Nummerical Programming II

Personal summary by Gilberto Lem

Summer 2018

Contents

1 ODE's: Initial Value Problems

1.1 General Theory

An Ordinary Differential Equation is presented in the following form:

$$y'(t) = f(t, y(t))$$

Where f is a given function, t is the independent (scalar) variable, and y is the dependent variable we want to find.

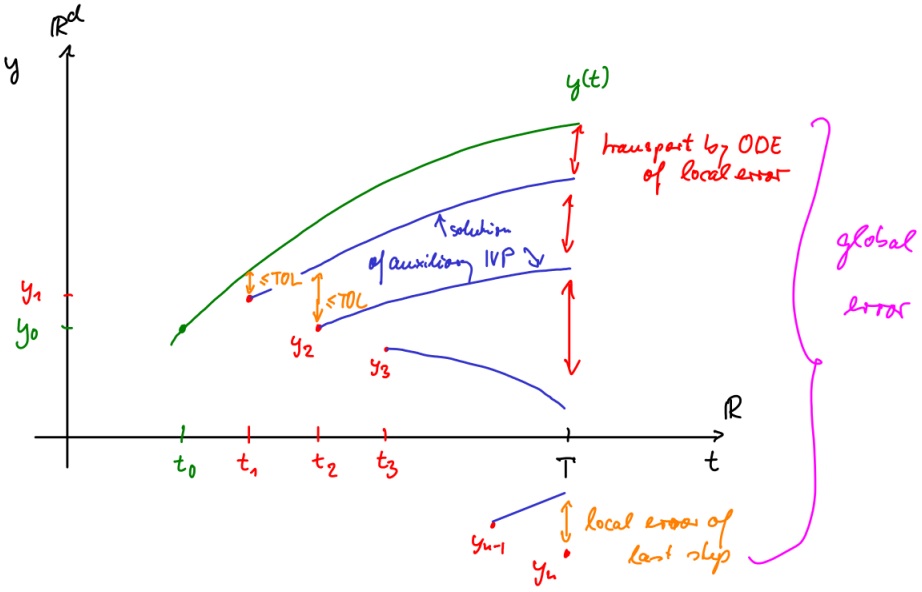
Solution lines for ODE's never cross, they can only touch, in which case uniqueness is lost.

An Initial Value Problem (IVP) means that we are provided with the value of the dependent variable at a specific value of the independent value: $y(t_0) = y_0$. IVP's have a general theory about existence and uniqueness:

- If f is continuous, then it **exists** a solution
- If both f and $\frac{\partial f}{\partial y}$ are continuous, then it exists a **unique** solution

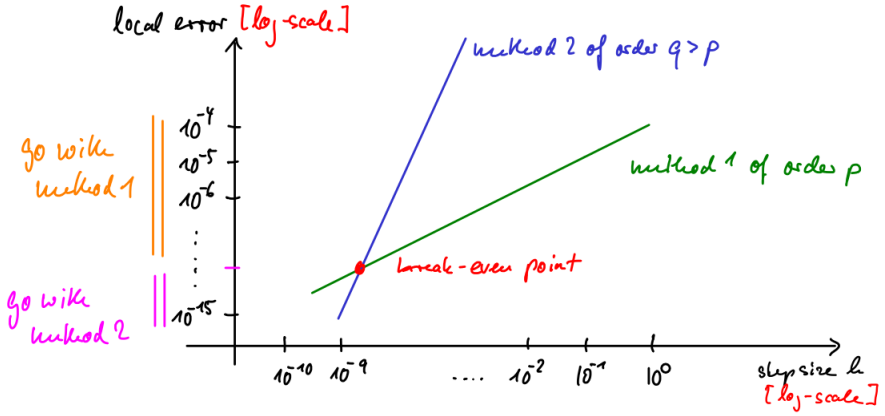
For already-made numeric implementations for solving IVP's, as the ones in MATLAB, it is important to consider the uniqueness criteria, for the machine to be able to obtain a valid solution, as well as the sensitivity to inputs of the ODE (take into account numerical error of initial values).

Local Error vs Global Error Before going into Error Estimation we need to define two types of errors: Local and Global. **Local Error** is the difference between the real solution of the ODE starting in the last grid-point and the new point y_k given by the numerical method. **Global Error** is the accumulation of those local errors, resulting in the difference between the last y_k given by the numerical method and the analytical solution of the ODE. Both kinds of error are illustrated in the following figure:



Order and Error of the Method The order p of the numerical method tells you how much the error changes when you modify the stepsize. One can express the error in terms of the order. The local error is $\mathcal{O}(h_k^{p+1})$, where h_k is the local stepsize, while the global error is $\mathcal{O}(h_{max}^p)$, where h_{max} is the maximum stepsize in the domain. It is important to note that a bigger order does not necessarily

imply a lower error. It depends on the stepsize. There comes a point where there is a break-even point, as we can see in the following figure:



In the IVP packages supplied for example by MATLAB, the machine controls just the **local** error to match the user supplied tolerance TOL. As in real life one does not have the analytical solution for the ODE to evaluate the real local error, one can use a better numerical method as the "real" solution. The standard today is to consider the difference between the methods as the local error and still use as a solution the one provided by the "better" numerical method, even though the error we calculated would correspond to the error of the "worse" numerical method. This results in a conservative error estimate, and is called **local extrapolation mode**.

Choosing Tolerances In practice, for the error estimation, one uses the sum of two tolerances: the Relative Tolerance, which takes into account the value of the solution itself, and the Absolute Tolerance, for not being too strict when the value of the solution (and thus the RelTOL) is very small. To choose the values of the different tolerances one can use the following rule of thumb:

- **RelTOL:** adjusted to the measurement errors in the physical system
- **AbsTOL:** adjusted to the problem and physical units

MATLAB ODE methods In MATLAB, IVP's can be solved with different implemented methods, here is a summary with some of their characteristics:

Name	Order (stepsize control)	Single/Multi step	Ex/Implicit
ode45	5(4)	s	explicit
ode23	3(2)	s	explicit
ode113	1-13	m	explicit
ode23s	3(2)	s	implicit
ode15s	1-5	m	implicit

1.2 Runge Kutta and Multistep Methods

Runge Kutta and Multistep Methods are used to solve IVP's. The idea behind them is to do a local estimation of the next value for the solution with the following exact expression:

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(\tau, y(\tau))\mathrm{d}\tau$$

Runge Kutta methods approximate the integral by a quadrature formula, while Multistep Methods replace f with a polynomial obtained interpolating its values in the last m steps.

1.2.1 Runge Kutta Methods

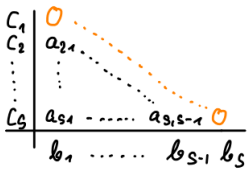
As said, Runge Kutta methods use a quadrature formula for approximating the integral. In a code-friendly form for evaluating the expression, one uses already-taken-care-of tables of coefficients c_i, a_{ij}, and b_j to compute:

$$y_{n+1} = y_n + h_n \sum_{j=1}^{n} b_j k_j$$

with

$$k_i = f(t_n + c_i h_n, \quad y_n + h_n \sum_{j=1}^{i-1} a_{ij} k_j)$$

The tables are commonly written in a Butcher-scheme:



As we have said, the estimation of the local error in all methods is done using two numerical methods of different order, and thus, in Runge Kutta methods we have to compute two different quadratures for each step. A nice thing of this approach is that between the two different methods only the b_j coefficients change, and thus we get to recycle the f-evaluations, which could be expensive.

Another trick used for making the computation efficient is is the **FSAL** (first-same-as-last). Here one uses the last f-evaluation of the last step as the first evaluation of the current step, to save one evaluation.

Another nice trick of Runge Kutta methods is that we can use them to generate dense outputs of y, i.e. not only get y_k, y_{k+1}, etc. but get also any value between those points too. This is achieved by changing the coefficients b_j for polynomials $b_j(\theta)$, where $\theta \in [0, 1]$ and $\theta = 0$ corresponds to y_k and $\theta = 1$ corresponds to y_{k+1}. We can construct the polynomials to match the same error as the corresponding to the order of our method.

1.2.2 Multistep Methods

On the other side, Multistep Methods, instead of a quadrature use an interpolation for f taking the information of the last m points. The error of the interpolation is $\mathcal{O}(h^m)$, and the method is order m. Note thas this needs the computing of only one f-evaluation per step, and thus these methods are useful for very expensive f.

A useful characteristic of multistep methods is that you can make very easily a numerical method of variable order, just by changing the number of past values you use to interpolate the polynomial. However, it has disadvantages, as that rapid stepsize changes degrade the quality of the interpolated polynomial, and thus step sizes have to be changed in a very conservative way.

1.3 Stiffness of IVP

A problem is stiff if the solution being sought varies slowly, but there are nearby solutions that vary rapidly, so the numerical method must take small steps to obtain satisfactory results, i.e. the problem is unstable. This characteristic depends on the differential equation, the initial conditions, and the numerical method used.

What can be done about stiff problems? You don't want to change the differential equation or the initial conditions, so you have to change the numerical method. Methods intended to solve stiff problems efficiently do more work per step, but can take much bigger steps. Stiff methods are implicit methods.

Stiffness implies that one needs certain restrictions on the stepsize for the numerical method to be stable. Specifically, we need the real part of the eigenvalues of the linearized operator $D_y f$ around the point of interest to be negative. This defines some regions of stability for each numerical method. In general, the restrictions have the following form:

$$h|\lambda| \leq c_\alpha$$

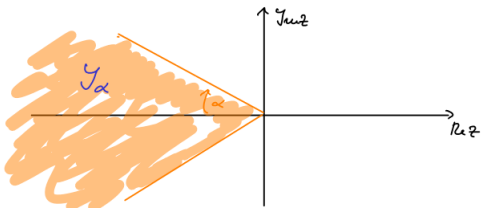
A method is called A-stable if the stability region is all space such that $\mathrm{Re} z < 0$:

For example, for the `ode45` method in MATLAB, $c_\alpha = 3.31$.

Stiff solvers use implicit methods to overcome the stability problem. This methods imply solving nonlinear systems of equations, which we should solve using Newton iterations (iterate by fixed-point iteration would be a disguised explicit method). The cost for each step of this kind of methods has the following components:

- Cost of $\#f$-evaluations
- Cost of building the $D_y f$ (Jacobian) matrix. (Automatic differentiation)
- Cost of the # steps in Newton iteration

Multistep methods add an additional restriction for the stability region. Depending on the number of steps taken, there is an angle α that restricts the region. Here the method is called A(α)-stable.



2 ODE's: Boundary Value Problems

Initial Value Problems have one or more conditions in one point of the independent variable domain. Boundary Value Problems have conditions in more than one point of the independent variable domain.

Unlike in IVP's, there is no general theory for BVP's regarding existence and uniqueness. There can be infinitely many solutions to a problem or none.

Many times, the problem is formulated with an additional parameter, which is a part of the ODE and thus will define the behaviour of the solution. Simmilarly to the conditions, the parameters are constant.

2.1 Solution Schemes

Shooting Method One solution technique to solving BVP's is the **shooting method**. It consists on solving the ODE as an IVP, proposing an initial condition (that matches one or more of the boundary conditions if possible). Then taking the result and setting with it a nonlinear system of equations, whose solution is the function that satisfies all the boundary conditions. The nonlinear system is solved using Newton's method. This method can lead to numerical instabilities.

Multiple Shooting Another solution scheme is **multiple shooting**. This adds multiple shooting points (each one a shooting method) and solves a more stable but bigger nonlinear system of equations.

Global Solvers for BVP's This is the most general scheme. They take an implicit method (for example the trapezoidal rule) and with that method applied to every point on the grid, plus the boundary conditions, define a $(N+1)d$-dimensional nonlinear system of equations, where N is the number of grid points and d the dimension of the problem. The nonlinear system is then solved by Newton iteration. This methods are available in MATLAB with the names bvp4c and bvp5c, where the number 4 or 5 refer to the order of the numerical scheme used (for example the trapezoidal rule would be order 2).

2.2 Practical Issues

2.2.1 Initial Guesses

Newton's method is iterative, and thus if we want to be efficient solving our problem, we need a useful first guess for the solutions and for the parameters needed. These guesses are made with some preliminary mathematical analysis (as taking approximations), or with prior experience from the problem.

Take for example the following ODE:

$$\epsilon y' = \sin^2(x) - \lambda \frac{\sin^4(x)}{y}$$

where λ is the parameter, and ϵ is a small number. As ϵ is small, we could approximate $\epsilon y' = 0$ and with that get an initial guess for $\lambda = 1$ of $y(x) = \sin^2(x)$. If we wanted to be more precise, we could generalize this to $\epsilon y' = \mathcal{O}(\epsilon)$ to see the order of magnitude of the approximation we are making. This approach is called **asymptotic analysis**.

Another technique is **Homotopy (parameter continuation)**. Imagine we have a problem that has a fixed parameter (for example a fixed Reynolds number in a fluid problem), which value makes the ODE difficult to solve numerically. One way to deal with this is to solve the BVP for a different value of the parameter (some value that makes the ODE easier to solve), and take that solution as an initial guess for solving the problem with a parameter a little closer to the real value, and iterate until achieving the desired value.

2.2.2 Parameters not part of the ODE

Imagine the case where the parameter is not part of the ODE, but it still part of the problem as the length of the interval of definition of the independent variable e.g. parameter T in $t \in [0, T]$. In this case, for solving the problem we could make a change of variable, using an **auxiliary independent variable** $t = Ts$, $s \in [0, 1]$, and modify the BVP accordingly.

2.3 Solving optimization problems as BVP's

Suppose we want to optimize a function R. Then, in the optimal point:

$$\nabla R = 0$$

Now, if we want to have an additional restriction on the solution, we could express the restriction as $G = 0$, where G is a function. To get a solution for the minimization problem that satisfies the restriction condition, we need the gradients of both functions R and G to be in the same direction::

$$\nabla R = \lambda \nabla G$$

where λ is a scalar called **Lagrange multiplier**. For compact notation, we can define an additional function:

$$I = R - \lambda G,$$

and state the problem as simply

$$\nabla I = 0$$

Now imagine that the problem could be too formulated as an integral equation. Then the optimization of I could be expressed as:

$$I = \int F\left(x, y, y'\right) dx = \min!$$

The solution to that problem is provided by techniques from **Calculus of Variations**, and is given by the y that solves the **Euler-Lagrange equation**:

$$\frac{\partial F}{\partial y} = \frac{\mathrm{d}}{\mathrm{d}x} \frac{\partial F}{\partial y'}$$

This equation transforms the integral problem into a differential problem. If we additionally had some restrictions for y in different parts of the domain of x, as $y(a) = y_a$ and $y(b) = y_b$, then this optimization problem is equivalent to solving a BVP!

If our problem needs to satisfy some restrictions, then our function F should be of the form $R(x, y, y') - \lambda G(x, y, y')$, where G is the restriction. Note that these R and G are not necessarily the same R and G we were considering before.

The general pipeline would be then:

- First we need a quantity that we want to optimize, and that can be expressed as an integral equation. Say we want to optimize

$$\int R\left(x, y, y'\right) dx = \min!$$

- We construct F as R plus a linear combination of the restrictions
- Solve the Euler-Lagrange equation for F as a BVP problem

Practical Trick One common trick for solving this kind of problems, when the restriction is given in an integral form, is to define an additional differential equation with boundary conditions, which represents the restriction.

2.4 BVPs as optimal control problems

Suppose we have a physical phenomenon described by an ODE:

$$x' = f\left(x(t), u(t)\right),$$

with its respective boundary conditions $x(0) = x_0$, $x(1) = x_1$. Here the differential part of the model is for $x(t)$, however, the problem also considers an additional function $u(t)$, which is called **control**. This could be for example a car moving, where the control is how much the accelerator is being pushed. The equivalent of having a variable parameter.

Suppose we have a quantity of interest from the problem which we want to optimize, for example, minimize the pollution of the car. And that the quantity can be expressed as a function of $x(t)$ and $u(t)$. Then we can formulate the optimization problem:

$$\mathcal{J} = \int_0^1 \varphi\left(x(t), u(t)\right) dt = \min!$$

Until now the approach is very similar to the optimization problem we described in the last section. However, in this case we want to find the function $u^*(t)$ which optimizes $\mathcal{J}$. We could then change our approach. Now we want to solve for u, and our restriction is going to be given by $x' = f$. We define then a Lagrangian (F in the last section):

$$\mathcal{L} = \varphi(x, u) + \lambda\left(f(x, u) - x'\right)$$

We can generalize and consider all x, λ and u as functions. Then we apply Euler-Lagrange equation with respect to each one of them and we get three expressions. For easying notation, we can define the **Hamilton function**[1] as:

$$H(x, \lambda, u) = \varphi(x, u) + \lambda f(x, u)$$

Then our three expressions are:

- w.r.t. x: We call it the **adjoint equation.** $\lambda' = -\frac{\partial H}{\partial x}$
- w.r.t. λ : The original ODE of the problem. $x' = f$
- w.r.t. u: An ODE on u, which solves for u^*. $\frac{\partial H}{\partial u} = 0$. i.e. we would algebraically solve this equation for $u^* = g(x, \lambda)$.

In practice we need to solve this problem with the computer. We have a system of ODE's (adjoint and original) and we have the computation of the optimal u for each of the numerical method's solutions for those two equation.

As this is a boundary value problem, we need an initial solution. To ensure that the set of three initial solutions we have to propose (u, x, and λ) makes sense, this is generally done proposing a clever guess for $u(t)$, and plugging it in the original ODE and the adjoint equation to solve for a corresponding x and λ.

[1] Note that this is very similar to the Lagrangian, just removing the x'.

3 PDE's: Poisson Equation

PDE's have more than one independent variable on a domain Ω with boundary $\delta\Omega$. This is often stated as one independent vector $x \in \mathbb{R}^n$, and time is often separated: $x = (x', t)$. There is no general theory for solving PDE's, even in the linear case i.e. when there is one spacial dimension and one time dimension.

3.1 General Aspects of PDE's

There are several types of PDE's, and each one has a different kind of set of conditions which lead to proper solvable problems.

- **Elliptic PDE's**, such as Laplace Equation or Poisson Equation, need **stationary boundary conditions**, such as Dirichlet or Neumann conditions on $\delta\Omega$.
- **Hiperbolic PDE's** need a time variable, for example the wave equation $u_{tt} = c^2\Delta u$, and need **only an initial condition** $u(x, t = 0) = u_0(x)$, $x \in \Omega$.
- **Parabolic PDE's** model infinite speed of transport of information, for example the heat equation $u_t = k\Delta u$. They need **initial and boundary conditions**, i.e. they need $u(x, t = 0) = u_0(x)$ on Ω, and $u(x, t) = \varphi_t(x)$ on $\delta\Omega$ for $t \geq 0$.

Well-Posedness The PDE is well posed if and only if there **exists** a **unique** solution for every input, and the solution **depends continuously** on the input i.e. noise in data does not blow up in the solution.

Principles of deducing PDE's One common way to deduce PDE's is the **variational principle**, which uses the Euler-Lagrange equations for reformulating an integral optimization problem into a differential problem. It may be useful to use Taylor expansions or other simplifications to be able to formulate a simpler problem.

One important tool used in PDE theory is multidimensional integration by parts:

$$\int_\Omega \frac{\partial u}{\partial x_j} v \mathrm{d}x = \int_{\delta\Omega} uv \cdot n_j d\sigma - \int_\Omega u \frac{\partial v}{\partial x_j} \mathrm{d}x$$

When u is a scalar field, this leads to the **divergence theorem.**

3.2 Poisson Equation: Fast Poisson Solvers

One of the most common PDE's is the Poisson equation. With homogeneous Dirichlet conditions it has the following form:

$$-\Delta u = f, \qquad \text{on } \Omega \subset \mathbb{R}^d$$

$$u|_{\delta\Omega} = 0$$

We will consider the solving of this equation in the case of the hypercube $\Omega = [-1, 1]^d$, for $d = 1$, $d = 2$, and $d > 2$.

3.2.1 1D problem

In the 1D problem, one can discretize and get the operator Δ as a matrix T_n with stencil $[-1, 2, -1]$, a tridiagonal symmetric positive definite matrix.

$$T_n u_n = h^2 f_n,$$

where h is the discretization step. This needs $\mathcal{O}(n)$ flops for solving, because of its sparcity.

$\mathcal{O}(n)$ is pretty cheap, however in the problem with $d > 1$, this nice complexity is not achieved. As an alternative we can diagonalize T_n in the following form:

$$S_n T_n S_n = D_n,$$

where D_n is a diagonal matrix with the eigenvalues of T_n, and S_n is a unitary matrix[2] with the eigenvectors of T_n, called the **discrete sine transform (DST)**. This transformation is useful because the DST can be easily obtained from to the Discrete Fourier Transform (DFT), and we can obtain the DFT with the Fast Fourier Transform algorithm in $\mathcal{O}(n \log n)$ operations. Note that $n \log n$ is bigger than n, so for 1D this is not very useful.

3.2.2 2D problem

In the 2D case, the Poisson equation becomes

$$-\Delta u = (-\frac{\partial^2}{\partial x_1^2} - \frac{\partial^2}{\partial x_2^2})u = f$$

Discretizing, u becomes a matrix, where x_1 varies in the first dimension and x_2 varies in the second dimension. We can apply the T_n matrix to the row indices by multiplying u by the left, and apply it to the columns by multiplying it by the right. The linear system is now:

$$T_n u_n + u_n T_n = h^2 f_n,$$

which has $N = n^2$ unknowns and equations to be solved, and as the system is not tridiagonal anymore (or even triangular), it needs $\mathcal{O}(N^3)$ flops to be solved. However, we can transform the problem with the DST. Multiplying the equation from both sides with S_n we get:

$$D_n \tilde{u}_n + \tilde{u}_n D_n = h^2 \tilde{f}_n$$

As D_n is diagonal, we can solve easily for the element j, k:

$$(\tilde{u}_n)_{jk} = h^2 \frac{(\tilde{f}_n)_{jk}}{\lambda_j + \lambda_k}$$

As an algorithm:

1. Compute $\tilde{f}_n = S_n f_n S_n$ [needs $\mathcal{O}(N \log N)$]
2. Compute $\tilde{u}_n = h^2 \tilde{f}_n / D_n$ (elementwise division) [needs $\mathcal{O}(N)$]
3. Backtransform $u_n = S_n \tilde{u}_n S_n$ [$\mathcal{O}(N \log N)$]

So in total this has $\mathcal{O}(N \log N)$, which is useful even for $N \sim 10^{6-9}$.

[2] Generalization of orthogonal matrices to the complex numbers: $U^\dagger U = I$

3.2.3 Hypercube

The same algorithm of the 2D case can used for $d > 2$, where we have $N = n^d$. The only difference is than the DST is now successily applied to each coordinate, and instead of matrices we have d-dimensional tensors.

Error

In general, the error of this method is given by the error of the central difference approximation made when discretizing Δ, i.e. $\mathcal{O}(h^2) = \mathcal{O}(N^{-2/d})$. Note that this is bad, because when $d \to \infty$, $N^{-2/d} \to 1$, and thus our algorithm has error of $\mathcal{O}(1)$. You decrease your step size by half and you get the same error! This is called the **curse of dimensionality**. If we wanted to achieve an absolute error of $TOL = N^{-2/d} \Rightarrow N = TOL^{-d/2}$, then we need a complexity of $\mathcal{O}(N \log N) = \mathcal{O}(d \cdot TOL^{-d/2} \log TOL)$.

3.3 Poisson Equation: Montecarlo Method "Random Walk on Spheres"

We take as a model the homogeneous Poisson equation (Laplace equation):

$$-\Delta u = 0, \qquad \text{on } \Omega \subset \mathbb{R}^d$$

$$u|_{\delta\Omega} = w$$

Functions that satisfy this equation are called harmonic, and satisfy the **mean-value property**. This property says that the value of $u(x_0)$ is equal to the mean value of $u(x)$ on a sphere with center on x_0. This added with brownian motion magic lead to the following algorithm, called Random Walk on Spheres (WoS):

1. Get the radius of the largest sphere that fits into Ω around x_j (if it's the first step, $x_j = x_0$)
2. Sample uniformily from that sphere to get a new x_{j+1}
3. If $x_{j+1} \in \delta\Omega$ (with a tolerance h), stop and return $w(x_{j+1})$ as an estimate of $u(x)$

Note that this method is only valid for Laplace Equation. If you wanted to solve a Poisson Equation you would have to make a change of variable. For example for solving the problem:

$$-\Delta u = 1, \qquad \text{on } \Omega \subset \mathbb{R}^d$$

$$u|_{\delta\Omega} = 0$$

you could define $u = v + g$, and then set the problem:

$$-\Delta v = 0, \qquad \text{on } \Omega \subset \mathbb{R}^d$$

$$v|_{\delta\Omega} = w$$

This would mean that $1 = -\Delta u = -\Delta v - \Delta g = -\Delta g$, and thus you could would choose any g with $-\Delta g = 1$. Furthermore, in the boundary:

$$0 = u|_{\delta\Omega} = v|_{\delta\Omega} + g|_{\delta\Omega} \to w = -g|_{\delta\Omega}$$

3.3.1 Error

Monte-Carlo methods have a statistical error of $\mathcal{O}(1/\sqrt{N})$, where N is the number of samples taken, and the error of introducing a tolerance in the boundary is $\mathcal{O}(h)$, thus the total error is $\mathcal{O}(1/\sqrt{N}) + \mathcal{O}(h)$. To make both error contributions equally large, we define $h \coloneqq 1/\sqrt{N}$, and finally our error is $\mathcal{O}(1/\sqrt{N})$.

3.3.2 Complexity

The expected number of steps to reach the boundary scales as $\mathcal{O}(d\,|\log h|) = \mathcal{O}(d\left|\log N^{-1/2}\right|) = \mathcal{O}(d \log N)$. Additionally, the number of flops per step is $\mathcal{O}(d)$. So, our total complexity is

$$N \cdot \mathcal{O}(d \log N) \cdot \mathcal{O}(d) = \mathcal{O}(d^2 N \log N)$$

If we want to achieve an absolute error $TOL = 1/\sqrt{N}$, then we have a complexity of $\mathcal{O}(d^2 TOL^{-2} \log TOL)$. Note that for WoS we do not have d in the exponent, whereas in the best grid-based method we have a complexity of $\mathcal{O}(TOL^{-d/2})$. This sets a break even point. For example for $TOL = 10^{-(2or3)}$ the breakeven is in $d \approx 5 - 6$, and thus for that tolerance, if $d \geq 7$, we would go with WoS.

3.3.3 Summary

WoS is nice because it is mildly dependent on d, embarassingly parallel, simple to program and uses only pointwise evaluations. However it has a limited accuracy.

4 PDE's: Finite Element

4.1 Introduction

4.1.1 Weak Formulation

The Finite Element Method works for differential equations with elliptic operators in the divergence form:

$$Pu(x) \coloneqq -\nabla \cdot (\mathbb{A}(x)\nabla u(x)) + a_0(x)u(x)$$

Where $\mathbb{A}(x)$ is called the diffusion matrix, and it is symmetric and uniformly positive definite. One can define a PDE with this kind of operator:

$$Pu = f \ \text{ on } \Omega, \qquad u|_{\Gamma_D} = 0, \ \frac{\partial u}{\partial n_{\mathbb{A}}} = g|_{\Gamma_N}$$

where f is the load function, Γ_D is a Dirichlet part of the boundary $\delta\Omega$ and Γ_N is the Neumann part of the boundary. $n_{\mathbb{A}}$ is the normal direction in the boundary rotated by $\mathbb{A}$ (if $\mathbb{A} = I$, then $n_{\mathbb{A}} = n$).

Our operator P contains second order derivatives (divergence and gradient). This can be problematic if we want to find a piecewise solution to the PDE, because its derivative (or weak derivative) might not be continuous, and thus its second derivative might not be well defined. The idea of the **weak formulation** is to relax this condition, reformulating the problemgoing from having two derivatives to having just one. For this first we are going to integrate the PDE on the whole domain:

$$Pu = f \implies \int_\Omega Puv\mathrm{d}x = \int_\Omega fv\mathrm{d}x$$

Here v is called a **test function**. We introduce v such that $v \in V$, where V is the space of functions which have a value of 0 in the Dirichlet boundary (to satisfy the PDE Dirichlet boundary). Applying integration by parts to the first part of the new approach we can throw one derivative from P to v. Applying the divergence theorem and the boundary conditions, we get the following form:

$$\int_\Omega \langle \mathbb{A}\nabla u, \nabla v\rangle \, dx + \int_\Omega a_0 uv dx = \int_\Omega fv dx + \int_{\partial\Omega} gv d\sigma$$

We notice that the result of the Left Hand Side of the equation is a number, and it is linear w.r.t. both u and v. This is called a **bilinear form.** The RHS is a **linear form** with respect to v. Naming the bilinear form a and the linear form λ, we can reexpress as:

$$a(u, v) = \lambda(v)$$

This is not the only way to get this last equation. An alternative way parts from the definition and the optimization of a quadratic functional dependent on u and $v \in V$. In this approach, the functional is minimized when our PDE is satisfied. However, according to Dirichlet's principle, not every quadratic functional bounded from below has a minimum. This means that our problem does not always have a solution. For ensuring that we can find a solution, we have to restrict the function space V to make the functional to have a minimum. We should have a solution if V is taken large enough, but also we do not want it to be too large because we want u to be unique and computable. **Sobolev spaces** satisfy this conditions.

Intuitively, a Sobolev space is a space of functions with sufficiently many **(weak)** derivatives for some application domain, such as partial differential equations, and equipped with a norm that measures both the size and regularity of a function.

For defining mathematically a Sobolev space, first we have to define the $L^2(\Omega)$ space. $L^2(\Omega)$ is a set of functions that map $\Omega \to \mathbb{R}$, and have finite squared norm. Functions in $L^2(\Omega)$ are equipped with a norm.

A Sobolev Space is notated as $H^m(\Omega)$, and consists on a subset of $L^2(\Omega)$, such that its functions have existing weak derivatives up to order m in $L^2(\Omega)$. The norm of a function w.r.t. a Sobolev Space $H^m(\Omega)$ is the sum of all the L^2 norms of its derivatives of order 0 to m.

For our particular problem, as we have now only first order derivatives, we only need to have one weak derivative well defined, and thus a Sobolev space $H^1(\Omega)$ is enough.

Another mathematical formality is that our boundary $\delta\Omega$ is one dimension less than our domain Ω, and our space of functions is defined only for Ω. This requires the existence of a trace map γ to be able to go from $H^1(\Omega)$ to $L^2(\delta\Omega)$. Note that in the boundaries we do not need to compute the derivatives, and thus $L^2(\delta\Omega)$ is enough.

Adding these restrictions on V to our approach, we get the **weak formulation:**

$$a(u,v) = \lambda(v), \qquad v \in V \coloneqq \left\{ v \in H^1(\Omega) : \ \gamma(v)|_{\Gamma_D} = 0 \right\}$$

If the problem has a solution, the weak formulation is equivalent to the original PDE. The belonging of our test functions to $H^1(\Omega)$ and the existence of the trace map γ are mere mathematical formatlities. In practice, choosing a pretty normal function space as a polynomial space would satisfy these restrictions automatically.

4.1.2 Ritz-Galerkin Method

For putting the ideas of the weak formulation in the computer, we need to define a finite subspace of functions. The idea of the Ritz-Galerkin method is to take a finite dimensional function subspace $V_h \subset V$, where h aludes to a grid size. Being $\{\varphi_i\}$ the basis of V_h, then we could express u_h and v_h (which are discrete representations of u and v) as a linear combination of these basis functions. Plugging these into our weak formulation would result in the definition of a linear system of equations $A_h u_h = f_h$.

The process or getting this discrete form is called **assemblying.** A_h is called the **stiffness matrix**, it is the result of assemblying the bilinear form, which means that it is symmetric and positive definite. On the other hand,f_h is the result of assemblying the linear form, and is called the **load vector**.

4.1.3 Finite Element Method

The Finite Element Method (FEM) would consist in applying the weak formulation to a PDE, then applying the Ritz-Galerkin Method and then solving for the discrete form of the problem.

For any problem that we would like to solve with FEM, defined in Ω, we would have the following steps to follow:

1. Choose the suitable geometric finite element spaces to represent our discretized domain (shape of the mesh elements)
2. Choose the suitable function spaces $V_h \subset H^1(\Omega)$ to represent our solution
3. Assembly (set up the linear system $A_h u_h = b_h$)
4. Solve the linear system
5. Estimate the error of our solution (a posteriori)
6. Adaptively refine V_h until reaching a desired error criteria

The following sections elaborate on these steps.

4.2 Choosing the Finite Element Spaces

For describing our domain, we must divide it in cells. For example, in 2D we could represent our domain with only triangles, only hexagons, or any combination of 2D figures. We call this discretization a **mesh**.

Here are some notations/conventions/considerations for the geometric aspects of FEM:

- There can be several refinement levels, which would lead to different accuracy levels. The set of cells at a certain level h is notated as J_h, and a specific cell of J_h as T.
- h represents the maximum diameter of a cell $T \in J_h$.
- The union of all the cells is an approximation of our domain, notated by $\bar{\Omega}$, and the intersection between different T_j is a null or lower dimensional space representing a boundary between cells.
- V_h is the space of functions on J_h. If the functions are piecewise polynomials it is denoted as X_h. A specific polynomial is $P_T = \{v_h|_T : v_h \in X_h\}$.
- A mesh can be characterized with its number of nodes, number of edges and number of cells. These variables are notated by n_0, n_1 and n_2 respectively.
- One could express restriction on the admissibility of a mesh in terms of the mesh variables. This restriction would be made through the degrees of freedom of the piecewise polynomial space, notated as $\dim X_h$. For example, if the piecewise polynomials chosen are of the Lagrange family of order m, then J_h is an admissible mesh if $\dim X_h = n_0 + (m-1)n_1 + \frac{1}{2}(m-1)(m-2)$. If they are chosen Cubic Hermite, which means having information of the value in each node and the derivative in each node, and an extra center of gravity, we would need $\dim X_h = 9n_0 + n_2$.
- In every 2D domain, the following geometric relation is satisfied:

  $$n_0 - n_1 + n_2 = 1 - n_{holes} =: \chi$$

  n_{holes} is the number of physical holes inside the domain (e.g. a doughnut shape would have one hole), independently of their size. χ is called the **Euler characteristic**, and it is a property of the domain, independently of its subdivision.

4.3 Choosing the function space $V_h \subset H^1(\Omega)$

As we have mentioned, the formal mathematical requirements of the weak formulation, as the belonging of our test functions to $H^1(\Omega)$ and the existence of the trace map γ are intrinsically satisfied by polynomial spaces.

These polynomials are chosen to be piecewise, to allow geometric flexibility and numerical stability. Furthermore, the fact of being polynomials allows different approximation powers by interpolation.

4.4 Assemblying the linear system

Assemblying means setting up the linear system $A_h u_h = b_h$. This requires a choice of basis. As we want A_h to be sparse for storage and computing reasons, we would like to choose a basis that is as local as possible.

One example of a choice for a basis (and one of the most standard) is the **nodal basis**. It is defined:

$$\varphi_j(x_k) = \delta_{jk}$$

This basis is commonly utilized because the coefficients of u_h with respect to this basis are just the function values at the grid points, so u_h is then just the piecewise interpolation of the values in the nodes.

However, independently of the chosen basis, the process of assemblying the system is the same. These are the two main ideas behind the process:

- Both the bilinear and the linear form are integrals in the whole domain and the boundary. As we have our domain discretized, we can express the integrals over the whole domain as a sum of integrals in each element $T \in J_h$.
- Every cell will be different (different size, angles, location, etc.), which complicates the problem. A solution for this is to transform T to a reference cell $\hat{T}$, to reach a simple, efficient framework and to be able to pre-compute a lot[3]. One can define this transformation to the reference cell with a map given by a Jacobian of the transformation, often noted in this context as B_T.

4.4.1 Load Vector

In the weak formulation, we had the load function in the term $\int_\Omega f v dx$[4]. After applying the Ritz-Galerkin method (expressing v as a linear combination of the basis of the function space V) we now want to find $(b_h)_j = \int_\Omega f \varphi_j dx$

We can express f too as a linear combination of our basis functions $f \approx \sum_k \alpha_k \varphi_k$ and then compute all the possible combinations of $\langle \varphi_j, \varphi_k \rangle$. For computational reasons, instead of iterating over j, i.e. going element by element of the resulting vector, and then check the cells in which the integral is not zero, we iterate once over all cells and then update accordingly. As an algorithm:

- $b_h = 0$
- for all $T \in J_h$:
  - for k, j being nodes of T :
    - update $(b_h)_j \to (b_h)_j + |\det B_T| \, M \alpha_k$

Where $M = \int_{\hat{T}} \hat{\varphi}_j \hat{\varphi}_k \mathrm{d}\hat{x}$ is precomputed within the reference cell using reference functions $\hat{\varphi}_i$, and it has invariant information about the integrals for different configurations.

4.4.2 Stiffness Matrix

One can iterate over the elements of the matrix, to have

$$\int_T \langle \nabla \varphi_k, \nabla \varphi_j \rangle \, \mathrm{d}x = |\det B_T| \int_{\hat{T}} \left\langle \left(B_T^T\right)^{-1} \nabla \hat{\varphi}_k, \left(B_T^T\right)^{-1} \nabla \hat{\varphi}_j \right\rangle \mathrm{d}\hat{x}$$

4.4.3 Matrix-Free Assembly in FEM

If the linear system $A_h u_h = b_h$ gotten from a FEM analysis of a PDE is solved by an iterative method, that employs just matrix-vector multiplications of the form $A_h r_h$, then a black-box for setting up the operation $r_h \to A_h r_h$ would suffice to solve it. This means that we do not need to assembly the matrix A_h, saving storage and possibly computing time. This way of computing also allows to reuse code from the assemble of the load vector. This is useful when the number of iterations k for solving the linear system is not too large:

$$k \lesssim \frac{1}{2} c_A \frac{\mathrm{nnz}(A)}{\#DOF}$$

where $\#DOF$ is the number of nodes, and $\mathrm{nnz}(A)$ is the number of nonzero values of A_h. c_A is a constant typically between 1 and 2.

[3] In FENICS the precomputing is done in the JIT-compile step

[4] Considering for simplicity homogeneous Neumann conditions

4.5 Solving the Linear System. Iterative Solvers and Conjugate Gradient

We part from the linear system of equations we got from the FEM approach, for a PDE with boundary conditions valid on $\mathbb{R}^d$:

$$A_h u_h = b_h,$$

where A_h is symmetric, positive definite and sparse. The dimension of the system is $N = \mathcal{O}(h^{-d})$. One way to solve this system would be to get the Cholesky decomposition of A_h, however, as the dimension of the system increases this starts being too computationally expensive.

The idea of an iterative method consists on having a solver that gets better with time, and that can be interrupted. We can compute the number of iterations necessary in order to reach the error we would get by solving the system directly, called **discretization error**[5]. The following table includes relevant information for some iterative solvers:

Method	**FLOPS (2D)**	**FLOPS (3D)**	**Iters. to reach discr. error**
Gauss-Seidel method	$\mathcal{O}(N^2)$	$\mathcal{O}(N^{5/3})$	$\mathcal{O}(h^{-2})$
Krylov-space method: CG	$\mathcal{O}(N^{3/2})$	$\mathcal{O}(N^{4/3})$	$\mathcal{O}(h^{-1})$
Multi-grid/level methods	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(1)$

4.5.1 Preconditioning

The number of iterations needed to reach a certain RelTOL is dependent on the condition number of the matrix ($\mathcal{O}(\sqrt{\kappa(A_h)})$). If we reduced the condition number we could reduce the number of iterations needed. This could be achieved by multiplying our equation by a matrix B_h:

$$B_h A_h u_h = B_h b_h$$

Where B_h is defined such that $\kappa(B_h A_h) \ll \kappa(A_h)$. The optimal thing would be to choose A_h^{-1} to get a condition number of 1, but that is unrealistic.

4.5.2 Classical Iterative Methods

Classical Iterative Methods consist on decomposing the matrix A. One option is $A = M - N$, which would yield:

$$Mx = Nx + b$$

We can establish with this equation a fixed point iteration and get

$$x^{k+1} = M^{-1}Nx^k + M^{-1}b$$

If we define M as the diagonal of the matrix A (call it D), and $-N$ to be the rest, then we would easily get $M^{-1} = D^{-1}$. This is called the **Jacobi Method.**

If we instead choose M to be the diagonal **and** the left triangular part of A (call them D and $-L$ respectively), then we would get $M^{-1} = (D-L)^{-1}$. This may sound like a difficult matrix to invert, but this configuration means that $M^{-1}r$ is simply a forward substitution, which would be $O(N)$ because of the sparsity. This is called **Gauss-Seidel Method.** A variant of that would be its **symmetric version** which consists on using $(D-R)^{-1}D(D-L)^{-1}$.

[5]This is our better approximation of the solution

Incomplete Cholesky Decomposition As we have commented, Cholesky decomposition is too expensive for large N, however we could define an alternative Cholesky decomposition where

$$A_h \text{ " = " } \tilde{L}_h \tilde{L}_h^T,$$

where that quoted equality is valid only in the non-zero elements of A_h. This provides a preconditioner $B = \tilde{L}\tilde{L}^T$,called **ICC preconditioner**. This takes $\mathcal{O}(N)$ operations.

4.5.3 Conjugate Gradient method (CG)

Conjugate Gradient method is an iterative algorithm to find the numerical solution of systems of linear equations whose matrix is symmetric and positive-definite. We can apply Galerkin's idea of taking the inner product of the whole equation with a vector y to get:

$$Ax = b$$

$$\langle Ax, y\rangle = \langle b, y\rangle$$

$$a(x_m, y_m) = \lambda(y_m), \quad \forall y_m \in V_m$$

The idea in CG is to define $V_m = \text{span}\left\{b, Ab, AAb, ..., A^{m-1}b\right\}$, where m would be the number of iteration. This provides that one only needs A and b to generate V_m.

The CG algorithm is the following:

- Define initial x_0, $r_0 = b - Ax_0$, $p_1 = r_0$
- for $m = 1, 2, 3, ...$
  - $\alpha_m = \frac{\langle r_{m-1}, r_{m-1}\rangle}{\langle p_{m-1}, Ap_m\rangle}$
  - $x_m = x_{m-1} + \alpha_m p_m$ (Update x)
  - $r_m = r_{m-1} - \alpha_m A p_m$ (Update residual)
  - $\beta_m = \frac{\langle r_m, r_m\rangle}{\langle r_{m-1}, r_{m-1}\rangle}$ (How much did the residual changed in the last iteration)
  - $p_{m+1} = r_m + \beta_m r_m$ (Modified residual that includes information about the change of the residual with respect to the last iteration)

Complexity The complexity of this method is the number of iterations it takes times the complexity per iteration.

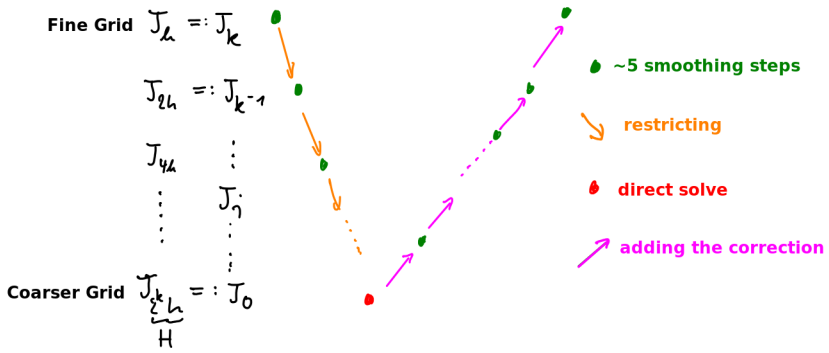
Each iteration needs $\mathcal{O}(N)$ flops for the inner product and scalings, plus the flops needed for matrix-vector-multiplication (which is $\mathcal{O}(N)$ in FE/FD methods, because of sparsity). So every iteration needs in total $\mathcal{O}(N)$.

The number of iterations needed with this method to reach a Relative Residual of TOL is $k \approx \frac{1}{2}\sqrt{\kappa(A_h)}\log\frac{2}{TOL}$. As the condition number of this kind of matrix is $\mathcal{O}(h^{-2})$, then the number of iterations is $\mathcal{O}(h^{-1})$.

In total, the algorithm needs $\mathcal{O}(h^{-1}N) = \mathcal{O}(N^{1+1/d})$

4.5.4 Multigrid Methods

The general idea of Multigrid Methods is to use the following procedure **as a preconditioner**: use a fixed number of smoothing steps (i.e. making the residual smooth with for example 5 Gauss-Seidel iterations) on a fine grid ($\mathcal{O}(1)$ operations), then switch to a coarser grid to make the residual rough again and repeat. The process is the following:



When you make a smoothing step, the high frequency residuals are eliminated more quickly than the low frequency residuals. Thus, if you eliminate the high frequency residuals, then go to a coarser grid and eliminate its high frequency residuals (that are now of lower frequency than the ones eliminated before) and then repeat the process, by the time you get to the coarsest grid you have eliminated the lowest residuals of the finest grid. Then, for the coarsest grid, you can easily apply a direct solver, and after that start refining again. The refinement would be done by taking the grid and the solution that was already computed in that grid level, adding a correction corresponding to the coarser grid, and smoothing.

This is called a V-cycle, and can be used as a preconditioner for CG. The complexity of the V-cycle is $\mathcal{O}(N_k) = \mathcal{O}(2^{dk}N_0)$, where N_0 is the number of points in the coarser grid.

The restriction and correction steps are very simple if we arrange for the FE spaces to have nested functions. That is why Multigrid Methods use a different basis of functions than the one we usually utilize in FEM. We use a **hierarchical basis** instead of a nodal basis:

At the left of the picture we have the nodal basis and at the right the hierarchical basis. In the hierarchical basis each color represents a different level.

For the case of the Poisson equation, applying the bilinear form in the hierarchical basis results in a diagonal A_h matrix. This means that if we take $D_h = A_h^{-1}$ as a preconditioner, we would get $B_h A_h = I$, which implies a condition number of 1. In general for elliptic PDEs in 1D, the matrix is no longer diagonal, but still the condition number that results of its preconditioning is independent of h (it is $\mathcal{O}(1)$).

There could be the case where we do not have access to a history of grid refinement. In this case we could apply an alternative method called **Algebraic Multi-Grid Method (AMG)**, which builds a refinement structure from the sparsity pattern of A_h on a purely algebraic level.

There could be also the case where there can be refinement but you cannot or do not want to keep it in storage. A well suited method for this case is the **Cascadic Multigrid.** This consists on **starting** with a coarse grid and solving the system directly. Then take that solution and interpolate it to a finer mesh, to take as initial guess for an iterative method (a preconditioned CG); then repeat the process until reaching the finest desired grid. Note that this method does not require the storing of all the past grid levels as the original multigrid. Also, this method does a CG in every level, while in the original V-cycle the CG is done until the end.

4.6 Error Estimates

From interpolation theory, we are supposed to know that to interpolate a function u using V_h, we get the *maximum* error

$$\|u - w_h\|_\infty \leq ch^{k+1}\|u^{(k+1)}\|_\infty,$$

where $w_h \in V_h$ is the interpolation of u. This would lead to think, as in FEM we are making an interpolation using V_h, that the error would be too $\mathcal{O}(h^{k+1})$.

Let us have $u \in V$ and $u_h \in V_h$. We compute the weak form for both:

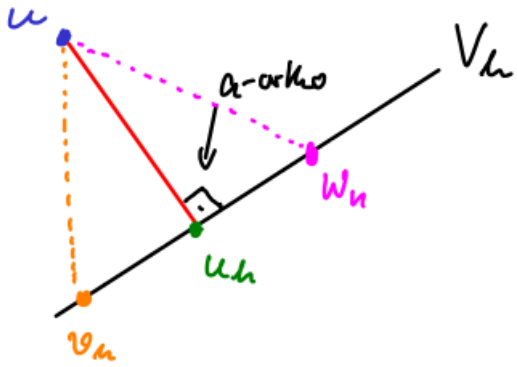
$$a(u, v_h) = \lambda(v_h)$$

$$a(u_h, v_h) = \lambda(v_h)$$

If we substract them we get that $a(u - u_h, v_h) = 0$, which means that the error is orthogonal to V_h with respect to a-norm. [6]This property of the error is called the **Galerkin orthogonality of the FE solution.** Note that this means that u_h is the projection of u on V_h, and thus:

$$\|u - u_h\|_a = \min_{v_h \in V_h} \|u - v_h\|_a$$

One can see this geometrically with the following figure:



Note that we could take the same approach for the interpolation w_h, which gives us the restriction $\|u - u_h\|_a \leq \|u - w_h\|_a$.

As we know, $\|u - w_h\|_\infty = \mathcal{O}(h^{k+1})$, and as the a-norm measures also the derivative, we loose one power of h, so we get $\|u - w_h\|_a = \mathcal{O}(h^k)$, and thus

$$\|u - u_h\|_a = \mathcal{O}(h^k)$$

$\|\cdot\|_a$ and $\|\cdot\|_{H^1}$ are equivalent up to constants[7], thus we have

$$\|u - u_h\|_{H^1} = \mathcal{O}(h^k)$$

This is valid for $u \in H^{k+1}$, which means that u is $k+1$ times differentiable. For example, for H^2 regular problems (which would need choosing $k = 1$ polynomials), we would have $\|u - u_h\|_{H^1} = \mathcal{O}(h)$.

For other norms as L^2 and L^∞, the approach to get the error is completely different.

The L^2 norm in general would be:

$$\|u - u_h\|_{L^2} = \mathcal{O}(h^{k+1})$$

[6]In applications, $\|\cdot\|_a$ often has a physical meaning. Note thas this norm is the bilinear form, which is computing an integral of the inner product of the gradients of two functions.

[7]This is a consequence of the continuity of $a(\cdot,\cdot)$ and the H^1 ellipticity.

4.6.1 Local Residual Based Error Estimation

Sometimes we want to estimate the error locally, to see which parts of the domain need to be refined. For doing this we take as a model problem the Poisson equation with homogeneous Dirichlet conditions on the boundary,

$$-\Delta u = f \text{ on } \Omega, \qquad u|_{\delta\Omega} = 0$$

which has a weak form

$$a(u, v) = \lambda(v)$$

The error would be given by $e = u - u_h$. For getting a sense of the size of e without knowing the exact solution u, we could probe it with a test function:

$$a(e, v) = a(u, v) - a(u_h, v) = \lambda(v) - a(u_h, v)$$

$$= \int_\Omega f v \mathrm{d}x - \int_\Omega \nabla u_h \nabla v \mathrm{d}x = \sum_{T \in J_h} \left(\int_T f v \mathrm{d}x - \int_T \nabla u_h \nabla v \mathrm{d}x \right)$$

Applying integration by parts and regrouping:

$$a(e, v) = \sum_{T \in J_h} \left(\int_T (f + \Delta u_h) v \mathrm{d}x - \int_{\delta T} \vec{n} \cdot \nabla u_h v \mathrm{d}s \right)$$

We can define $R(u_h) \coloneqq f + \Delta u_h$ and $r(u_h) \coloneqq -\vec{n} \cdot \nabla u_h$. Note that these two quantities are totally computable from our solution u_h. R represents the local residual of the PDE, and r the jump of the derivative in the normal direction. Then we get:

$$a(e, v) = \sum_{T \in J_h} \left(\langle R, v \rangle_{L^2(\Omega)} + \langle r, v \rangle_{L^2(\Omega)} \right)$$

Because of the Galerkin orthogonality ($(u - u_h) \perp_a V_h$), we could add any function $\varphi_h \in V_h$ without affecting the formula:

$$a(e, v) = \sum_{T \in J_h} \left(\langle R, v - \varphi_h \rangle_{L^2(T)} + \langle r, v - \varphi_h \rangle_{L^2(\delta T)} \right)$$

Then by Cauchy-Schwarz inequality:

$$|a(e, v)| \leq \sum_{T \in J_h} \left(\|R\|_{L^2(T)} \|v - \varphi_h\|_{L^2(T)} + \|r\|_{L^2(\delta T)} \|v - \varphi_h\|_{L^2(\delta T)} \right)$$

Note that we are with the inequality sacrificing accuracy for the error estimation, but we can counteract that by choosing a nice pair of v and φ_h such that $\|v - \varphi_h\|$ is small. We choose φ_h to be the nodal interpolation of v. Then, with some fancy trace theorem and another Cauchy-Schwarz inequatlity, we get:

$$|a(e, v)| \leq c \sqrt{\sum_T \eta_T^2} \, |v|_{H^1(\Omega)}$$

$$\eta_T^2 \coloneqq h_T^2 \|R\|_{L^2(T)}^2 + h_T \|r\|_{L^2(\delta T)}^2$$

Where η_T is a purely local indicator, and as it depends only on R, r, and the maximum diameter of the cell h_T, it is **computable**.

This is valid for H^2 regular problems. Now we only have to choose v for estimating our error. Choosing $v = e$, and using the equivalence between the energy norm and the H^1 norm, we get:

$$\|e\|_{H^1} \leq c\sqrt{\sum_T \eta_T^2}$$

Note that by making η smaller we guarantee that the error gets smaller, which gives us **reliability** of the error estimate. Note that as η_T^2 is $\mathcal{O}(h^2)$, then we have

$$\|e\|_{H^1} = \mathcal{O}(h)$$

Computing η

Recall that

$$\eta_T^2 = h_T^2 \int_T R^2 dx + h_T \int_{\delta T} r^2 ds$$

For computing this efficiently and be able to reuse our assembling code, we want to probe it with a test function:

$$H(v_T) = \eta_T^2 = \sum_T \left(h_T^2 \int_T R^2 v_h dx + h_T \int_{\delta T} r^2 v_h ds \right)$$

Where v_h would have to be piece-wise **constant**, for not modifying our result, with a basis:

$$v_T(x) = \begin{cases} 1 & x \in T \\ 0 & x \notin T \end{cases}$$

Furthermore, for evaluating the surface integral, we have to specify the computer the computation of the interior edges. With this our computable version of η would be:

$$\eta_T^2 \overset{\text{assembly}}{=} \sum_T \left(h_T^2 \int_T R^2 v_h dx + \sum_{\text{int edge}} \int_{\text{edge}} r^2 \left(h_T^+ v_h^+ + h_T^- v_h^- \right) ds \right)$$

4.6.2 Goal-oriented error control: dual weighted residuals

Suppose that you have a functional $J(v)$ which you want to evaluate in the domain. You have a PDE, which has a real function u, and we can solve for a solution u_h, however, more than in the solution per se, we are interested in evaluating the functional J. Then for evaluating your error you are interested in evaluating $J(u - u_h) = J(e)$.

For estimating $J(e)$ you take our FE machinery and adapt it. We have formulas for evaluating $a(e, v)$, so we look for a function $w \in V$ such that $J(e) = a(e, w)$. For being able to formulate a weak problem, we generalize e to v :

$$J(v) = a(v, w) \quad \forall v \in V$$

Then we have a weak formulation that we want to solve, but now for w. This is called the **dual weak problem**. Note that in our original weak problem we had $\lambda(v) = a(u, v)$. In our case, as our bilinear form is symmetric, the problem is exactly the same, but with different linear form.

Simmilarly as in the residual based error control, we can express:

$$|J(e)| = |a(e, w)| \leq \sum_{T \in J_h} \left(\|R\|_{L^2(T)} \|w - \varphi_h\|_{L^2(T)} + \|r\|_{L^2(\delta T)} \|w - \varphi_h\|_{L^2(\delta T)} \right)$$

This can be furtherly bounded to get:

$$|J(e)| \leq \sum_{T \in J_h} \rho_T \omega_T$$

$$\rho_T \coloneqq \|R\|_{L^2(T)} + \frac{1}{h_T}\|r\|_{L^2(\delta T)}$$

$$\omega_T^2 \coloneqq h_T^3 \|r(w_h)\|_{L^2(\delta T)}^2$$

Where ρ_T is called the residual, and is computable from the solution u_h (note that the computation of ρ_T is very similar to η_T, with just an h_T^2 factor of difference), and ω_T are called the effective weights which includes the infromation original stated as norms of $w - \varphi_h$, and thus it is dependent only on w_h.

In summary, if we have a functional of interest J, we could compute its error by solving the primal problem $\lambda(v) = a(u, v)$ and the dual problem $J(v) = a(v, w)$. With that we could get two solutions: u_h from the primal problem and w_h from the dual problem. Then the error would be given by:

$$|J(e)| \leq \sum_{T \in J_h} \rho_T \omega_T$$

where ρ_T is computed with u_h, and ω_T is computed with w_h.

Example for getting the L^2 error We could use this machinery for getting an expression for the L^2 error of our solution, if we choose the functional to be $J(e) = \|e\|_{L^2}$:

$$\|e\|_{L^2} \leq c \sqrt{\sum_T h_T^4 \rho_T^2} = \mathcal{O}(h^2)$$

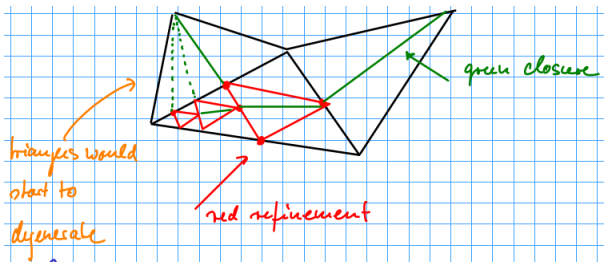
Compare against the $\mathcal{O}(h)$ of the H^1 error of the solution. Note that this is valid for H^2 regular problems.

4.7 Adaptivity

Adaptivity helps to get optimal complexity if Ω has localized source terms or geometric singularities. The steps of this process are:

1. Estimate the local error in every cell
2. Mark the cells where the error is too large
3. Refine the marked cells
4. Solve for the new mesh

There are two kinds of refinements: red refinement and bisection. Red refinement, for example in a triangular mesh, consists on making a new triangle inside the triangle that needs to be refined, with vertices in the middle of the original edges. The addition of those new nodes needs addition of additional triangles in the neighbor cells. This is called green closure.



As red triangles start getting smaller, the green triangles start degenerating, which means having higher error and thus they might have to be replaced with red triangles. This means that this kind of refinment might spread into Ω.

Bisection is simpler, just bisecting the longest edge of a triangle, which allows locallity.

4.8 FEniCS

Here is a model problem for FEM using FEniCS:

```
from fenics import *
import mshr

# Define the mesh
circle = mshr.Circle(Point(0,0), 1)
rectangle = mshr.Rectangle(Point(-2,-2), Point(0,0))
mesh = mshr.generate_mesh(circle-rectangle, 20)

# Define the function space
V = FunctionSpace(mesh, 'P', degree)

# Define instances of the trial and test functions
u = TrialFunction(V)
v = TestFunction(V)

# Define problem (f)
f = Expression('exp(a*x[0])',a=3, degree=1)

# Define problem (boundary conditions)
g_d = Expression('sin(x[1])', degree = 1)
g_n = Experession('cos(x[0]+x[1])', degree=1)
dirichlet_b = lambda x, onb: x[0]<FENICS_EPS and onb
bc = DirichletBC(V, g_d, dirichlet_b)

# Define problem (bilinear and linear form)
D = as_matrix([[2,0], [0,1]])
a=inner(D*grad(u),grad(v))*dx
L = f*v*dx + g*v*ds

# Assembly if necessary
A = assemble(a, V)
b = assemble(L, V)

# Define instance of function
u = Function(V)

# Define Solver if necessary
solver = KrylovSolver("cg", "icc")
solve.parameters[“maximum_iterations”]=10000
solve.parameters[“relative_tolerance”]=5e-3

# Solve the linear system
solve(a==L, u, bc)   # or
solver.solve(A, u.vector(), b)

# Compare
mesh.hmax()
errornorm(u, u_an, norm_type='H1', degree_rise=0, mesh=mesh)

# Compute local errors
Constants = FunctionSpace(mesh, "DG", 0)
v = TestFunction(Constants)
h = CellSize(mesh)
form = h**2 * R**2 * v * dx + r**2 * (h('+') * v('+') + h('-') * v('-')) * dS
eta2 = assemble(form)
error_estimate = sqrt(sum(eta2_T for eta2_T in eta2))

# Adapt if necessary
if error_estimate > tolerance:
    largest_eta = max(eta)
    cell_markers = MeshFunction("bool", mesh, mesh.topology().dim())
    for T in cells(mesh):
        cell_markers[T] = eta[T.index()] > (fraction * largest_eta)
    mesh = refine(mesh, cell_markers)
    # And repeat the solving again
```