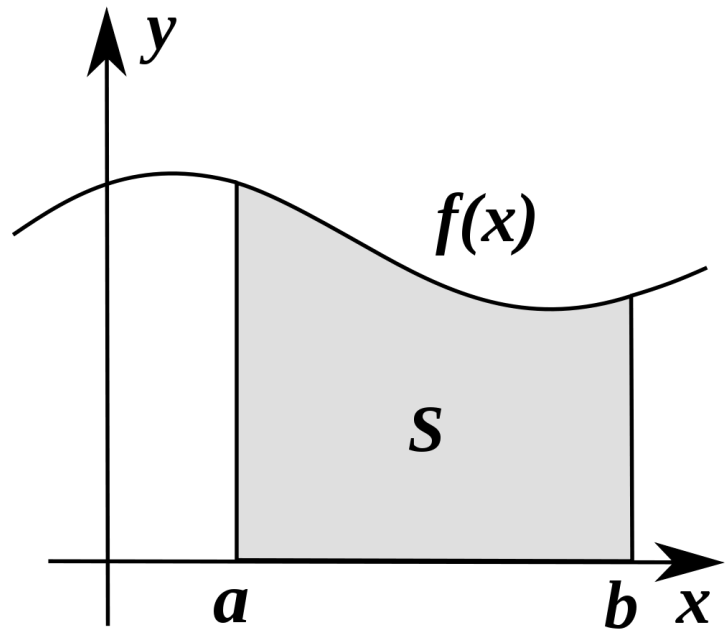
Specialized Numerical Methods for Transport Phenomena

Integrating over cells



Bruno Blais and Laura Prieto Saavedra

Department of Chemical Engineering
Polytechnique Montréal

October 22, 2025

Outline

Motivation

Gauss-Legendre quadratures in 1D

Extension to 2D and 3D

Using quadratures in `deal.II`

Conclusion

Outline

Finite element method

As we will see in the following weeks, the finite element method will require us to carry out integral over the domains (1D, 2D or 3D):

$$\int_\Omega \varphi_i \varphi_j \mathrm{d}\Omega = \int_\Omega \varphi_i Q \mathrm{d}\Omega$$

This will enable us to solve equations. Don't worry, we will see in the following weeks what this is and where it comes from. For now, we need to know we will be integrating over domain Ω. We saw in the previous lectures that we divide this domain into cells. It is over the cells that we will integrate.

Numerical integration

For now, let's simplify our problem to this:

$$\int_\Omega f(\boldsymbol{x}) \mathrm{d}\Omega = I$$

where we wish to calculate I. We triangulate the domain, so I is approximated by:

$$I = \int_\Omega f(\boldsymbol{x}) \mathrm{d}\Omega = \sum_e \int_{\Omega_e} f(\boldsymbol{x}) \mathrm{d}\Omega_e$$

where Ω_e are the cells. The integral over the domain, is the sum of the integrals over the cells.

Integrating over a cell

Integration methods

In your curriculum you have seen ways to integrate numerically:

- Mid-point rule
- Trapezoidal rule
- Simpson's 1/3 rule
- Gauss quadratures

Integrating over a cell

Integration methods

In your curriculum you have seen ways to integrate numerically:

- Mid-point rule
- Trapezoidal rule
- Simpson's 1/3 rule
- Gauss quadratures

In FEM (and FVM) what we integrate are polynomials in the end. So we will use Gauss quadratures.

Outline

Gauss quadratures: Concept

A review

You should have learnt about Gauss quadratures at one point in your curriculum if you took an intro to scientific computing class. The only new thing we will learn here is how they are generalized to 2D and 3D integration.

Gauss-Legendre Quadratures in 1D

Gauss-Legendre quadratures are a different integration technique than interval decomposition strategies (e.g. trapezoids). For a given integral:

$$\int_a^b f(x)dx$$

They aim at choosing the optimal points x_i where the function $f(x)$ is evaluated.

Gauss-Legendre Quadratures in 1D

Gauss-Legendre quadratures are a different integration technique than interval decomposition strategies (e.g. trapezoids). For a given integral:

$$\int_a^b f(x)dx$$

They aim at choosing the optimal points x_i where the function $f(x)$ is evaluated.

This concept is critical to decrease the computational cost of the FEM.

Idea

Gauss quadratures replace the integral of a function $f(t)$ by a sum of this function evaluated at points t_i multiplied by weights w_i:

$$\int_{-1}^{1} f(t)\mathrm{d}t \approx \sum_{i=1}^{n} w_i f(t_i)$$

The n integration weights w_i and the points t_i are chose to integrate exactly a polynomial of order $2n-1$.

This integral is between -1 and 1, what will happen if we wish to integrate over other intervals?

Changing the interval

An integral I:

$$I = \int_a^b f(x)dx$$

Can be rewritten, after a change of coordinates, in the following form:

$$I = \int_{-1}^{1} g(t)dt = \int_{-1}^{1} f\left(\frac{(b-a)\,t + (a+b)}{2}\right)\frac{b-a}{2}dt$$

So we can change the bounds of any integral so that they are -1 and 1.

Example: changing the bounds

Let be the following integral:

$$I = \int_0^4 x dx$$

Perform the change of bounds to obtain an integral from -1 to 1 and compute the integral

Gauss Quadratures

We have just seen that any definite integral can be reduced to an integral whose bounds are in the form of an integral whose bounds are -1 and 1. The only thing missing is the choice of the points t_i and the weights w_i of the Gauss quadrature

$$\int_{-1}^{1} f(t)\mathrm{d}t \approx \sum_{i=1}^{n} w_i f(t_i)$$

The t_i and w_i, of which there are n, will be chosen so as to integrate exactly one polynomial of degree $2n-1$ or less.

One-point quadrature

The **one** point quadrature is constructed to integrate exactly a polynomial of degree 0 or degree 1.

$$\int_{-1}^{1} 1dt = w_1 f(t_1) = w_1$$
$$\int_{-1}^{1} tdt = w_1 f(t_1) = w_1 t_1$$

What should the values of t_1 and w_1 be?

Derivation of the two-point quadrature

The **two** quadrature integrates exactly polynomials of degree 0, 1, 2 or 3.

$$\int_{-1}^{1} 1dt = \sum_{i=1}^{2} w_i f(t_i) = w_1 + w_2$$

$$\int_{-1}^{1} tdt = \sum_{i=1}^{2} w_i f(t_i) = w_1 t_1 + w_2 t_2$$

$$\int_{-1}^{1} t^2 dt = \sum_{i=1}^{2} w_i f(t_i) = w_1 t_1^2 + w_2 t_2^2$$

$$\int_{-1}^{1} t^3 dt = \sum_{i=1}^{2} w_i f(t_i) = w_1 t_1^3 + w_2 t_2^3$$

Let's define the system of equations to determine the values of t_1, t_2, w_1 et w_2.

Quadrature formulas

With this procedure, we construct quadrature formulas.

n	Integration points t_i	Weights w_i	Degree
1	$t_1 = 0$	$w_1 = 2$	1
2	$t_1 = -\sqrt{1/3}$ $t_2 = \sqrt{1/3}$	$w_1 = 1$ $w_2 = 1$	3
3	$t_1 = -\sqrt{15}/5$ $t_2 = 0$ $t_3 = \sqrt{15}/5$	$w_1 = 5/9$ $w_2 = 8/9$ $w_3 = 5/9$	5
4	$t_1 = -\left(\sqrt{525 + 70\sqrt{30}}\right)/35$ $t_2 = -\left(\sqrt{525 - 70\sqrt{30}}\right)/35$ $t_3 = \left(\sqrt{525 - 70\sqrt{30}}\right)/35$ $t_4 = \left(\sqrt{525 + 70\sqrt{30}}\right)/35$	$w_1 = \left(18 - \sqrt{30}\right)/36$ $w_2 = \left(18 + \sqrt{30}\right)/36$ $w_3 = \left(18 + \sqrt{30}\right)/36$ $w_4 = \left(18 - \sqrt{30}\right)/36$	7

Example 1

Calculate the following integral I using a one point quadrature:

$$I = \int_0^1 e^x dx$$

Outline

Extending to multiple dimensions

Extending Gauss quadratures to 2D and 3D is easy in the case of tensor elements (quads and hexahedra).
The quadrature formula in 2D will be the tensor (dyadic) product of the quadrature formula in 1D for x and 1D for y.

Example with 2 points in 2D

Assuming we have a two point quadrature along both axis. For x we have r_0 and r_1 such that:

$$r_0 = -\frac{1}{\sqrt{3}}, \quad r_1 = \frac{1}{\sqrt{3}}$$

And for y we have s_0 and s_1

$$s_0 = -\frac{1}{\sqrt{3}}, \quad s_1 = \frac{1}{\sqrt{3}}$$

The resulting points will be the tensor product between $\boldsymbol{s}$ and $\boldsymbol{r}$: (r_i, s_j)

Example with 2 points in 2D

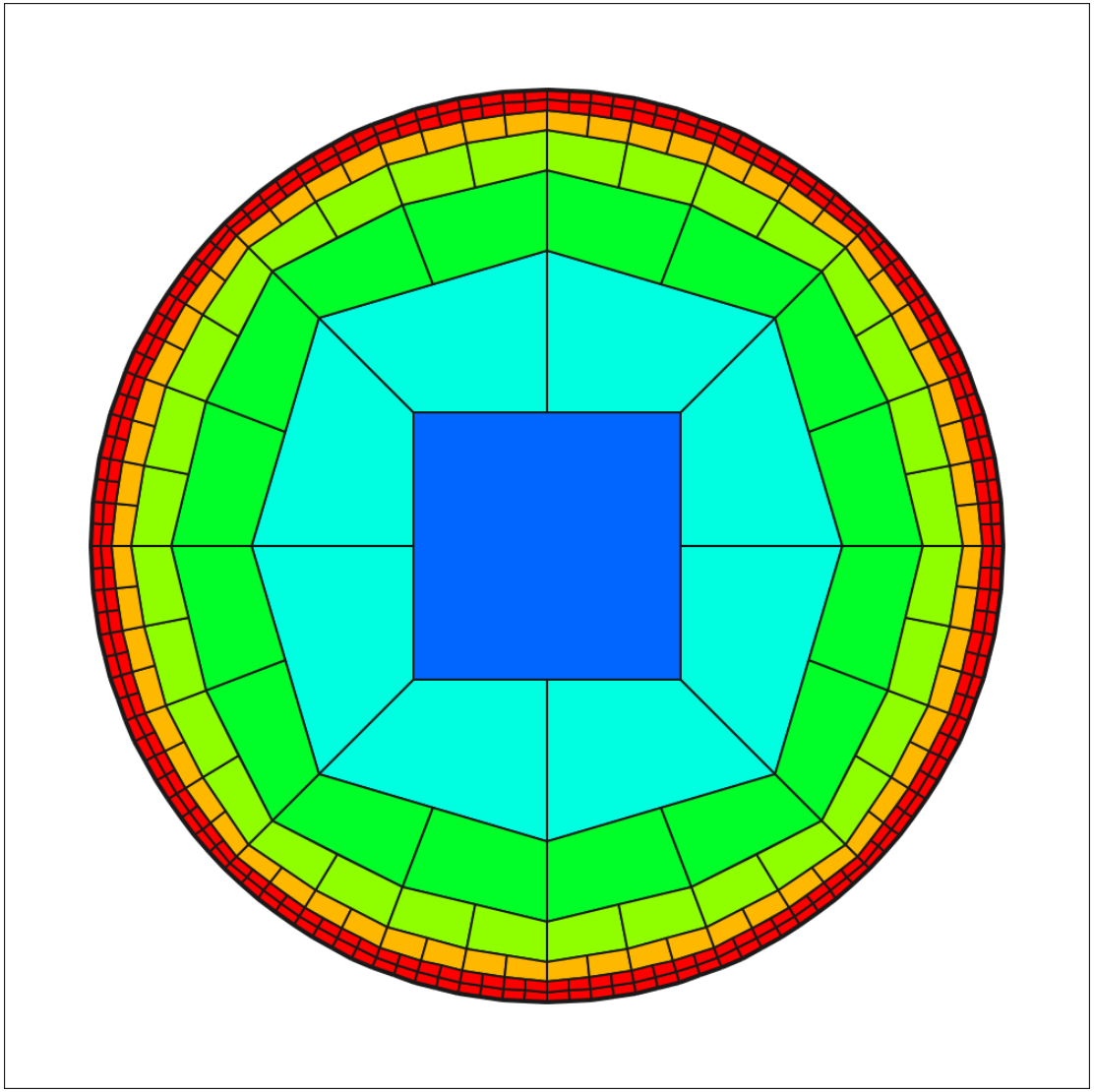
In this case, the four points we obtain are:

$$\boldsymbol{p_0} = (r_0, s_0) = \left(-\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}\right)$$
$$\boldsymbol{p_1} = (r_1, s_0) = \left(\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}\right)$$
$$\boldsymbol{p_2} = (r_0, s_1) = \left(-\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$$
$$\boldsymbol{p_3} = (r_1, s_1) = \left(\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$$

We had two points per dimensions, thus we obtain 4 points in 2D and 8 points in 3D.

A real mesh

In a quadrilateral mesh, the cells are not perfect unit squares.



The unit cell

All cells can be mapped to a unit cell. To achieve this easily, we need to use interpolation, which we will see extensively next week. Assuming we have a 2D Lagrange polynomial on my unit square, with four Lagrange polynomials:

$$\varphi_0 = \frac{1}{4}(1-\xi)(1-\eta)$$
$$\varphi_1 = \frac{1}{4}(1+\xi)(1-\eta)$$
$$\varphi_2 = \frac{1}{4}(1-\xi)(1+\eta)$$
$$\varphi_3 = \frac{1}{4}(1+\xi)(1+\eta)$$

where ξ and η are the independent coordinates on the unit square. The resulting weights are also the products of the original weight associated with each point in a coordinate.

The unit cell

We can express the position in the unit cell using the location of the four vertices of the original quadrangle:

$$\boldsymbol{x} = (x, y) = \sum_{i=0}^{4} \varphi_i \boldsymbol{x}_i = \sum_{i=0}^{4} \varphi_i \left(x_i, y_i\right)$$

This defines a transformation from $\boldsymbol{x}$ to $\boldsymbol{\xi}$ for which the Jacobian is:

$$\mathcal{J}(\xi, \eta) = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{bmatrix}$$

You may remember these notions from your Calculus class (Calcul 2 at Poly).

Redefining the integral

Consequently, the following integral

$$I = \int_{y=c}^{y=d} \int_{x=a}^{x=b} f(x) \mathrm{d}x \mathrm{d}y$$

becomes

$$I = \int_{\eta=-1}^{\eta=1} \int_{\xi=-1}^{\xi=1} f(\xi, \eta) \left\| J(\xi, \eta) \right\| \mathrm{d}\xi \mathrm{d}\eta$$

Which, using quadrature formula becomes:

$$I \approx \sum_{q=0}^{3} w_q f(\xi_q, \eta_q) \left\| J(\xi_q, \eta_q) \right\| \tag{1}$$

This works for any quadrilateral, even if the $a = a(y)$ and $b = b(y)$.

Outline

The code

deal.II provides nice abstraction for integration. The following code calculates the area of a 2D triangulation using quadratures.

```
FE_Q<2> fe(1);
QGauss<2> quadrature_formula(2);
FEValues<2> fe_values(fe, quadrature_formula, update_values |
    update_JxW_values | update_quadrature_points);

 const unsigned int n_q_points = fe_values.n_quadrature_points;
 double integration_result=0;

 for (auto cell: triangulation.active_cell_iterators())
 {
 ^^Ife_values.reinit(cell);
 ^^Ifor (unsigned int q = 0; q < n_q_points; q++)
 ^^I{
 ^^I^^Iintegration_result += fe_values.JxW(q)
 ^^I}
 }
```

Explaining the code

Generate the Finite Element interpolation support (we will see more about this next week). Here, 2 is the number of dimensions (2D) and the 1 implies first order.

```
FE_Q<2> fe(1);
```

Generate a Gauss quadrature in 2D. The quadrature uses 2 points per dimension.

```
QGauss<2> quadrature_formula(2);
```

Thus, it will have 4 points in 2D.

Explaining the code

The FEValues object takes care of coupling the interpolation object (FE) and the quadratures. It automatically calculates everything related to the geometrical transformation.

```
FEValues<2> fe_values(fe, quadrature_formula, update_values |
    update_JxW_values | update_quadrature_points);
^^I
```

It's a complicated object and has many arguments. The first one is the Finite Element object, then the quadrature object, then we specify what we will update when we loop through cells. We will see this more in the following weeks !

Explaining the code

```
FEValues<2> fe_values(fe, quadrature_formula, update_values |
    update_JxW_values | update_quadrature_points);
^^I
```

From this object, we can gather the number of quadrature point. Here we would have 4.

```
const unsigned int n_q_points = fe_values.n_quadrature_points;
^^I
```

Explaining the code

Finally, we carry out integration by looping over the cells and summing over the quadrature points.

```
double integration_result=0;
for (auto cell: triangulation.active_cell_iterators())
{
^^Ife_values.reinit(cell);
^^Ifor (unsigned int q = 0; q < n_q_points; q++)
^^I{
^^I^^Iintegration_result += fe_values.JxW(q)
^^I}
}
^^I
```

The above code is mathematically equivalent to :

$$\int_{y=-1}^{y=1}\int_{x=-1}^{x=1} 1 \cdot \|J(\xi,\eta)\| \,\mathrm{d}\xi\mathrm{d}\eta \approx \sum_{q=0}^{n_q} w_q \,\|J(\xi_q,\eta_q)\|$$

Outline

Conclusion

- Integration is fundamental to the Finite Element Method.
- Quadratures are the efficient way to carry out this integration.
- We have seen how they can be used in 1D, 2D and 3D.
- We have also seen how `deal.II` manages to abstract their complexity in any dimensions with just a few objects.
- In the homework, you get to manipulate these objects.
- Next week, they will be key to solving our first FEM problems.

Some things to clarify:

The Gauss quadrature requires:

- Certain number of integration points and weights in the unit coordinates.
- Transformation between physical coordinates and unit coordinates: to be able to move from any cell to the reference cell.

What are the equivalent of those in the code?

Components in the code

When performing numerical integration in `deal.II` the main ingredients are:

- A triangulation to perform integration in each of the cells:

```
Triangulation<2> tria;
```

- The quadrature formula with the points required in 1D:

```
QGauss<2> quadrature_formula(n_q_points);
```

- `FEValues` object that allows us to get the number of quadrature points in the appropriate dimension, the weights and the Jacobian information:

```
FEValues<2> fe_values(fe, quadrature_formula, update_values
    | update_JxW_values | update_quadrature_points);
```

Is the `FEValues` object magic?

Let us look at the syntax:

```
^^IFEValues<2> fe_values(fe, quadrature_formula, update_values |
    update_JxW_values | update_quadrature_points);
```

Several questions arise:

- What is this weird `fe` object?
- Why does this object know how to transform from physical to reference coordinates? Where are the lagrange polynomials that were used to map the cells to unit cells?
- What are those weird flags?

To understand this better, let us look at the documentation...

The `FEValues` constructor in `deal.II`:

There are four available constructors for this class due to overloading:

Public Member Functions

```
FEValues (const Mapping< dim, spacedim > &mapping, const
FiniteElement< dim, spacedim > &fe, const Quadrature< dim >
&quadrature, const UpdateFlags update_flags)
```

```
FEValues (const Mapping< dim, spacedim > &mapping, const
FiniteElement< dim, spacedim > &fe, const hp::QCollection< dim
> &quadrature, const UpdateFlags update_flags)
```

```
FEValues (const FiniteElement< dim, spacedim > &fe, const
Quadrature< dim > &quadrature, const UpdateFlags update_flags)
```

```
FEValues (const FiniteElement< dim, spacedim > &fe, const
hp::QCollection< dim > &quadrature, const UpdateFlags
update_flags)
```

Let us look at the third one, since it is the one we have been using so far...

The third `FEValues` constructor in `deal.II`:

The documentation page shows the following:

◆ FEValues() [3/4]

```
template<int dim, int spacedim = dim>
FEValues< dim, spacedim >::FEValues ( const FiniteElement< dim, spacedim > &  fe,
                                      const Quadrature< dim > &               quadrature,
                                      const UpdateFlags                       update_flags
                                    )
```

Constructor. This constructor is equivalent to the other one except that it makes the object use a Q_1 mapping (i.e., an object of type MappingQ(1)) implicitly.

As it can be seen we use a **Q1 mapping**, i.e., a polynomial mapping of degree 1. These are the functions we saw last class.
Since there is another constructor, we can also specify it directly.

Specifying a mapping:

The first constructor has this parameter:

◆ FEValues() [1/4]

```
template<int dim, int spacedim = dim>
FEValues< dim, spacedim >::FEValues ( const Mapping< dim, spacedim > &       mapping,
                                      const FiniteElement< dim, spacedim > & fe,
                                      const Quadrature< dim > &              quadrature,
                                      const UpdateFlags                      update_flags
                                    )
```

Constructor. Gets cell independent data from mapping and finite element objects, matching the quadrature rule and update flags.

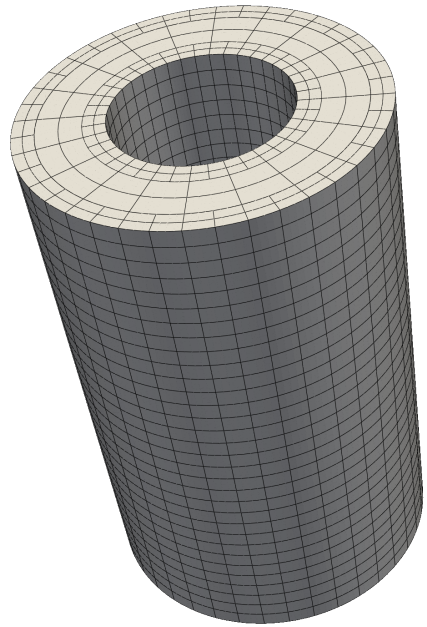
How to use it in the example?

```
MappingQ<dim>    mapping(1);
FEValues<2> fe_values(mapping, fe, quadrature_formula,
    update_values | update_JxW_values |
    update_quadrature_points);
```

This is equivalent to the previous call but giving an explicit mapping parameter...

Do we always want mapping of degree 1?

Increasing the order of the mapping is useful when we have elements with complex shapes (e.g. curved boundaries).
It will allow to have a more accurate transformation from the physical coordinates to the reference element improving the precision of the solution.



In the homework we have simple meshes with no curvature. You can try to increase the order and no improvement will be seen. Moreover, it requires also to provide more information about how interior edges and faces of the mesh should be curved.

What about the other parameters?

- The `FE_Q` object is not used at all when only doing integration. It was needed only for compilation since the `FEValues` requires it. Today we will learn what is its role in FEM.
- The ingredients needed for quadrature are all in the quadrature formula and the mapping. The `FEValues` is in charge of taking information from both to provide the Jacobian and the weights for each cell via `JxW`.
- The flags tell the `FEValues` what information is needed on each cell when iterating over all of them.