  "$<BUILD_INTERFACE:${Julia_INCLUDE_DIRS}>"
)
target_link_libraries(${TARGET} $<BUILD_INTERFACE:${Julia_LIBRARY}>)
```

Mapping from C++ to Julia types and back

- Primitive types match directly to the Julia equivalents
  - `double` ⇔ `Float64` (`Cdouble`)
  - `int` ⇔ `Int32` (`Cint`)
  - `double*` ⇔ `Ptr{Float64}`
  - ...
- "POD"-structs (plain old data) map directly to Julia structs
  - `struct Foo { double a; int b };` ⇔ `struct Foo; a::Float64; b::Int32; end`
  - `std::array{T, N}` ⇔ `NTuple{N, T}`
- Arrays shared with pointers (no copying needed)

Converting from deal.ii to Julia types and back

- Deal.ii `Tensor`s
  - Julia package [Tensors.jl] similar to deal.ii Tensor type.
  - "Unroll" into `std::array` in Julia tensor order send over to Julia
  - If element order is same in Julia and deal.ii, could potentially be passed directly
- Deal.ii arrays (`FullMatrix`, `Vector`)
  - Pass pointers to auxiliary `std::vector` to Julia, mutate them there, then copy over to deal.ii with `operator[]`.
  - With some assumptions about memory layout, memory could directly be shared between Julia and deal.ii.

Pseudo code for the assembly loop

```
template <int dim>
void HyperelasticitySim<dim>::assemble_system(const Vector<double>&
solution_delta) {
  // Setup FEValues etc
  // Allocate deal.ii arrays and Julia arrays for residual and tangent
  for (const auto& cell : dof_handler.active_cell_iterators()) {
    // Zero out assembly arrays
    fe_values.reinit(cell);
    // Compute function values etc:
    fe_values[displacement].get_function_gradients(total_solution, grad_u);
    for (unsigned int q_point = 0; q_point < n_q_points; ++q_point) {
      auto grad_u_q = grad_u[q_point];
      // Calculate shape values + shape gradients
      // Copy deal.ii data to Julia arrays
      // Call Julia with the Julia arrays, updating the residual and tangent
      this->jl_assemble(...)
    }
    // Copy data from Julia arrays to deal.ii arrays
    constraints.distribute_local_to_global(...) // etc
  }
}
```

Concrete example: Solid mechanics – large strain hyperelasticity

Weak form: Solid mechanics large strains

$$\int_{\Omega} [\nabla_{\mathbf{X}} \delta \mathbf{u}] : \mathbf{P}(\mathbf{u}) \, \mathrm{d}\Omega = \int_{\Omega} \delta \mathbf{u} \cdot \mathbf{b} \, \mathrm{d}\Omega + \int_{\Gamma_{\mathrm{N}}} \delta \mathbf{u} \cdot \mathbf{t} \, \mathrm{d}\Gamma \quad \forall \delta \mathbf{u} \in \mathbb{U}^0,$$

- ***u***: Displacement
- ***P***: First Piola-Kirchoff stress
- ***b***: body load
- ***t:*** surface load (traction)
- Gradients are taken w.r.t the reference configuration

Material model

$$\Psi(\mathbf{C}) = \frac{\mu}{2}(I_1 - 3) - \mu \ln(J) + \frac{\lambda}{2} \ln(J)^2$$

Neo-Hookean material

- $\mathbf{C} = \mathbf{F}^{\mathrm{T}} \cdot \mathbf{F}$ right Cauchy Green tensor where $\mathbf{F} = \mathbf{I} + \nabla_{\mathbf{X}} \mathbf{u}$
- $I_1 = \mathrm{tr}(\mathbf{C})$ first invariant
- $J = \sqrt{\det(\mathbf{C})}$
- μ and λ are material parameters
- $\mathbf{S} = 2\frac{\partial \Psi}{\partial \mathbf{C}}$
- $\mathbf{P} = \mathbf{F} \cdot \mathbf{S}$
- Given **F** we can compute **P** needed in the weak form

Julia code

Data from deal.ii

$$\Psi(\mathbf{C}) = \frac{\mu}{2}(I_1 - 3) - \mu \ln(J) + \frac{\lambda}{2}\ln(J)^2$$

```julia
struct NeoHooke
    μ::Float64 # double
    λ::Float64 # double
end

function Ψ(C, mp::NeoHooke)
    Ic, J = tr(C), sqrt(det(C))
    return mp.μ / 2 * (Ic - 3) - mp.μ * log(J) + mp.λ / 2 * log(J)^2
end

function constitutive_driver(C, mp::NeoHooke)
    # Automatic Differentiation
    ∂²Ψ∂C², ∂Ψ∂C = Tensors.hessian(y -> Ψ(y, mp), C, :all)
    S, ∂S∂C = 2.0 * ∂Ψ∂C, 2.0 * ∂²Ψ∂C²

    return S, ∂S∂C
end
```

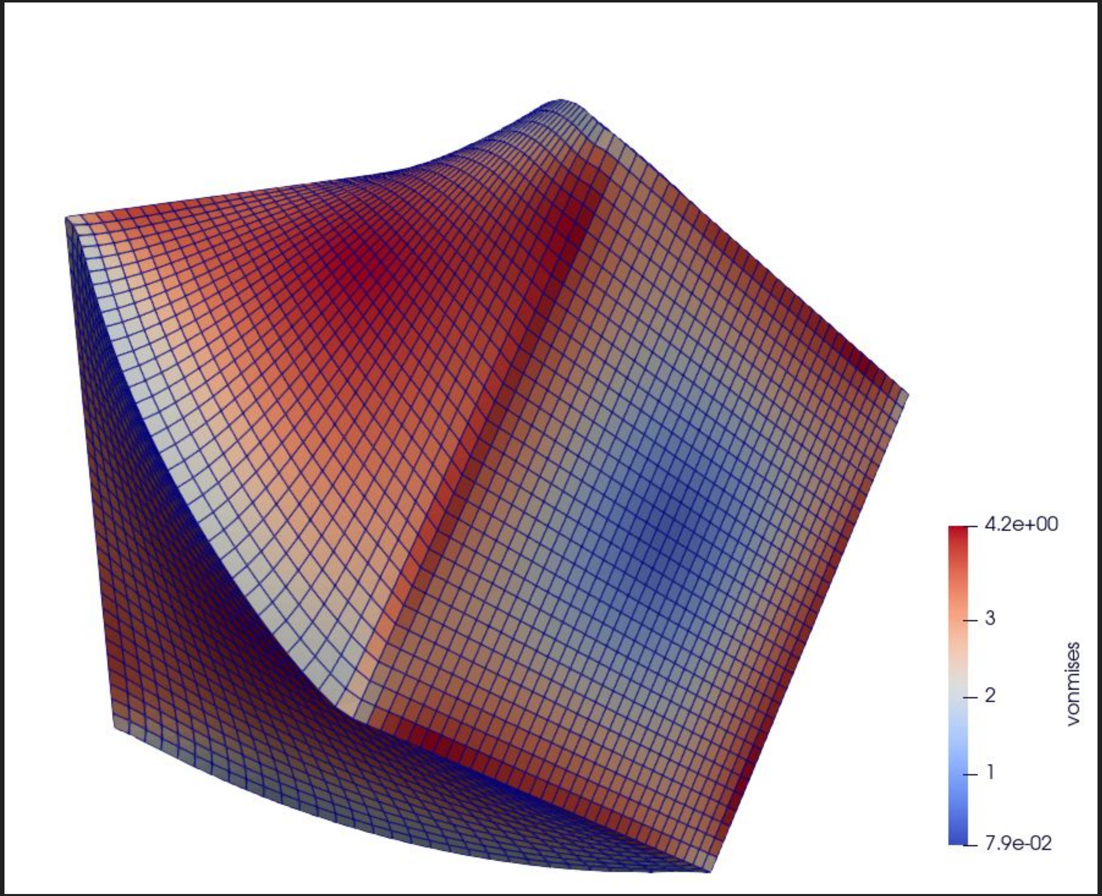
```julia
function do_assemble!(
        ge::Vector{Float64}, ke::Matrix{Float64},
        ∇u::Tensor{2}, δuis::Vector{<:Vec}, ∇δuis::Vector{<:Tensor{2}},
        ndofs, dΩ::Float64, mp::NeoHooke)
    )

    # Compute deformation gradient F and right Cauchy-Green tensor C
    F = one(∇u) + ∇u
    C = tdot(F) # F' ⋅ F

    # Compute stress and tangent
    S, ∂S∂C = constitutive_driver(C, mp)

    P = F ⋅ S
    I = one(S)
    ∂P∂F = otimesu(I, S) + 2 * otimesu(F, I) ⊡ ∂S∂C ⊡ otimesu(F', I)

    # Assemble contributions
    for i in 1:ndofs
        δui, ∇δui  = δuis[i], ∇δuis[i]
        # Residual
        ge[i] += ( ∇δui ⊡ P ) * dΩ

        # Tangent
        for j in 1:ndofs
            ∇δuj = ∇δuis[j]
            ke[i, j] += ( ∇δui ⊡ ∂P∂F ⊡ ∇δuj ) * dΩ
        end
    end
    return
end
```

$$(\underline{g})_i = \int_\Omega [\nabla_\mathbf{X} \delta \mathbf{u}_i] : \mathbf{P}\, \mathrm{d}\Omega$$

$$(\underline{\underline{K}})_{ij} = \int_\Omega [\nabla_\mathbf{X} \delta \mathbf{u}_i] : \frac{\partial \mathbf{P}}{\partial \mathbf{F}} : [\nabla_\mathbf{X} \delta \mathbf{u}_j]\, \mathrm{d}\Omega.$$

Results

Von Mise stress for the unit cube with applied torsion (Dirichlet boundary conditions)



Results

```
+---------------------------------------------+------------+------------+
| Total wallclock time elapsed since start    | 1.837e+02s |            |
|                                             |            |            |
| Section                         | no. calls |  wall time | % of total |
+---------------------------------+-----------+------------+------------+
| Global assembly                 |        56 | 1.104e+02s |  6.01e+01% |
|   Cell loop                     |   1835008 | 1.089e+02s |  5.93e+01% |
|     Julia kernel                |  14680064 | 8.319e+01s |  4.53e+01% |
| Solving linear system           |        46 | 7.061e+01s |  3.84e+01% |
+---------------------------------+-----------+------------+------------+
```

Conclusions and TODOs

- Using Julia to offload parts of the core computations in deal.ii is feasible
    - Interoperability and data sharing easy
    - Implementing material routine(s) in Julia arguably more convenient
- TODOs
    - Use jlluna for proper modern C++ bindings to Julia over Julia’s C-API.
    - Allocate material states on the Julia side more natural if using a “material model” library in Julia
    - Create something like `julia_dealII.h` to put the various conversions functions in.