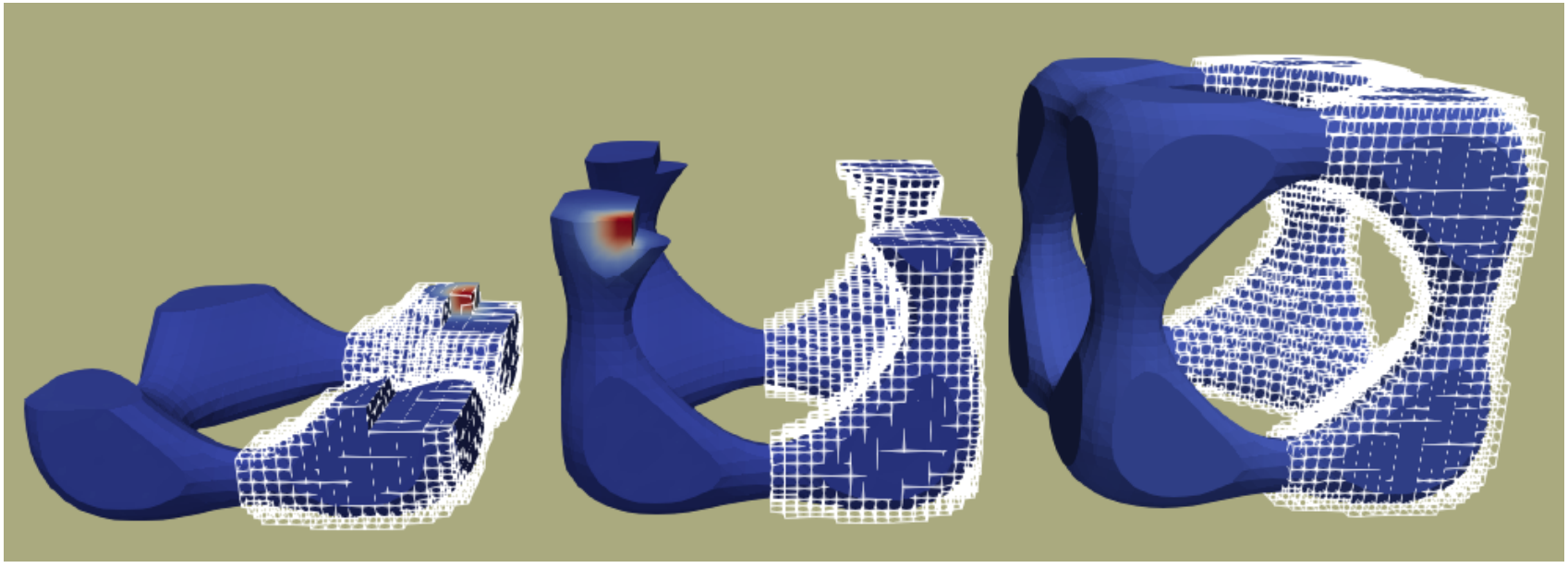
Gridap: Towards productivity and performance in Julia

Santiago Badia



UNSW Comp Math Webinar, April 21st 2020

MONASH University

Disclaimer

My concerns about **poor productivity wrt software development**

Workflow

Design new method $\rightarrow$ analyse it $\rightarrow$ implement it (rapid prototyping) $\rightarrow$ exploit it in (large scale) applications (performance)

Probably not your case: Focused on analysis (academic examples) or application side (existing libraries OK)

Numerical PDE solvers

Numerical PDE software implementations *are complex*

PDEs: multiphysics, multiscale, constitutive models

Discretisation: (grad, curl, div)-conforming, dG, unfitted FEM, hybrid and virtual elements, h/p-adaptivity

(Non)linear solvers: multiscale/multilevel solvers strongly coupled to PDE structure (not black-box)

Large scale computations: distributed-memory implementations, accelerators, ...

Combined w/: forward/inverse UQ, data-driven parameter identification, ...

Scientific computing teams

PhD students (3-4y), postdocs (1-3y), no computer scientists

Software dev policies

Start from scratch: Academic codes in **dynamic languages** (MATLAB, Python...), wasting previous work, no performance, usually not accessible code (**no reproducible science**)

Existing numerical PDE libraries

Software dev policies

Reuse: Excellent pool of high-performance libraries: deal.ii, Fenics, FEMPAR, MOOSE, libmesh, Firedrake, DUNE, NGSolve, etc.

- **Static languages** (C++, FORTRAN08...) for *performance*
- Excellent if they provide all you need (Python interfaces)
- Far more involved if not (**productivity loss**)

Productivity vs performance

Productivity

Related to **dynamic languages** (Python, MATLAB...): More expressive, no compilation step, interactive development (debugging on-the-fly), better for math-related bugs (no benefit from static compilation), no set-up of environment (compilers, system libraries, etc)

Performance

Related to **static languages** (C/C++,FORTRAN,...): Compilers generate *highly optimised* code

Compromise solutions

Dynamic-static combinations:

1. vectorised PDE solvers in Python + external pre-compiled libraries (NumPy in C)
    - Constraints over the dynamic code (e.g. vectorisation)
2. high-level Python interface of a static PDE library (Fenics in C++), etc.
    - **Two-language barrier:** When changes require to get into static library

https://julialang.org/

21st century FORTRAN, designed for numerical computation (MIT, 2011-)

All-in-one (?)

Productive: Dynamic language (as Python, MATLAB...)

Performant: Advanced type-inference system + just-in-time (JIT) compilation

Julia universe

- Own REPL (MATLAB-like), package manager is awesome
- Seamless integration with **Github** (register packages, automatic documentation deployment...)
- Built-in unit testing, performance tools, own parallel mechanisms...

Julia features

- **Not OO:** No inheritance of concrete types (only abstract types), *use composition, not inheritance*, *classify by their actions, not their attributes*...
- **Multiple dispatching paradigm:** functions not bound to types, dispatching wrt all arguments

Julia features

- **Not OO:** No inheritance of concrete types (only abstract types), *use composition, not inheritance*, *classify by their actions, not their attributes*...
- **Multiple dispatching paradigm:** functions not bound to types, dispatching wrt all arguments

Let us play a little with with Julia...

Gridap seed started in Christmas 2018 trying to increase productivity in my team

Some key decisions based on previous experience and Julia capabilities:

- Functional-like style i.e. **immutable objects**, no *state diagram* (just cache arrays for performance)
- **Lazy evaluation** of expressions (implement unary/binary expression trees for types)

In the spirit of the lazy matrix example...

CellField

Given a *cell* in a partition $\mathcal{T}$ of a domain $\mathcal{D}$ (e.g. cells, faces, edges in a mesh), it provides a `Field`. A `Field` assigns a *physical quantity (n-tensor) per space(-time) point* in the manifold.

Key method, lazy evaluation: Given an array of points per cell in $\mathcal{T}$, we can evaluate a `CellField`, returning an array of scalars/vectors/tensors (`FieldValue`) per cell per point

```
Evaluate(cf::CellField,ps::CellPoints)
::CellArray{FieldValue}
```

We also implement operations:

- Unary operations: e.g. $\nabla()$, $\nabla \times ()$, $\nabla \cdot ()$, etc.
- Binary operations: inner$(,)$, $\times$, etc.

With these types, we represent *FE functions, FE bases, constitutive models, etc.*

Applying a `CellField` to a `CellPoints` (integration points) plus expression trees we can integrate forms and assemble matrices

We also implement operations:

- Unary operations: e.g. $\nabla()$, $\nabla \times ()$, $\nabla \cdot ()$, etc.
- Binary operations: inner$(,)$, $\times$, etc.

With these types, we represent *FE functions, FE bases, constitutive models, etc.*

Applying a `CellField` to a `CellPoints` (integration points) plus expression trees we can integrate forms and assemble matrices

Let us look at Gridap Tutorial 1

Gridap status

Gridap is pretty comprehensive (big thanks to F Verdugo's amazing work at UPC):

- Lagrangian, Raviart-Thomas, Nedelec, dG
- Embedded methods (own constructive solid geometry engine)
- Multifield or multiphysics methods
- Interaction with GMesh, Pardiso, PETSc...
- dimension-agnostic (5-dim Laplacian), order-agnostic

Quite rich documentation, tutorials, automatic testing, etc.

After 1 year and two developers (part time!)... *highly productive environment*

Gridap for teaching

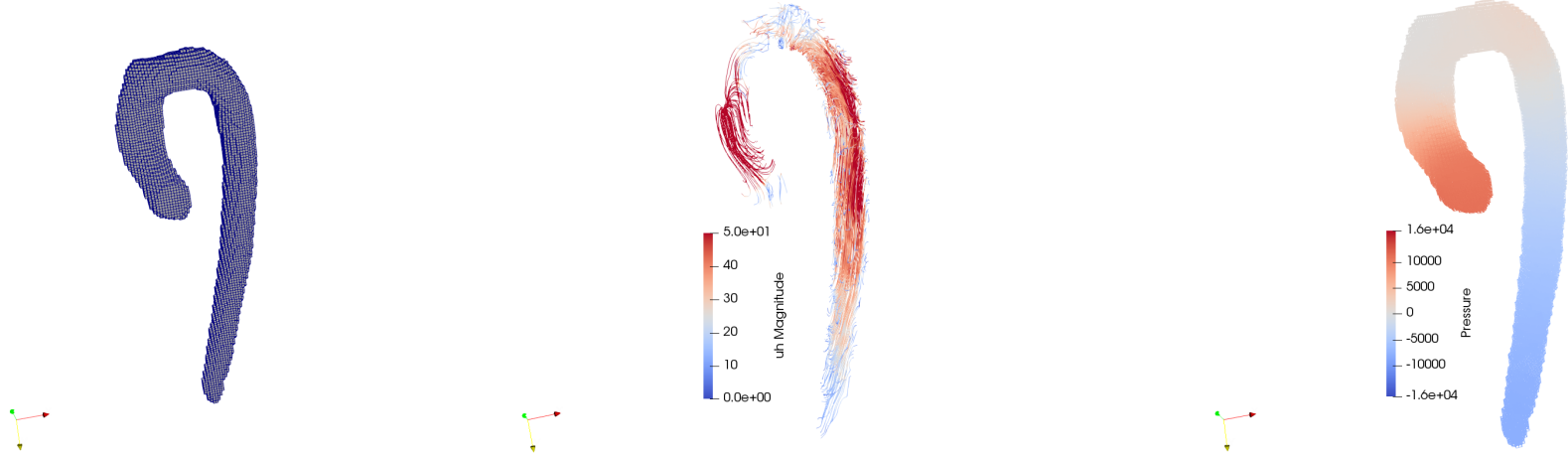
Objective: same **software for research and teaching**

- Designing FE tutorials in *MTH5321 - Methods of computational mathematics*

Gridap for teaching

Objective: same **software for research and teaching**

- One undergrad AMSI project on *Gridap*: No idea about FEs/coding $\rightarrow$ from patient-specific MRI data of aorta velocity field to pressure field (Navier-Stokes solver...) in about 2 months



Gridap future

This is just the beginning:

- Distributed-memory integration/assembly (ongoing)
- Parallel hp-adaptivity (ongoing)
- Space-time discretisations (ongoing)
- Virtual element methods
- Interaction with other Julia packages (optimisation, ML, UQ, ODE, automatic diff...)
- ...

Performance analysis:

- Poisson solver w/ 1st order FEs on 145^3 mesh in 30 sec (CG+AMG about 60%), similar for 30^4 mesh
- Trying to write performant code (type stable), but NO optimisation yet
- *Performance analysis* on the way (x2-3 performance hit OK if x2-3 productivity, but does not seem to be the case)
- Further topic: In fact, type stability + JIT compilation *eliminates virtualisation* overhead in static languages

Further reading

Learning Julia

`julialang.org`

Gridap

`github.com/gridap/Gridap.jl`

Gridap tutorials

`github.com/gridap/Tutorials`

Further reading

Learning Julia

`julialang.org`

Gridap

`github.com/gridap/Gridap.jl`

Gridap tutorials

`github.com/gridap/Tutorials`

Thanks!

Contact me with your comments/suggestions!

santiago.badia@monash.edu