# GalerkinToolkit.jl: Towards a multi-platform finite element API with deep integration with the Julia package ecosystem

Francesc Verdugo (and collaborators)

VRIJE UNIVERSITEIT AMSTERDAM

GalerkinToolkit
High-performance finite element toolbox in Julia

netherlands eScience center

Grant ID: NLESC.SS.2023.008

NUM-SCARS workshop · 2026-01-23

# The team

Main developer

- Francesc Verdugo (VU)

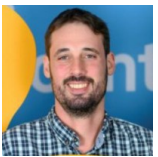
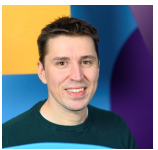
GalerkinToolkit form compiler

- Xiaowei Ouyang (VU)
- Tiziano de Matteis (VU)

Software sustainability & GPU

- Robin Richardson (NLeSc)
- Stijn Heldens (NLeSc)
- Alessio Sclocco (NLeSc)




# Why another FEM library?

- Deal.ii
- GalerkinToolkit
- Gridap
- FEniCS
- Firedrake
- MFEM
- …

# Today’s agenda

- Which are the benefits of our API?
- What is new in our code generation strategy?
- Towards GPU support

# Simple FEM in math notation

Let $f, g : \mathbb{R}^d \longrightarrow \mathbb{R}$ be two functions.
Let $\Omega \subset \mathbb{R}^d$ be a computational domain.
Let $\mathcal{T}$ be a *triangulation* of $\Omega$
Let $V := \{v \in H^1(\Omega) : \ v(x)$ is a polynomial for all $x \in T \in \mathcal{T}\}$
Let $V_0 := \{v \in V : \ v(x) = 0$ for $x \in \partial\Omega\}$
Let $V_g := \{v \in V : \ v(x) = g(x)$ for $x \in \partial\Omega\}$

Find $u \in V_g$ such that:
$\int_\Omega \nabla u(x) \cdot \nabla v(x)\, \mathrm{d}x = \int_\Omega f(x)v(x)\, \mathrm{d}x$ for all $v \in V_0$

# Implementation with Deal.ii

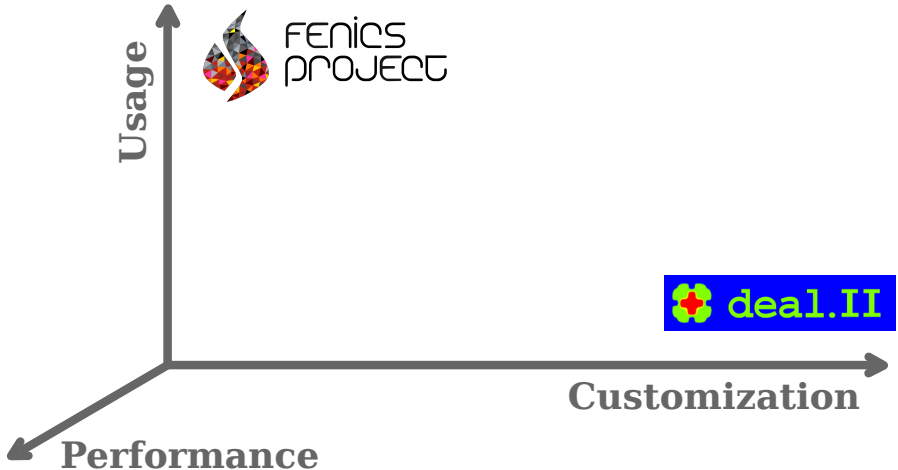
```cpp
#include <deal.II/grid/tria.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/tria_accessor.h>
#include <deal.II/grid/tria_iterator.h>
#include <deal.II/dofs/dof_accessor.h>
#include <deal.II/fe/fe_q.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/function.h>
#include <deal.II/numerics/vector_tools.h>
#include <deal.II/numerics/matrix_tools.h>
#include <deal.II/lac/vector.h>
#include <deal.II/lac/full_matrix.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/lac/dynamic_sparsity_pattern.h>
#include <deal.II/lac/solver_cg.h>
#include <deal.II/lac/precondition.h>
#include <deal.II/numerics/data_out.h>
#include <fstream>
#include <iostream>
using namespace dealii;
class Step3
{
public:
  Step3();
  void run();
private:
  void make_grid();
  void setup_system();
  void assemble_system();
  void solve();
  void output_results() const;
  Triangulation<2> triangulation;
  FE_Q<2>          fe;
  DoFHandler<2>    dof_handler;
  SparsityPattern      sparsity_pattern;
  SparseMatrix<double> system_matrix;
  Vector<double> solution;
  Vector<double> system_rhs;
};
Step3::Step3()
  : fe(1)
  , dof_handler(triangulation)
{}
void Step3::make_grid()
{
  GridGenerator::hyper_cube(triangulation, -1, 1);
  triangulation.refine_global(5);
  std::cout << "Number of active cells: " << triangulation.n_active_cells()
            << std::endl;
}
void Step3::setup_system()
{
  dof_handler.distribute_dofs(fe);
  std::cout << "Number of degrees of freedom: " << dof_handler.n_dofs()
            << std::endl;
  DynamicSparsityPattern dsp(dof_handler.n_dofs());
  DoFTools::make_sparsity_pattern(dof_handler, dsp);
  sparsity_pattern.copy_from(dsp);
  system_matrix.reinit(sparsity_pattern);
  solution.reinit(dof_handler.n_dofs());
  system_rhs.reinit(dof_handler.n_dofs());
}
void Step3::assemble_system()
{
  QGauss<2> quadrature_formula(fe.degree + 1);
  FEValues<2> fe_values(fe,
                        quadrature_formula,
                        update_values | update_gradients | update_JxW_values);
  const unsigned int dofs_per_cell = fe.dofs_per_cell;
  FullMatrix<double> cell_matrix(dofs_per_cell, dofs_per_cell);
  Vector<double>     cell_rhs(dofs_per_cell);
  std::vector<types::global_dof_index> local_dof_indices(dofs_per_cell);
  for (const auto &cell : dof_handler.active_cell_iterators())
    {
      fe_values.reinit(cell);
      cell_matrix = 0;
      cell_rhs    = 0;
      for (const unsigned int q_index : fe_values.quadrature_point_indices())
        {
          for (const unsigned int i : fe_values.dof_indices())
            for (const unsigned int j : fe_values.dof_indices())
              cell_matrix(i, j) +=
                (fe_values.shape_grad(i, q_index) * // grad phi_i(x_q)
                 fe_values.shape_grad(j, q_index) * // grad phi_j(x_q)
                 fe_values.JxW(q_index));           // dx
          for (const unsigned int i : fe_values.dof_indices())
            cell_rhs(i) += (fe_values.shape_value(i, q_index) * // phi_i(x_q)
                            1.* // f(x_q)
                            fe_values.JxW(q_index));           // dx
        }
      cell->get_dof_indices(local_dof_indices);
      for (const unsigned int i : fe_values.dof_indices())
        for (const unsigned int j : fe_values.dof_indices())
          system_matrix.add(local_dof_indices[i],
                            local_dof_indices[j],
                            cell_matrix(i, j));
      for (const unsigned int i : fe_values.dof_indices())
        system_rhs(local_dof_indices[i]) += cell_rhs(i);
    }
  std::map<types::global_dof_index, double> boundary_values;
  VectorTools::interpolate_boundary_values(dof_handler,
                                           0,
                                           Functions::ZeroFunction<2>(),
                                           boundary_values);
  MatrixTools::apply_boundary_values(boundary_values,
                                     system_matrix,
                                     solution,
                                     system_rhs);
}
void Step3::solve()
{
  SolverControl solver_control(1000, 1e-12);
  SolverCG<Vector<double>> solver(solver_control);
  solver.solve(system_matrix, solution, system_rhs, PreconditionIdentity());
}
void Step3::output_results() const
{
  DataOut<2> data_out;
  data_out.attach_dof_handler(dof_handler);
  data_out.add_data_vector(solution, "solution");
  data_out.build_patches();
  std::ofstream output("solution.vtk");
  data_out.write_vtk(output);
}
void Step3::run()
{
  make_grid();
  setup_system();
  assemble_system();
  solve();
  output_results();
}
int main()
{
  deallog.depth_console(2);
  Step3 laplace_problem;
  laplace_problem.run();
  return 0;
}
```

deal.II

# Implementation with FEniCS

```
1 from fenics import *
2 mesh = UnitSquareMesh(8, 8)
3 V = FunctionSpace(mesh, 'P', 1)
4 u_D = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]', degree=2)
5 def boundary(x, on_boundary):
6     return on_boundary
7 bc = DirichletBC(V, u_D, boundary)
8 u = TrialFunction(V)
9 v = TestFunction(V)
10 f = Constant(-6.0)
11 a = dot(grad(u), grad(v))*dx
12 L = f*v*dx
13 u = Function(V)
14 solve(a == L, u, bc)
15 vtkfile = File('poisson/solution.pvd')
16 vtkfile << u
```

# API trade-offs



# Implementation with GalerkinToolkit

```
using LinearAlgebra
import Gmsh
import GalerkinToolkit as GT
import LinearSolve
import ForwardDiff
import GLMakie
mesh = GT.mesh_from_msh("model.msh")
Ω = GT.interior(mesh)
Γd = GT.boundary(mesh)
f = GT.analytical_field(x->1.0,Ω)
g = GT.analytical_field(x->2.0,Ω)
k = 1
V = GT.lagrange_space(Ω,k;dirichlet_boundary=Γd)
```

```
T = Float64
uhd = GT.zero_dirichlet_field(T,V)
GT.interpolate_dirichlet!(g,uhd)
degree = 2*k
dΩ = GT.measure(Ω,degree)
∇ = ForwardDiff.gradient
a = (u,v) -> GT.∫( x->∇(u,x)⋅∇(v,x), dΩ)
l = v -> GT.∫( x->v(x)*f(x), dΩ)
p = GT.SciMLBase_LinearProblem(uhd,a,l)
sol = LinearSolve.solve(p)
uh = GT.solution_field(uhd,sol)
GT.makie_surfaces(Ω;color=uh)
```

```
a = (u,v) -> GT.∫( x->∇(u,x)⋅∇(v,x), dΩ)
l = v -> GT.∫( x->v(x)*f(x), dΩ)
p = GT.SciMLBase_LinearProblem(uhd,a,l)
```

# Implementation with GalerkinToolkit (v2)

```julia
using LinearAlgebra
import Gmsh
import GalerkinToolkit as GT
import LinearSolve
import ForwardDiff
import GLMakie as Makie
mesh = GT.mesh_from_msh("model.msh")
Ω = GT.interior(mesh)
Γd = GT.boundary(mesh)
f = GT.analytical_field(x->1.0,Ω)
g = GT.analytical_field(x->2.0,Ω)
k = 1
V = GT.lagrange_space(Ω,k;dirichlet_boundary=Γd)
T = Float64
uhd = GT.zero_dirichlet_field(T,V)
GT.interpolate_dirichlet!(g,uhd)
degree = 2*k
dΩ = GT.measure(Ω,degree)
A_alloc = GT.allocate_matrix(T,V,V,Ω)
free_or_dirichlet=(GT.FREE,GT.DIRICHLET)
Ad_alloc = GT.allocate_matrix(T,V,V,Ω;free_or_dirichlet)
b_alloc = GT.allocate_vector(T,V,Ω,Γn)
function assemble_in_Ω!(A_alloc,Ad_alloc,b_alloc,V,f,dΩ)
    tabulate = (GT.value,∇)
    compute = (GT.coordinate,)
    V_faces = GT.each_face(V,dΩ;tabulate,compute)
    n = GT.max_num_reference_dofs(V)
    T = Float64
    Auu = zeros(T,n,n)
    bu = zeros(T,n)
    for V_face in V_faces
        dofs = GT.dofs(V_face)
        fill!(Auu,zero(T))
        fill!(bu,zero(T))
        for V_point in GT.each_point(V_face)
            x = GT.coordinate(V_point)
            dV = GT.weight(V_point)
            dof_s = GT.shape_functions(GT.value,V_point)
            dof_∇s = GT.shape_functions(∇,V_point)
            for (i,dofi) in enumerate(dofs)
                v = dof_s[i]
                ∇v = dof_∇s[i]
                bu[i] += f.definition(x)*v*dV
                for (j,dofj) in enumerate(dofs)
                    ∇u = dof_∇s[j]
                    Auu[i,j] += ∇v⋅∇u*dV
                end
            end
        end
        GT.contribute!(A_alloc,Auu,dofs,dofs)
        GT.contribute!(Ad_alloc,Auu,dofs,dofs)
        GT.contribute!(b_alloc,bu,dofs)
    end
end
assemble_in_Ω!(A_alloc,Ad_alloc,b_alloc,V,f,dΩ)
A = GT.compress(A_alloc)
Ad = GT.compress(Ad_alloc)
b = GT.compress(b_alloc)
xd = GT.dirichlet_values(uhd)
b .= b .- Ad*xd
p = LinearSolve.LinearProblem(A,b)
sol = LinearSolve.solve(p)
uh = GT.solution_field(uhd,sol)
GT.makie_surfaces(Ω;color=uh)
```

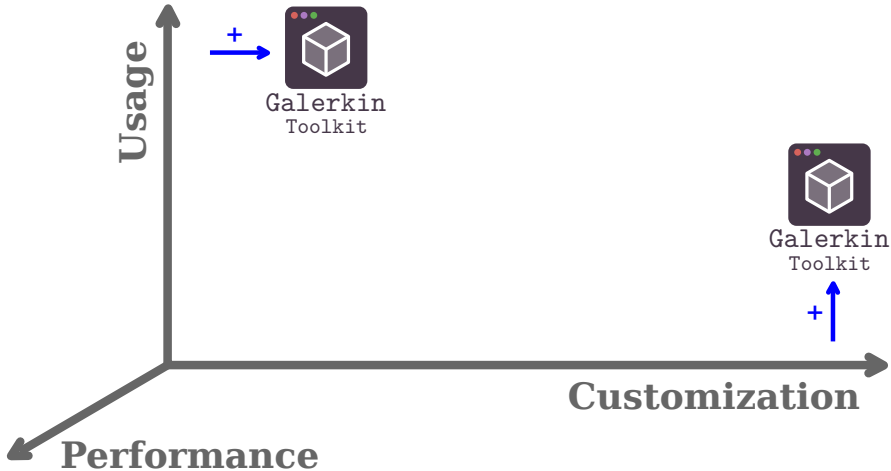
Galerkin Toolkit

# Explicit assembly loop in GalerkinToolkit

```
23 function assemble_in_Ω!(A_alloc,Ad_alloc,b_alloc,V,f,dΩ)
24     tabulate = (GT.value,∇)
25     compute = (GT.coordinate,)
26     V_faces = GT.each_face(V,dΩ;tabulate,compute)
27     n = GT.max_num_reference_dofs(V)
28     T = Float64
29     Auu = zeros(T,n,n)
30     bu = zeros(T,n)
31     for V_face in V_faces
32         dofs = GT.dofs(V_face)
33         fill!(Auu,zero(T))
34         fill!(bu,zero(T))
35         for V_point in GT.each_point(V_face)
36             x = GT.coordinate(V_point)
37             dV = GT.weight(V_point)
38             dof_s = GT.shape_functions(GT.value,V_point)
39             dof_∇s = GT.shape_functions(∇,V_point)
40             for (i,dofi) in enumerate(dofs)
41                 v = dof_s[i]
42                 ∇v = dof_∇s[i]
43                 bu[i] += f.definition(x)*v*dV
44                 for (j,dofj) in enumerate(dofs)
45                     ∇u = dof_∇s[j]
46                     Auu[i,j] += ∇v⋅∇u*dV
47                 end
48             end
49         end
50         GT.contribute!(A_alloc,Auu,dofs,dofs)
51         GT.contribute!(Ad_alloc,Auu,dofs,dofs)
52         GT.contribute!(b_alloc,bu,dofs)
53     end
54 end
```

```
26     V_faces = GT.each_face(V,dΩ;tabulate,compute)
35         for V_point in GT.each_point(V_face)
```

# Enhanced API trade-offs



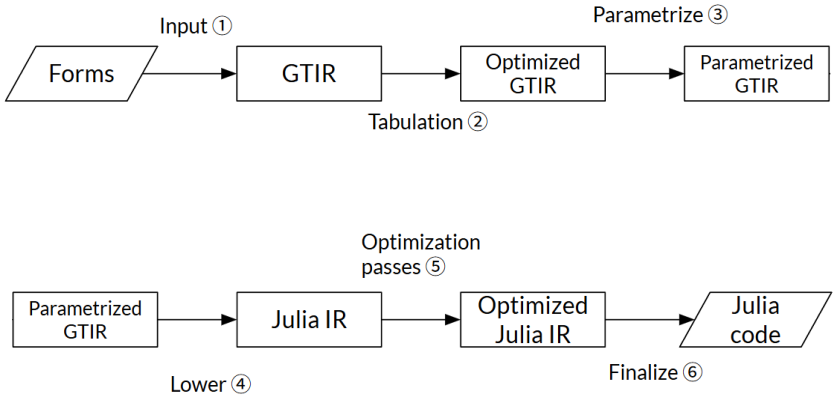
# GalerkinToolkit form compiler (GTFC)

```
GT.∫( x->∇(u,x)⋅∇(v,x), dΩ)
```

```
function assemble_in_Ω!(A_alloc,V,dΩ)
    n = GT.max_num_reference_dofs(V)
    Auu = zeros(Float64,n,n)
    for V_face in GT.each_face(V,dΩ;tabulate=(∇,))
        dofs = GT.dofs(V_face)
        fill!(Auu,zero(Float64))
        for V_point in GT.each_point(V_face)
            dV = GT.weight(V_point)
            dof_∇s = GT.shape_functions(∇,V_point)
            for (i,dofi) in enumerate(dofs)
                ∇v = dof_∇s[i]
                for (j,dofj) in enumerate(dofs)
                    ∇u = dof_∇s[j]
                    Auu[i,j] += ∇v⋅∇u*dV
                end
            end
        end
        GT.contribute!(A_alloc,Auu,dofs,dofs)
    end
end
```

* The actual generated code is slightly different

# GalerkinToolkit form compiler (GTFC)



# User-defined functions and types

Let $\sigma : \mathbb{R}^d \longrightarrow \mathbb{R}^d$ be a user-defined flux

The weak form transforms into

$$\int_\Omega \sigma(\nabla u(x)) \cdot \nabla v(x)\, dx = \int_\Omega v(x)\, dx$$

```
function σ(∇u)
    # You can implement here
    # anything that
    # returns a vector of
    # the same length as ∇u
end
```

```
GT.∫(dΩ) do x
    σ_u = GT.external(σ,∇(u,x))
    σ_u⋅∇(v,x)
end
```

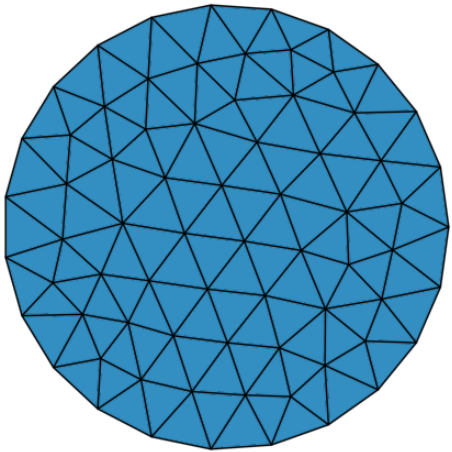
# User-defined functions and types

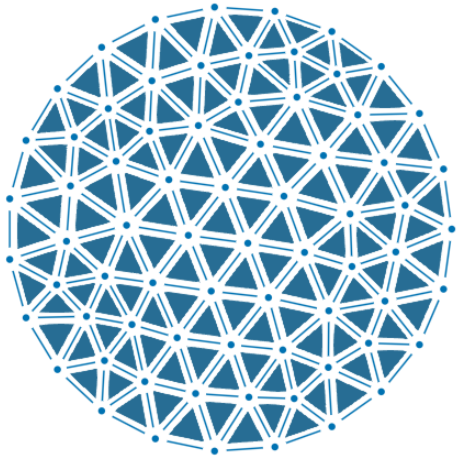
```
function assemble_in_Ω!(A_alloc,V,dΩ)
    n = GT.max_num_reference_dofs(V)
    Auu = zeros(Float64,n,n)
    σ_u = # allocate σ_u
    for V_face in GT.each_face(V,dΩ;tabulate=(∇,))
        dofs = GT.dofs(V_face)
        fill!(Auu,zero(Float64))
        for V_point in GT.each_point(V_face)
            dV = GT.weight(V_point)
            dof_∇s = GT.shape_functions(∇,V_point)
            for (j,dofj) in enumerate(dofs)
                ∇u = dof_∇s[j]
                σ_u[j] = σ(∇u)
            end
            for (i,dofi) in enumerate(dofs)
                ∇v = dof_∇s[i]
                for (j,dofj) in enumerate(dofs)
                    Auu[i,j] += σ_u[j]⋅∇u*dV
                end
            end
        end
        GT.contribute!(A_alloc,Auu,dofs,dofs)
    end
end
```

```
dof_∇s = GT.shape_functions(∇,V_point)
for (j,dofj) in enumerate(dofs)
    ∇u = dof_∇s[j]
    σ_u[j] = σ(∇u)
end
for (i,dofi) in enumerate(dofs)
    ∇v = dof_∇s[i]
    for (j,dofj) in enumerate(dofs)
        Auu[i,j] += σ_u[j]⋅∇u*dV
    end
end
```

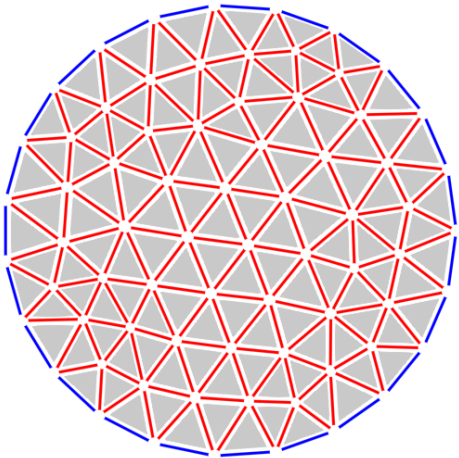
* The actual generated code is slightly different

# Polyhedral complexes



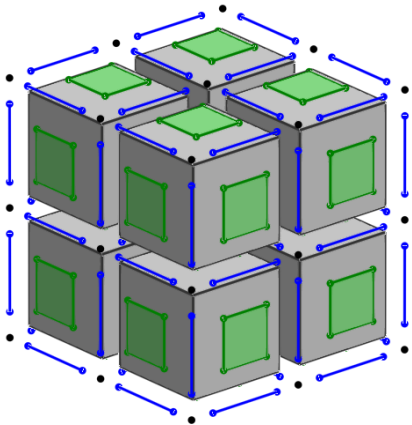


# Polyhedral complexes



```
GT.∫(x->v(x),dΩ)

GT.∫(x->v(x),dΓ)

GT.∫(x->v[1](x)-v[2](x),dΛ)

GT.∫(x->v(x)+q(x),dΓ)
```

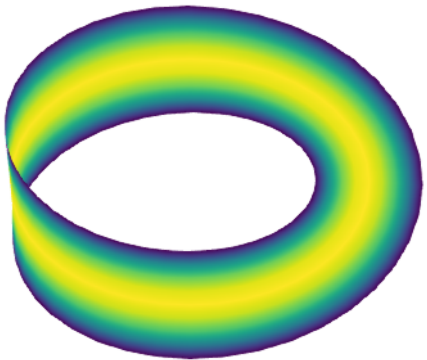
# Polyhedral complexes

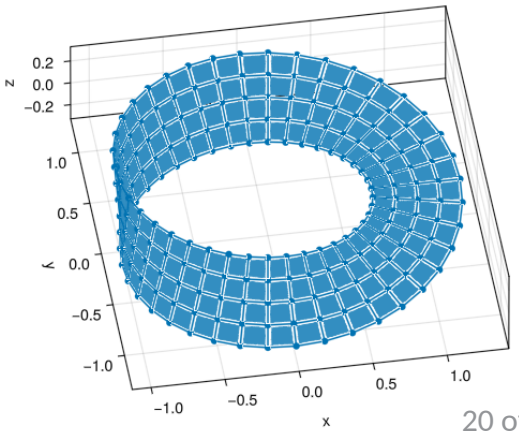


```
# 3D
shrink = 0.8
linewidth = 3
GT.makie_surfaces!(ax,mesh;shrink,dim=3,color="gray90")
GT.makie_edges!(ax,mesh;shrink,dim=3,color="gray20",linewidth)

# 2D
shrink = 0.4
markersize = 12
GT.makie_surfaces!(ax,mesh;shrink,dim=2,color="palegreen")
GT.makie_edges!(ax,mesh;shrink,dim=2,color="green",linewidth)
GT.makie_vertices!(ax,mesh;shrink,dim=2,color="green",markersize)

# 1D
shrink = 0.6
GT.makie_edges!(ax,mesh;shrink,dim=1,color="blue",linewidth)
GT.makie_vertices!(ax,mesh;shrink,dim=1,color="blue",markersize)

# 0D
GT.makie_vertices!(ax,mesh;dim=0,color="black",markersize)
```

# Manifolds

Example: Laplace-Beltrami on a Möbius strip





# Parametrizable code generation

Example: Residual of a non-linear problem ($p$-Laplacian).

$r(v) = \int_\Omega \nabla v \cdot \left( |\nabla u_h|^{p-2} \nabla u_h \right) \, \mathrm{d}\Omega$

```
uh = # Define your FE field
flux(∇u) = norm(∇u)^(p-2) * ∇u
r = v -> GT.∫( x-> ∇(v,x)⋅GT.external(flux,∇(uh,x)), dΩ)
# Assembly (code gets generated here)
b,cache = GT.assemble_vector(r,Float64,V;parameters=(uh,))


uh2 = # Another FE field
# Reuse the code generated above
GT.update_vector!(b,cache;parameters=(uh2,))
```

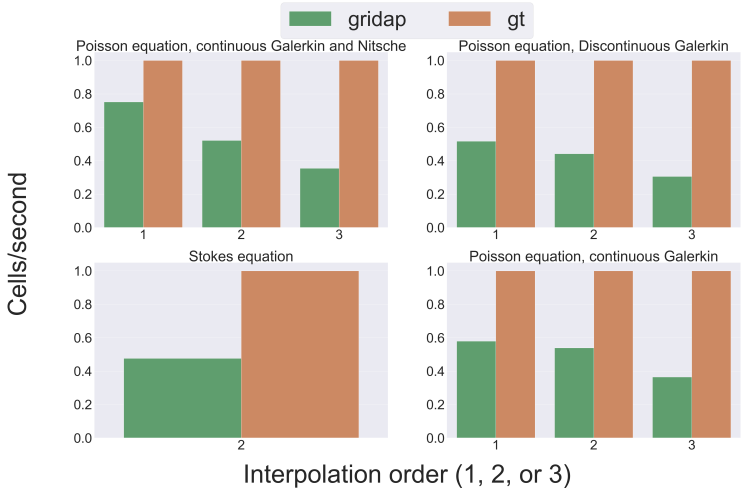
# Building SciML problems

```
# Linear problem
a = (u,v) -> GT.∫( x->∇(u,x)⋅∇(v,x), dΩ)
l = v -> GT.∫( x->v(x)*f(x), dΩ)
p = GT.SciMLBase_LinearProblem(uhd,a,l)
sol = LinearSolve.solve(p)
uh = GT.solution_field(uhd,sol)

# Nonlinear problem
r = u -> v -> GT.∫( x-> ∇(v,x)⋅GT.external(flux,∇(u,x)),dΩ)
j = u -> (du,v) -> GT.∫( x-> ∇(v,x)⋅GT.external(dflux,∇(du,x),∇(u,x)),dΩ)
p = GT.SciMLBase_NonlinearProblem(uh,r,j)
sol = NonlinearSolve.solve(p)
uh = GT.solution_field(uh,sol)
```

# Building SciML problems

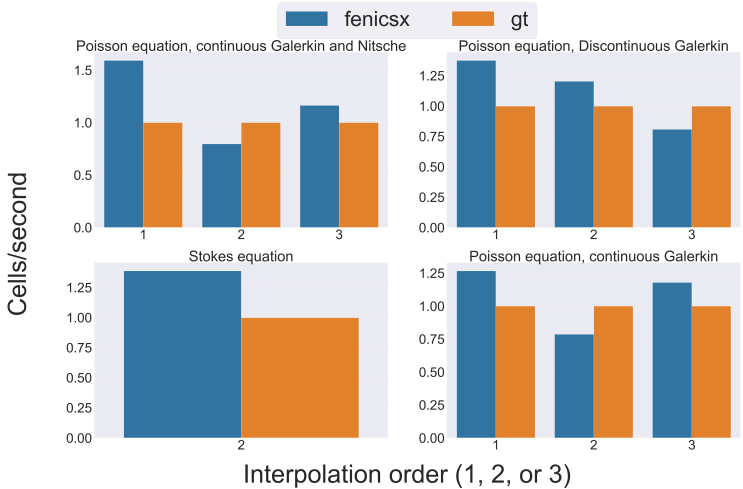
```
# ODE Problem
m = (u,v) -> GT.∫(x->C*v(x)*u(x),dΩ)
r = (uh,t) -> v -> -1*GT.∫(x->∇(uh,x)⋅∇(v,x),dΩ)
j = (uh,t) -> (u,v) -> -1*GT.∫(x->∇(u,x)⋅∇(v,x),dΩ)
tspan = (0.0,T)
problem = GT.SciMLBase_ODEProblem(tspan,uh,m,r,j;dirichlet_dynamics!)
integrator = DifferentialEquations.init(problem)
for integrator in integrator
    uh = GT.solution_field(integrator)
end
```

# Performance: Gridap vs GalerkinToolkit




Higher is better. Results for unstructured hexahedal meshes. Internal git branch.

# Performance: FEniCSx vs GalerkinToolkit




Higher is better. Results for unstructured hexahedal meshes. Internal git branch.

# Porting low-level API to GPUs

Example: Integrate a user-defined function on the CPU

$$\int_\Omega f(x)\,dx$$

```
Ω = # define domain here
f(x) = 2*sin(sum(x))

degree = 5
dΩ = GT.quadrature(Ω,degree)
s = 0.0
for dΩ_face in GT.each_face(dΩ)
    for dΩ_point in GT.each_point(dΩ_face)
        x = GT.coordinate(dΩ_point)
        dx = GT.weight(dΩ_point)
        s += f(x)*dx
    end
end
```

# Porting low-level API to GPUs

Example: Integrate a user-defined function on the GPU

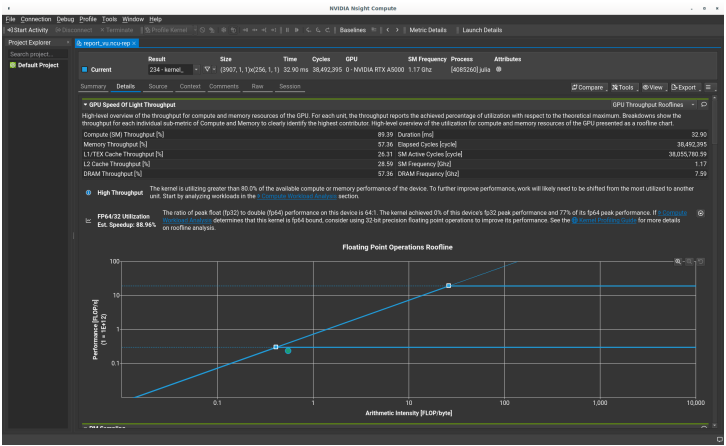
```
function kernel!(contributions,dΩ_faces_gpu)
    face_id = (blockIdx().x-1)*blockDim().x +
              threadIdx().x
    if face_id > length(dΩ_faces_gpu)
        return nothing
    end
    dΩ_face = dΩ_faces_gpu[face_id]
    s = 0.0
    for dΩ_point in GT.each_point(dΩ_face)
        x = GT.coordinate(dΩ_point)
        dx = GT.weight(dΩ_point)
        s += f(x)*dx
    end
    contributions[face_id] = s
    return nothing
end
```

```
dΩ_faces_cpu = GT.each_face(dΩ)
dΩ_faces_gpu = CUDA.cu(dΩ_faces_cpu)
nfaces = length(dΩ_faces_gpu)
contributions = CUDA.zeros(Float64,nfaces)
threads_in_block = 256
blocks_in_grid = ceil(Int, nfaces/256)
@cuda threads=threads_in_block
      blocks=blocks_in_grid
      kernel!(contributions,dΩ_faces_gpu)
s = sum(contributions)
```

Internal git branch.

# Porting low-level API to GPUs

Example: Integrate a user-defined function on the GPU





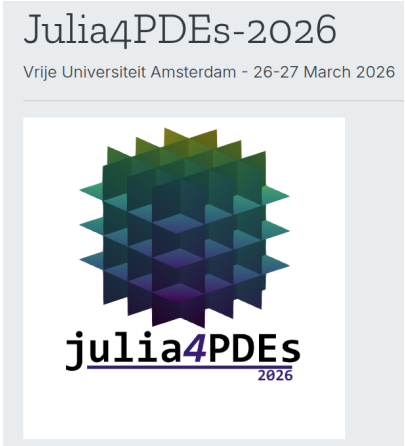

Internal git branch.

# Summary

GalerkinToolkit

- Multi-purpose
- Modern
- Easy to use / customizable / performant
- Inter-operable
- Towards multi-platform
- It is FOSS!

Registration and call for abstract are open!




netherlands eScience center

Grant ID: NLESC.SS.2023.008

Co-organized with O. Colomes, A. Palha, R. Richardson, and D. Toshniwal