UNIVERSITY of HOUSTON

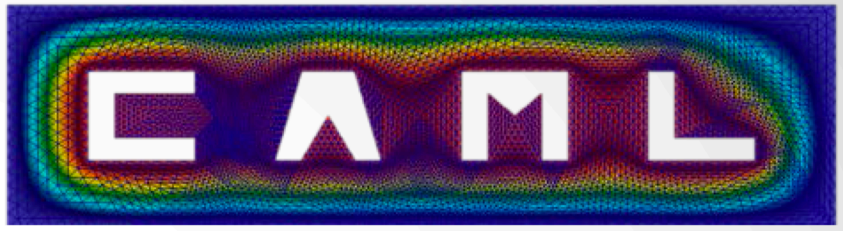
CENTER FOR ADVANCED COMPUTING & DATA SCIENCE

FEniCS workshop

June 2018

Mohammad Sarraf Joshaghani
CACDS Fellow
Graduate student @CAML
Advisor: Prof. Kalyana Nakshatrala

Computational & Applied Mechanics Laboratory



Discover through advance computing and mathematical modeling

FENICS PROJECT

What is FEniCS?

FEniCS is an automated programming environment for differential equations

- C++/Python library
- Initiated 2003 in Chicago
- 1000–2000 monthly downloads
- Part of Debian and Ubuntu
- Licensed under the GNU LGPL

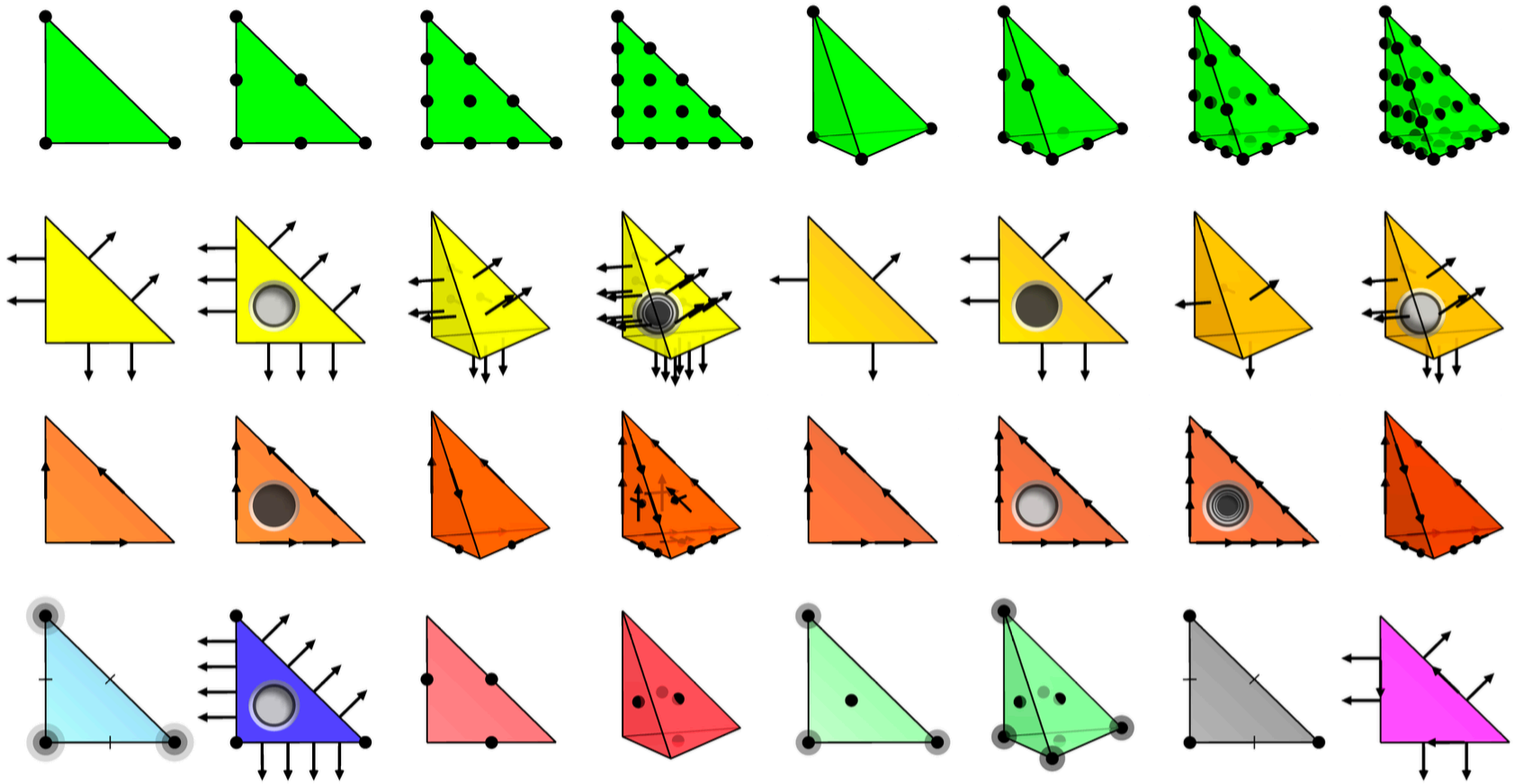
`http://fenicsproject.org/`

Collaborators

Simula Research Laboratory, University of Cambridge, University of Chicago, Texas Tech University, KTH Royal Institute of Technology, Chalmers University of Technology, Imperial College London, University of Oxford, Charles University in Prague, ...

FEniCS is automated FEM

- Automated generation of basis functions
- Automated evaluation of variational forms
- Automated finite element assembly
- Automated adaptive error control



Hello World in FEniCS: problem formulation

Poisson's equation

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega \end{aligned}$$

Finite element formulation

Find $u \in V$ such that

$$\underbrace{\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x}_{a(u,v)} = \underbrace{\int_\Omega f \, v \, \mathrm{d}x}_{L(v)} \quad \forall \, v \in V$$

Hello World in FEniCS: implementation

```
from fenics import *

mesh = UnitSquareMesh(32, 32)

V = FunctionSpace(mesh, "Lagrange", 1)
u = TrialFunction(V)
v = TestFunction(V)
f = Expression("x[0]*x[1]", degree=2)

a = dot(grad(u), grad(v))*dx
L = f*v*dx

bc = DirichletBC(V, 0.0, DomainBoundary())

u = Function(V)
solve(a == L, u, bc)
plot(u)
```

Basic API

- `Mesh`, `Vertex`, `Edge`, `Face`, `Facet`, `Cell`
- `FiniteElement`, `FunctionSpace`
- `TrialFunction`, `TestFunction`, `Function`
- `grad()`, `curl()`, `div()`, ...
- `Matrix`, `Vector`, `KrylovSolver`, `LUSolver`
- `assemble()`, `solve()`, `plot()`

- Python interface generated semi-automatically by SWIG
- C++ and Python interfaces almost identical

Three survival advices

Use the right Python tools

Explore the documentation

Ask, report and request

Use the right Python tools!

Python tools

Doc tools

- Standard terminal:
  ```
  > pydoc dolfin
  > pydoc dolfin.Mesh
  ```
- Python console
  ```
  >>> help(dolfin)
  >>> help(dolfin.Mesh)
  ```

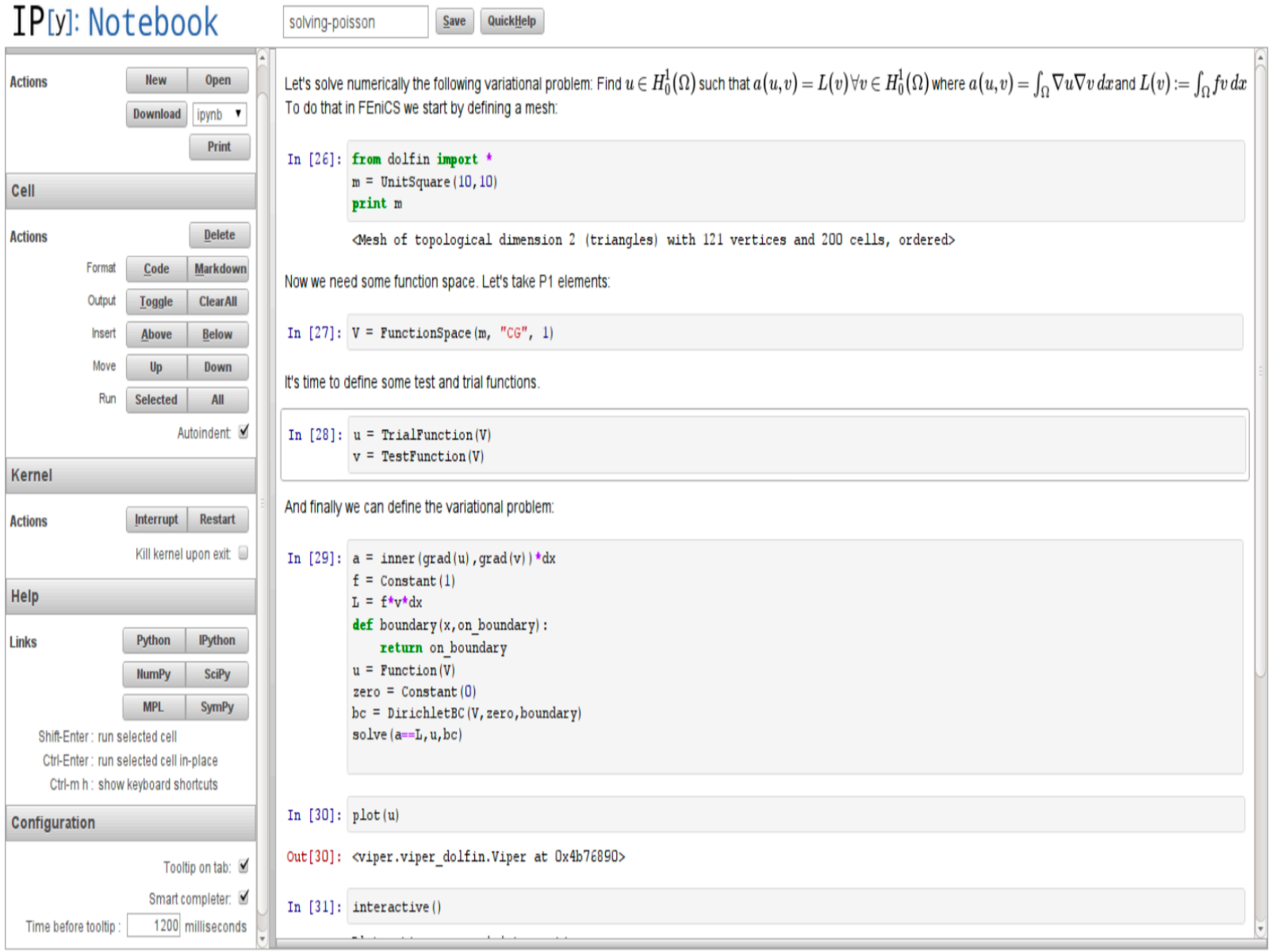
Sophisticated Python environments

IDLE the official (but rather limited) Python IDE

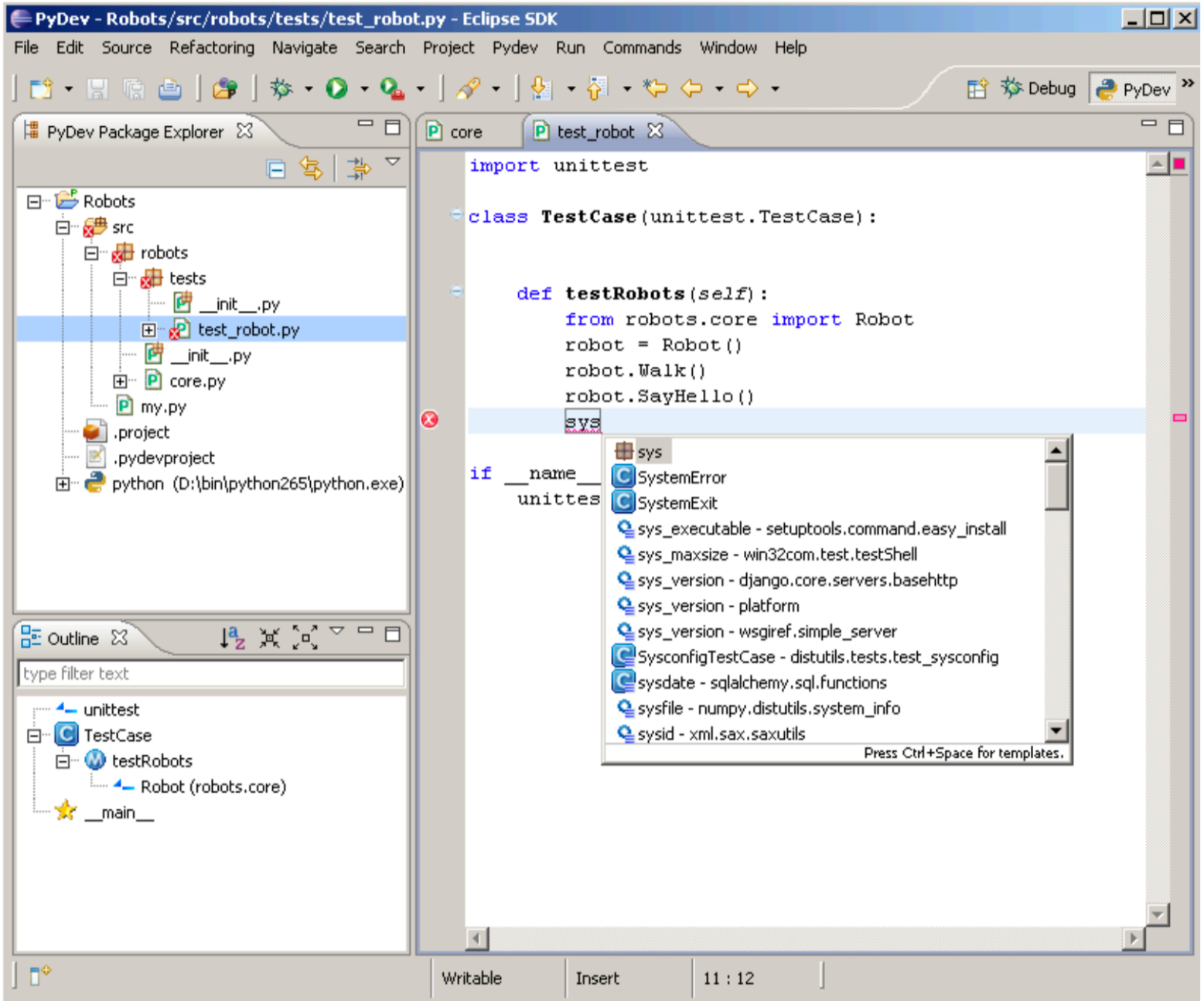
IPython `http://ipython.org/` provides a Python shell and notebook including syntax highlighting, tab-completion, object inspection, debug assisting, history ...

Eclipse plugin `http://pydev.org/` includes syntax highlighting, code completion, unit-testing, refactoring, debugger ...

IPython/Jupyter Notebook



Eclipse plugin Pydev



Explore the FEniCS documentation!

FEniCS project

About Download Documentation Applications Contributing Citing Support

Documentation for FEniCS 1.3.0

Our documentation includes a book, a collection of documented demo programs, and complete references for the FEniCS application programming interface (API). Note that the FEniCS API is documented separately for each FEniCS component. The most important interfaces are those of the C++/Python problem solving environment *DOLFIN* and the form language *UFL*.

(*This page accesses the FEniCS 1.3.0 documentation. Not the version you are looking for? See all versions.*)

The FEniCS Tutorial

A good starting point for new users is the *FEniCS Tutorial*. The tutorial will help you get quickly up and running with solving differential equations in FEniCS. The tutorial focuses exclusively on the FEniCS Python interface, since this is the simplest approach to exploring FEniCS for beginners.

The FEniCS Book

The FEniCS Book, *Automated Solution of Differential Equations by the Finite Element Method*, is a comprehensive (700 pages) book documenting the mathematical methodology behind the FEniCS Project and the software developed as part of the FEniCS Project. The FEniCS Tutorial is included as the opening chapter of the FEniCS Book.

The FEniCS Manual

The FEniCS Manual is a 200-page excerpt from the FEniCS Book, including the FEniCS Tutorial, an introduction to the finite element method and documentation of DOLFIN and UFL.

Additional Documentation

Mixing software with FEniCS is a tutorial on how to combine FEniCS applications in Python with software written in other languages.

Demos

A simple way to build your first FEniCS application is to copy and modify one of the existing demos:

- Documented DOLFIN demos (Python)
- Documented DOLFIN demos (C++)

The demos are *already installed on your system* or can be found in the `demo` directory of the DOLFIN source tree.

Quick Programmer's References

Some of the classes and functions in DOLFIN are more frequently used than others. To learn more about these, take a look at the

- Basic classes and functions in DOLFIN (Python)
- Basic classes and functions in DOLFIN (C++)

Complete Programmer's References

- All classes and functions in DOLFIN (Python)
- All classes and functions in DOLFIN (C++)
- All classes and functions in UFL

http://fenicsproject.org/documentation/

Documentation for FEniCS 1.3.0

Our documentation includes a book, a collection of documented demo programs, and complete references for the FEniCS application programming interface (API). Note that the FEniCS API is documented separately for each FEniCS component. The most important interfaces are those of the C++/Python problem solving environment *DOLFIN* and the form language *UFL*.

(*This page accesses the FEniCS 1.3.0 documentation. Not the version you are looking for? See all versions.*)

The FEniCS Tutorial

A good starting point for new users is the *FEniCS Tutorial*. The tutorial will help you get quickly up and running with solving differential equations in FEniCS. The tutorial focuses exclusively on the FEniCS Python interface, since this is the simplest approach to exploring FEniCS for beginners.

The FEniCS Book

The FEniCS Book, *Automated Solution of Differential Equations by the Finite Element Method*, is a comprehensive (700 pages) book documenting the mathematical methodology behind the FEniCS Project and the software developed as part of the FEniCS Project. The FEniCS Tutorial is included as the opening chapter of the FEniCS Book.

The FEniCS Manual

The FEniCS Manual is a 200-page excerpt from the FEniCS Book, including the FEniCS Tutorial, an introduction to the finite element method and documentation of DOLFIN and UFL.

Additional Documentation

Mixing software with FEniCS is a tutorial on how to combine FEniCS applications in Python with software written in other languages.

Demos

A simple way to build your first FEniCS application is to copy and modify one of the existing demos:

- Documented DOLFIN demos (Python)
- Documented DOLFIN demos (C++)

The demos are *already installed on your system* or can be found in the `demo` directory of the DOLFIN source tree.

Quick Programmer's References

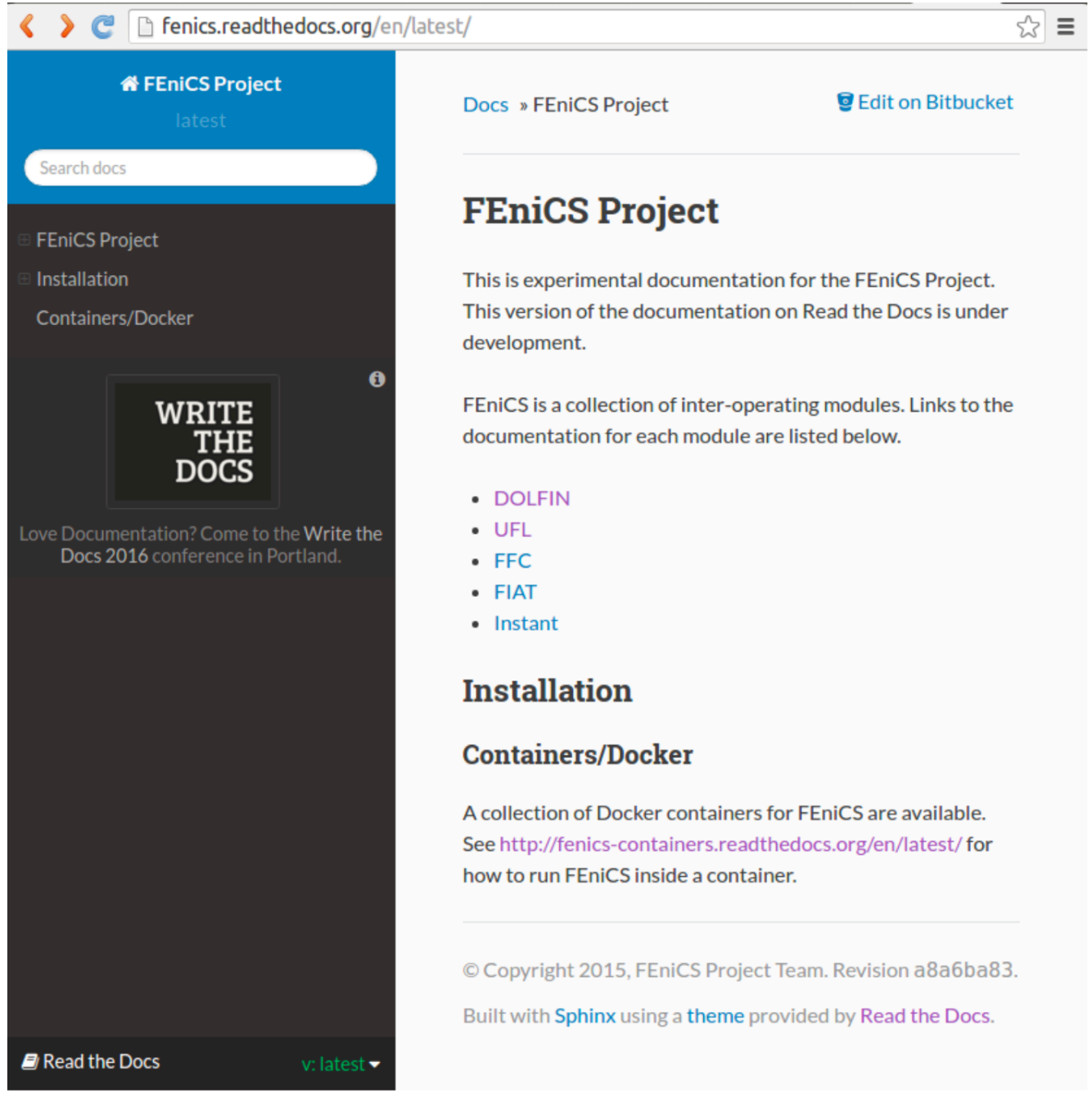
Some of the classes and functions in DOLFIN are more frequently used than others. To learn more about these, take a look at the

- Basic classes and functions in DOLFIN (Python)
- Basic classes and functions in DOLFIN (C++)

Complete Programmer's References

- All classes and functions in DOLFIN (Python)
- All classes and functions in DOLFIN (C++)
- All classes and functions in UFL

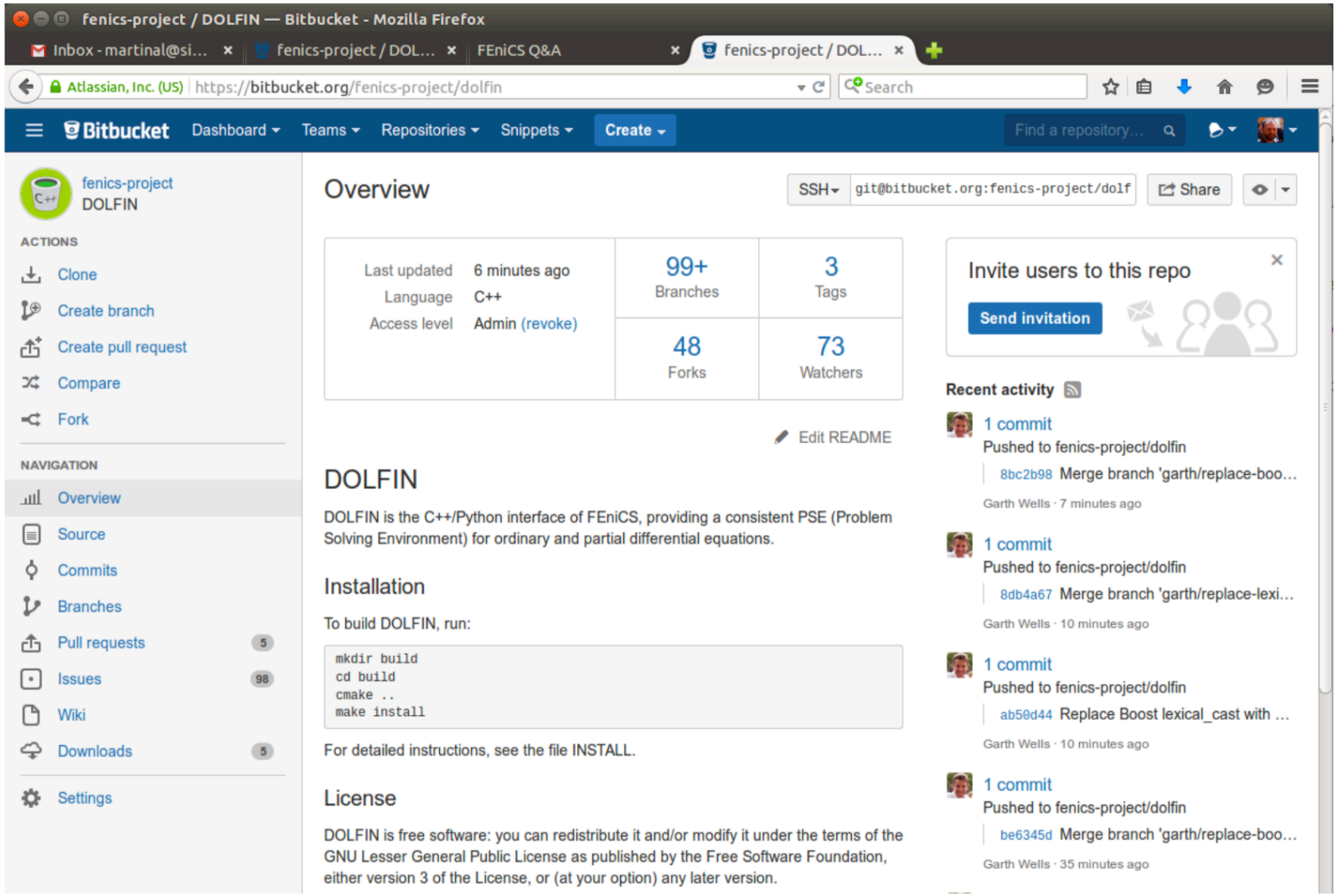
http://fenicsproject.org/documentation/



http://fenics.readthedocs.org/

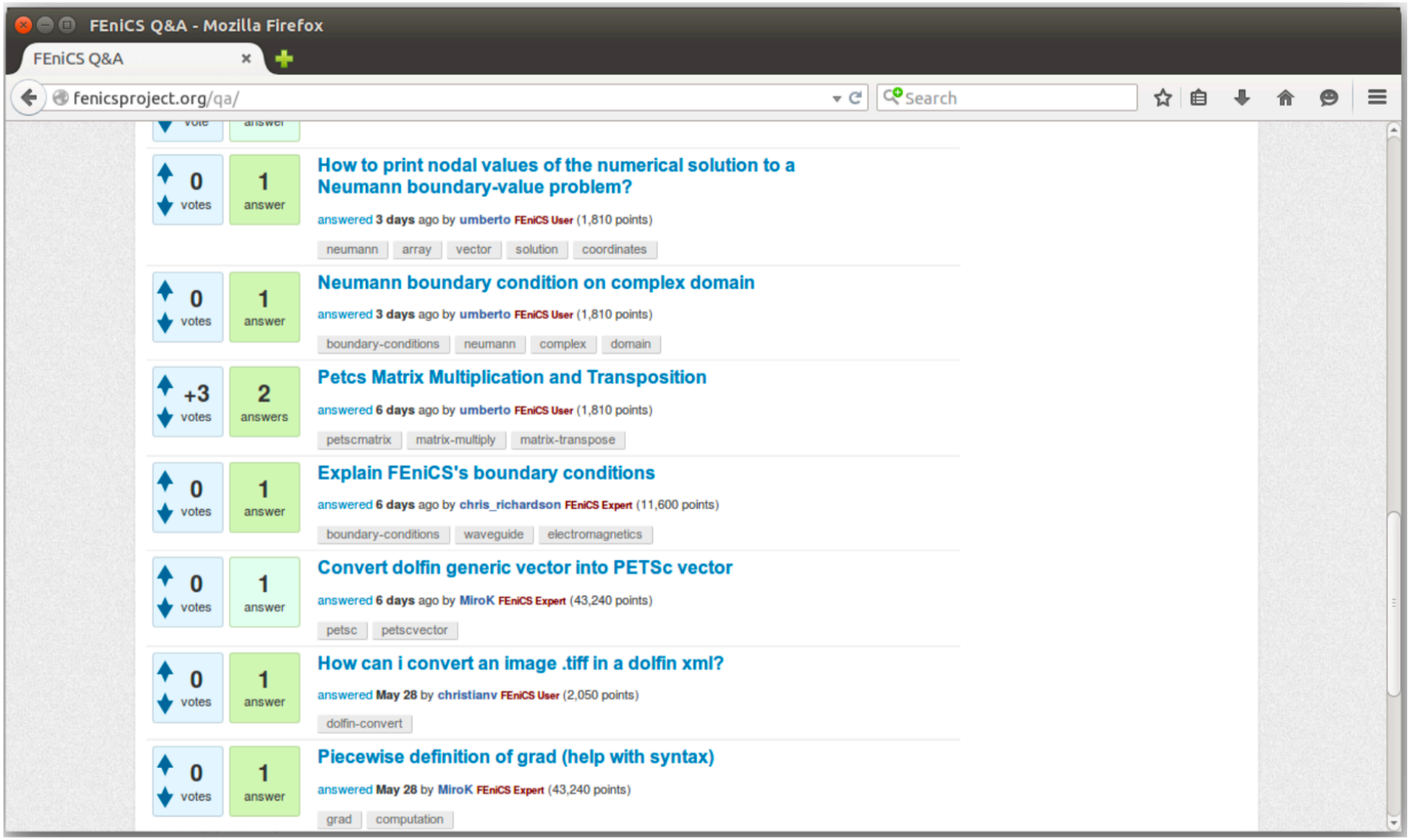
Ask questions, report bugs and request new features!

Development community is organized via bitbucket.org



http://bitbucket.org/fenics-project/

Community help is available via QA forum



https://fenicsproject.org/qa

Let’s start

Installation alternatives

☞ Docker images on Linux, Mac, Windows

☞ Build from source with Hashdist (fenics-install.sh)

☞ PPA with apt packages for Debian and Ubuntu

☞ Drag and drop installation on Mac OS X

`http://fenicsproject.org/download/`

Installation using Docker

Follow instructions to install Docker on linux, mac, or windows:

https://docs.docker.com/linux/ or `mac/`, `windows/`

Download and open a terminal in a clean FEniCS environment:

Bash code

```
$ docker run -ti quay.io/fenicsproject/dev
```

More instructions on using FEniCS Docker images here:

http://fenics-containers.readthedocs.org

TASK

Step 1. Sharing files from the host (your computer) into the docker container

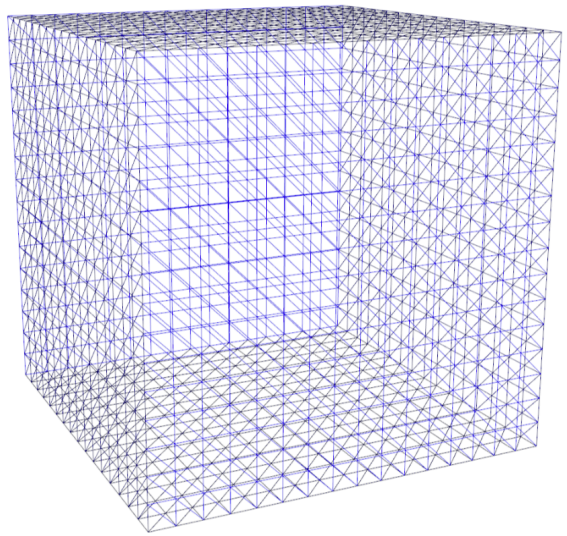
```
docker run -ti -v $(pwd):/home/fenics/shared quay.io/fenicsproject/stable
```

Step 2. Use an editor and generate the following < >.py file.

```
from fenics import *

mesh = UnitCubeMesh(16, 16, 16)
plot(mesh)
interactive()
```

Step 3. run the code



How to set up Jupyter?

1- First of all we run a new Docker container with the jupyter-notebook

```
docker run --name notebook -w /home/fenics -v $(pwd):/home/fenics/shared -d -p
127.0.0.1:8888:8888 quay.io/fenicsproject/stable 'jupyter-notebook --ip=0.0.0.0'
```

2- Figure out what is your container ID

```
docker ps
```

3- Find out the access token

```
docker logs <cointainer ID>
```

4- Paste log-in token in browser

5- For visualization, import pylab

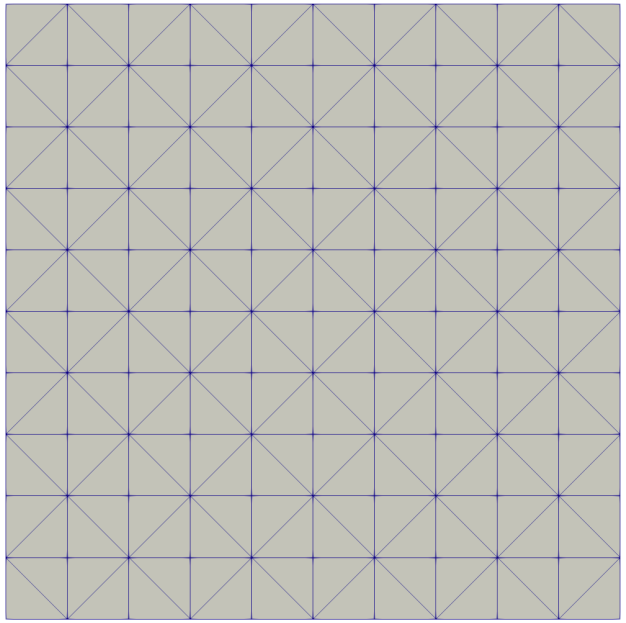
```
import pylab
%matplotlib inline
parameters["plotting_backend"] = "matplotlib"
```

The solution and the mesh may be plotted by simply calling:

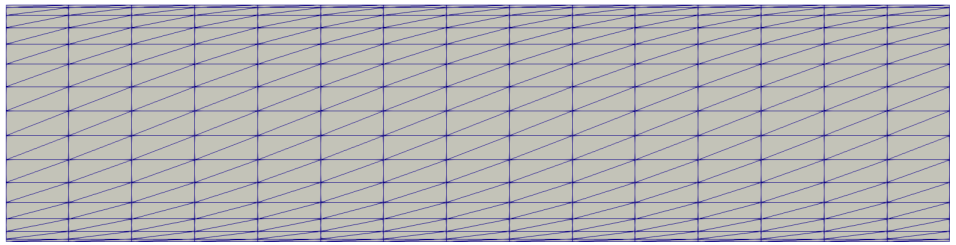
```
plot(u)
pylab.show()
plot(mesh)
pylab.show()
```

TASK Mesh generation

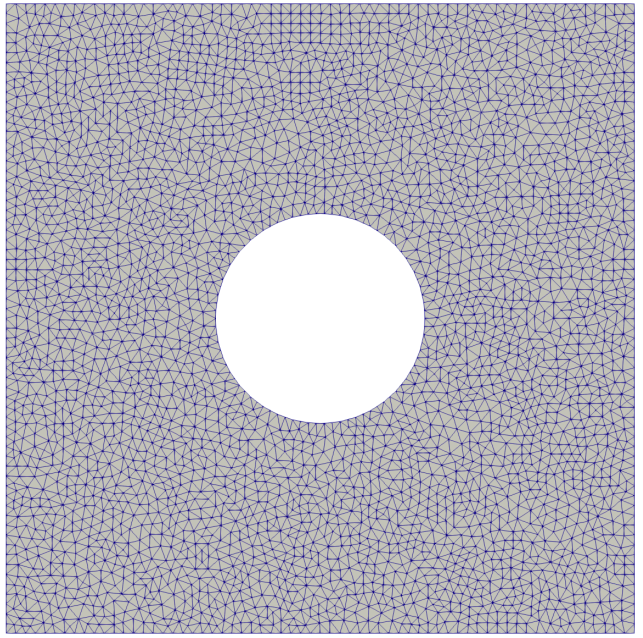
Ex. Write a python code to generate the following domains and meshes:



Mesh 1



Mesh 2



Mesh 3

```
git clone https://github.com/msarrafj/FEniCS.git
```

What is Finite element!

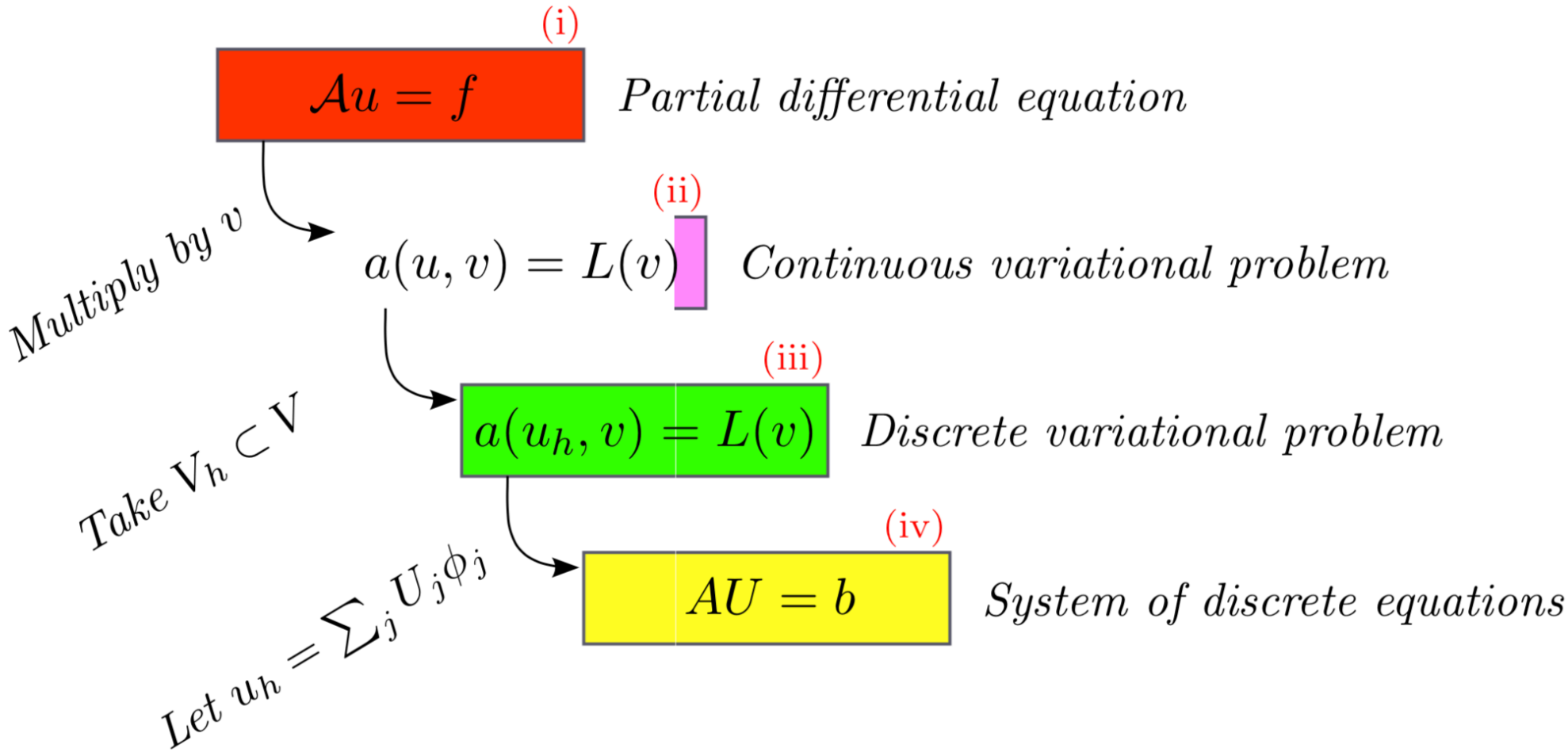
What is FEM?

The finite element method is a framework and a recipe for discretization of differential equations

- Ordinary differential equations
- Partial differential equations
- Integral equations

- A recipe for discretization of PDE
- PDE $\rightarrow Ax = b$
- Different bases, stabilization, error control, adaptivity

The FEM cookbook



The PDE (i)

Consider Poisson's equation, the Hello World of partial differential equations:

$$\begin{aligned} -\Delta u &= f \quad && \text{in } \Omega \\ u &= u_0 && \text{on } \partial\Omega \end{aligned}$$

Poisson's equation arises in numerous applications:

- heat conduction, electrostatics, diffusion of substances, twisting of elastic rods, inviscid fluid flow, water waves, magnetostatics, ...
- as part of numerical splitting strategies for more complicated systems of PDEs, in particular the Navier–Stokes equations

From PDE (i) to variational problem (ii)

The simple recipe is: multiply the PDE by a test function v and integrate over Ω:

$$-\int_\Omega (\Delta u) v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x$$

Then integrate by parts and set $v = 0$ on the Dirichlet boundary:

$$-\int_\Omega (\Delta u) v \,\mathrm{d}x = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \underbrace{\int_{\partial\Omega} \frac{\partial u}{\partial n} v \,\mathrm{d}s}_{=0}$$

We find that:

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x$$

The variational problem (ii)

Find $u \in V$ such that

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x$$

for all $v \in \hat{V}$

The trial space V and the test space $\hat{V}$ are (here) given by

$$V = \{v \in H^1(\Omega) : v = u_0 \text{ on } \partial\Omega\}$$
$$\hat{V} = \{v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega\}$$

From continuous (ii) to discrete (iii) problem

We approximate the continuous variational problem with a discrete variational problem posed on finite dimensional subspaces of V and $\hat{V}$:

$$V_h \subset V$$
$$\hat{V}_h \subset \hat{V}$$

Find $u_h \in V_h \subset V$ such that

$$\int_\Omega \nabla u_h \cdot \nabla v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x$$

for all $v \in \hat{V}_h \subset \hat{V}$

From discrete variational problem (iii) to discrete system of equations (iv)

Choose a basis for the discrete function space:

$$V_h = \text{span}\,\{\phi_j\}_{j=1}^{N}$$

Make an ansatz for the discrete solution:

$$u_h = \sum_{j=1}^{N} U_j \phi_j$$

Test against the basis functions:

$$\int_\Omega \nabla(\underbrace{\sum_{j=1}^{N} U_j \phi_j}_{u_h}) \cdot \nabla \phi_i \,\mathrm{d}x = \int_\Omega f \phi_i \,\mathrm{d}x$$

From discrete variational problem (iii) to discrete system of equations (iv), contd.

Rearrange to get:

$$\sum_{j=1}^{N} U_j \underbrace{\int_\Omega \nabla\phi_j \cdot \nabla\phi_i \,\mathrm{d}x}_{A_{ij}} = \underbrace{\int_\Omega f\phi_i \,\mathrm{d}x}_{b_i}$$

A linear system of equations:

$$AU = b$$

where

$$A_{ij} = \int_\Omega \nabla\phi_j \cdot \nabla\phi_i \,\mathrm{d}x \tag{1}$$

$$b_i = \int_\Omega f\phi_i \,\mathrm{d}x \tag{2}$$

The canonical abstract problem

(i) Partial differential equation:

$$\mathcal{A}u = f \quad \text{in } \Omega$$

(ii) Continuous variational problem: find $u \in V$ such that

$$a(u, v) = L(v) \quad \text{for all } v \in \hat{V}$$

(iii) Discrete variational problem: find $u_h \in V_h \subset V$ such that

$$a(u_h, v) = L(v) \quad \text{for all } v \in \hat{V}_h$$

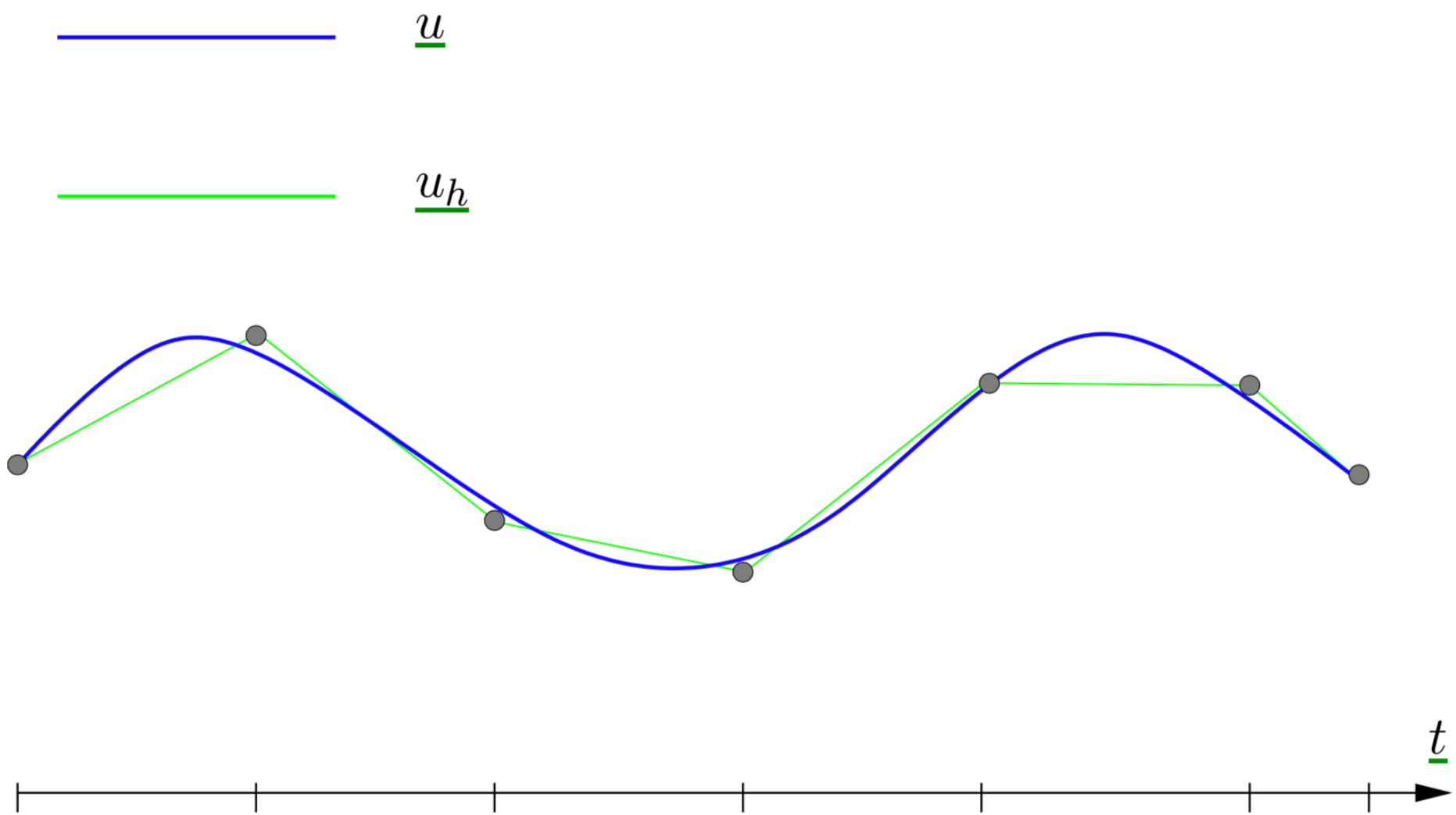
(iv) Discrete system of equations for $u_h = \sum_{j=1}^{N} U_j \phi_j$:

$$\begin{aligned} AU &= b \\ A_{ij} &= a(\phi_j, \phi_i) \\ b_i &= L(\phi_i) \end{aligned}$$

Important topics

- *How to choose* V_h*?*
- *How to compute* A *and* b
- *How to solve* $AU = b$*?*
- *How large is the error* $e = u - u_h$*?*
- Extensions to nonlinear problems

Finite element function spaces



The finite element definition (Ciarlet 1975)

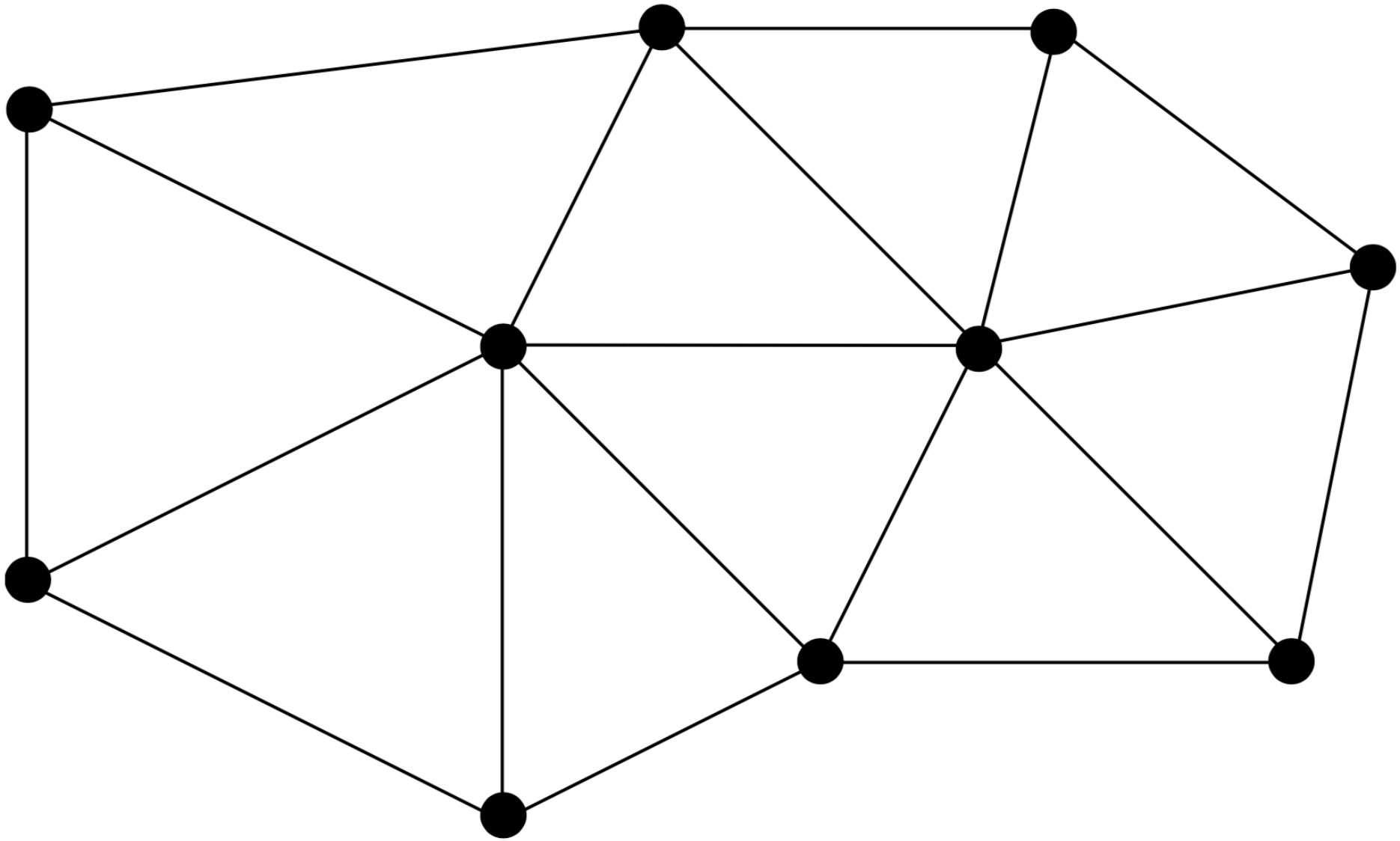
A finite element is a triple $(T, \mathcal{V}, \mathcal{L})$, where

- the domain T is a bounded, closed subset of $\mathbb{R}^d$ (for $d = 1, 2, 3, \dots$) with nonempty interior and piecewise smooth boundary
- the space $\mathcal{V} = \mathcal{V}(T)$ is a finite dimensional function space on T of dimension n
- the set of degrees of freedom (nodes) $\mathcal{L} = \{\ell_1, \ell_2, \dots, \ell_n\}$ is a basis for the dual space $\mathcal{V}'$; that is, the space of bounded linear functionals on $\mathcal{V}$

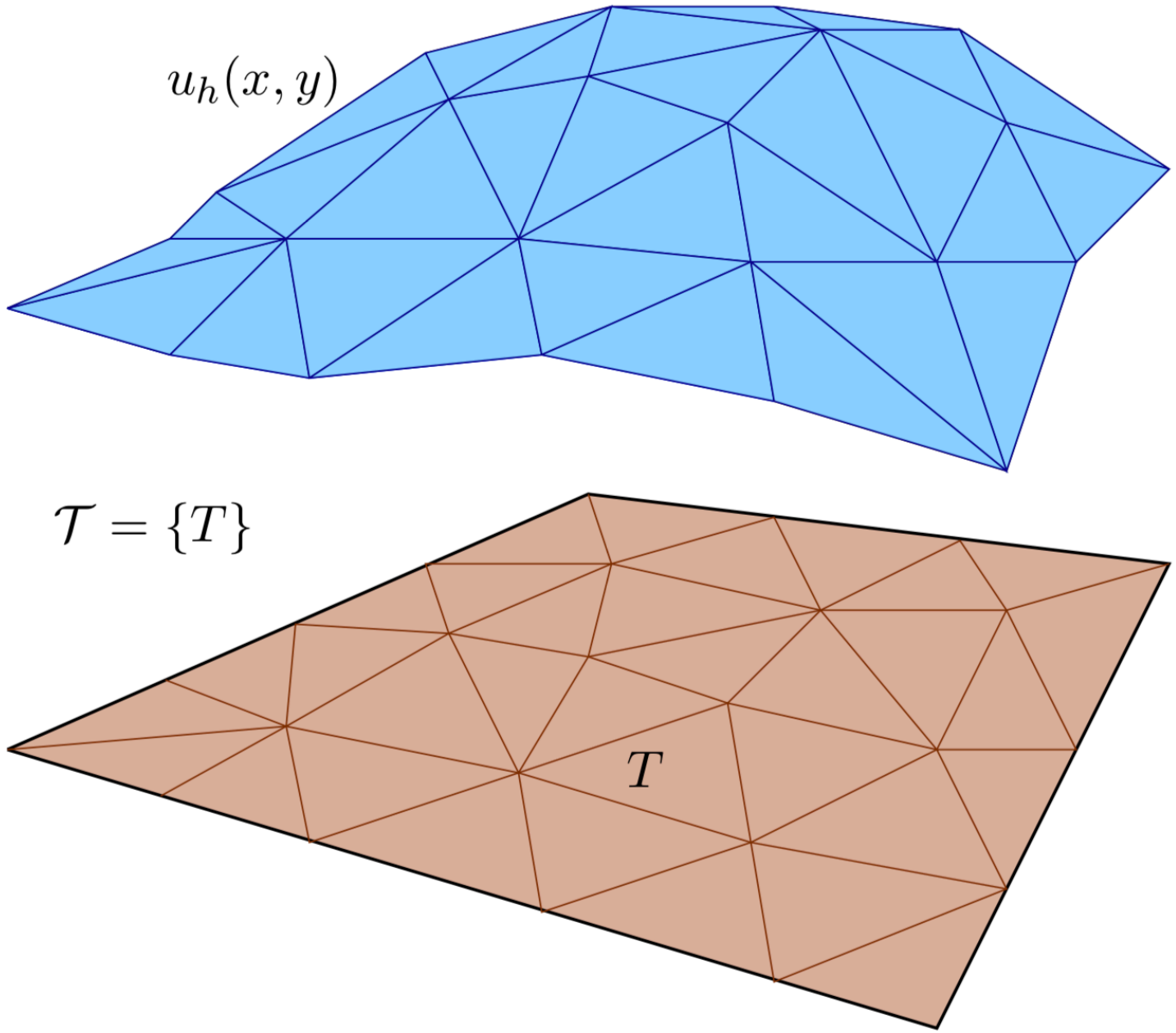
The linear Lagrange element: $(T, \mathcal{V}, \mathcal{L})$

- T is a line, triangle or tetrahedron
- $\mathcal{V}$ is the first-degree polynomials on T
- $\mathcal{L}$ is point evaluation at the vertices

The linear Lagrange element: $\mathcal{L}$



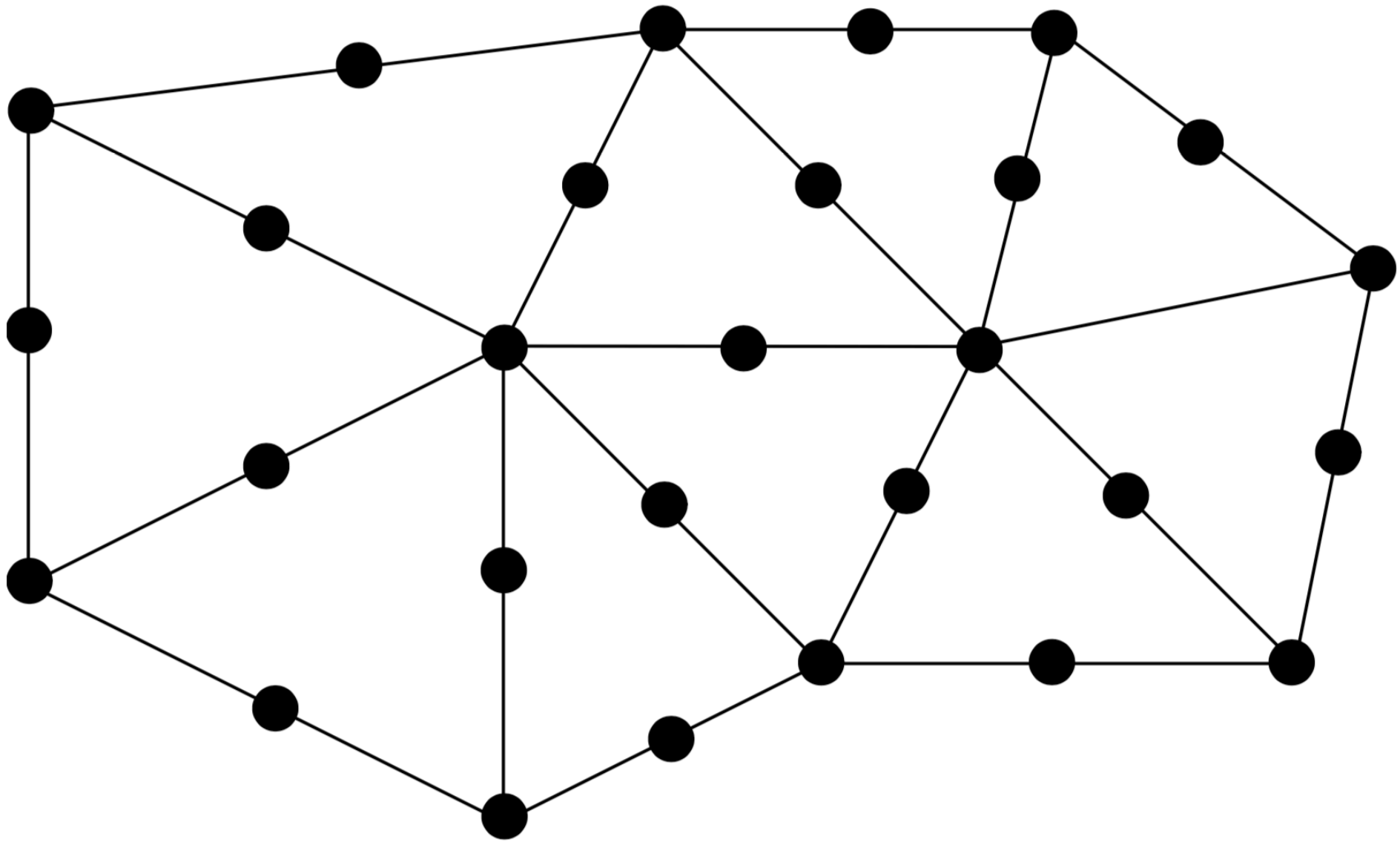
The linear Lagrange element: V_h



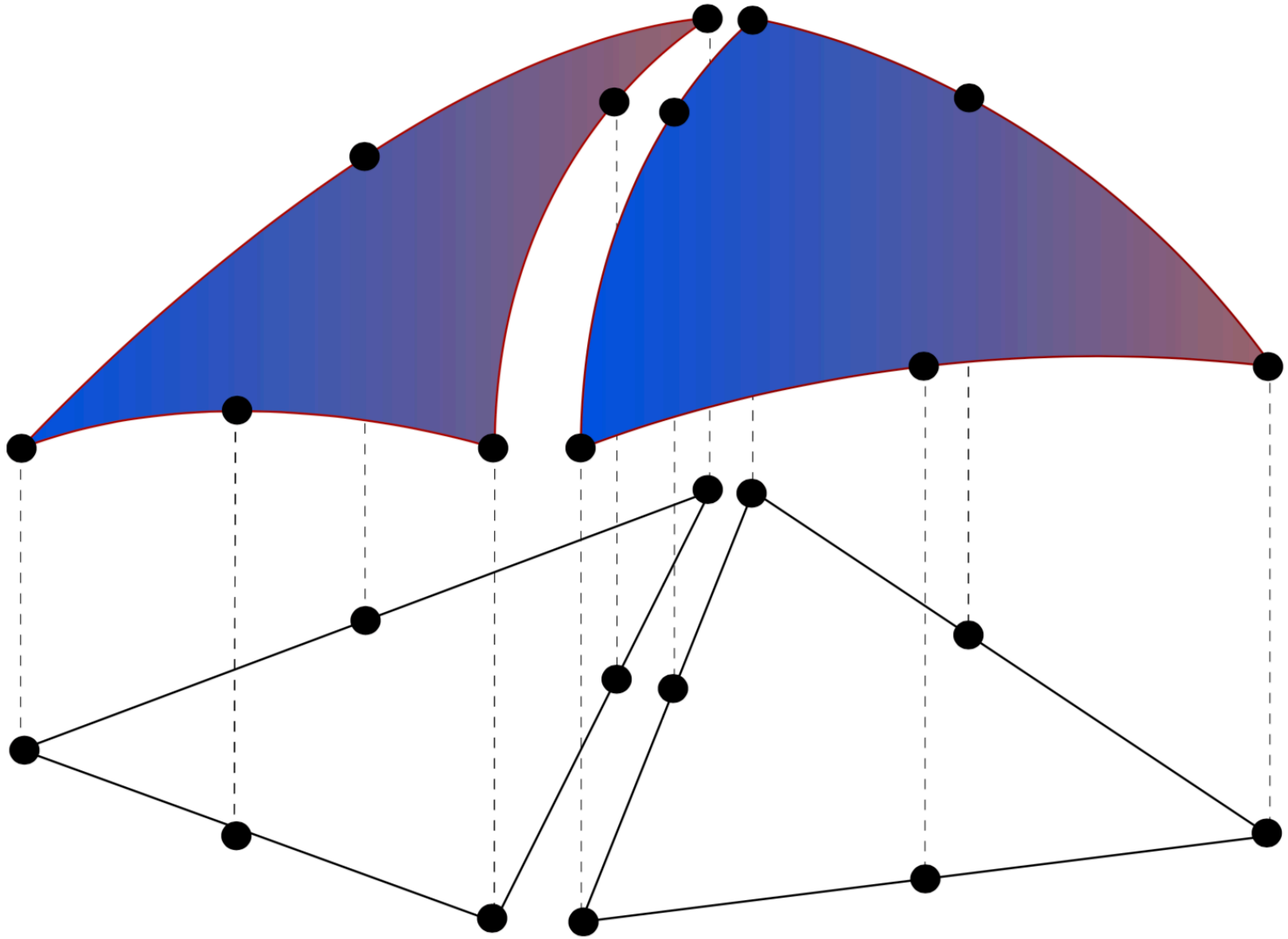
The quadratic Lagrange element: $(T, \mathcal{V}, \mathcal{L})$

- T is a line, triangle or tetrahedron
- $\mathcal{V}$ is the second-degree polynomials on T
- $\mathcal{L}$ is point evaluation at the vertices and edge midpoints

The quadratic Lagrange element: $\mathcal{L}$

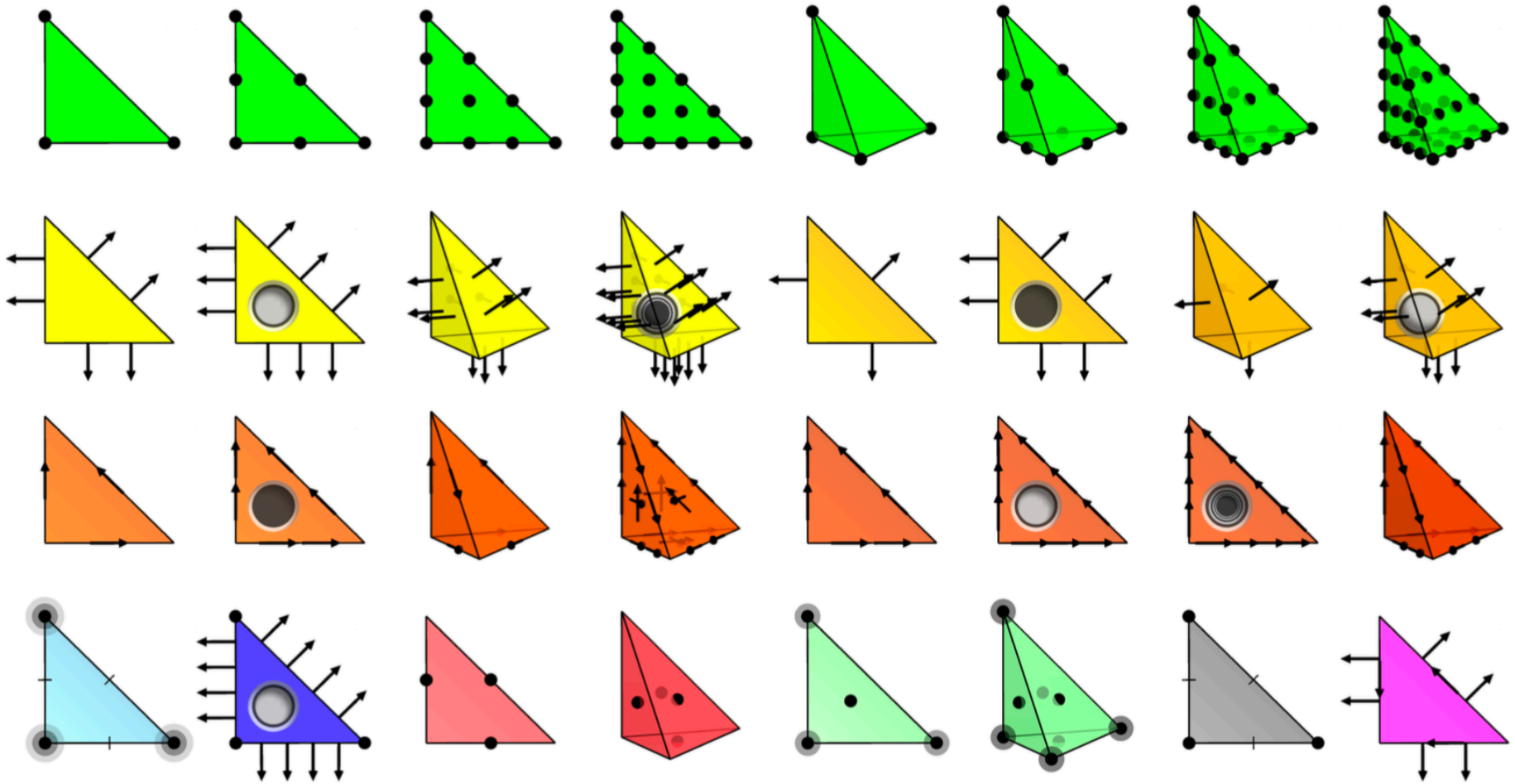


The quadratic Lagrange element: V_h



Families of elements

Families of elements



The element matrix

The global matrix A is defined by

$$A_{ij} = a(\phi_j, \phi_i)$$

The *element matrix* A_T is defined by

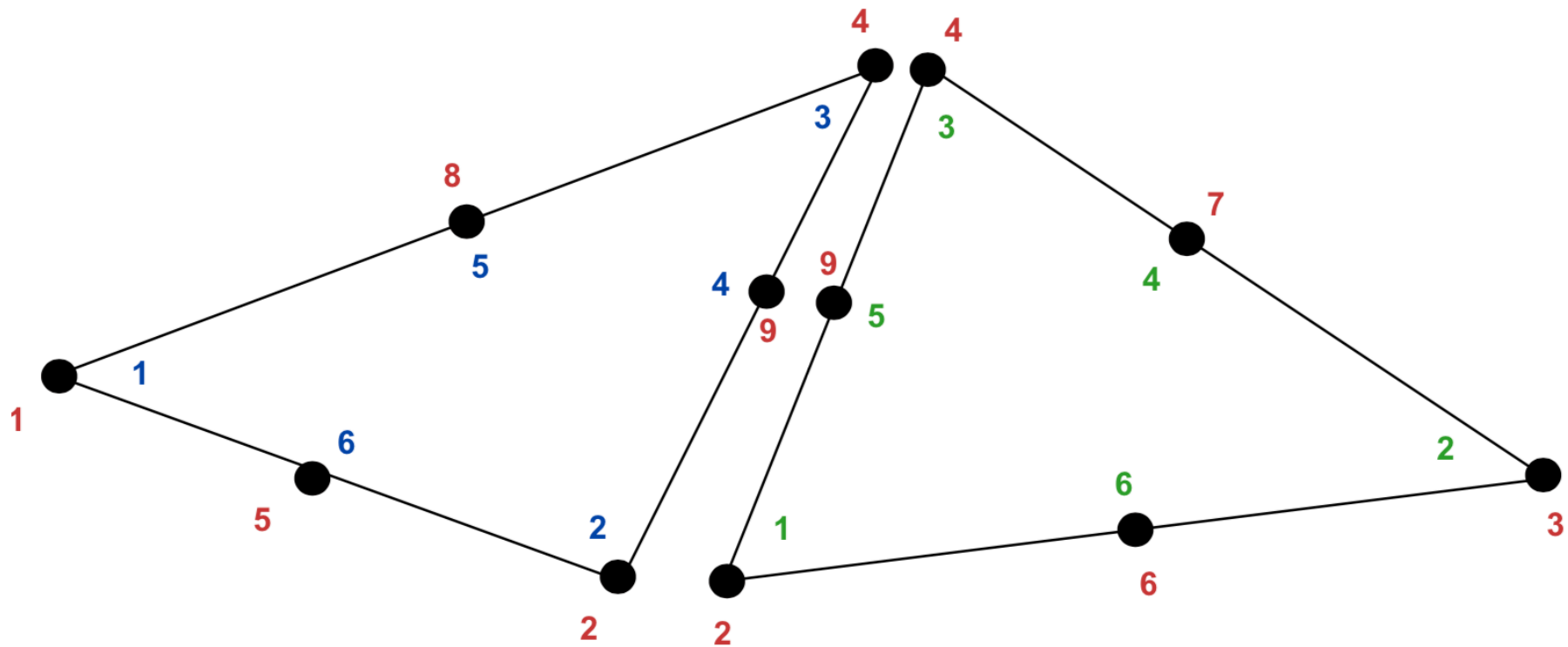
$$A_{T,ij} = a_T(\phi_j^T, \phi_i^T)$$

The local-to-global mapping

The global matrix ι_T is defined by

$$I = \iota_T(i)$$

where I is the *global index* corresponding to the *local index* i



The assembly algorithm

$A = 0$

for $T \in \mathcal{T}$

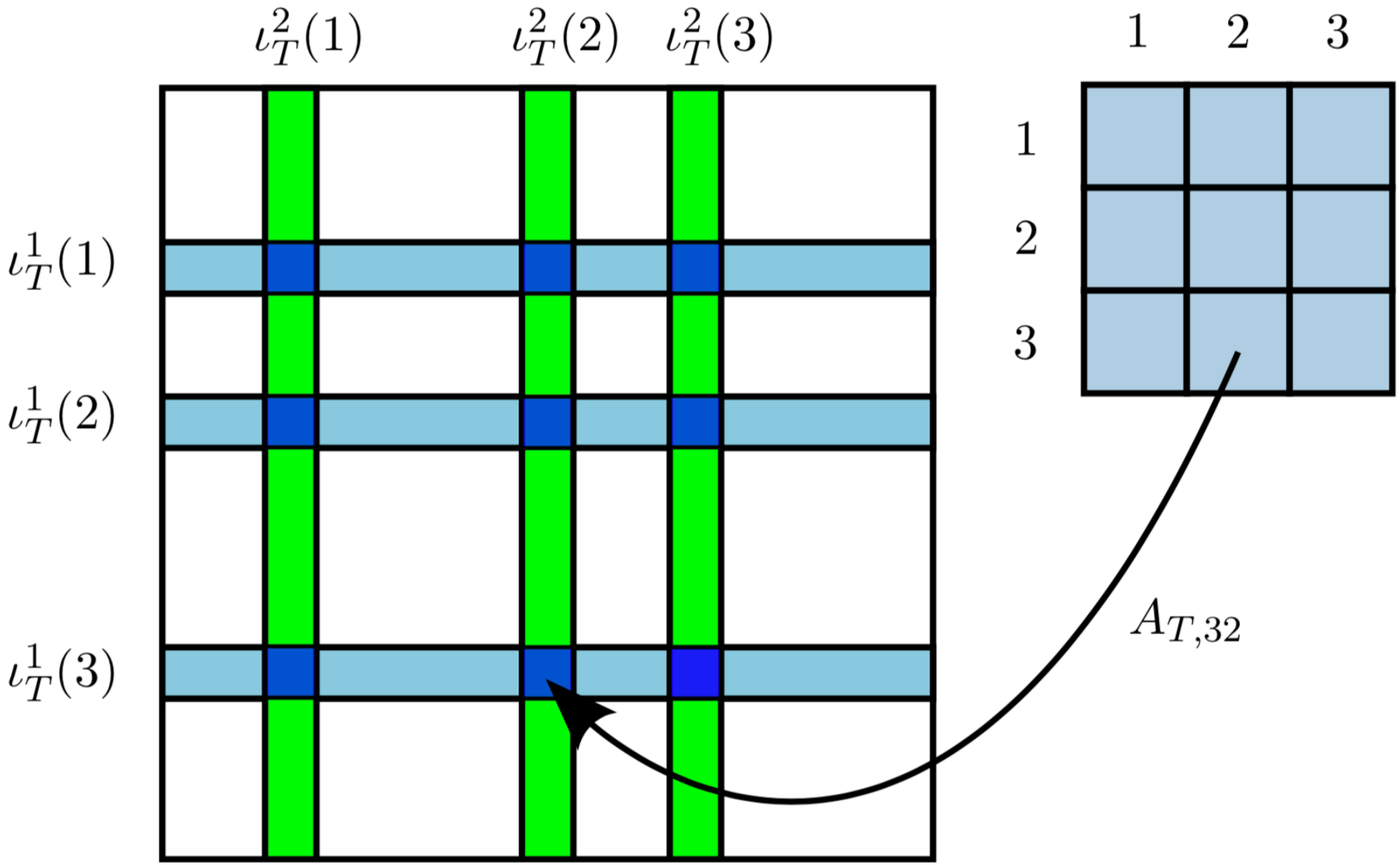
Compute the element matrix A_T

Compute the local-to-global mapping ι_T

Add A_T to A according to ι_T

end for

Adding the element matrix A_T



Solving static PDEs

Solving PDEs in FEniCS

Solving a physical problem with FEniCS consists of the following steps:

1. Identify the PDE and its boundary conditions
2. Reformulate the PDE problem as a variational problem
3. Make a Python program where the formulas in the variational problem are coded, along with definitions of input data such as f, u_0, and a mesh for Ω
4. Add statements in the program for solving the variational problem, computing derived quantities such as ∇u, and visualizing the results

Deriving a variational problem for Poisson's equation

The simple recipe is: multiply the PDE by a test function v and integrate over Ω:

$$-\int_\Omega (\Delta u) v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x$$

Then integrate by parts and set $v = 0$ on the Dirichlet boundary:

$$-\int_\Omega (\Delta u) v \,\mathrm{d}x = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \underbrace{\int_{\partial\Omega} \frac{\partial u}{\partial n} v \,\mathrm{d}s}_{=0}$$

We find that:

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x$$

Variational problem for Poisson's equation

Find $u \in V$ such that

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x$$

for all $v \in \hat{V}$

The trial space V and the test space $\hat{V}$ are (here) given by

$$V = \{v \in H^1(\Omega) : v = u_0 \text{ on } \partial\Omega\}$$
$$\hat{V} = \{v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega\}$$

Discrete variational problem for Poisson's equation

We approximate the continuous variational problem with a discrete variational problem posed on finite dimensional subspaces of V and $\hat{V}$:

$$V_h \subset V$$
$$\hat{V}_h \subset \hat{V}$$

Find $u_h \in V_h \subset V$ such that

$$\int_\Omega \nabla u_h \cdot \nabla v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x$$

for all $v \in \hat{V}_h \subset \hat{V}$

Canonical variational problem

The following canonical notation is used in FEniCS: find $u \in V$ such that

$$a(u, v) = L(v)$$

for all $v \in \hat{V}$

For Poisson's equation, we have

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x$$
$$L(v) = \int_\Omega f v \, \mathrm{d}x$$

$a(u, v)$ is a *bilinear form* and $L(v)$ is a *linear form*

A test problem

We construct a test problem for which we can easily check the answer. We first define the exact solution by

$$u(x, y) = 1 + x^2 + 2y^2$$

We insert this into Poisson's equation:

$$f = -\Delta u = -\Delta(1 + x^2 + 2y^2) = -(2 + 4) = -6$$

This technique is called the *method of manufactured solutions*

TASK

$$-\mathrm{div}[\mathrm{grad}[u]] = f$$

$$-\mathrm{div}[\mathrm{grad}[u]] = -6 \qquad \text{in } \Omega$$

Boundary conditions satisfy: $u(x, y) = 1 + x^2 + 2y^2$

Ex. Solve and visualize results in ParaView

Ex. Calculate grad(u) field and visualize in ParaView glyph

Ex. Impose these BCs using class

$$u(x = 0, y) = 1$$
$$u(x = 1, y) = \sin(y)$$

Ex. How much is flux of grad(u) on the right boundary?

Step by step: the first line

The first line of a FEniCS program usually begins with

```
from fenics import *
```

This imports key classes like `UnitSquareMesh`, `FunctionSpace`, `Function` and so forth, from the FEniCS user interface (DOLFIN)

Step by step: creating a mesh

Next, we create a mesh of our domain Ω:

```
mesh = UnitSquareMesh(8, 8)
```

This defines a mesh of $8 \times 8 \times 2 = 128$ triangles of the unit square.

Other useful classes for creating built-in meshes include `UnitIntervalMesh`, `UnitCubeMesh`, `UnitCircleMesh`, `UnitSphereMesh`, `RectangleMesh` and `BoxMesh`

More complex geometries can be built using Constructive Solid Geometry (CSG) through the FEniCS component `mshr`:

```
from mshr import *
r = Rectangle(Point(0.5, 0.5), Point(1.5, 1.5))
c = Circle(Point(1.0, 1.0), 0.2)
g = r - c
mesh = generate_mesh(g, 10)
```

Step by step: creating a function space

The following line creates a finite element function space relative to this mesh:

```
V = FunctionSpace(mesh, "Lagrange", 1)
```

The second argument specifies the type of element, while the third argument is the degree of the basis functions on the element

Other types of elements include `"Discontinuous Lagrange"`, `"Brezzi-Douglas-Marini"`, `"Raviart-Thomas"`, `"Crouzeix-Raviart"`, `"Nedelec 1st kind H(curl)"` and `"Nedelec 2nd kind H(curl)"`

Step by step: defining expressions

Next, we define an expression for the boundary value:

```
u0 = Expression("1 + x[0]*x[0] + 2*x[1]*x[1]",
    degree=2)
```

The formula must be written in C++ syntax, and the polynomial degree must be specified.

The `Expression` class is very flexible and can be used to create complex user-defined expressions. For more information, try

```
from fenics import *
help(Expression)
```

in Python or, in the shell:

```
$ pydoc fenics.Expression
```

Step by step: defining a boundary condition

The following code defines a Dirichlet boundary condition:

```
bc = DirichletBC(V, u0, "on_boundary")
```

This boundary condition states that a function in the function space defined by `V` should be equal to `u0` on the domain defined by `"on_boundary"`

Note that the above line does not yet apply the boundary condition to all functions in the function space

Step by step: more about defining domains

For a Dirichlet boundary condition, a simple domain can be defined by a string

```
"on_boundary" # The entire boundary
```

Alternatively, domains can be defined by subclassing `SubDomain`

```
class Boundary(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary
```

You may want to experiment with the definition of the boundary:

```
"near(x[0], 0.0)" # x_0 = 0
"near(x[0], 0.0) || near(x[1], 1.0)"
```

There are many more possibilities, see

```
help(SubDomain)
help(DirichletBC)
```

Step by step: defining the right-hand side

The right-hand side $f = -6$ may be defined as follows:

```
f = Expression("-6.0", degree=0)
```

or (more efficiently) as

```
f = Constant(-6.0)
```

Step by step: defining variational problems

Variational problems are defined in terms of *trial* and *test* functions:

```
u = TrialFunction(V)
v = TestFunction(V)
```

We now have all the objects we need in order to specify the bilinear form $a(u, v)$ and the linear form $L(v)$:

```
a = inner(grad(u), grad(v))*dx
L = f*v*dx
```

Step by step: solving variational problems

Once a variational problem has been defined, it may be solved by calling the `solve` function:

```
u = Function(V)
solve(a == L, u, bc)
```

Note the reuse of the variable name `u` as both a `TrialFunction` in the variational problem and a `Function` to store the solution.

Error analysis

TASK

The computational domain is a unit bi-square plate.
Consider homogeneous Dirichlet BCs on the whole domain.
Using methods of manufactured solution, $f = 2\pi^2 \sin(\pi x) \sin(\pi y)$.
The boundary value problem takes the following form:

$$-\text{div}[\text{grad}[u]] = f$$

$$u(x = y = 0) = u(x = 0, y = 1) = u(x = 1, y = 0) = u(x = y = 1) = 0$$

with analytical solution of:

$$u = \sin(\pi x) \sin(\pi y)$$

Ex. Visualize u in ParaView and extract centerline profile

Ex. What are the maximum and minimum value of u; and u(0.5,0.3)=?

Ex. Calculate degree-of-freedom and time_to_solution

Ex. Calculate L2 and H1 norms

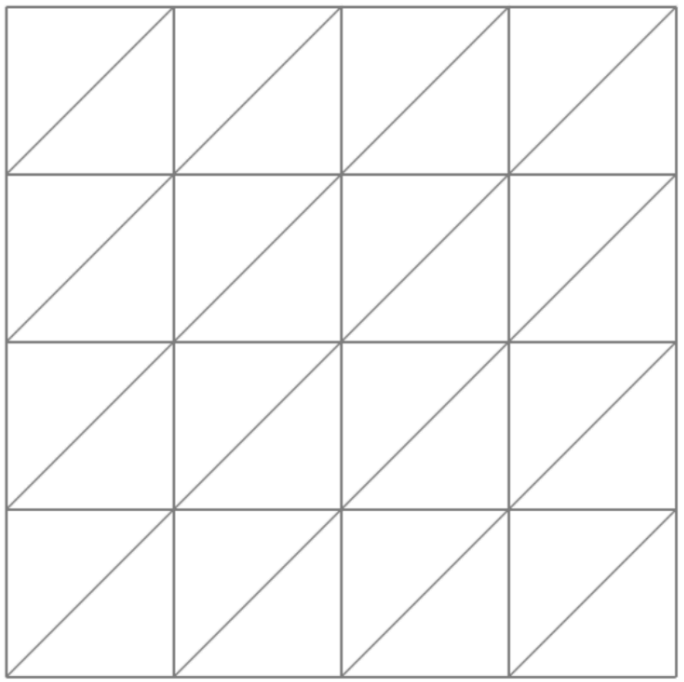
$$L_2^{\text{error}} = ||u - u_h|| = \sqrt{\int_\Omega (u_h - u)^2 \, \mathrm{d}\Omega}$$

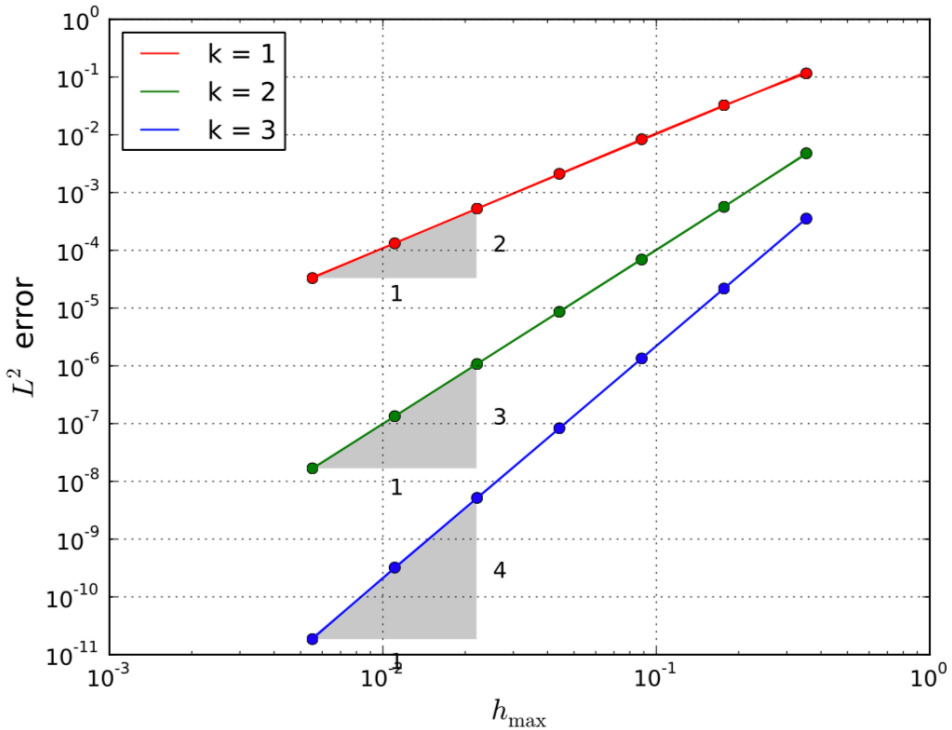
$$H_1^{\text{error}} = ||\nabla u - \nabla u_h|| = \sqrt{\int_\Omega (\nabla_x u_h - \nabla_x u)^2 + (\nabla_y u_h - \nabla_y u)^2 \, \mathrm{d}\Omega}$$

A priori estimates

If $u \in H^{k+1}(\Omega)$ and $V_h = P^k(\mathcal{T}_h)$ then

$$|||u - u_h||| \leqslant Ch^k \|u\|_{\Omega,k+1}$$
$$\|u - u_h\|_{L^2(\Omega)} \leqslant Ch^{k+1} \|u\|_{\Omega,k+1}$$





Ex. Calculate “rate of convergence” for this problem

Ex. Write a bash file for performing h-refinement analysis

Ex. Increase polynomial order, how rate of convergence changes?

How to changes solvers?

Direct methods

- Gaussian elimination
  - Requires $\sim \frac{2}{3}N^3$ operations
- LU factorization: $A = LU$
  - Solve requires $\sim \frac{2}{3}N^3$ operations
  - Reuse L and U for repeated solves
- Cholesky factorization: $A = LL^\top$
  - Works if A is symmetric and positive definite
  - Solve requires $\sim \frac{1}{3}N^3$ operations
  - Reuse L for repeated solves

Iterative methods

Krylov subspace methods

- GMRES (Generalized Minimal RESidual method)
- CG (Conjugate Gradient method)
  - Works if A is symmetric and positive definite
- BiCGSTAB, MINRES, TFQMR, ...

Multigrid methods

- GMG (Geometric MultiGrid)
- AMG (Algebraic MultiGrid)

Preconditioners

- ILU, ICC, SOR, AMG, Jacobi, block-Jacobi, additive Schwarz, ...

Which method should I use?

Rules of thumb

- Direct methods for small systems
- Iterative methods for large systems
- Break-even at ca 100–1000 degrees of freedom
- Use a symmetric method for a symmetric system
  - Cholesky factorization (direct)
  - CG (iterative)
- Use a multigrid preconditioner for Poisson-like systems
- GMRES with ILU preconditioning is a good default choice

- Preconditioned Krylov solvers is a type of popular iterative methods that are easily accessible in FEniCS programs.
- The Poisson equation results in a symmetric, positive definite system matrix, for which the optimal Krylov solver is the Conjugate Gradient (CG) method.
- For non-symmetric problems, a Krylov solver for non-symmetric systems, such as GMRES, is a better choice.
- Incomplete LU factorization (ILU) is a popular and robust all-round preconditioner.

```
list_linear_solver_methods()
list_krylov_solver_preconditioners()
```

TASK

Ex. For the Poisson problem solved earlier, try GMRES-ILU solver

```
solve(a == L, u, bc)
      solver_parameters={'linear_solver': 'cg',
                         'preconditioner': 'ilu'})
```

Name	Method
'icc'	Incomplete Cholesky factorization
'ilu'	Incomplete LU factorization
'petsc_amg'	PETSc algebraic multigrid
'sor'	Successive over-relaxation

Name	Method
'bicgstab'	Biconjugate gradient stabilized method
'cg'	Conjugate gradient method
'gmres'	Generalized minimal residual method
'minres'	Minimal residual method
'petsc'	PETSc built in LU solver
'richardson'	Richardson method
'superlu_dist'	Parallel SuperLU
'tfqmr'	Transpose-free quasi-minimal residual method
'umfpack'	UMFPACK

Solving time-dependent PDEs

The heat equation

We will solve the simplest extension of the Poisson problem into the time domain, the heat equation:

$$\frac{\partial u}{\partial t} - \Delta u = f \quad \text{in } \Omega \text{ for } t > 0$$

$$u = g \quad \text{on } \partial\Omega \text{ for } t > 0$$

$$u = u^0 \text{ in } \Omega \text{ at } t = 0$$

The solution $u = u(x, t)$, the right-hand side $f = f(x, t)$ and the boundary value $g = g(x, t)$ may vary in space ($x = (x_0, x_1, ...)$) and time (t). The initial value u^0 is a function of space only.

Time-discretization of the heat equation

We discretize in time using the implicit Euler (dG(0)) method:

$$\frac{\partial u}{\partial t}(t^n) \approx \frac{u^n - u^{n-1}}{\Delta t}, \quad u(t^n) \approx u^n, \quad f^n = f(t^n)$$

Semi-discretization of the heat equation:

$$\frac{u^n - u^{n-1}}{\Delta t} - \Delta u^n = f^n$$

Algorithm

1. Start with u^0 and choose a timestep $\Delta t > 0$.
2. For $n = 1, 2, \ldots$, solve for u^n:

$$u^n - \Delta t \Delta u^n = u^{n-1} + \Delta t f^n$$

Variational problem for the heat equation

Find $u^n \in V^n$ such that

$$a(u^n, v) = L^n(v)$$

for all $v \in \hat{V}$ where

$$a(u, v) = \int_\Omega uv + \Delta t \nabla u \cdot \nabla v \, \mathrm{d}x$$
$$L^n(v) = \int_\Omega u^{n-1} v + \Delta t f^n v \, \mathrm{d}x$$

Note that the bilinear form $a(u, v)$ is constant while the linear form L^n depends on n

Detailed time-stepping algorithm for the heat equation

Define the boundary condition
Compute u^0 *as the projection of the given initial value*
Define the forms a *and* L
Assemble the matrix A *from the bilinear form* a
$t \leftarrow \Delta t$
while $t \leqslant T$ **do**
  Assemble the vector b *from the linear form* L
  Apply the boundary condition
  Solve the linear system $AU = b$ *for* U *and store in* u^1
  $t \leftarrow t + \Delta t$
  $u^0 \leftarrow u^1$ (get ready for next step)
end while

Test problem

We construct a test problem for which we can easily check the answer. We first define the exact solution by

$$u = 1 + x^2 + \alpha y^2 + \beta t$$

We insert this into the heat equation:

$$f = \dot{u} - \Delta u = \beta - 2 - 2\alpha$$

The initial condition is

$$u^0 = 1 + x^2 + \alpha y^2$$

This technique is called the *method of manufactured solutions*

Handling time-dependent expressions

We define a time-dependent expression for the boundary value:

```
alpha = 3; beta = 1.2
t = 0.0
g = Expression("1 + x[0]*x[0] + \
               alpha*x[1]*x[1] + beta*t",
               alpha=alpha, beta=beta, t=t,
               degree=2)
```

Then, we must explicitly update t and g:

```
t = 1.0
g.t = t
```

An alternative (robust) approach is to define t as a Constant:

```
t = Constant(0.0)
g = Expression("...", ..., t=t, ...)
t.assign(1.0)
# No need to update g itself
```

Projection and interpolation

We need to project the initial value into V_h:

```
u0 = project(g, V)
```

We can also interpolate the initial value into V_h:

```
u0 = interpolate(g, V)
```

A closer look at `solve`

For linear problems, this code

```
solve(a == L, u, bcs)
```

is equivalent to this

```
# Assembling a bilinear form yields a matrix
A = assemble(a)
# Assembling a linear form yields a vector
b = assemble(L)

# Applying boundary condition info to system
for bc in bcs:
    bc.apply(A, b)

# Solve Ax = b
solve(A, u.vector(), b)
```

Implementing the variational problem

```
# Decide on a time step
dt = 0.3

# Create Functions for previous and current sol.s
u0 = project(g, V)
u1 = Function(V)

# Define the variational formulation
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(beta - 2 - 2*alpha)
a = u*v*dx + dt*inner(grad(u), grad(v))*dx
L = u0*v*dx + dt*f*v*dx

# Define the boundary condition
bc = DirichletBC(V, g, "on_boundary")

# Assemble only once, before time-stepping
A = assemble(a)
```

Implementing the time-stepping loop

```
T = 2            # Set end time
t.assign(dt) # Solve on [0, dt] first

while t <= T:
    b = assemble(L)   # Assemble the rhs vector
    bc.apply(A, b)    # Apply boundary conditions

    # Solve linear system
    solve(A, u1.vector(), b)

    # Update time and previous solution
    t1 = float(t + dt)
    t.assign(t1)            # t := t1 + dt
    u0.assign(u1)           # u0 := u1
```

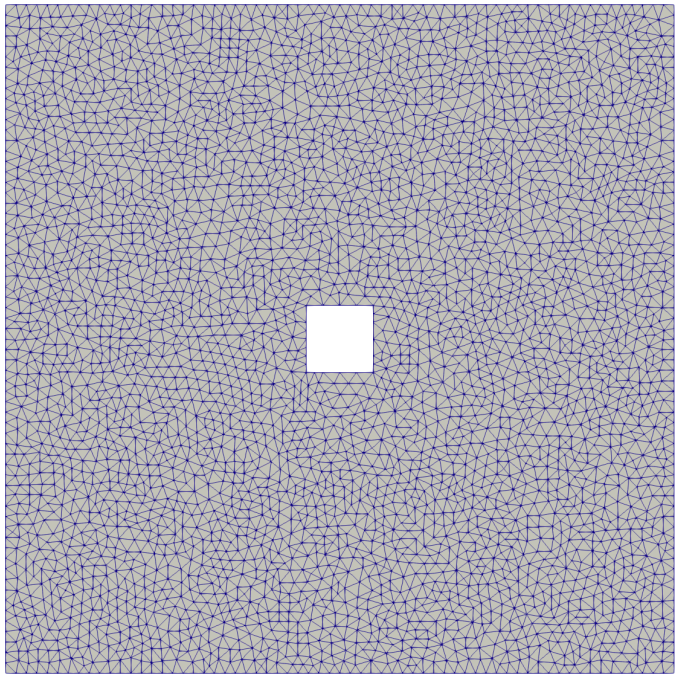
TASK

Ex. Write the code for above transient diffusion problem using backward Euler methods.

Ex. Visualize results for the first 3 seconds, with dt = 0.1 in ParaViw.

Ex. Compute the same solution on a same setup for a cubic domain.

Ex. Solve this problem in a square domain with square-hole in the center.



Solving non-linear PDEs

$$F(u) = au^2 + bu + c = 0,$$

Picards' method

$$F(u) \approx \hat{F}(u) = au^- u + bu + c = 0\,.$$

Newton's method

$$u^{k+1} = u^k - \frac{F(u^k)}{F'(u^k)}, \quad k = 0, 1, \ldots$$

$$F'(u) = 2au + b\,.$$

$$u = u^- - \frac{a(u^-)^2 + bu^- + c}{2au^- + b}, \quad u^- \leftarrow u\,.$$

Stopping criteria

$$|u - u^-| \leq \epsilon_u,$$

$$|F(u)| = |au^2 + bu + c| < \epsilon_r$$

Variational formulation

$$-\nabla \cdot (q(u)\nabla u) = f.$$

$$q(u) = (1+u)^m,\ f = 0,\ u = 0 \text{ for } x_0 = 0,\ u = 1 \text{ for } x_0 = 1$$

$$F(u;v) = 0 \quad \forall v \in \hat{V},$$

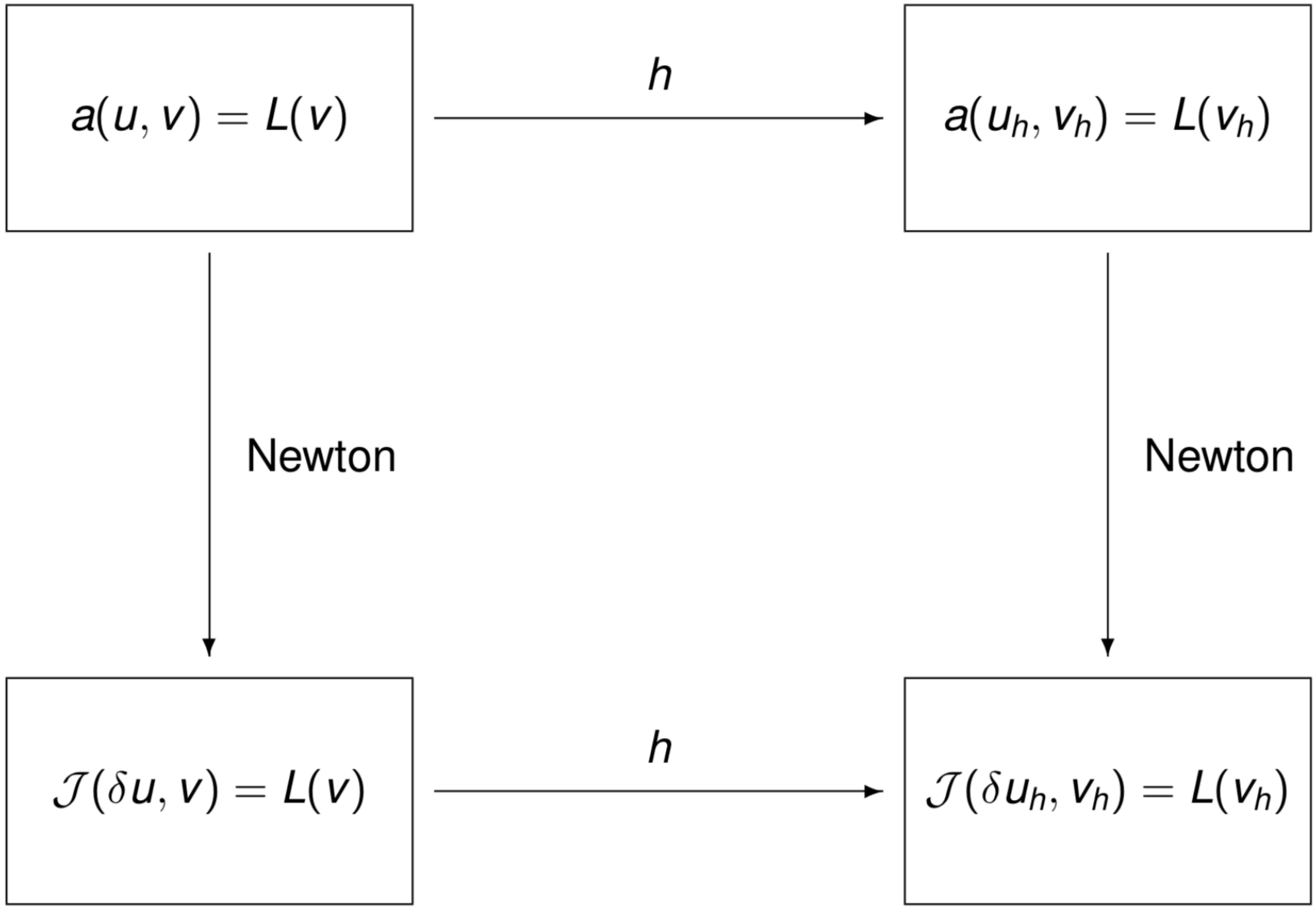
where

$$F(u;v) = \int_\Omega q(u)\nabla u \cdot \nabla v \, \mathrm{d}x,$$

and

$$\hat{V} = \{v \in H^1(\Omega) : v = 0 \text{ on } x_0 = 0 \text{ and } x_0 = 1\}.$$

The diagram commutes!



Picards' method

$$\tilde{F}(u^{k+1}; v) = \int_\Omega q(u^k) \nabla u^{k+1} \cdot \nabla v \, \mathrm{d}x.$$

Since this is a linear problem in the unknown u^{k+1}, we can equivalently use the formulation

$$a(u^{k+1}, v) = L(v),$$

with

$$\begin{aligned} a(u, v) &= \int_\Omega q(u^k) \nabla u \cdot \nabla v \, \mathrm{d}x, \\ L(v) &= 0. \end{aligned}$$

```
def q(u):
    return (1+u)**m

# Define variational problem for Picard iteration
u = TrialFunction(V)
v = TestFunction(V)
u_k = interpolate(Constant(0.0), V)  # previous (known) u
a = inner(q(u_k)*nabla_grad(u), nabla_grad(v))*dx
f = Constant(0.0)
L = f*v*dx

# Picard iterations
u = Function(V)     # new unknown function
eps = 1.0           # error measure ||u-u_k||
tol = 1.0E-5        # tolerance
iter = 0            # iteration counter
maxiter = 25        # max no of iterations allowed
while eps > tol and iter < maxiter:
    iter += 1
    solve(a == L, u, bcs)
    diff = u.vector().array() - u_k.vector().array()
    eps = numpy.linalg.norm(diff, ord=numpy.Inf)
    print "iter=%d: norm=%g" % (iter, eps)
    u_k.assign(u)   # update for next iteration
```

Newton's method (at the PDE level)

$$u^{k+1} = u^k + \delta u$$

$$q(u^{k+1}) = q(u^k) + q'(u^k)\delta u + \mathcal{O}((\delta u)^2) \approx q(u^k) + q'(u^k)\delta u,$$

and dropping other nonlinear terms in δu, we get

$$\nabla \cdot \left(q(u^k)\nabla u^k\right) + \nabla \cdot \left(q(u^k)\nabla \delta u\right) + \nabla \cdot \left(q'(u^k)\delta u \nabla u^k\right) = 0.$$

We may collect the terms with the unknown δu on the left-hand side,

$$\nabla \cdot \left(q(u^k)\nabla \delta u\right) + \nabla \cdot \left(q'(u^k)\delta u \nabla u^k\right) = -\nabla \cdot \left(q(u^k)\nabla u^k\right),$$

$$a(\delta u, v) = \int_\Omega \left(q(u^k)\nabla \delta u \cdot \nabla v + q'(u^k)\delta u \nabla u^k \cdot \nabla v\right) \mathrm{d}x,$$

$$L(v) = -\int_\Omega q(u^k)\nabla u^k \cdot \nabla v \,\mathrm{d}x.$$

There is a shortcut!

```
du = TrialFunction(V)
v  = TestFunction(V)
u_ = Function(V)      # the most recently computed solution
F  = inner(q(u_)*nabla_grad(u_), nabla_grad(v))*dx

J  = derivative(F, u_, du)  # Gateaux derivative in dir. of du
```

```
problem = NonlinearVariationalProblem(F, u, bcs, J)
solver  = NonlinearVariationalSolver(problem)
solver.solve()
```

$$\frac{d}{d\epsilon}\int_\Omega \nabla v \cdot \left(q(u^k+\epsilon\delta u)\nabla(u^k+\epsilon\delta u)\right)\,\mathrm{d}x$$

$$\lim_{\epsilon\to 0}\frac{d}{d\epsilon}F_i(u^k+\epsilon\delta u;v)$$

TASK

Consider $\Omega \in [0, 1]$ is an interval.
Calculate the following non-linear PDE:

$$-\nabla \cdot (q(u)\nabla u) = f.$$

$$q(u) = (1+u)^m,\ f = 0,\ u = 0 \text{ for } x_0 = 0,\ u = 1 \text{ for } x_0 = 1$$

Ex. Using Picard method

Ex. Using Newton's methods

Ex. Compare maximum errors for Picard and Newton

Mixed finite element

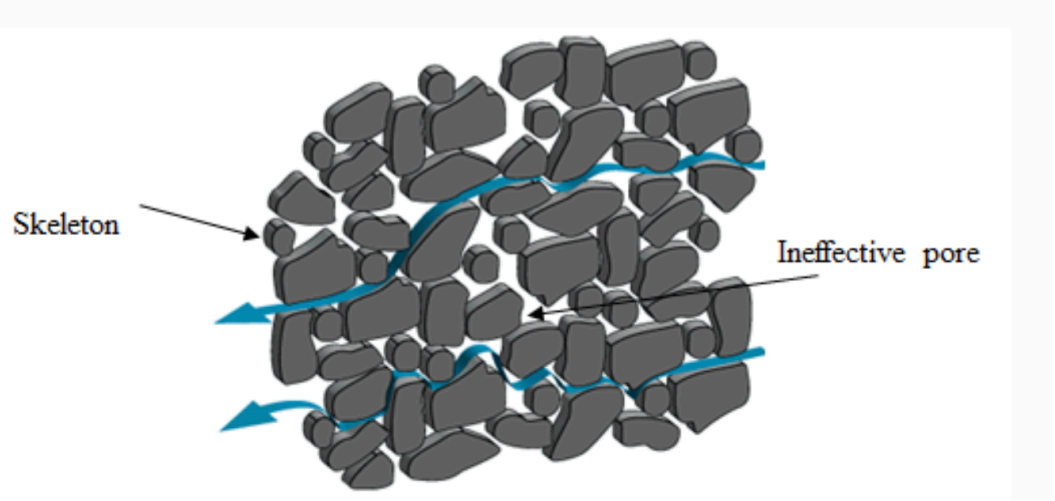
Darcy equation

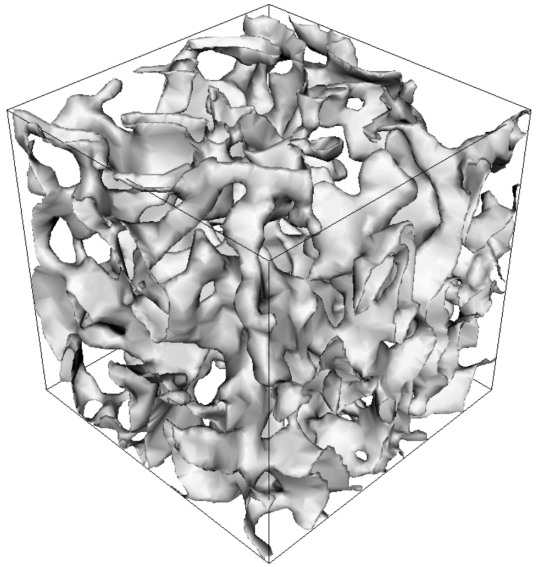
$$\alpha \boldsymbol{v} + \mathrm{grad}[p] = \rho(\boldsymbol{x})\boldsymbol{b}(\boldsymbol{x}) \qquad in\,\Omega$$

$$\mathrm{div}[\boldsymbol{v}] = 0 \qquad in\,\Omega$$

$$\boldsymbol{v}(\boldsymbol{x}) \cdot \hat{\boldsymbol{n}}(\boldsymbol{x}) = v_n(\boldsymbol{x}) \qquad on\,\Gamma^v$$

$$p(\boldsymbol{x}) = p_0(\boldsymbol{x}) \qquad on\,\Gamma^p$$





Weak form

$$\int_\Omega \boldsymbol{w}\cdot\alpha\boldsymbol{v}\,\mathrm{d}\Omega - \int_\Omega \mathrm{div}[w]p\,\mathrm{d}\Omega - \int_\Omega \mathrm{div}[v]q\,\mathrm{d}\Omega$$

$$= \int_\Omega \boldsymbol{w}\cdot\rho\boldsymbol{b}\,\mathrm{d}\Omega - \int_{\Gamma^p} \boldsymbol{w}\cdot\hat{\boldsymbol{n}}p_0\,\mathrm{d}\Gamma \qquad \forall \boldsymbol{w}(\boldsymbol{x})\in\mathcal{W},\ q(\boldsymbol{x})\in\mathcal{P}$$

$$\mathcal{P} := \{p(\boldsymbol{x})\in H^1(\Omega)|p(\boldsymbol{x}) = p_0(\boldsymbol{x}) \qquad on\ \Gamma^p\},$$

$$\mathcal{Q} := \{q(\boldsymbol{x})\in H^1(\Omega)|q(\boldsymbol{x}) = 0 \qquad on\ \Gamma^p\},$$

$$\mathcal{V} := \{\boldsymbol{v}(\boldsymbol{x})\in L_2(\Omega))^{nd}| \quad \mathrm{div}[v]\in L_2(\Omega), \boldsymbol{v}(x)\cdot\widehat{\boldsymbol{n}}(x) = v_n(x) \qquad on\ \Gamma^v\},$$

$$\mathcal{W} := \{\boldsymbol{w}(\boldsymbol{x})\in L_2(\Omega))^{nd}| \quad \mathrm{div}[w]\in L_2(\Omega), \boldsymbol{w}(x)\cdot\hat{\boldsymbol{n}}(x) = 0 \qquad on\ \Gamma^v\}.$$

Useful FEniCS tools (I)

Mixed elements:

```
V = VectorFunctionSpace(mesh, "Lagrange", 2)
Q = FunctionSpace(mesh, "Lagrange", 1)
W = V*Q
```

Defining functions, test and trial functions:

```
up = Function(W)
(u,p) = split(up)
```

Shortcut:

```
(u, p) = Functions(W)
# similar for test and trial functions
(u, p) = TrialFunctions(W)
(v, q) = TestFunctions(W)
```

Useful FEniCS tools (II)

Access subspaces:

```
W.sub(0) #corresponds to V
W.sub(1) #corresponds to Q
```

Splitting solution into components:

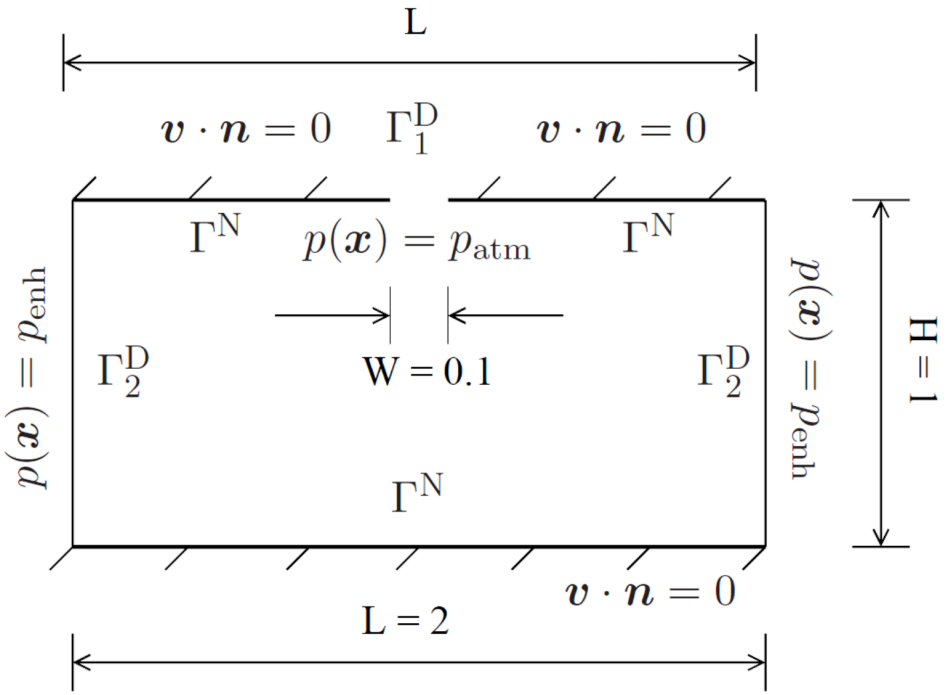
```
w = Function(W)
solve(a == L, w, bcs)
(u, p) = w.split()
```

Rectangle mesh:

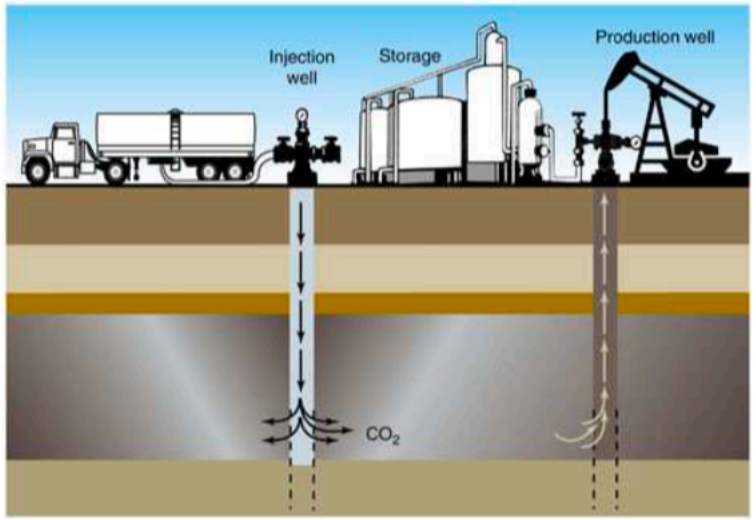
```
mesh = RectangleMesh(0.0, 0.0, 5.0, 1.0, 50, 10)
```

```
h = CellSize(mesh)
```

TASK



Reservoir simulation



Ex. Use continuous Galerkin weak form and calculate total flux at production The well.

Ex. Visualize p and v fields in ParaView

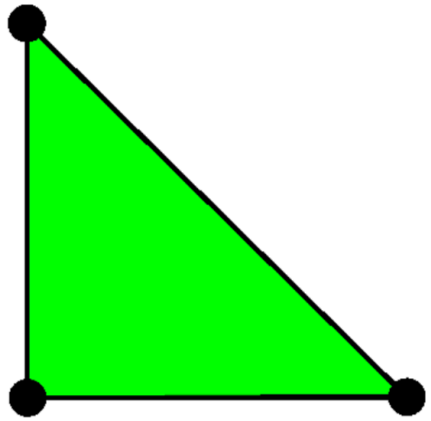
Parameter	Description	Value	Unit
k	permeability	0.001	m^2
μ_0	dynamic viscosity	0.001	$Pa \cdot s$
ρ	fluid Density(oil)	900	$\frac{Kg}{m^3}$
$\boldsymbol{b}(\boldsymbol{x})$	body force	**0**	N
P_{inj}	Injection pressure	p[atm]*101325	Pa
ϕ	Porosity [shale]	0.1	

What elements are available in FEniCS?

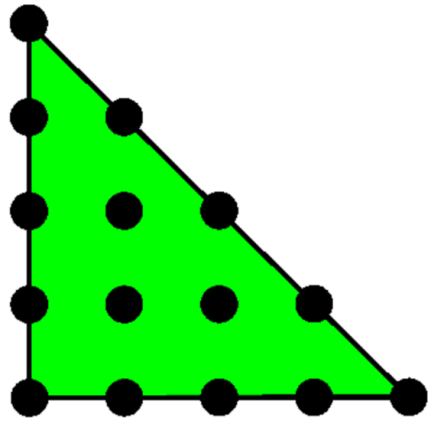
Name	Symbol	Dimension	Degree
Argyris	ARG	2	5
Arnold-Winther	AW	2	
Brezzi-Douglas-Marini	BDM	2,3	1-6
Crouzeix-Raviart	CR	2,3	1
Discontinuous Lagrange	DG	2,3	1-6
Hermite	HER	2,3	
Lagrange	CG	2,3	1-6
Mordal-Tai-Winther	MTW	2	
Morley	MOR	2	
Nédélec 1st kind H(curl)	N1curl	2,3	6
Nédélec 2nd kind H(curl)	N2curl	2,3	6
Raviart-Thomas	RT	2,3	6

Lagrange (CG) elements

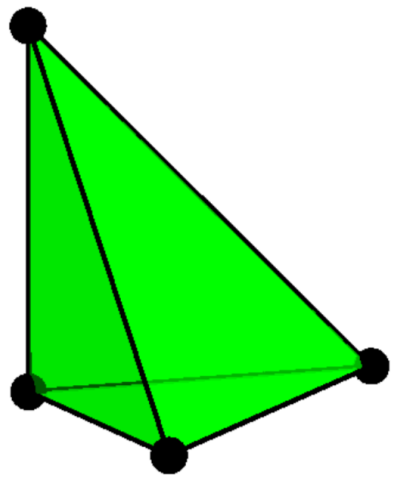
First order on triangles



Fourth order on triangles

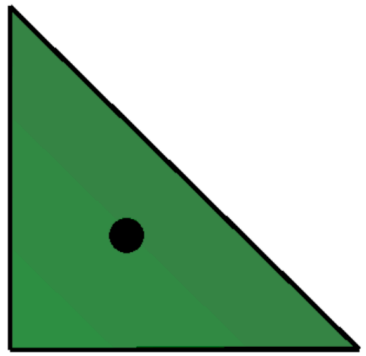


First order on tetrahedra

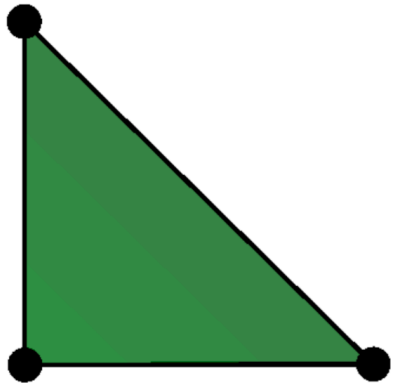


Discontinuous Lagrange (DG) elements

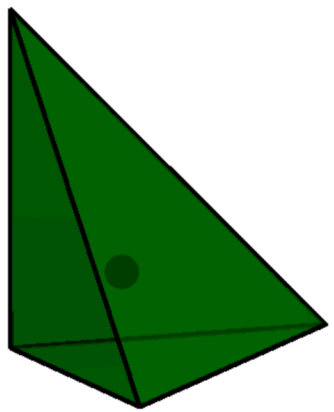
Constant on triangles



Linear on triangles



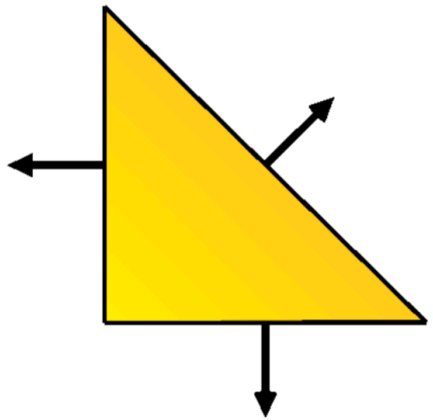
Constant on tetrahedra



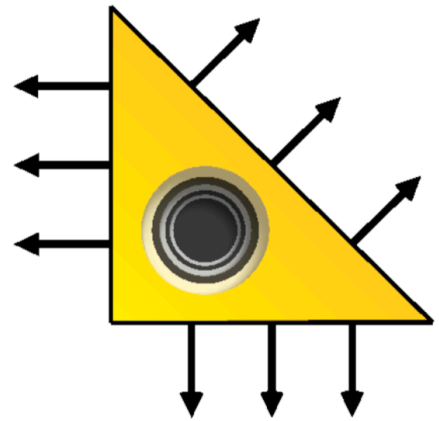
Raviert-Thomas (RT) elements

These elements are $H(\mathrm{div})$-conforming.

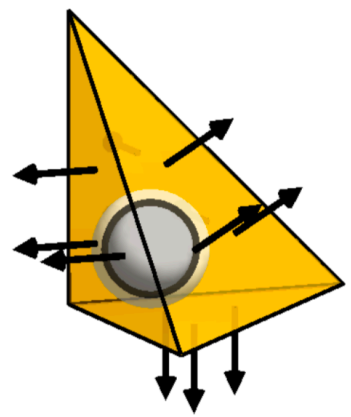
First order on triangles



Third order on triangles



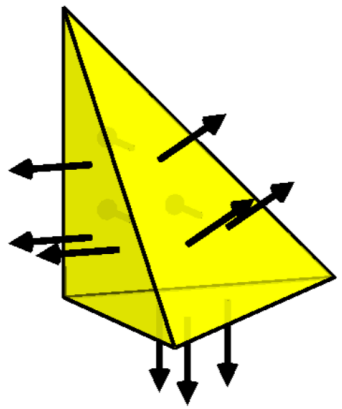
Second order on tetrahedra



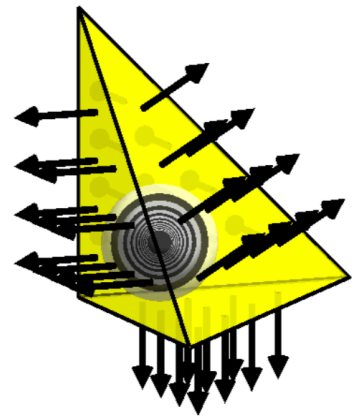
Brezzi-Douglas-Marini (BDM) elements

These elements are $H(\mathrm{div})$-conforming.

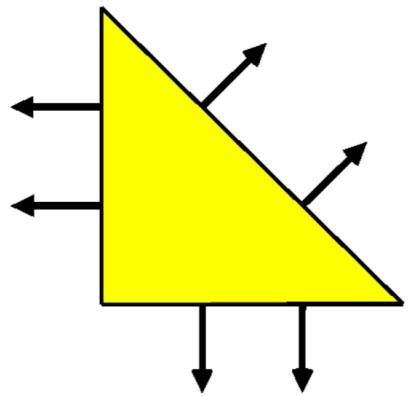
First order on tetrahedra



Third order on tetrahedra

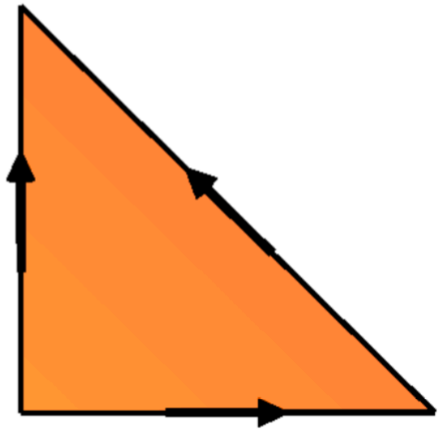


Second order on triangles

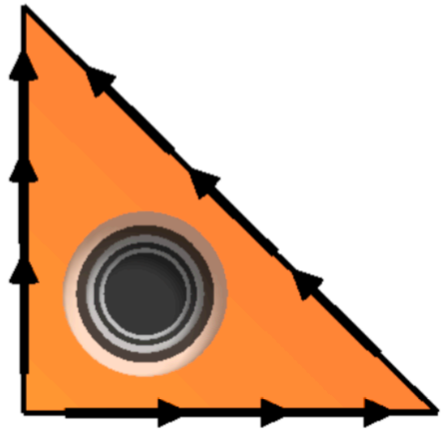


These elements are $H(\mathrm{curl})$-conforming.

First order on triangles



Third order on triangles

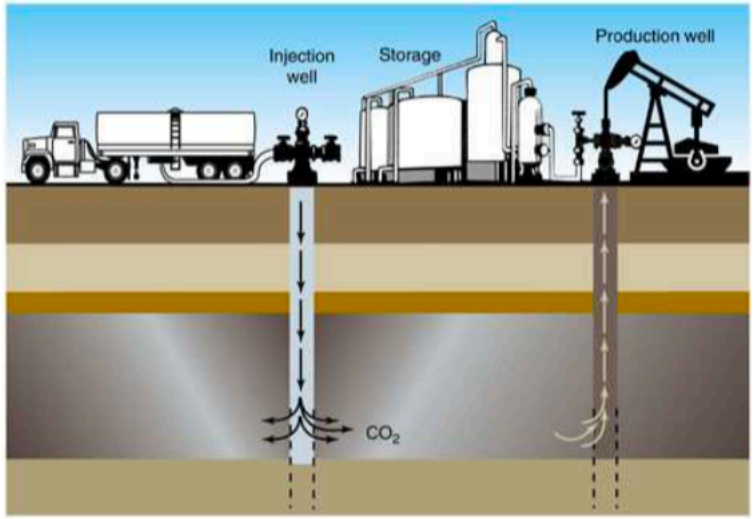


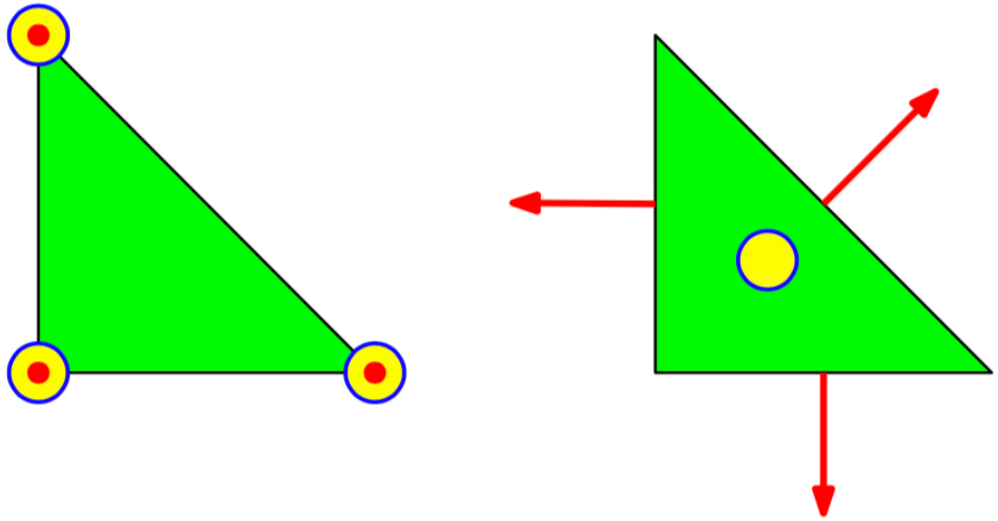
First order on tetrahedra

TASK

Ex.

Use RTO mixed finite element
and solve the reservoir problem





Thank you

Contact

Mohammad S. Joshaghani
msarrafjoshaghani@uh.edu
Graduate Student
Computational and Applied Mechanics Laboratory
Department of Civil & Environmental Engineering
University of Houston, Houston, Texas.