An introduction to Computational mechanics

Alessandro Tasora
alessandro.tasora@unipr.it

May 29, 2020

Abstract

This document contains some short notes for the seminar on Computational Mechanics for the PhD students at the University of Parma. It starts covering a review of tensor notation, then it moves to PDE applications in engineering, such as FEA and CFD, highlighting how in many cases they share a common underlying mathematical structure. Then it presents basic concepts about strong-weak formulations and discretization, and it discusses numerical methods for solving large scale linear problems. Finally it presents a review of the most relevant integration methods for time-dependant problems and it discusses the implementation details of for index-2 ODEs and DAEs that are of particular interest for the mechanical engineering community.

Note: this is a first release, it will be extended in future. Please report typos and suggestions for future revisions.

1. Introduction

Most problems in applied and theoretical mechanics share a common structure, ad require going through three steps:

- find a mathematical formulation of the physical problem
- find a discretized version of the formulation
- implement solvers for the discretized formulation

An interesting aspect is that, even if applied to completely different contexts, a problem of fluid dynamics, a problem of heat diffusion or a problem of elasticity might share the same mathematical formulation. Often these formulations are expressed in differential form. This will be discussed in the section on *Classes of problems*.

One of the biggest challenges of computational mechanics (and of computational sciences in general) is how to transform those mathematical formulations into *discretized* problems with a *finite* number of unknowns. This is the rationale of the finite element methods, or the finite volume methods, or the meshless particle methods, for example. This will be discussed in the sections on *Discretization*. Additionally, time-dependant problems might require a time integration process: this is discussed in the *Time integration* section.

Going a step further: once the problem is discretized and turned into a solution procedure, such procedure might embed numerical problems that often fall into a limited number of classes - for instance finite element methods require the solution of a system of linear equations. This brings in some issues: how to efficiently solve a linear system? How to handle very large and sparse matrices? Topics like this are discussed in the section on *Linear solvers* etc.

The following sections represent a primer on computational mechanics, starting from general general mathematical tools such as tensor notation and vector algebra, then showing a taxonomy of mechanical problems, and finally presenting some insight on the numerical methods that can solve computational primitives that often are encountered while solving computational mechanics: sparse large linear systems, performing time integration, etc.

2. Notation, tensors and other mathematical tools

In the following we list some basic notations that will be used later when dealing with computer methods. When possible, we highlight the similarities between the vector/matrix notation and the tensor notation.

2.1 Symbols

A cheat sheet with notation that will be used in the following:

- s is a scalar
- $\boldsymbol{v}$ is a generic vector
- $\boldsymbol{v}^*$ is a covariant vector (or covariant base vector)
- $\underline{\boldsymbol{v}}$ is a contravariant vector (or contravariant base vector)
- A is a matrix
- A_{ij} is a (scalar) element of matrix A
- $\boldsymbol{T}$ is a generic tensor
- $T^i{}_j{}^k$ is a (scalar) component of a tensor $\boldsymbol{T}$
- $\underline{\boldsymbol{t}}$ is an euclidean tensor, rank 1.
- $\underline{\underline{\boldsymbol{T}}}$ is an euclidean tensor, rank 2.

2.2 Index notation

First, we need to introduce the index notation.

Index notation is used as an alternative to the conventional notation in matrix algebra. It is used also in tensor formulas, and one might be tempted to think that tensors are just like matrices, i.e. multidimensional arrays, however tensors are not just like matrices because they convey more information about the underlying coordinate systems (see contravariance and covariance later), and vice-versa there are matrices that could not be interpreted as tensors.

By the way, index notation (despite more intricate than matrix notation at the first glance) offer more possibilities later when coming to formulas for differential geometry.

- The *Einstein*[1] *summation convention* is used: it assumes that repeated indexes are automatically summed, so the $\sum$ symbol can be omitted:

$$\boldsymbol{v} = \sum_{i=1}^{3} v_i \boldsymbol{e}_i \quad \Rightarrow \quad \boldsymbol{v} = v_i \boldsymbol{e}_i$$

[1] It is remarkable how many bright scientists attended the University of Göttingen.

- With index notation, one can express, for example, a product of matrix by vector in this way:

$$\boldsymbol{v} = A\boldsymbol{b} \quad \Rightarrow \quad v_i = A_{ij} b_j$$

  and a product between matrices as:

$$\boldsymbol{C} = AB \quad \Rightarrow \quad C_{ij} = A_{ik} B_{kj}$$

- The *transpose* of a matrix translates into switching the order of the indexes:

$$C_{ij} = (C^T)_{ji}$$

- The *trace* of a matrix translates into:

$$\mathrm{tr} A = A_{ii}$$

- The *Kronecker delta* is the equivalent of the identity matrix $I_{ij} = \delta_{ij}$ in index notation:

$$\delta_{ij} = \begin{cases} 0 & \text{if } i \neq j, \\ 1 & \text{if } i = j. \end{cases}$$

- With index notation, and with tensors in general, one does not use bold symbols because all formulas are effectively speaking about scalars (ex. the i-th element of a vector, in formula above, where the fact that i expands into multiple values is implicitly assumed by the *range convention*).

- Index notation can be easily translated into computer code: *Einstein convention* and *range convention* lead to `for` loops, for instance

$$D_{ij} = A_{ik} B_{kl} C_{lj} \quad \Rightarrow \quad D_{ij} = \sum_{k=1}^{m} \sum_{l=1}^{m} A_{ik} B_{kl} C_{lj}$$

  where the two $\sum$ become two nested `for` loops with sums (also *reduction* primitives in a parallel computing environment), all enclosed in two additional `for` loops over the i and j indexes for storing results of sums into the C data structure.

- The `for` loops with sums can become *reduction* primitives in a parallel computing environment.

- The repeated indexes are called *dummy indexes*.

- The non-repeated indexes are called *free indexes*.

- In a tensor expression indexes are repeated maximum two times (if dummy indexes).

- The *outer product* is also called dyadic product or tensor product. Assuming vectors with basis $\boldsymbol{e}_i$, it is:

$$\boldsymbol{a} \otimes \boldsymbol{b} = \boldsymbol{a}\boldsymbol{b}^{\mathsf{T}} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \begin{pmatrix} b_1 & b_2 & b_3 \end{pmatrix} = \begin{pmatrix} a_1 b_1 & a_1 b_2 & a_1 b_3 \\ a_2 b_1 & a_2 b_2 & a_2 b_3 \\ a_3 b_1 & a_3 b_2 & a_3 b_3 \end{pmatrix}. \quad (1)$$

$$[\boldsymbol{a} \otimes \boldsymbol{b}]_{ij} = a_i b_j \quad (2)$$

$$\boldsymbol{a} \otimes \boldsymbol{b} = a_i b_j \boldsymbol{e}_i \otimes \boldsymbol{e}_j \quad (3)$$

- Some properties of dyadic product:

$$(\mathbf{u} \otimes \mathbf{v})^{\mathsf{T}} = (\mathbf{v} \otimes \mathbf{u}) \quad (4)$$

$$(\mathbf{v} + \mathbf{w}) \otimes \mathbf{u} = \mathbf{v} \otimes \mathbf{u} + \mathbf{w} \otimes \mathbf{u} \quad (5)$$

$$\mathbf{u} \otimes (\mathbf{v} + \mathbf{w}) = \mathbf{u} \otimes \mathbf{v} + \mathbf{u} \otimes \mathbf{w} \quad (6)$$

$$c(\mathbf{v} \otimes \mathbf{u}) = (c\mathbf{v}) \otimes \mathbf{u} = \mathbf{v} \otimes (c\mathbf{u}) \quad (7)$$

2.3 Contravariant and covariant transformations

- We introduce a *manifold* M parametrized by n coordinates θ_i.
- When changing coordinates from θ_i to new coordinates $\hat{\theta}_i$, we say that an entity such as a vector a_i changes with a *contravariant transformation rule* if it holds:

$$\hat{a}^i = \frac{\partial \hat{\theta}_i}{\partial \theta_j} a^j$$

- We say that an entity a_i changes with a *covariant transformation rule* if it holds:

$$\hat{a}_i = \frac{\partial \theta_j}{\partial \hat{\theta}_i} a_j$$

- Indexes denoting covariant components are denoted with *superscripts*. Indexes denoting contravariant components are denoted with *subscripts*. More on this later, when discussing tensors, that can have multiple indexes and mixed contravariant/covariant properties.
- Example: the derivative of a scalar function, $a_j = \frac{\partial f(\cdot)}{\partial \theta_j}$, transforms covariantly. Same for base vectors $\boldsymbol{g}_i^* = \frac{\partial}{\partial \theta_i}$, tangents to the (maybe curvilinear) grid of coordinates. Differential forms transform contravariantly.

2.4 Coordinate basis

- We introduce the *basis*, a set of linearly independent vectors spanning a vector space V; usually V is the Euclidean 3D space $\mathbb{R}^3$,
- We denote a set of *covariant basis vectors* as $\{\boldsymbol{g}_1^*, \boldsymbol{g}_2^*, \boldsymbol{g}_3^*\} = \{\boldsymbol{g}_i^*\}$, these can be computed as tangents to the coordinate grid: $\boldsymbol{g}_i^* = \frac{\partial}{\partial \theta_i}$.

- Covariant basis vectors transform covariantly as:

$$\hat{\boldsymbol{g}^*}_i = \frac{\partial \theta_j}{\partial \hat{\theta}_i} \boldsymbol{g}_j^*$$

- We denote a set of *contravariant basis vectors* as $\{\underline{\boldsymbol{g}}^1, \underline{\boldsymbol{g}}^2, \underline{\boldsymbol{g}}^3\} = \{\underline{\boldsymbol{g}}^i\}$ such that

$$\underline{\boldsymbol{g}}^i \cdot \boldsymbol{g}_j^* = \delta_j^i \tag{8}$$

- Contravariant basis vectors transform contravariantly as:

$$\hat{\underline{\boldsymbol{g}}}^i = \frac{\partial \hat{\theta}_i}{\partial \theta_j} \underline{\boldsymbol{g}}^j \tag{9}$$

- Note that, apart from using $\underline{\boldsymbol{g}}^i \cdot \boldsymbol{g}_j^* = \delta_j^i$, these might be computed also as $\underline{\boldsymbol{g}}^i = \nabla_{\boldsymbol{r}} \theta_i$ if one could have a function $\boldsymbol{\theta}(\boldsymbol{r})$ with $\boldsymbol{r}$ as manifold point; more often however one has $\boldsymbol{r}(\boldsymbol{\theta})$. More easily, the contravariant basis, if needed, can be computed just from the covariant basis by passing through the inverse of the metric tensor (see later).

- A vector is represented as a linear combination of covariant basis, or equivalently, as a linear combination of contravariant basis:

$$\boldsymbol{v} = v^i \boldsymbol{g}_i^* = v_i \underline{\boldsymbol{g}}^i$$

- The vector with covariant basis $v^i \boldsymbol{g}_i^*$ is called *contravariant vector* because, for representing the same $\boldsymbol{v}$ after a change of coordinates, its components v^i must follow the contravariant transformation, hence are *contravariant components*:

$$\hat{v}^i = \frac{\partial \hat{\theta}_i}{\partial \theta_j} v^i$$

  Similarly, the vector with contravariant basis $v_i \underline{\boldsymbol{g}}^i$ is called *covariant vector* because its components v^i must follow the contravariant transformation, hence are *covariant components*:

$$\hat{v}_i = \frac{\partial \theta_j}{\partial \hat{\theta}_i} v_j$$

- Covariant vectors are also called *covectors*.

- Covariant and contravariant vectors differ respect to the way they are transformed with a change of basis coordinates. For a vector $\boldsymbol{v}$ to remain invariant under coordinate changes, the components v^i should transform following a contravariant rule and the components v_i should transform following a covariant rule.

- Examples. Velocity is a contravariant vector. A gradient is a covariant vector. In fact, say you shrink the basis of a velocity vector by a factor 0.1, you must then scale by a factor of 10 its components to keep it the same. Viceversa, if you shrink the basis by factor 0.1, the gradient scales 0.1 times as well.
- As a rule of thumb: vectors that contain distance at the numerator in a dimensional expression (ex. velocity = distance / time) transform contravariantly; vectors that have the dimension of distance at the denominator (ex. the gradient) transform covariantly.

2.5 Tensors

- We assume that tensors represent physical entities whose meaning does not change when changing frame of reference, so it must come equipped with some type of information about how it is transformed when its basis is changed, just like we did for vectors (which are, indeed, special cases of tensors).
- Given basis vectors $\{\underline{\boldsymbol{g}}^i\}$ in $\underline{\mathrm{E}}$ and dual basis vectors $\{\boldsymbol{g}_i^*\}$ in E^*, both of dimension m, a p-times contravariant and q-times covariant **tensor** $\boldsymbol{T}$ is:

$$\boldsymbol{T} = T^{i_1,i_2,\ldots,i_p}_{j_1,j_2,\ldots,j_q} \boldsymbol{g}^*_{i_1} \otimes \boldsymbol{g}^*_{i_2} \otimes \ldots \otimes \boldsymbol{g}^*_{i_p} \otimes \underline{\boldsymbol{g}}^{j_1} \otimes \underline{\boldsymbol{g}}^{j_2} \otimes \ldots \otimes \underline{\boldsymbol{g}}^{j_p}$$

- The *type* or *valence* of tensor is given by the couple (p, q).
- The *rank* of the tensor is the sum $n = p + q$, also *order* [2].
- The *dimension* of the tensor is m, ex. $m = 3$ for most problems in three dimensional space, $m = 4$ in general relativity, etc.
- The number of *components* is m^n.
- Tensors with covariant components only, as in R_{ab}, are said *covariant* tensors.
- Tensors with contravariant components only, as in R^{ab}, are said *contravariant* tensors.
- Tensors with both covariant and contravariant components, as in $R^a{}_{bcd}$ are said *mixed* contravariant/covariant tensors.
- Often the $\boldsymbol{g}^*_{i_1} \otimes \boldsymbol{g}^*_{i_2} \otimes \ldots \otimes \boldsymbol{g}^*_{i_p} \otimes \underline{\boldsymbol{g}}^{j_1} \otimes \underline{\boldsymbol{g}}^{j_2} \otimes \ldots \otimes \underline{\boldsymbol{g}}^{j_p}$ dyadic products are omitted and tensors are written simply as:

$$T^{i_1,i_2,\ldots,i_p}_{j_1,j_2,\ldots,j_q}$$

[2] Order and rank are often used interchangeably, but some Authors distinguish rank from order where the former does not count repeated indices (for contractions - see later).

We call this the *squeezed* simplified notation. Using this purely indicial notation, an example of tensor could be

$$T^{ijk}_{lmn}$$

- Another simplified notation, again purely indicial, uses vertically aligned superscripts and subscripts as, for example,

$$T^{ij}{}_{lm}{}^{k}{}_{n}$$

In fact we assumed a dyadic product of all covariant bases followed by all contravariant bases, although this is arbitrary. For example, as dyadic product is not commutative, the tensor $T^{ij}_k \boldsymbol{g}^*_i \otimes \boldsymbol{g}^*_j \otimes \underline{\boldsymbol{g}}^k$ is not exactly $T^{ij}_k \boldsymbol{g}^*_i \otimes \underline{\boldsymbol{g}}^k \otimes \boldsymbol{g}^*_j$, yet if we write them with the squeezed simplified notation we obtain the same symbol T^{ij}_k, and we loose the information on how we ordered contravariant/covariant bases. For this reason, some authors introduce small spaces in the index super/subscripts to retain the ordering information, so the two examples become $T^{ij}{}_k$ and $T^i{}_k{}^j$, respectively. For example $R^i{}_j$ is not $R_j{}^i$ unless symmetric, whereas R^i_j would be equivocal.

In the following we will use simplified notations only in the cases where there is no risk of misunderstanding.

- *Two point tensors* are tensors where some of the base vectors belong to a coordinate reference A, and some other belong to a coordinate reference B. Often, the distinction is done by using uppercase/lowercase both for the dyads and for the indexes, so for example the former base vectors are $\boldsymbol{g}^*_i$ or $\underline{\boldsymbol{g}}_j$, and latter are $\boldsymbol{G}^*_I$ or $\underline{\boldsymbol{G}}_J$. An example is the deformation tensor[3] in nonlinear elasticity:

$$\boldsymbol{F} = F^i{}_I \boldsymbol{g}^*_i \otimes \underline{\boldsymbol{G}}^I$$

- From the point of view of computer implementation, a tensor $\boldsymbol{T}$ can be stored in memory just like a *multi-dimensional array* of scalars, ie. $T^{i_1,i_2,\ldots,i_p}_{j_1,j_2,\ldots,j_q}$, and there is no need to store the basis vectors $\underline{\boldsymbol{g}}^i$ and $\boldsymbol{g}^*_j$ of the dyadic products. In fact it is implicitly assumed that who write the tensor expression is aware of the basis (whose components can be stored once in a single place, if needed). Also there is a binary information per each index (covariant or contravariant?) that can be omitted as it is implicitly assumed that who writes the formulas in the computer code is aware of the permitted transformations.

- Similarly to vectors, covariant and contravariant components of tensors differ respect to the way they are transformed with a change of basis. For a vector $\boldsymbol{v}$ to remain invariant under coordinate changes, contravariant

[3] This holds assuming a material point in coordinates of the reference configuration is $\boldsymbol{X} = X^I \boldsymbol{G}^*_I$, and in current configuration is $\boldsymbol{x} = x^i \boldsymbol{g}^*_i$, for $d\boldsymbol{x} = \boldsymbol{F} d\boldsymbol{X}$ or equivalently $dx^i = \frac{\partial x^i}{\partial X^I} dX^I$.

components should transform following a contravariant rule and covariant components should transform following a covariant rule. Formally, when changing coordinates $\boldsymbol{\theta} \to \hat{\boldsymbol{\theta}}$, one passes from $\boldsymbol{T}$ to $\hat{\boldsymbol{T}}$, hence from ${T^{abc}}_{def}\boldsymbol{g}^*_a \otimes \boldsymbol{g}^*_b \otimes \boldsymbol{g}^*_c \otimes \underline{\boldsymbol{g}}^d \otimes \underline{\boldsymbol{g}}^e \otimes \underline{\boldsymbol{g}}^f$ to ${\hat{T}^{abc}}_{def}\hat{\boldsymbol{g}}^*{}_a \otimes \hat{\boldsymbol{g}}^*{}_b \otimes \hat{\boldsymbol{g}}^*{}_c \otimes \hat{\underline{\boldsymbol{g}}}^d \otimes \hat{\underline{\boldsymbol{g}}}^e \otimes \hat{\underline{\boldsymbol{g}}}^f$ but for invariance $\boldsymbol{T} = \hat{\boldsymbol{T}}$ to hold, transforming all bases as in (9) and (8), one obtains the *transformation rule* of tensor components:

$$\hat{T}^{abc\dots}{}_{def\dots} = T^{ijk\dots}{}_{lmn\dots}\frac{\partial\hat{\theta}_a}{\partial\theta_i}\frac{\partial\hat{\theta}_b}{\partial\theta_j}\frac{\partial\hat{\theta}_c}{\partial\theta_k}\dots\frac{\partial\theta_l}{\partial\hat{\theta}_d}\frac{\partial\theta_m}{\partial\hat{\theta}_e}\frac{\partial\theta_n}{\partial\hat{\theta}_f}\dots \tag{10}$$

- More succinctly, one can define $\frac{\partial\hat{\theta}_i}{\partial\theta_j} = \beta^i_j$ and $\frac{\partial\theta_i}{\partial\hat{\theta}_j} = \alpha^i_j$, which are basically matrices, like jacobians, and write:

$$\hat{T}^{abc\dots}{}_{def\dots} = T^{ijk\dots}{}_{lmn\dots}\beta^a_i\beta^b_j\beta^c_k\dots\alpha^l_d\alpha^m_e\alpha^n_f\dots \tag{11}$$

- Note that one might obtain α^i_j matrices directly with some simple procedures, without need of passing through $\boldsymbol{\theta}$ coordinates and derivatives. For example, a common scenario is a rotation in Cartesian coordinates via a $\mathsf{SO}(3)$ matrix R, ie.e a 3x3 array, that would lead to $\alpha^i_j = {R^i}_j$.

- One can compute $\beta = \alpha^{-1}$ by matrix inversion with linear algebra, since one can show $\alpha^k_i\beta^j_k = \delta^j_i$. In the example with the rotation matrix, given orthogonality of R, one has $R^{-1} = R^T$ and ${(R^T)^i}_j = {R_j}^i$, so it would be $\beta^i_k = (\alpha^{-1})^i_k = {R_j}^i$.

- An example of change of basis with orthogonal transformation (rotation matrix R), for a case of a (0,1) tensor like a velocity, corresponding to well known linear algebra $\hat{\boldsymbol{v}} = R^T\boldsymbol{v}$:

$$\hat{v}^i = {R_j}^i v^j$$

- An example of change of basis with orthogonal transformation (rotation matrix R), for a case of a (0,2) tensor like stress tensor, corresponding to well known linear algebra $\hat{\sigma} = R^T\boldsymbol{\sigma}R$:

$$\hat{\sigma}^{ij} = {R_k}^i {R_m}^j \sigma^{km}$$

- Some examples of tensors.
  - s : a scalar is a order 0 tensor,
  - v^i : velocity is a first order contravariant three dimensional tensor, hence of type $(1, 0)$, with $n = 3$ components,
  - d_i : gradients are first order covariant tensors, hence of type $(0, 1)$, with $n = 3$ components if in 3D space,
  - $g_{\alpha\beta}$: the *metric tensor* is a covariant second order tensor, hence of type $(0, 2)$

  - R_{ij} : the Ricci curvature tensor is a covariant second order tensor, of type $(0, 2)$
  - $g^{\alpha\beta}$: the *inverse metric tensor* is a contravariant second order tensor, of type $(2, 0)$
  - J^{ij} : the *inertia tensor* is a contravariant second order tensor, of type $(2, 0)$
  - σ^{ij} : stress tensors (v.Piola-Kirchhoff) are contravariant second order tensors, of type $(2, 0)$
  - ϵ_{kl} : strain tensors (v.Green-Lagrange) are covariant second order tensors, of type $(0, 2)$
  - $R^{\alpha}{}_{\beta\gamma\nu}$: the Riemann[4] curvature tensor is a mixed tensor of type $(1, 3)$
  - $\delta^{i}{}_{j}$: the Kronecker delta tensor is a mixed order tensor, of type $(1, 1)$
  - $R^{i}{}_{j}$: a rotation matrix is a mixed order tensor, of type $(1, 1)$

- Tensor **multiplication by a scalar** is associative and commutative, for ex. (omitting dyads):

$$a\boldsymbol{T} \;=\; \boldsymbol{T}a \;= aT^{ij}{}_{k} \tag{12}$$

- Tensor **addition and subtraction** is associative and commutative, and can be done only for tensors of same rank and type, for ex. (omitting dyads):

$$\boldsymbol{A} + \boldsymbol{B} - \boldsymbol{C} \;=\; A^{ij}{}_{k} + B^{ij}{}_{k} - C^{ij}{}_{k} \tag{13}$$

- Note that tensor addition requires same super/subscripts. Dummy indexes do not count, ex. $A^{ij}{}_{k}+B^{ij}{}_{k}+C^{mij}{}_{mk}$ is legal too. In fact see contraction, later, for which $C^{mij}{}_{mk} = D^{ij}{}_{k}$.

- An example of tensor expression with addition and scalar multiplication: the Ricci flow (with g_{ij} metric tensor, R_{ij} Ricci curvature):

$$\partial_t g_{ij} + 2R_{ij} = 0$$

- Tensor **transpose**, defined for 2nd rank tensors, is obtained swapping the order of indexes; this correspond simply to swapping columns and rows in the T matrix of components. Note that also corresponding bases in dyadic products are switched, so the contravariant/covariant property of each index persists:

$$({}^{t}T)_{ij} = T_{ji} \quad ({}^{t}T)^{ij} = T^{ji} \quad ({}^{t}T)^{i}{}_{j} = T_{j}{}^{i} \tag{14}$$

[4] It is remarkable how many bright scientists attended the University of Göttingen.

- Tensor **product**, denoted with $\otimes$, takes two vectors of types (p_a, q_a) and (p_b, q_b) and returns a tensor of type $(p_a + p_b, q_a + q_b)$, appending dyadic products (and components) as in this example:

$$\boldsymbol{A} \otimes \boldsymbol{B} = \boldsymbol{C} \tag{15}$$

$$= A^{ij}{}_k B^m{}_n \boldsymbol{g}_i^* \otimes \boldsymbol{g}_j^* \otimes \underline{\boldsymbol{g}}^k \otimes \boldsymbol{g}_m^* \otimes \underline{\boldsymbol{g}}^n \tag{16}$$

$$= C^{ij}{}_k{}^m{}_n \boldsymbol{g}_i^* \otimes \boldsymbol{g}_j^* \otimes \underline{\boldsymbol{g}}^k \otimes \boldsymbol{g}_m^* \otimes \underline{\boldsymbol{g}}^n \tag{17}$$

  that is, in simplified indicial notation:

$$A^{ij}{}_k B^m{}_n = C^{ij}{}_k{}^m{}_n \tag{18}$$

- The tensor product, as in dyadic product seen before, is associative but *not commutative*. Hence, except special cases:

$$\boldsymbol{A} \otimes \boldsymbol{B} \neq \boldsymbol{B} \otimes \boldsymbol{A} \tag{19}$$

- Note that one can commute the order of terms in the indicial notation of the product without problems, such as in:

$$A^i{}_j B^k{}_l = B^k{}_l A^i{}_j$$

  and this independence of order of terms holds for all tensor expressions in indicial form. Where is the trick? The reason is that the indicial form deals with simple scalar products, hence commutative, yet the hidden sorcery is that the order of tensor product is preserved *because we do not change the indexes*, in fact: $\boldsymbol{A} \otimes \boldsymbol{B} = A^i{}_j B^k{}_l = B^k{}_l A^i{}_j = C^i{}_j{}^k{}_l$, but $\boldsymbol{B} \otimes \boldsymbol{A} = B^i{}_j A^k{}_l = C^i{}_j{}^k{}_l$, note how the same $C^i{}_j{}^k{}_l$ term has two different expressions in A and B components depending on the order of multiplication?

- This said, by the way, there is no need to prefer a "well ordered" expression $A^i{}_j B^k{}_l = C^i{}_j{}^k{}_l$ respect to $B^k{}_l A^i{}_j = C^i{}_j{}^k{}_l$, although the former might be more intuitive.

- Also the name of indexes are arbitrary, for instance $A^i B^k = C^{ik}$ would work equally well as $A^p B^m = C^{pm}$, and repeated indexes (that will elide because of by Einstein notation, implying contraction - see later) are allowed everywhere as in $A^{iz}{}_{jz} B^k = C^i{}_j{}^k$.

- Tensor **contraction** is an operation where two indices in a tensor terms are set the same, hence implying summation over that indices by Einstein convention. The repeated indices and corresponding bases are elided if rewriting the tensor term. This means that it takes a tensor (p, q) and returns a tensor of $(p-1, q-1)$ type:

$$\text{contract}_{j,l}(A^{ij}{}_{kl}) = A^{ic}{}_{kc} = A^i{}_k \tag{20}$$

  Note that choosing the pair of indices to contract is not arbitrary: one must have a "physical" reason to do so.

- Contraction can be done only between a covariant and a contravariant index, or viceversa.
- In general an index does not repeat more than two times in a tensor term. Expressions like T_i^{ii} are illegal.
- The **trace** of a rank-2 matrix in matrix algebra can be written as a contraction using tensors, es:
$$\mathrm{tr}(\boldsymbol{A}) = A_i^i$$
- Tensor **contracted product**, denoted $\cdot$, is just a shorthand notation for representing a tensor product (that gives an "'inflated"' tensor) followed by a contraction over a pair of indices (that "'deflates"' the tensor), as for example:
$$\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B} = A^{ij}{}_{kl} B^l_m = C^{ij}{}_{km} \tag{21}$$
Note that there is no agreed rule on the pair of indices must be contracted, but usually one writes the expressions such that the contracted indices are the last of the first term, and the first index of the second term.
- Sometimes a more compact notation is used, where the dot is skipped if the meaning is clear, thus $\boldsymbol{A} = \boldsymbol{B} \cdot \boldsymbol{C} \cdot \boldsymbol{D}$ could be written as
$$\boldsymbol{A} = \boldsymbol{BCD}$$
- Tensor contracted product $\cdot$ is not commutative just like the tensor product.
- A special case: the contracted product of a covariant $(0,1)$ tensor $\boldsymbol{a} \in \underline{E}$ and a contravariant $(1,0)$ tensor $\boldsymbol{b} \in E^*$, where the couple of indices to contract is obvious, returns a scalar, i.e. a tensor of (0,0) type, hence it is equivalent to a scalar product between the two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$:
$$c = \boldsymbol{a} \cdot \boldsymbol{b} = a_i b^i$$
Some authors call this a **tensor dot product**.
- A special case: the contracted product of a second order mixed tensor $\boldsymbol{T} \in E^* \otimes \underline{E}$ and a first order tensor $\boldsymbol{v} \in E^*$, is the equivalent of a matrix-by-vector product $\boldsymbol{u} = T\boldsymbol{v}$ with linear algebra:
$$\boldsymbol{u} = \boldsymbol{T} \cdot \boldsymbol{v} = T^i{}_j v^j \boldsymbol{g}^*_i;$$
Similarly, for the contracted product of a second order mixed tensor $\boldsymbol{T} \in \underline{\mathrm{E}} \otimes E^*$ and a first order tensor $\boldsymbol{w} \in \underline{E}$:
$$\boldsymbol{z} = \boldsymbol{T} \cdot \boldsymbol{w} = T_i{}^j w_j \underline{\boldsymbol{g}}^i;$$

- A special case: the contracted product of a second covariant tensor $\boldsymbol{T} \in \underline{E} \otimes \underline{E}$ and two first order tensors $\boldsymbol{u} \in E^*$, $\boldsymbol{v} \in E^*$, that is equivalent to the expression $\boldsymbol{u}^T T \boldsymbol{v}$ with linear algebra:

$$c = \boldsymbol{u} \cdot \boldsymbol{T} \cdot \boldsymbol{v} = T_{ij} u^i v^j$$

- The previous example also explain that a tensor is also a *multilinear map*: a p-contravariant q-covariant tensor, $\boldsymbol{T} \in E_1^* \otimes E_2^* \otimes \ldots \otimes E_p^* \otimes \underline{E}_1 \otimes \underline{E}_2 \otimes \ldots \otimes \underline{E}_q$ is a multilinear form on vector space $(\underline{\mathrm{E}})^p \times (E^*)^q$, in fact in the above example the tensor $\boldsymbol{T} \in \underline{E} \otimes \underline{E}$ is a multilinear map of type $c = \boldsymbol{T}(\boldsymbol{u}, \boldsymbol{v})$ hence a multilinear map $E^* \times E^* \to \mathbb{R}$.

- An example of tensor expression with products and contractions: the Einstein field equation in general relativity, with $R_{\mu\nu}$ Ricci curvature tensor, R Ricci scalar curvature -a scalar given by double contraction $g^{\mu\nu} R_{\mu\nu}$, $g_{\mu\nu}$ is the metric tensor, Λ is the cosmological constant, $T_{\mu\nu}$ is the stress-energy tensor ex. depending on mass density, G is Newton gravitation constant, c is speed of light constant:

$$R_{\mu\nu} - \tfrac{1}{2} R g_{\mu\nu} + \Lambda g_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu}$$

- The **Kroeneker symbol** δ^i_j does not perform any transformation on the components, just like in linear algebra with an identity matrix $A = IA$ for $I_{ij} = \delta_{ij}$, but just keep in mind that it changes the name of an index, as in:

$$a_i \delta^i_j = a_j$$

- The **Levi-Civita symbol** $\varepsilon_{a_1 a_2 a_3 \ldots a_n}$ is a pseudotensor defined as

$$\varepsilon_{a_1 a_2 a_3 \ldots a_n} = \begin{cases} +1 & \text{if } (a_1, a_2, a_3, \ldots, a_n) \text{ even permutation of } (1, 2, 3, \ldots, n) \\ -1 & \text{if } (a_1, a_2, a_3, \ldots, a_n) \text{ odd permutation of } (1, 2, 3, \ldots, n) \\ 0 & \text{otherwise} \end{cases} \tag{22}$$

  and in the simple two-dimension case, for example, it becomes a 2x2 hemisymmetric matrix $[0, 1; -1, 0]$. Also note $\varepsilon_{a_1 a_2 a_3 \ldots a_n} = \varepsilon^{a_1 a_2 a_3 \ldots a_n}$.

- The **cross product** of two vectors $\boldsymbol{a} \in \mathbf{R}^3$, $\boldsymbol{b} \in \mathbf{R}^3$, denoted $\times$ in vector algebra, can be represented using the Levi-Civita symbol and tensor notation - that also generalizes to other dimensions:

$$\boldsymbol{a} \times \boldsymbol{b} = a^j b^k \varepsilon_{ijk} \boldsymbol{g}_i^* \tag{23}$$

$$[\boldsymbol{a} \times \boldsymbol{b}]^i = a^j b^k \varepsilon_{ijk} \tag{24}$$

- The cross product commutes as

$$\boldsymbol{a} \times \boldsymbol{b} = -\boldsymbol{b} \times \boldsymbol{a}$$

- Just for curiosity:

$$\boldsymbol{a} \cdot (\boldsymbol{b} \times \boldsymbol{c}) = a^i b^j c^k \varepsilon_{ijk} \tag{25}$$

- The Levi-Civita symbol ε is a *pseudo*tensor because it changes sign after an improper orthogonal transformation of coordinates, ex. a reflection like in a mirror.

- The determinant of a second rank tensor can be computed with the Levi-Civita symbol as

$$\det(\boldsymbol{A}) = \varepsilon_{ijk\ldots m} A_1^i A_2^j A_3^k \ldots A_m^m \tag{26}$$

- Tensor **double contracted product**, denoted :, is just a shorthand notation for representing a tensor product followed by two contractions over two pairs of indices, as for example:

$$c = \boldsymbol{A} : \boldsymbol{B} = A^{ij} B_{ji} \tag{27}$$

  In most cases this operation is done on two second-rank tensors. Note that most Authors agree to assume the two indexes being ^{ij} and ^{ji}, that is, swapped, but other Authors assume ^{ij} and ^{ij}. With out notation the latter would rather $A^{ij} B_{ij} = \boldsymbol{A} : {}^t\boldsymbol{B}$. For the latter, another symbol could be used:

- Tensor **double contracted product**, denoted $\cdot\cdot$, corresponds to:

$$c = \boldsymbol{A} \cdot\cdot \boldsymbol{B} = A^{ij} B_{ij} \tag{28}$$

- The **invariants** of a second order mixed tensor of type (1,1) are scalars computed as

$$I_1 = \mathrm{tr}\boldsymbol{A} = A_i^i \tag{29}$$

$$I_2 = \frac{1}{2}\mathrm{tr}(\boldsymbol{A} \cdot \boldsymbol{A}) = \frac{1}{2} A_j^i A_i^j \tag{30}$$

$$I_3 = \frac{1}{3}\mathrm{tr}(\boldsymbol{A} \cdot \boldsymbol{A} \cdot \boldsymbol{A}) = \frac{1}{3} A_j^i A_k^j A_i^k \tag{31}$$

$$I_4 = \ldots \tag{32}$$

- A manifold $(M, (G))$ is said a *Riemannian manifold* if it is a differentiable manifold M equipped with an everywhere non-degenerate, smooth, symmetric, positive-definite metric tensor (G). If the metric tensor is not necessarily positive-definite, it is said *pseudo-Riemannian* manifold: a special case is the Lorentzian manifold in relativity.

- The **metric tensor** is a type (0,2) tensor, a covariant second order tensor defined as

$$\boldsymbol{G} = g_{ij} \underline{\boldsymbol{g}}^i \otimes \underline{\boldsymbol{g}}^j \tag{33}$$

where the components are defined as

$$g_{ij} = \underline{\boldsymbol{g}}^i \cdot \underline{\boldsymbol{g}}^j \tag{34}$$

- We can also compute the inverse of the metric tensor as

$$g^{ij} = \boldsymbol{g}_i^* \cdot \boldsymbol{g}_j^* \tag{35}$$

and noting that $g_{ij}g^{jk} = \delta_i^j$, this also means that one can compute g^{ij} using linear algebra as a matrix inverse of g_{ij}, and vice versa.

- Contracted product by the metric tensor $\boldsymbol{G}$ gives an **index lowering**, ex. getting a covariant tensor from a contravariant vector:

$$\boldsymbol{a} = a_i \underline{\boldsymbol{g}}^i = g_{ij} b^j \underline{\boldsymbol{g}}^i \tag{36}$$

or shortly: $a_i = g_{ij}b^j$.

- Contracted product by the inverse metric tensor $\boldsymbol{G}^{-1}$ gives an **index raising**, ex. getting a contravariant tensor from a covariant vector:

$$\boldsymbol{b} = b^i \boldsymbol{g}_i^* = g^{ij} a_j \boldsymbol{g}_i^* \tag{37}$$

or shortly: $b^i = g^{ij}a_j$.

- The index lowering/raising via tensor metric is also called **musical isomorphism**, with notation

$$\boldsymbol{a} = \boldsymbol{b}^\flat \tag{38}$$

$$\boldsymbol{b} = \boldsymbol{a}^\sharp \tag{39}$$

- The $\mathrm{d}s$ distance on a manifold is defined as

$$\mathrm{d}s^2 = \mathrm{d}\boldsymbol{r} \cdot \boldsymbol{G} \cdot \mathrm{d}\boldsymbol{r} = g_{ij}\mathrm{d}\theta^i \mathrm{d}\theta^j \tag{40}$$

also written as a quadratic form with map $G(\mathrm{d}\boldsymbol{r})$ or $\langle \mathrm{d}\boldsymbol{r}, \mathrm{d}\boldsymbol{r} \rangle_G$. For a Cartesian system, with ortonormal basis, it boils down to $\mathrm{d}s^2 = \mathrm{d}r^i \delta_{ij} \mathrm{d}r^i = \sum_i \mathrm{d}r_i^2$.

- Note that the metric tensor can be computed as a matrix product of transformation jacobian by its transpose, $G = J^T J$,

$$g_{ij} = \alpha_i^k \alpha_j^k$$

, for example in the case of a surface in 3D space parametrized as $\boldsymbol{x} = \boldsymbol{x}(\boldsymbol{\theta})$ one has $\alpha_j^i = \frac{\partial x_i}{\partial \theta_j}$.

- In the special case where the manifold is a surface in 3D space parametrized by θ_1, θ_2, infinitesimal area is scaled by square root of determinant of the metric tensor as in

$$\mathrm{d}A = \sqrt{\det(g_{ij})} d\theta_1 d\theta_2 \tag{41}$$

$$\mathrm{d}A = \det(\alpha^i_j) d\theta_1 d\theta_2 \quad \alpha^i_j = \frac{\partial x_i}{\partial \theta_j} \tag{42}$$

  and this for example allows to compute the area of a surface, given its metric, as $A = \int\int \sqrt{|\det(g_{ij})|} d\theta_1 d\theta_2$. Note that if the local reference is Cartesian, $\det(g_{ij}) = 1$.

- In the special case where the manifold is a curved volume in 3D space parametrized by $\theta_1, \theta_2, \theta_3$, infinitesimal volume is scaled by square root of determinant of the metric tensor as in

$$\mathrm{d}V = \sqrt{\det(g_{ij})} d\theta_1 d\theta_2 d\theta_3 \tag{43}$$

$$\mathrm{d}V = \det(\alpha^i_j) d\theta_1 d\theta_2 d\theta_3 \quad \alpha^i_j = \frac{\partial x_i}{\partial \theta_j} \tag{44}$$

- In the special case where the manifold is a surface in 3D space, the metric tensor is also called *first fundamental form* of the surface, $\boldsymbol{I}$, that in an intuitive sense tells how the surface "stretches".

- A vector space equipped with an inner product $s = \boldsymbol{a} \cdot \boldsymbol{b}$ is called an *inner product space*. An inner product $\boldsymbol{a} \cdot \boldsymbol{a}$ induces a *quadratic form*, if the form is positive we say that the vector space is *Euclidean*.

- In a Euclidean space we can define the **dot product** using the metric tensor as in:

$$\boldsymbol{a} \cdot \boldsymbol{b} = g_{ij} a^i b^i$$

  This is sometimes written as a map $G(\boldsymbol{a}, \boldsymbol{b})$.

- If a basis is *ortonormal*, it is identical to its dual basis, and $\boldsymbol{G} = \boldsymbol{G}^{-1}$, In terms of matrix algebra, $\boldsymbol{G}$ would be a unit matrix, as in $g_{ij} = \delta_{ij}$.

- *Cartesian reference systems* have ortonormal basis, thus in a Cartesian reference one could forget the distinction between contravariant and covariant components, and super/subindexes could be written all as subscripts. This simplifies the notation quite a bit, for example A^i_j could be written A_{ij}, or ${C_{ij}}^{kl}$ could be written C_{ijkl}, etc.

- The all-subscript notation is often used in computational mechanics where problems are formulated in a Cartesian system. However this simplification might not be used, for instance, when using polar coordinates, etc.

2.6 Differential operators

- We introduce the **Christoffel symbols of the 2nd kind**, denoted as ${\Gamma^k}_{ij}$ and defined as

$$\frac{\partial \boldsymbol{g}_s^*}{\partial \theta^n} = {\Gamma^p}_{sn} \boldsymbol{g}_p^* \tag{45}$$

  intuitively representing how a basis is changing along coordinates of a manifold, expressed in the basis itself.

- We introduce the **Christoffel symbols of the 1st kind**, denoted as Γ_{ijk} and defined as

$$\frac{\partial \underline{\boldsymbol{g}}^s}{\partial \theta^n} = \Gamma_{psn} \underline{\boldsymbol{g}}^p \tag{46}$$

  intuitively representing how a basis is changing along coordinates of a manifold, expressed along the dual basis. One has $\Gamma_{psn} = g_{pd} {\Gamma^d}_{sn}$.

- Sometimes Christoffel symbols of the 2nd kind ${\Gamma^p}_{sn}$ are written as $\left\{ {}^{p}_{sn} \right\}$.

- Sometimes Christoffel symbols of the 1st kind Γ_{psn} are written as $[sn, p]$.

- One can compute Christoffel symbols of 2nd kind, if one knows $\boldsymbol{\theta} = \boldsymbol{\theta}(\boldsymbol{x})$ with $\boldsymbol{x}$ coordinates of a Cartesian system, as:

$${\Gamma^p}_{sn} = \frac{\partial^2 x^m}{\partial \theta^s \partial \theta^n} \frac{\partial \theta^p}{\partial x^n}$$

  or directly from the metric tensor as:

$$\Gamma_{cab} = \frac{1}{2} \left(\frac{\partial g_{ca}}{\partial x^b} + \frac{\partial g_{cb}}{\partial x^a} - \frac{\partial g_{ab}}{\partial x^c} \right)$$

- The **covariant derivative** of a contravariant vector field $\boldsymbol{u} = u^i \boldsymbol{g}_i^*$ is $\frac{\partial \boldsymbol{u}}{\partial \theta^n} = \frac{\partial u^i}{\partial \theta^n} \boldsymbol{g}_i^* + u^i \frac{\boldsymbol{g}_i^*}{\partial \theta^n}$, hence:

$$\nabla_j \boldsymbol{u} = \left(\frac{\partial u^k}{\partial \theta^j} + u^i {\Gamma^k}_{ij} \right) \boldsymbol{g}_k^* \tag{47}$$

- The **covariant derivative** of a covariant vector field $\boldsymbol{u} = u_i \underline{\boldsymbol{g}}^i$ is

$$\nabla_j \boldsymbol{u} = \left(\frac{\partial u_k}{\partial \theta^j} - u_i {\Gamma^i}_{kj} \right) \underline{\boldsymbol{g}}^k \tag{48}$$

- Note that the Christoffel symbols in (47) (48) account for the fact that the basis might change direction as in curved manifolds. In a straight manifold (ex. in a Cartesian reference) they vanish to zero.

- The **covariant directional derivative** of a contravariant $\boldsymbol{u}$ along a direction $\boldsymbol{v}$ is:

$$\nabla_{\boldsymbol{v}}\boldsymbol{u} = \left(v^j \frac{\partial u^k}{\partial \theta^j} + v^j u^i \Gamma^k{}_{ij}\right) \underline{\boldsymbol{g}}_k \tag{49}$$

  and the covariant directional derivative of a covariant $\boldsymbol{u}$ along a direction $\boldsymbol{v}$ is:

$$\nabla_{\boldsymbol{v}}\boldsymbol{u} = \left(v^j \frac{\partial u_k}{\partial \theta^j} - v^j u_i \Gamma^i{}_{kj}\right) \underline{\boldsymbol{g}}^k \tag{50}$$

- For a more compact notation, we introduce the following semicolon ";" notation for the (components of) the covariant derivative:

$$u^k{}_{;j} = \left(\frac{\partial u^k}{\partial \theta^j} + u^i \Gamma^k{}_{ij}\right) \tag{51}$$

$$u_{k;j} = \left(\frac{\partial u_k}{\partial \theta^j} - u_i \Gamma^i{}_{kj}\right) \tag{52}$$

  To sum up things: alternative notations for the covariant derivative:

$$\nabla_j \boldsymbol{u} = \nabla_{\boldsymbol{g}_j}\boldsymbol{u} = u^i|_j \boldsymbol{g}_i^* = u^i{}_{;j}\boldsymbol{g}_i^* \tag{53}$$

  Moreover, some Authors call only the component part $u^i{}_{;j}$ as "covariant derivative", not including $\boldsymbol{g}_i^*$.

- One can see the covariant derivative as an extended version of the partial derivative, the latter often being denoted with a "," in shorthand notation instead of semicolon ";":

$$\frac{\partial u^i}{\partial \theta^j} = \partial_j u^i = u^i{}_{,j} \tag{54}$$

- Using the simplified notation above, one can express the following special cases:

  - The covariant derivative of a scalar field is also the partial derivative regardless of the curvature,

$$\phi_{,j} = \phi_{;j} = \partial_j \phi$$

  - The covariant derivative of a contravariant vector field:

$$u^i{}_{;j} = u^i{}_{,j} + u^k \Gamma^i{}_{kj}$$

  - The covariant derivative of a covariant vector field:

$$u_{i;j} = u_{i,j} - u_k \Gamma^k{}_{ij}$$

  - The covariant derivative of a contravariant vector field in a Cartesian system simplifies to:
$$u^i{}_{;j} = u^i{}_{,j}$$
The covariant derivative of a covariant vector field in a Cartesian system simplifies to:
$$u_{i;j} = u_{i,j}$$
  - In a Cartesian reference, as we have seen before, contravariant and covariant basis are the same so we can take a further simplification and write indifferently contravariant and covariant vector components with subscripts anyway, as often happens in computational mechanics where derivatives are written as $a_{ij,k}$, $v_{i,j}$, etc.

- The covariant derivative for a generic (p,q) type tensor can be expressed as
$$\nabla_{\boldsymbol{g}_n} \boldsymbol{T} = T^{ij\cdot\cdot}{}_{kl..;n} \boldsymbol{g}_i^* \otimes \boldsymbol{g}_j^* \ldots \otimes \underline{\boldsymbol{g}}^k \otimes \underline{\boldsymbol{g}}^l \ldots$$
and again, introducing the conventional partial derivatives $T^{ij\cdot\cdot}{}_{kl..,n} = \frac{\partial T^{ij\cdot\cdot}{}_{kl..}}{\partial \theta^n}$, one corrects them with the Christoffel symbols as:
$$\begin{aligned} T^{ij\cdot\cdot}{}_{kl..;n} =& T^{ij\cdot\cdot}{}_{kl..,n} && (55)\\ &+ T^{sj\cdot\cdot}{}_{kl..}\Gamma^i{}_{sn} + T^{is\cdot\cdot}{}_{kl..}\Gamma^j{}_{sn} + \ldots && (56)\\ &- T^{ij\cdot\cdot}{}_{sl..}\Gamma^s{}_{kn} - T^{ij\cdot\cdot}{}_{ks..}\Gamma^s{}_{ln} - \ldots && (57) \end{aligned}$$

- The ∇ is a *connection*, see the *Levi-Civita connection* in Riemannian geometry.

- In the special case where the manifold is a surface in 3D space, the *second fundamental form* is the tensor $\boldsymbol{II}$, and this is related to the covariant derivative as $\boldsymbol{II}(\boldsymbol{u}, \boldsymbol{v}) = \langle \nabla_{\boldsymbol{u}} \boldsymbol{v}, \boldsymbol{n} \rangle$, with $\boldsymbol{n}$ normal vector field to the manifold.

- The **gradient** operator is obtained with the covariant derivative. Some examples in different notations in the following; note the added tensor product by $\underline{\boldsymbol{g}}^j \otimes \ldots$ in the dyads.

  - The gradient of a scalar is a covariant vector:
$$\text{grad}(\phi) = \nabla\phi = \phi_{;i} \underline{\boldsymbol{g}}^i$$
  - The gradient of a contravariant vector is a rank-2 tensor of mixed type (1,1):
$$\text{grad}(\boldsymbol{v}) = \nabla \boldsymbol{v} = v^i{}_{;j}\, \underline{\boldsymbol{g}}^j \otimes \boldsymbol{g}_i^*$$
  - The gradient of a covariant vector is a rank-2 tensor of covariant type (0,2):
$$\text{grad}(\boldsymbol{v}) = \nabla \boldsymbol{v} = v_{i;j}\, \underline{\boldsymbol{g}}^j \otimes \underline{\boldsymbol{g}}^i$$

  - The gradient of a n-rank tensor of type (p,q), in general, is a (n+1) rank tensor of type (p,q+1):

$$\text{grad}(\boldsymbol{T}) = \nabla \boldsymbol{T} = T^{ij..}{}_{kl..;n}\, \underline{\boldsymbol{g}}^n \otimes \boldsymbol{g}_i^* \otimes \boldsymbol{g}_j^* \ldots \otimes \underline{\boldsymbol{g}}^k \otimes \underline{\boldsymbol{g}}^l \ldots$$

- The **divergence** operator takes a (n) rank tensor and returns a (n-1) rank tensor, basically it is a contraction -usually on the first index- after the covariant derivative. Some examples in different notations in the following.

  - The divergence of a scalar field is not defined.
  - The divergence of a contravariant vector field is a scalar:

$$\text{div}(\boldsymbol{v}) = \nabla \cdot \boldsymbol{v} = \phi^i_{;i}$$

  - The divergence of a covariant vector field is a scalar - note the metric tensor for index raising to allow contraction:

$$\text{div}(\boldsymbol{v}) = \nabla \cdot \boldsymbol{v} = g^{ni}\phi_{i;n}$$

  - The divergence of a tensor field in general - note the covariant basis vector $\boldsymbol{g}_i^*$ that disappears from dyads:

$$\text{div}(\boldsymbol{T}) = \nabla \cdot \boldsymbol{T} = T^{ij..}{}_{kl..;i}\, \boldsymbol{g}_j^* \ldots \otimes \underline{\boldsymbol{g}}^k \otimes \underline{\boldsymbol{g}}^l \ldots \tag{58}$$

$$= g^{ni} T_i{}^{j..}{}_{kl..;n}\, \boldsymbol{g}_j^* \ldots \otimes \underline{\boldsymbol{g}}^k \otimes \underline{\boldsymbol{g}}^l \ldots \tag{59}$$

- The **rotor** operator takes a (n) rank tensor and returns a (n) rank tensor, using the Levi-Civita symbol. The rotor of a tensor in general is:

$$\text{curl}(\boldsymbol{T}) = \nabla \times \boldsymbol{T} = \underline{\boldsymbol{g}}^n \times T^{ij..}{}_{kl..;n}\, \boldsymbol{g}_i^* \otimes \boldsymbol{g}_j^* \ldots \otimes \underline{\boldsymbol{g}}^k \otimes \underline{\boldsymbol{g}}^l \ldots \tag{60}$$

$$= \varepsilon_{nim} T^{ij..}{}_{kl..;l}\, g^{nl} \underline{\boldsymbol{g}}^m \otimes \boldsymbol{g}_i^* \otimes \boldsymbol{g}_j^* \ldots \otimes \underline{\boldsymbol{g}}^k \otimes \underline{\boldsymbol{g}}^l \ldots \tag{61}$$

  Some special cases in the following.

  - The rotor of a contravariant vector is a covariant vector:

$$\text{curl}(\boldsymbol{v}) = \nabla \cdot \boldsymbol{v} = \varepsilon_{ijk} v^j{}_{;n} g^{ni} \underline{\boldsymbol{g}}^k$$

  - The rotor of a covariant vector is a contravariant vector:

$$\text{curl}(\boldsymbol{v}) = \nabla \cdot \boldsymbol{v} = \varepsilon^{ijk} v_{j;n} \boldsymbol{g}_k^*$$

- The **Laplacian** operator ∇^2 takes a (n) rank tensor and returns a (n) rank tensor, and is defined in different notations

$$\nabla^2 \boldsymbol{T} = \delta \boldsymbol{T} = \nabla \cdot \nabla \boldsymbol{T} = \text{div}(\text{grad}\boldsymbol{T})$$

  Some examples in the following:

– The Laplacian of a scalar field, hence divergence of a gradient vector, is a scalar:

$$\nabla^2\phi = \phi_{;ii}$$

– The Laplacian of a contravariant vector field is a contravariant vector:

$$\nabla^2\boldsymbol{v} = v^i{}_{;nl}g^{nl}\boldsymbol{g}^*_i$$

– The Laplacian of a covariant vector field is a covariant vector:

$$\nabla^2\boldsymbol{v} = v_{i;nl}g^{nl}\underline{\boldsymbol{g}}^i$$

– The Laplacian of a tensor field in general is again a tensor of same type:

$$\nabla^2\boldsymbol{T} = \nabla\cdot\nabla\boldsymbol{T} = T^{ij\cdot\cdot}{}_{kl..;nl}\ g^{nl}\boldsymbol{g}^*_i\otimes\boldsymbol{g}^*_j\ldots\otimes\underline{\boldsymbol{g}}^k\otimes\underline{\boldsymbol{g}}^l\ldots \tag{62}$$

where the components of the double covariant derivative are represented by the complex expression

$$T^{ij\cdot\cdot}{}_{kl..;nl} = T^{ij\cdot\cdot}{}_{kl..,nl} \tag{63}$$

$$+ T^{sj\cdot\cdot}{}_{kl..;l}\Gamma^i{}_{sn} + T^{sj\cdot\cdot}{}_{kl..}\frac{\partial\Gamma^i{}_{sn}}{\partial\theta^l} \tag{64}$$

$$+ T^{is\cdot\cdot}{}_{kl..;l}\Gamma^j{}_{sn} + T^{is\cdot\cdot}{}_{kl..}\frac{\partial\Gamma^j{}_{sn}}{\partial\theta^l} \tag{65}$$

$$+\ldots \tag{66}$$

Hovever the expression above simplifies in case of Cartesian coordinate systems as:

$$T^{ij\cdot\cdot}{}_{kl..;nl} = T^{ij\cdot\cdot}{}_{kl..,nl} = \frac{\partial^2 T^{ij\cdot\cdot}{}_{kl}}{\partial\theta^n\partial\theta^l}$$

2.7 Special case: Cartesian coordinates

Many problems in computational mechanics are expressed in Cartesian coordinates, for example finite element problems, multibody problems, etc. Under this assumption, many of the expressions of the previous sections will simplify, in fact in a Cartesian coordinate system one has ortonormal base vectors, hence: covariant and contravariant bases are cohincident, the g_{ij} and g^{ij} metric tensors can be eliminated from formulas (or just them in place enough to raise/lower indices for formal correctness), Christoffel symbols can be omitted in derivatives.

In this simplified setting, one has the following expressions (where we use subscripts regardless if components are covariant or contravariant):

Table 1: Table of simplified formulas for Cartesian coordinate system

	Tensor notation	Indicial notation
inner product	$\boldsymbol{u} \cdot \boldsymbol{v} = u_i v_i$	$\boldsymbol{u} \cdot \boldsymbol{v} = u_i v_i$
outer product	$\boldsymbol{u} \otimes \boldsymbol{v} = u_i v_j \boldsymbol{g}_i \otimes \boldsymbol{g}_j$	$[\boldsymbol{u} \otimes \boldsymbol{v}]_{ij} = u_i v_j$
cross product	$\boldsymbol{u} \times \boldsymbol{v} = u_i v_j \varepsilon_{ijk} \boldsymbol{g}_k$	$[\boldsymbol{u} \times \boldsymbol{v}_k]_k = u_i v_j \varepsilon_{ijk}$
matrix product	$\boldsymbol{A}\boldsymbol{v} = A_{ij} v_j \boldsymbol{g}_i$	$[\boldsymbol{A}\boldsymbol{v}]_i = A_{ij} v_j$
matrix product	$\boldsymbol{A}^T \boldsymbol{v} = A_{ji} v_j \boldsymbol{g}_i$	$[\boldsymbol{A}^T \boldsymbol{v}]_i = A_{ji} v_j$
matrix product	$\boldsymbol{u}^T \boldsymbol{A}^T \boldsymbol{v} = A_{ji} u_i v_j$	$\boldsymbol{u}^T \boldsymbol{A}^T \boldsymbol{v} = A_{ji} u_i v_j$
matrix product	$\boldsymbol{A} \cdot \boldsymbol{B} = A_{ij} B_{jk} \boldsymbol{g}_i \otimes \boldsymbol{g}_j$	$[\boldsymbol{A} \cdot \boldsymbol{B}]_{ij} = A_{ij} B_{jk}$
matrix trace	$\mathrm{tr}(\boldsymbol{A}) = A_{ii}$	$\mathrm{tr}\boldsymbol{A} = A_{ii}$
matrix determinant	$\det(\boldsymbol{A}) = \varepsilon_{ijk} A_{i1} A_{j2} A_{j3}$	$\det(\boldsymbol{A}) = \varepsilon_{ijk} A_{i1} A_{j2} A_{j3}$
grad(ϕ)	$\nabla\phi = \phi_{,i}\, \boldsymbol{g}_i$	$[\nabla\phi]_i = \phi_{,i}$
grad($\boldsymbol{v}$)	$\nabla\boldsymbol{v} = v_{i,j}\, \boldsymbol{g}_i \otimes \boldsymbol{g}_j$	$[\nabla\boldsymbol{v}]_{ij} = v_{i,j}$
grad($\boldsymbol{A}$)	$\nabla\boldsymbol{A} = A_{ij,k}\, \boldsymbol{g}_i \otimes \boldsymbol{g}_j \otimes \boldsymbol{g}_k$	$[\nabla\boldsymbol{A}]_{ijk} = A_{ij,k}$
div($\boldsymbol{v}$)	$\nabla \cdot \boldsymbol{v} = v_{i,i}$	$\nabla \cdot \boldsymbol{v} = v_{i,i}$
div($\boldsymbol{A}$)	$\nabla \cdot \boldsymbol{A} = A_{ij,j}\, \boldsymbol{g}_i$	$\nabla \cdot \boldsymbol{A} = A_{ij,j}$
curl($\boldsymbol{v}$)	$\nabla \times \boldsymbol{v} = v_{i,j} \varepsilon_{jik} \boldsymbol{g}_k$	$[\nabla \times \boldsymbol{v}]_k = v_{i,j} \varepsilon_{jik}$
curl($\boldsymbol{A}$)	$\nabla \times \boldsymbol{A} = A_{ij,k} \varepsilon_{kjl}\, \boldsymbol{g}_i \otimes \boldsymbol{g}_l$	$[\nabla \times \boldsymbol{A}]_{il} = A_{ij,k} \varepsilon_{kjl}\, \boldsymbol{g}_i \otimes \boldsymbol{g}_l$
div(grad(ϕ))	$\nabla \cdot \nabla\phi = \nabla^2\phi = \phi_{,jj}$	$\nabla \cdot \nabla\phi = \phi_{,ii}$
div(grad($\boldsymbol{v}$))	$\nabla \cdot \nabla\boldsymbol{v} = \nabla^2\boldsymbol{v} = v_{j,ii}\boldsymbol{g}_j$	$[\nabla \cdot \boldsymbol{v}]_j = v_{j,ii}$

2.8 Properties

Here is a list of important properties of vector calculus, as often met in physical problems in Cartesian coordinates with vectors $\mathbf{v}, \mathbf{u}$, etc. and scalars ψ, ϕ, etc.

Gradient properties

$$\nabla(\psi + \phi) = \nabla\psi + \nabla\phi \tag{67}$$
$$\nabla(\psi\phi) = \phi\nabla\psi + \psi\nabla\phi \tag{68}$$
$$\nabla(\psi\mathbf{v}) = \nabla\psi \otimes \mathbf{v} + \psi\nabla\mathbf{v} \tag{69}$$
$$\nabla(\mathbf{v} \cdot \mathbf{u}) = (\mathbf{v} \cdot \nabla)\mathbf{u} + (\mathbf{u} \cdot \nabla)\mathbf{v} + \mathbf{v} \times (\nabla \times \mathbf{u}) + \mathbf{u} \times (\nabla \times \mathbf{v}) \tag{70}$$

Divergence properties

$$\nabla \cdot (\mathbf{v} + \mathbf{u}) = \nabla \cdot \mathbf{v} + \nabla \cdot \mathbf{u} \tag{71}$$
$$\nabla \cdot (\psi\mathbf{v}) = \psi\nabla \cdot \mathbf{v} + \mathbf{v} \cdot \nabla\psi \quad \leftarrow \text{see use in weak formulations} \tag{72}$$
$$\nabla \cdot (\mathbf{v} \times \mathbf{u}) = (\nabla \times \mathbf{v}) \cdot \mathbf{u} - (\nabla \times \mathbf{u}) \cdot \mathbf{v} \tag{73}$$
$$\nabla \cdot (\nabla \times \mathbf{v}) = 0 \tag{74}$$

Curl properties

$$\nabla \times (\mathbf{v} + \mathbf{u}) = \nabla \times \mathbf{v} + \nabla \times \mathbf{u} \tag{75}$$
$$\nabla \times (\psi \mathbf{v}) = \psi (\nabla \times \mathbf{v}) + \nabla \psi \times \mathbf{v} \tag{76}$$
$$\nabla \times (\psi \nabla \phi) = \nabla \psi \times \nabla \phi \tag{77}$$
$$\nabla \times (\mathbf{v} \times \mathbf{u}) = \mathbf{v} (\nabla \cdot \mathbf{u}) - \mathbf{u} (\nabla \cdot \mathbf{v}) + (\mathbf{u} \cdot \nabla) \mathbf{v} - (\mathbf{v} \cdot \nabla) \mathbf{u} \tag{78}$$
$$\nabla \times (\nabla \psi) = \mathbf{0} \tag{79}$$

Laplacian properties

$$\nabla^2 \psi = \nabla \cdot (\nabla \psi) \tag{80}$$
$$\nabla^2 \mathbf{v} = \nabla (\nabla \cdot \mathbf{v}) - \nabla \times (\nabla \times \mathbf{v}) \tag{81}$$
$$\nabla \cdot (\phi \nabla \psi) = \phi \nabla^2 \psi + \nabla \phi \cdot \nabla \psi \tag{82}$$
$$\psi \nabla^2 \phi - \phi \nabla^2 \psi = \nabla \cdot (\psi \nabla \phi - \phi \nabla \psi) \tag{83}$$
$$\nabla^2 (\phi \psi) = \phi \nabla^2 \psi + 2 (\nabla \phi) \cdot (\nabla \psi) + \left(\nabla^2 \phi\right) \psi \tag{84}$$
$$\nabla^2 (\psi \mathbf{v}) = \mathbf{v} \nabla^2 \psi + 2 (\nabla \psi \cdot \nabla) \mathbf{v} + \psi \nabla^2 \mathbf{v} \tag{85}$$
$$\nabla^2 (\mathbf{v} \cdot \mathbf{u}) = \mathbf{v} \cdot \nabla^2 \mathbf{u} - \mathbf{u} \cdot \nabla^2 \mathbf{v} + 2 \nabla \cdot ((\mathbf{u} \cdot \nabla) \mathbf{v} + \mathbf{u} \times (\nabla \times \mathbf{v})) \tag{86}$$

2.9 Integral theorems

In the following we present some formulas that are often used in computational mechanics, ex. to transform surface integrals into volume integrals etc. We show that using the tensor notation one can generalize the divergence theorem (aka Ostrogradsky–Gauss[5] theorem) and the Stokes theorem (aka Stokes-Cartan theorem).

- The **divergence theorem** states that, given a continuously differentiable vector field $\boldsymbol{v}$ on a compact volume V, with piecewise smooth boundary $S = \partial V$ with normal $\boldsymbol{n}$, it holds:

$$\int_V \nabla \cdot \boldsymbol{v} dV = \int_S \boldsymbol{v} \cdot \boldsymbol{n} \, dS \tag{87}$$

  We can express this with tensor notation and generalize it:

  - The divergence theorem for vector fields:

$$\int_V v_{i,i} dV = \int_S v_i n_i \, dS \tag{88}$$

  - The divergence theorem for rank-two tensor fields:

$$\int_V T_{ij,j} dV = \int_S T_{ij} n_j \, dS \tag{89}$$

[5] It is remarkable how many bright scientists attended the University of Göttingen.

- The **Stokes theorem** states that, given a continuously differentiable vector field $\boldsymbol{v}$ on a surface S, with piecewise smooth contour boundary $C = \partial S$ with tangent $\boldsymbol{r}$, it holds:

$$\int_S (\nabla \times \boldsymbol{v}) \cdot \boldsymbol{n}\, dS = \int_C \boldsymbol{v} \cdot d\boldsymbol{r} \tag{90}$$

  We can express this with tensor notation and generalize it:

  - The Stokes theorem for vector fields:

$$\int_S \varepsilon_{ijk} v_{k,j} n_i\, dS = \int_C v_i dx_i \tag{91}$$

  - The Stokes theorem for rank-two tensor fields:

$$\int_S \varepsilon_{ijk} T_{kl,j} n_i\, dS = \int_C T_{il} dx_i \tag{92}$$

3. Classes of problems

Thie following is a short overview of important special cases of PDEs.

3.1 Partial differential equations

Given an unknown function $u(\boldsymbol{x}, t)$, depending on independent variables $\boldsymbol{x}$ where one can have time t among these, in the most general case a **Partial Differential Equation (PDE)** is a (possibly nonlinear) equation that include partial derivatives of u as in:

$$f\left(x_1, \dots x_n, u, \frac{\partial u}{\partial x_1}, \dots \frac{\partial u}{\partial x_n}, \frac{\partial^2 u}{\partial x_1 \partial x_1}, \dots \frac{\partial^2 u}{\partial x_1 \partial x_n}; \dots\right) = 0$$

If f depends only linearly to the partial derivatives, this becomes a *linear PDE*, as in the Laplace equation, Poisson equation, etc., and it is a case of big interest.

The *degree* of the PDE is the highest derivative degree, usually is 2 in many cases of practical interest.

In most cases, PDEs are paired with *boundary conditions*. Assume a boundary $\partial\Omega$ of the Ω domain, and assume that the boundary can be split in disjoint parts $\Gamma_1, \Gamma_2, ...$ with $\Gamma_i \in \partial\Omega$, $\partial\Omega = \cup_i \Gamma_i$, one can impose one or multiple (mixed) boundary conditions of the type:

- The **Dirichlet**[6] **boundary condition** imposes the value b of the function on a part of the boundary:

$$u(\boldsymbol{x}) = b(\boldsymbol{x}) \quad \forall \boldsymbol{x} \in \Gamma_i \tag{93}$$

- The **Neumann** [7] **boundary condition** imposes the value d of the derivative of the function on a part of the boundary:

$$\frac{\partial u(\boldsymbol{x})}{\partial \boldsymbol{n}} = d(\boldsymbol{x}) \quad \forall \boldsymbol{x} \in \Gamma_i \tag{94}$$

- The **Robin boundary condition** imposes the a linear combination of the function and its derivative on a part of the boundary:

$$a_1 u(\boldsymbol{x}) + a_2 \frac{\partial u(\boldsymbol{x})}{\partial \boldsymbol{n}} = r(\boldsymbol{x}) \quad \forall \boldsymbol{x} \in \Gamma_i \tag{95}$$

- The **Cauchy boundary condition** imposes at the same time the value of the function and its derivative on a part of the boundary:

$$\begin{cases} u(\boldsymbol{x}) = b(\boldsymbol{x}) \\ \dfrac{\partial u(\boldsymbol{x})}{\partial \boldsymbol{n}} = d(\boldsymbol{x}) \end{cases} \quad \forall \boldsymbol{x} \in \Gamma_i \tag{96}$$

In the following we will see how few classes of PDE can cover different physical problems under similar mathematical structures.

[6] It is remarkable how many bright scientists attended the University of Göttingen.
[7] It is remarkable how many bright scientists attended the University of Göttingen.

3.2 Poisson equation

The **Poisson equation** is a PDE of the following type:

$$\nabla^2 u = f \tag{97}$$

where u is the unknown function $u(\boldsymbol{x})$ (usually scalar, but also tensor-valued) and f is a given term (scalar, but also tensor-valued) over a manifold, and it is often paired with Dirichlet and Neumann boundary conditions.

In the following we list some physical problems that share the same mathematical structure of the Poisson equation, all assumed in a 3D domain Ω, with $\boldsymbol{x} \in \Omega \subset \mathbb{R}^3$:

- The **steady state heat equation**:

  $$k\nabla^2 T = -q$$

  where $T(\boldsymbol{x})$ is the unknown temperature field in [K] units, k is the thermal conductivity [W/(m K)], q is the heat-flux [W/m^3] of sources in Ω if any - for example heat from uranium fission.

  - a Dirichlet boundary condition $T = T_0,\ \forall \boldsymbol{x} \in \Gamma_i$ means imposing a temperature on a boundary surface Γ_i,
  - a Neumann boundary condition $-k\partial T/\partial \boldsymbol{n} = q_0\ \forall \boldsymbol{x} \in \Gamma_i$ means imposing a heat flux q_0 [W/m^2] through a boundary zone Γ_i, remembering $\boldsymbol{q} = -k\nabla T$.

- The **electrostatics equation**:

  $$\epsilon\nabla^2 \phi = -\rho$$

  where $\phi(\boldsymbol{x})$ is the unknown scalar electric potential field in [V] units, ϵ is the permittivity [F/m] of the material, ρ is the total volume charge density of sources in Ω, in [C/m^3], if any.

  - a Dirichlet boundary condition $\phi = \phi_0,\ \forall \boldsymbol{x} \in \Gamma_i$ means imposing an electric potential [V] on a boundary surface Γ_i,
  - a Neumann boundary condition $\partial\phi/\partial \boldsymbol{n} = -E_0\ \forall \boldsymbol{x} \in \Gamma_i$ means imposing an electric field [V/m] E_0 through a boundary zone Γ_i, remembering that electric field in [V/m] is $\boldsymbol{E} = -\nabla\phi$.

- The **magnetostatics equation**:

  $$\nabla^2 \boldsymbol{A} = -\mu \boldsymbol{J}$$

  where $\boldsymbol{A}(\boldsymbol{x})$ is the unknown vector magnetic potential field in [V s / m] units, μ is the permeability [H/m] of the material, $\boldsymbol{J}$ is the volume current density, a vector in [A / m^2] units.

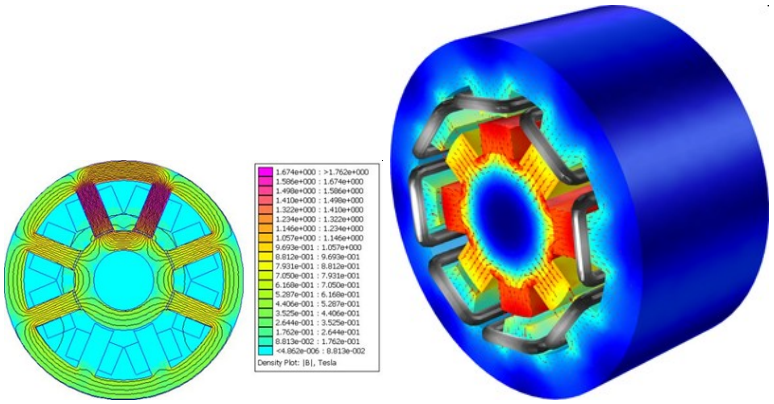

Figure 1: Examples of finite element solution to electrostatics/magnetostatics problems (from FEMM and COMSOL documentation)

  - a Dirichlet boundary condition $\boldsymbol{A} = \boldsymbol{A}_0,\ \forall \boldsymbol{x} \in \Gamma_i$ means imposing a vector magnetic potential [A] on a boundary surface Γ_i,
  - Once the vector magnetic potential is computed, one can get the magnetic field intensity in [A/m] as $\boldsymbol{H} = \frac{1}{\mu}\nabla \times \boldsymbol{A}$ and magnetic flux density as $\boldsymbol{B} = \nabla \times \boldsymbol{A}$.

- The **Newtonian gravitational field equation**:

$$\nabla^2 \phi = 4\pi\rho G$$

  where $\phi(\boldsymbol{x})$ is the unknown scalar gravitational potential field, ρ is the density of the material, G is the gravitational constant.

  - a Dirichlet boundary condition $\phi = \phi_0,\ \forall \boldsymbol{x} \in \Gamma_i$ means imposing a gravitational potential on a boundary surface Γ_i,
  - a Neumann boundary condition $\partial\phi/\partial\boldsymbol{n} = g_0\ \forall \boldsymbol{x} \in \Gamma_i$ means imposing a gravitational field g_0 perpendicular to a boundary zone Γ_i, remembering that $\boldsymbol{g} = -\nabla\phi$.

- The **Reynolds equation on lubrication of thin films**:

$$\nabla \cdot (-\boldsymbol{q}) = \rho W$$

  where $\boldsymbol{q} \in \mathbb{R}^2$ is a mass flow rate $\boldsymbol{q} = \rho\left\{\frac{Uh}{2} - \frac{h^3}{12\eta}\frac{\partial P}{\partial x}, \frac{Vh}{2} - \frac{h^3}{12\eta}\frac{\partial P}{\partial y}\right\}$ that stems from the original Reynolds equation that is often written in in the more conventional form

$$\frac{\partial}{\partial x}\left(\frac{\rho h^3}{12\eta}\frac{\partial P}{\partial x}\right) + \frac{\partial}{\partial y}\left(\frac{\rho h^3}{12\eta}\frac{\partial P}{\partial y}\right) = \frac{\partial}{\partial x}\left(\frac{\rho U h}{2}\right) + \frac{\partial}{\partial y}\left(\frac{\rho V h}{2}\right) + \rho W$$

for P as pressure, ρ as density, U, V, W as (tangential,tangential,vertical) relative surface velocities, h film thickness, η viscosity. Boundary conditions often constrain the pressure on the boundary.

- The Laplace equation is a Poison equation without the source term
- The thin elastic membrane on rigid support is $K\nabla^2 z = -f$ with z vertical displacement and f is vertical force.

3.3 Diffusion equations

Diffusion equations are parabolic PDEs of the following type:

$$\frac{\partial u}{\partial t} = \nabla \cdot (D\nabla u) + R \tag{98}$$

where u is the unknown $u(\boldsymbol{x}, t)$, often scalar-valued but might also be tensor-valued, D is a diffusivity coefficient, $\boldsymbol{V}$ is the velocity, R is a source (sink) term expressing loads per units of volume.

In the following we list some physical problems that share the same mathematical structure of the diffusion equation, all assumed in a 3D domain Ω, with $\boldsymbol{x} \in \Omega \subset \mathbb{R}^3$:

- The transient **heat equation**:

$$\rho c_p \frac{\partial T}{\partial t} - k\nabla^2 T = q \tag{99}$$

  where $T(\boldsymbol{x})$ is the unknown temperature field in [K] units, ρ is the material density [kg/m^3], c_p is the material specific heat capacity [J/(K kg)], k is the thermal conductivity [W/(m K)], q is the heat-flux [W/m^3] of sources in Ω if any - for example heat from uranium fission. Note that with no q source, and by introducing the thermal diffusivity $\alpha = k/(\rho c_p)$, in [m^2/s], one has the more conventional heat equation $\frac{\partial T}{\partial t} - \alpha\nabla^2 T = 0$, and here one recognize that α is the D is a diffusivity coefficient in (98). Also note that assuming steady state, $\frac{\partial T}{\partial t} = 0$, hence the Poisson equation for steady state heat $k\nabla^2 T = -q$ is a special case of this. Boundary and initial conditions:

  - a Dirichlet boundary condition $T = T_i,\ \forall \boldsymbol{x} \in \Gamma_i$ means imposing a temperature on a boundary surface Γ_i,
  - a Neumann boundary condition $-k\partial T/\partial \boldsymbol{n} = q_0\ \forall \boldsymbol{x} \in \Gamma_i$ means imposing a heat flux q_0 [W/m^2] through a boundary zone Γ_i, remembering $\boldsymbol{q} = -k\nabla T$.
  - a initial condition $T = T_0$ at time $t = 0$.

- the **second Fick equation**, governing the transport of mass through diffusive means (dilution in liquids, gases, asdorption in porous media, etc.)

$$\frac{\partial \phi}{\partial t} - D\nabla^2 \phi = 0$$

where $\phi(\boldsymbol{x})$ is the unknown concentration of a substance (ex. in [mol/m^3]), D is the diffusion coefficient, in [m^2/s].

3.4 Convection-diffusion equations

Convection-diffusion equations, sometimes advection-diffusion depending on the context, are PDEs of the following type:

$$\frac{\partial u}{\partial t} = \nabla \cdot (D \nabla u) - \nabla \cdot (\boldsymbol{v} u) + R \tag{100}$$

where u is the unknown $u(\boldsymbol{x}, t)$, often scalar-valued but might also be tensor-valued, D is a diffusivity coefficient, $\boldsymbol{v}$ is the velocity, R is a source (sink) term expressing loads per units of volume.

In the following we list some physical problems that share the same mathematical structure of the diffusion equation, all assumed in a 3D domain Ω, with $\boldsymbol{x} \in \Omega \subset \mathbb{R}^3$:

- the **Smoluchowski equation** of drift-diffusion equation,

  $$\frac{\partial \phi}{\partial t} = \nabla \cdot (D \nabla \phi) - \nabla \cdot \left(\zeta^{-1} \boldsymbol{F} \phi\right) + R$$

  similar to the second Fick equation governing the transport of mass through diffusive means (dilution in liquids, gases, asdorption in porous media, etc.) but also adding a force field $\boldsymbol{F}$ as in electrophoresis and assuming a viscous drag so that velocity is $\boldsymbol{v} = \zeta^{-1} \boldsymbol{F}$. Again, $\phi(\boldsymbol{x})$ is the unknown concentration of a substance (ex. in [mol/m^3]), D is the diffusion coefficient, in [m^2/s]. This is also similar to the **Fokker-Plank equation** if one considers the probability density function of velocities of particles in brownian motion instead of concentrations.

- the incompressible Navier-Stokes equation is closely related to a convection-diffusion problem as:

  $$\frac{\partial \boldsymbol{M}}{\partial t} = \frac{\mu}{\rho} \nabla^2 \boldsymbol{M} - \boldsymbol{v} \cdot \nabla \boldsymbol{M} + (\boldsymbol{f} - \nabla P)$$

  where $\boldsymbol{M} = \rho \boldsymbol{v}$ is fluid momentum per unit of volume, μ is viscosity, P is pressure, $\boldsymbol{f}$ is a load per unit of volume, for instance gravity.

3.5 Linear elasticity equations

A basic representation of **linear elasticity** is given by the following PDE, embedding an equilibrium equation, a strain-displacement equation, and a constitutive equation:

$$\begin{cases} \nabla \cdot \boldsymbol{\sigma} + \boldsymbol{b} = \boldsymbol{0} \\ \boldsymbol{\epsilon} = \dfrac{1}{2} \left(\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T \right) \\ \boldsymbol{\sigma} = \boldsymbol{C} : \boldsymbol{\epsilon} \end{cases} \tag{101}$$

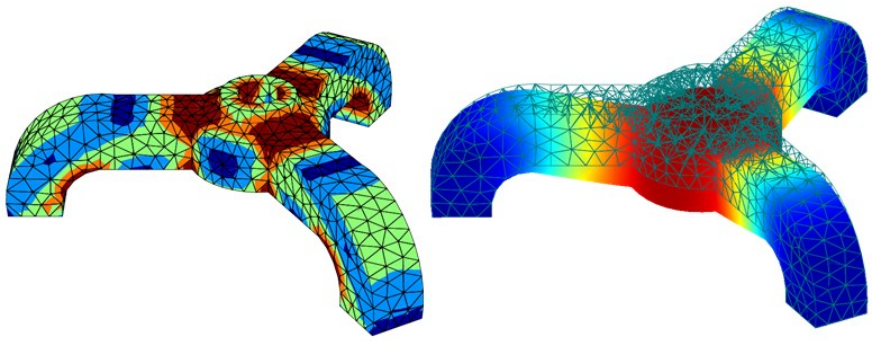

Figure 2: Example of finite element solution to a linear elasticity problem (from GETFem++ documentation)

where $\boldsymbol{u}$ is the unknown vector displacement field, $\boldsymbol{b}$ is an applied per-unit-of-volume load, $\boldsymbol{\epsilon}$ is the strain rank-2 tensor, $\boldsymbol{\sigma}$ is the stress rank-2 tensor, $\boldsymbol{C}$ is the stiffness rank-4 tensor.

For isotropic elasticity, where $\boldsymbol{C}$ can be made dependant on just two parameters and $\boldsymbol{\sigma} = 2\mu\boldsymbol{\epsilon} + \lambda\ \mathrm{tr}(\boldsymbol{\epsilon})\boldsymbol{I}$, one can also write the more compact expression as **Navier-Lamé equation**:

$$\mu\nabla^2\boldsymbol{u} + (\lambda+\mu)\nabla(\nabla\cdot\boldsymbol{u}) + \boldsymbol{b} = \boldsymbol{0}$$

with the Lamé constants $\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}$ and $\mu = G = \frac{E}{2(1+\nu)}$. This is again a PDE.

- a Dirichlet boundary condition $\boldsymbol{u} = \boldsymbol{u}_0,\ \forall\boldsymbol{x} \in \Gamma_i$ means imposing a known displacement (hence a known position) on a boundary surface Γ_i,
- a Neumann boundary condition $\boldsymbol{\sigma}\cdot\boldsymbol{n} = \boldsymbol{t}_0\ \forall\boldsymbol{x} \in \Gamma_i$ means imposing a known traction $\boldsymbol{t}_0$ on the boundary surface Γ_i.

3.6 Navier-Stokes equations

The **compressible Navier-Stokes equation** describes the motion of fluids under very general assumptions:

$$\rho\frac{\partial\boldsymbol{v}}{\partial t} + \rho\,\boldsymbol{v}\cdot\nabla\boldsymbol{v} - \mu\,\nabla^2\boldsymbol{v} + \nabla\bar{p} - \tfrac{1}{3}\mu\,\nabla(\nabla\cdot\boldsymbol{v}) - \boldsymbol{f} = \boldsymbol{0} \tag{102}$$

It stems from the following assumptions:

- the Cauchy momentum equation $\rho\frac{D\boldsymbol{v}}{Dt} = \nabla\cdot\boldsymbol{\sigma} + \boldsymbol{f}$, with material derivative $D\boldsymbol{v}/Dt = \partial\boldsymbol{v}/\partial t + \rho\boldsymbol{v}\cdot\nabla\boldsymbol{v}$,

- the rate of strain is $\boldsymbol{\epsilon} = \frac{1}{2}(\nabla \boldsymbol{v} + \nabla \boldsymbol{v}^T)$,
- the linear stress constitutive equation is a isotropic function of rate of strain as $\boldsymbol{\sigma} = \zeta(\nabla \cdot \boldsymbol{v})\mathbf{I} + \mu\left(\nabla \boldsymbol{v} + (\nabla \boldsymbol{v})^T - \frac{2}{3}(\nabla \cdot \boldsymbol{v})\boldsymbol{I}\right)$
- the continuity equation for balance of mass: $\frac{\partial \rho}{\partial t} = \nabla \cdot (\rho \boldsymbol{v})$

where $\boldsymbol{v}$ is velocity, μ is dynamic viscosity, λ is bulk viscosity, ζ is the "dilational" second viscosity as in $\zeta \equiv \lambda + \frac{2}{3}\mu$, pressure is $\bar{p} = p - \zeta\lambda \cdot \boldsymbol{v}$, external loads per unit of volume are $\boldsymbol{f}$, in many cases $\boldsymbol{f} = \rho \boldsymbol{g}$ where $\boldsymbol{g}$ is gravity acceleration field.

- The **incompressible Navier-Stokes equation** is a special case of the Navier-Stokes equations. Except for particular problems at high Mach numbers, in fact, the compressibility of the fluid can be neglected. Again with the assumption of isotropic Newtonian fluids, and recovering the continuity condition as $\nabla \cdot \boldsymbol{v} = 0$

$$\begin{cases} \rho \dfrac{\partial \boldsymbol{v}}{dt} + \rho\, \boldsymbol{v} \cdot \nabla \boldsymbol{v} - \mu\, \nabla^2 \boldsymbol{v} + \boldsymbol{\nabla} p - \boldsymbol{f} = \mathbf{0} \\ \nabla \cdot \boldsymbol{v} = 0 \end{cases} \tag{103}$$

- The **steady state Navier-Stokes equation** is a special case of the Navier-Stokes equations with no dependence on time. With the simplification of Newtonian fluid and incompressible flow one has:

$$\begin{cases} \rho\, \boldsymbol{v} \cdot \nabla \boldsymbol{v} - \mu\, \nabla^2 \boldsymbol{v} + \boldsymbol{\nabla} p - \boldsymbol{f} = \mathbf{0} \\ \nabla \cdot \boldsymbol{v} = 0 \end{cases} \tag{104}$$

- The **Stokes equations** is a special case of the steady state Navier-Stokes equations where the advective terms are negligible respect to the viscous forces, as in slow creeping fluids with low Reynolds numbers $Re \to 0$. With simplification of Newtonian fluid and incompressible flow one has:

$$\begin{cases} -\mu\, \nabla^2 \boldsymbol{v} + \boldsymbol{\nabla} p - \boldsymbol{f} = \mathbf{0} \\ \nabla \cdot \boldsymbol{v} = 0 \end{cases} \tag{105}$$

- The **inviscid Euler equations** is a special case of the Navier-Stokes equations where the diffusive terms are negligible, as in high Reynolds numbers $Re \to \infty$. With simplification of Newtonian fluid and incompressible flow one has:

$$\begin{cases} \rho \dfrac{\partial \boldsymbol{v}}{dt} + \rho \boldsymbol{v} \cdot \nabla \boldsymbol{v} + \boldsymbol{\nabla} p - \boldsymbol{f} = \mathbf{0} \\ \nabla \cdot \boldsymbol{v} = 0 \end{cases} \tag{106}$$

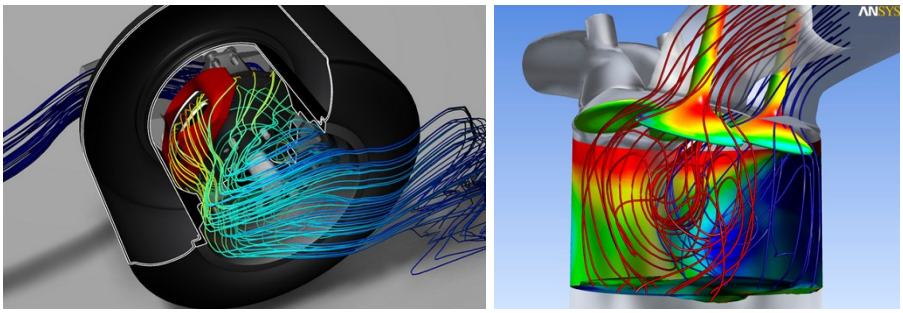

Figure 3: Examples of computational fluid dynamics, left: with finite elements (Autodesk CFD simulator), right: with finite volume method (from ANSYS Fluent documentation)

4. Discretization

There are different ways to cast the previous infinite dimensional problems into finite dimensional problems that can be attacked with computational tools. Among the different strategies we mention the finite element method, the finite different method, the finite volume method, etc. In the following we will focus on the finite element Galerkin method.

4.1 Function spaces

The following is a primer on basic concepts of functional analysis, this is useful because literature on FEA often cite concepts like Sobolev spaces etc.

- A **vector space**, over a scalar field $K \in \mathbb{R}$, is a set V equipped with addition $+ : V \times V \to V$, commutative and associative, and scalar multiplication $\cdot : F \times V \to V$, distributive and associative.
- An **inner product space** is a vector space equipped with an inner product, $\langle \cdot, \cdot \rangle : V \times V \to F$.
- A **normed vector space** is a vector space equipped with a norm, $\| \cdot \| : V \to \mathbb{R}$.
- A L^p-norm is a norm defined as $\|\boldsymbol{x}\|_p = (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{1/p}$, the Euclidean L^2 norm being the most common.
- A space V is **complete** if all Cauchy sequences converge to an element in V.
- A **Banach** space is a complete normed vector space.
- A **Hilbert**[8] space is a complete normed inner product space.

[8] It is remarkable how many bright scientists attended the University of Göttingen.

- A **function space** is a topological space whose points are functions. Some examples will follow.
  - B is the space of *bounded* functions
  - C is the space of *continuous* functions
  - C_c is the space of *continuous* functions with compact support
  - C^r is the space of functions that are *continuous* up to the first r *derivatives*
  - C^∞ is the space of *smooth functions*, ex. $f = \sin(x)$.
  - L^p is the **Lebesgue space** of measurable functions with finite norm $\|f\|_{L^p} = \left(\int_\Omega |f|^p d\mu\right)^{1/p}$, these functions are not necessarily continuous.
    $$L^p(\Omega) = \left\{ f \;\middle|\; \int_\Omega |f|^p d\mu < \infty \right\}$$
  - Lebesgue L^p space is a Banach space with norm $\|f\|_{L^p} = \left(\int_\Omega |f|^p d\mu\right)^{1/p}$.
  - Lebesgue L^2 space is a Hilbert space with inner product $\langle u, v\rangle_{L^2} = \int_\Omega uv d\mu$.
  - Let $f \in L^2(\Omega)$, the *weak derivative* $\partial^\alpha f$ of order α is a function in $L^2(\Omega)$ that satisfies:
    $$\int_\Omega \partial^\alpha f \psi d\mu = (-1)^{|\alpha|} \int_\Omega f \partial^{|\alpha|} \psi d\mu, \quad \forall \psi \in C_c^\infty$$
    It does not require the function to be differentiable, it is enough that it is integrable.
  - $W^{m,p}$ is the **Sobolev space** is the subspace of $L^p(\Omega)$ functions which possess up to order-m weak derivatives ∂^α, with $\alpha \le m$.
    $$W^{m,p}(\Omega) = \{ f \in L^p(\Omega) \mid \partial^\alpha f \in L^p(\Omega), 1 \le \alpha \le m \}$$
  - H^m is the **Hilbert-Sobolev space** more in detail is a $W^{m,2}$ space:
    $$H^m(\Omega) = \left\{ f \in L^2(\Omega) \mid \partial^\alpha f \in L^2(\Omega), 1 \le \alpha \le m \right\}$$
  - The norm in H^m is defined as
    $$\|f\|_{H^m} = \left(\sum_{\alpha=0}^{m} \int_\Omega |\partial^\alpha f|^2 d\mu \right)^{1/2} = \left(\sum_{\alpha=0}^{m} \int_\Omega ||\partial^\alpha f||_{L^2}^2 d\mu \right)^{1/2}$$
  - The importance of Sobolev space is that solutions of PDEs are often searched in H^m space (so called *weak*, or integral solutions) rather than in the C^k space (as in *strong* solutions for order-k PDEs), thus relaxing the continuity requirement.

- We denote with H_0^m the subspace of H^m with $m-1$ derivatives equal to zero at the boundary.
- In detail H_0^1 is the subspace of H^1 functions that are zero at the boundary.
- In detail H_g^1 is the subspace of H^1 functions that have a prescribed value g at the boundary.
- Functions in H^m are required to be in C^{m-1}.
- Example: the step function is in L^2 but not in H^1 (as derivative contains Dirac deltas), the hat function is in L^2 and in H^1, meanwhile it is in C^0.

4.2 The weak formulation

The weak formulation reduces the regularity assumptions for the solution of the PDE: whereas the solution of the original PDE must be in C^2, the weak solution must be only in H_0^1, a requirement that makes the problem much more practical. This is the positive side effect of using an integral (variational) form of the equations.

We introduce the weak formulation starting from the case of a generic Poisson equation (a PDE like the steady state heat equation or the electrostatic problem already discussed previously) with unknown scalar function u, paired with Dirichlet and Neumann boundary conditions:

$$\boxed{\begin{cases} k\nabla^2 u + f = 0 & \forall \boldsymbol{x} \in \Omega \\ u = u_D, & \forall \boldsymbol{x} \in \Gamma_D \\ k\partial u/\partial \boldsymbol{n} = q_N & \forall \boldsymbol{x} \in \Gamma_N \end{cases}} \tag{107}$$

A solution to the strong form above should satisfy $u \in C^2(\Omega)$, that is, at least it should be twice differentiable over the entire domain Ω.

We will rewrite it as a weak formulation, that requires less stringent requirement on the functional class of the solution -letting it to be approximated by a piecewise polynomial $u^h \approx u$ at the end. In fact we will assume a $u \in H^1_{u_D}(\Omega)$, that is we assume the solution to be a function with just the first derivative being square integrable, and with prescribed value u_D at the boundary in order to satisfy the Dirichlet boundary condition.

We also introduce a *test function*, namely $v \in H_0^1(\Omega)$, assumed null at the Γ_D boundary. For compactness we introduce functional spaces $\mathcal{U}$ and $\mathcal{U}_0$ as:

$$u \in \mathcal{U} = \left\{ u \in H^1(\Omega) \ : \ u = u_D \forall \boldsymbol{x} \in \Gamma_D \right\} \tag{108}$$

$$v \in \mathcal{U}_0 = \left\{ v \in H^1(\Omega) \ : \ v = 0 \forall \boldsymbol{x} \in \Gamma_D \right\} \tag{109}$$

We start multiplying the terms of the PDE by the test function v and we integrate over the domain, obtaining:

$$\int_\Omega v\, k\nabla^2 u \, d\Omega + \int_\Omega v\, f \, d\Omega = 0 \tag{110}$$

The first term can be integrated by parts (or just applying the Leibnitz product rule, that leads here to the known property of divergence operator: $\nabla \cdot (v\mathbf{A}) = v\nabla \cdot \mathbf{A} + \mathbf{A} \cdot \nabla v$ where our $\nabla^2 u$ is in fact $\nabla \cdot \mathbf{A}$ for $\mathbf{A} = \nabla u$) thus obtaining:

$$\int_\Omega k\nabla \cdot (v\nabla u)\, d\Omega - \int_\Omega k\nabla v \cdot \nabla u\, d\Omega + \int_\Omega v\, f\, d\Omega = 0 \tag{111}$$

Then, we use the Gauss divergence theorem to turn the first term into a boundary integral:

$$\int_\Omega k\nabla \cdot (v\nabla u)\, d\Omega = \int_\Gamma kv\nabla u \cdot \boldsymbol{n}\, d\Gamma$$

where on the Dirichlet boundaries Γ_D it simplifies to zero because $v = 0$ on Γ_D by construction, whereas on the boundaries Γ_N it assumes the value prescribed by Neumann condition as $k\partial u/\partial \boldsymbol{n} = k\nabla u \cdot \boldsymbol{n} = q_N$:

$$\int_{\Gamma_D} kv\nabla u \cdot \boldsymbol{n}\, d\Gamma_D = 0 \tag{112}$$

$$\int_{\Gamma_N} kv\nabla u \cdot \boldsymbol{n}\, d\Gamma_N = \int_{\Gamma_N} v\, q_N\, d\Gamma_N \tag{113}$$

Substitute the above in (111) and see that the weak formulation simplifies to:

$$\int_\Omega k\nabla v \cdot \nabla u\, d\Omega = \int_\Omega v\, f\, d\Omega + \int_{\Gamma_N} v\, q_N\, d\Gamma_N \tag{114}$$

We note in passing that going to the weak formulation of the Poisson equation, the original ∇^2 term disappeared, and now it does not require anything more than the first derivative of u to be integrable (and same for the test function v).

Also note that if no Neumann conditions are required it, the weak formulation simplifies even more:

$$\int_\Omega k\nabla v \cdot \nabla u\, d\Omega = \int_\Omega v\, f\, d\Omega$$

The weak formulation (114) can be rewritten with a more compact notation, introducing a bilinear operator $\mathcal{B} : \mathcal{U} \times \mathcal{U}_0 \to \mathbb{R}$ and a linear operator $\mathcal{L} : \mathcal{U}_0 \to \mathbb{R}$ that, in our Poisson case above, are:

$$\mathcal{B}(u, v) = \int_\Omega k\nabla v \cdot \nabla u\, d\Omega \tag{115}$$

$$\mathcal{L}(v) = \int_\Omega v\, f\, d\Omega + \int_{\Gamma_N} v\, q_N\, d\Gamma_N \tag{116}$$

This said, the weak form can be written as

$$\boxed{\mathcal{B}(u,v) = \mathcal{L}(v)} \tag{117}$$

In the following we see that most of the PDE problems seen in the previous sections can be expressed with a sum of bilinear and linear terms like the above (but the expressions of the bilinear and linear operators can be different in the sense that the integrands could have different expressions).

We remark in passing that, for symmetrical $\mathcal{B}(u,v)$, (117) is equivalent to a variational problem $u = \text{argmin}_u I[u]$ with the functional $I[u] = \frac{1}{2}\mathcal{B}(u,u) - \mathcal{L}(u)$ representing some form of energy, and whose stationary condition is $\mathcal{B}(u,\partial u) - \mathcal{L}(\partial u) = 0$; note how in elasticity the ∂u can be considered a virtual displacement, and this turns into the principle of virtual work.

4.3 Discretization

Now, one can introduce an approximation of u and v, to pass from an infinite-dimensional problem to a finite-dimensional problem that depends on a countable set of variables, hence more computer-friendly. We assume a discretized approximate solution $u^h \approx u$, with $u^h \in \mathcal{U}^h \subset \mathcal{U}$. Here h represents the discretization size, and $u^h \to u$ as $h \downarrow 0$. Similarly, $v^h \approx v$, with $v^h \in \mathcal{U}_0^h \subset \mathcal{U}_0$.

The finite-dimensional function spaces $\mathcal{U}^h, \mathcal{U}_0^h$ are built using n *shape functions*, here named $N_i(\boldsymbol{x})$, as in:

$$u^h(\boldsymbol{x}) = \sum_i^n \hat{u}_i N_i(\boldsymbol{x}) \tag{118}$$

$$v^h(\boldsymbol{x}) = \sum_j^n \hat{v}_j N_j(\boldsymbol{x}) \tag{119}$$

Thank to this discretization, the unknown is not a function u anymore, but it is rather a vector with a finite amount of $\hat{u}_i$ unknowns. We also remark that one can split the entire domain Ω in multiple finite elements, each with its $\hat{u}_i$ unknowns, leading to a finer discretization and to a larger amount of $\hat{u}_i$ unknowns.

Shape functions can be linear, quadratic, etc., depending on the finite element type.

This variational setting is attributed to W.Ritz[9] and B.Galërkin[10], hence the name **Galerkin** method. More in detail, if the same approximation space is used for both u and v, this is sometimes referred as **Bubnov-Galerkin** method, and if a different approximation space is chosen for v, this is called **Petrov-Galerkin** method. The modern mathematical setting and terminology for the finite element method is credited to R.Courant[11].

[9] It is remarkable how many bright scientists attended the University of Göttingen.

[10] The pronounciation of ë in russian sounds like *yo*, so one should pronounce *Galyorkin*, I do not why everyone spells it Gal*e*rkin.

[11] It is remarkable how many bright scientist attended the University of Göttingen.

Using the discretization, the weak form can be written:

$$\mathcal{B}(u^h, v^h) = \mathcal{L}(v^h) \tag{120}$$

$$\mathcal{B}\left(\sum_i^n u_i N_i(\boldsymbol{x}), \sum_j^n v_j N_j(\boldsymbol{x})\right) = \mathcal{L}\left(\sum_j^n v_j N_j(\boldsymbol{x})\right) \tag{121}$$

Exploiting the fact that $\mathcal{B}$ is bilinear and $\mathcal{L}$ is linear, one can take the $\hat{u}_i$ and $\hat{v}_i$ weights outside integrals, leading to:

$$\sum_j^n \hat{v}_j \sum_i^n \hat{u}_i \mathcal{B}\left(N_i(\boldsymbol{x}), N_j(\boldsymbol{x})\right) = \sum_j^n \hat{v}_j \mathcal{L}\left(N_j(\boldsymbol{x})\right) \tag{122}$$

One can simplify the $\hat{v}_j$ terms and obtain a very simple form of the discretized weak form where one can see that the bilinear and linear operators act only on the shape functions:

$$\sum_i^n \hat{u}_i \mathcal{B}\left(N_i(\boldsymbol{x}), N_j(\boldsymbol{x})\right) = \mathcal{L}\left(N_j(\boldsymbol{x})\right) \tag{123}$$

In sake of further compactness one can group all $\hat{u}_i$ in the vector of unknowns $\hat{\boldsymbol{u}}$, introduce a matrix $K \in \mathbb{R}^{n \times n}$ and a vector $\boldsymbol{b} \in \mathbb{R}^n$, and obtain a linear system:

$$\boxed{\begin{aligned} &K\hat{\boldsymbol{u}} = \boldsymbol{b} \\ &K_{ij} = \mathcal{B}\left(N_i, N_j\right) \\ &b_j = \mathcal{L}\left(N_j\right) \end{aligned}} \tag{124}$$

This $K\boldsymbol{u} = \boldsymbol{b}$ linear system gives the solution $\boldsymbol{u}$ to the discretized PDE problem. Of course, the finer the discretization, the larger the size of $\hat{\boldsymbol{u}}$, hence the larger the matrix K, and the more computationally intensive is the solution. To this end, one might consider using special types of solvers, such as Krylov solvers, that exploit the fact that in many cases the K matrix is sparse and contains many structural zeros: please go to the section on linear solvers for a primer on this topic.

4.4 Quadrature

One can see that, for computing the elements of K, in our example of the Poisson equation, one needs to compute

$$K_{ij} = \mathcal{B}\left(N_i, N_j\right) = \int_\Omega k \nabla N_i(\boldsymbol{x}) \cdot \nabla N_j(\boldsymbol{x}) d\Omega$$

just like for the elements of $\boldsymbol{b}$ one computes

$$b_j = \mathcal{L}\left(N_j\right)) = \int_\Omega N_i(\boldsymbol{x})\, f\, d\Omega + \int_{\Gamma_N} N_i(\boldsymbol{x})\, q_N\, d\Gamma_N$$

This amounts to performing some type of integration over a domain (the finite element volume, for instance) per each term. This can be done via analytical methods only in limited cases, otherwise using Newton-Cotes quadrature, or Gauss-Legendre quadrature, or other approximations. We discuss the Gauss quadrature briefly, ie. an approximation of the integral via a weighted sum of values sampled in some Gauss points $\boldsymbol{x}_g$ scattered in the volume domain.

In general, Gauss integration over nGP Gauss points with corresponding w_g weights at $\boldsymbol{x}_g$ abscyssae points is done with a sum, for example:

$$\int_\Omega k\nabla N_i(\boldsymbol{x}) \cdot \nabla N_j(\boldsymbol{x}) d\Omega = \sum_{g=1..n_{GP}} w_g \nabla N_i(\boldsymbol{x}_g) \cdot \nabla N_j(\boldsymbol{x}_g)$$

and the same for other bilinear/linear terms.

The lower the degree of shape functions, the smaller the amount of Gauss point samples that are needed to correctly compute the quadrature; for example, a tetrahedral finite element with its four linear shape functions requires only a single Gauss point in the middle, etc.

Note, however, that most often the w_g and values are tabulated assuming generic abscyssae ζ_g ranging in the $[-1, +1]$ interval, hence one has to express $\boldsymbol{x} = \boldsymbol{x}(\boldsymbol{\zeta})$ (as happens in the *isoparametric* finite element approach, where $\boldsymbol{\zeta}$ for example contains the three parametric coordinates along the direction of a brick each ranging in $[-1, +1]$ and shape functions are functions of these parametric coordinates) and simply scale the result using jacobians $\partial\boldsymbol{x}/\partial\boldsymbol{\zeta}$ as in

$$\int_a^b f(x)dx = \int_{-1}^{+1} J_\zeta f(\zeta) d\zeta \approx J_\zeta \sum_{g=1..n_{GP}} w_g f(\zeta_g)$$

that in our case would lead to:

$$\int_\Omega k\nabla N_i(\boldsymbol{x}) \cdot \nabla N_j(\boldsymbol{x}) d\Omega \approx J_{\boldsymbol{\zeta}} \sum_{g=1..n_{GP}} w_g \nabla N_i(\boldsymbol{\zeta}_g) \cdot \nabla N_j(\boldsymbol{\zeta}_g)$$

The reader interested in quadrature should look also at the following connected problems: *hourglass modes* (zero-energy spurious modes), selective and *reduced integration*, *shear locking* (ex.in Timoshenko beams) and how to get rid of it using ad-hoc integration.

4.5 Constraints

Apart from the already mentioned treatment of Dirichlet and Neumann conditions, a straightforward and powerful way to apply generic constraints to nodes of a FE mesh -especially in a context of mechanical problems where nodes represent points of a deformable material- is to couple the discretized PDE with algebraic constraints. A (possibly linear) constraint equation is a function that gives zero residual when satisfied:

$$\boldsymbol{C}(\hat{\boldsymbol{u}}) = \boldsymbol{0}$$

For example, in the context of 3D continuum elasticity, if one requires that the displacement $\hat{\boldsymbol{u}}_k$ of the k-th node is equal to some $\overline{\boldsymbol{u}}$ value, one would write:

$$\boldsymbol{C}(\hat{\boldsymbol{u}}) = \hat{\boldsymbol{u}}_k - \overline{\boldsymbol{u}}$$

Another example, again with a node in a 3D continuum: if one requires the node to be able to move on the yz plane of a coordinate system with rotation matrix R, while precluding the x displacement relative to such coordinate system, one would write:

$$\boldsymbol{C}(\hat{\boldsymbol{u}}) = [1, 0, 0]R^T\hat{\boldsymbol{u}}_k$$

One can add multiple constraints in this way, obtaining a large $\boldsymbol{C}(\hat{\boldsymbol{u}})$ vector of m scalar constraints, $\boldsymbol{C} : \mathbb{R}^n \to \mathbb{R}^m$. The constraint jacobian is a matrix $C_u \in \mathbb{R}^{m\times n}$:

$$C_u = \frac{\partial \boldsymbol{C}(\hat{\boldsymbol{u}})}{\partial \hat{\boldsymbol{u}}}$$

Not all m constraints act on all the n variables of the system-wide $\hat{\boldsymbol{u}}$ vector, so the jacobian is often a very sparse matrix.

By linearizing the constraint equation, one has

$$\boldsymbol{C}(\hat{\boldsymbol{u}}) \approx C_u(\hat{\boldsymbol{u}} - \hat{\boldsymbol{u}}_0) + \boldsymbol{C}(\hat{\boldsymbol{u}}_0)$$

where here we pose $\hat{\boldsymbol{u}}_0 = \boldsymbol{0}$, thus the constraint equations leads to a linear algebra equation $C_u\hat{\boldsymbol{u}} + \boldsymbol{C}(\hat{\boldsymbol{u}}_0) = \boldsymbol{0}$.

Roughly speaking, the transpose of the jacobian matrix multiplied by a unknown multiplier $\boldsymbol{\lambda}$ gives a reaction force in global coordinates, where the unknown multiplier is the reaction in the coordinates of the constraint, therefore the final linear system to be solved becomes:

$$\boxed{\begin{bmatrix} K & C_u^T \\ C_u & 0 \end{bmatrix} \begin{Bmatrix} \hat{\boldsymbol{u}} \\ \boldsymbol{\lambda} \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{b} \\ -\boldsymbol{C}(\boldsymbol{u}_0) \end{Bmatrix}} \tag{125}$$

The structure of the linear system above, featuring a square matrix K bordered to the right and below by constraint jacobians C_u and additional unknown multipliers $\boldsymbol{\lambda}$ (the constraint reaction forces in this case) like in a Karush Kuhn Tucker (KKT) matrix, is very frequent in other problems involving constraints - not only in structural mechanics. Solving the KKT matrices efficiently is a key to a robust and fast computational tool.

The question is: we added constraints using the KKT matrix by enforcing constraints at level of an already discretized problem, but is it possible to recover the same KKT machinery also if the constraints are defined in the previous step, i.e. when the formulation is still infinite-dimensional? The answer is yes, but the variational procedure is left to the user.

4.6 Weak formulations for other classes of problems

So far, we have seen how to translate a Poisson problem (stationary heat problem, magnetostatics, Reynolds lubrication etc.) into a corresponding weak form,

and from the weak form we obtained a discretized problem to be solved via a linear system. One can see that other weak formulations can be derived for other classes of problems. For sake of compactness here we present just a few.

- Weak form of the Navier-Lamé (linar elasticity) equation:

$$\int_\Omega \boldsymbol{\sigma}(\boldsymbol{u}) : \boldsymbol{\epsilon}(\boldsymbol{v})\, d\Omega = \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v}\, d\Omega + \int_\Gamma \boldsymbol{\sigma}(\boldsymbol{u})\boldsymbol{n} \cdot \boldsymbol{v}\, d\Gamma \tag{126}$$

  This can be again expressed as $\mathcal{B}(\boldsymbol{u}^h, \boldsymbol{v}^h) = \mathcal{L}(\boldsymbol{v}^h)$, and therefore also to a linear system

$$K\hat{\boldsymbol{u}} = \boldsymbol{b}$$

  once discretized (but look how the expression of $\mathcal{B}$ changes respect to the Poisson problem).

- Weak form of the steady-state incompressible Navier-Stokes equation:

$$\begin{cases} \displaystyle\int_\Omega \mu \nabla \boldsymbol{u} \cdot \nabla \boldsymbol{v} + \int_\Omega \rho(\boldsymbol{u} \cdot \nabla)\boldsymbol{u} \cdot \boldsymbol{v} - \int_\Omega p \nabla \cdot \boldsymbol{v} = \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v} + \int_{\Gamma_N} \boldsymbol{h} \cdot \boldsymbol{v} \\ \displaystyle\int_\Omega q \nabla \cdot \boldsymbol{u} = 0 \end{cases} \tag{127}$$

  assuming a Dirichlet condition $\boldsymbol{\sigma}(\boldsymbol{u}, p)\boldsymbol{n} = \boldsymbol{h}$ in Γ_N. Note that this required an additional test function q, for the incompressibility constraint, and note that the discretization will lead to a KKT matrix,

$$\begin{bmatrix} K & C_u^T \\ C_u & 0 \end{bmatrix} \begin{Bmatrix} \hat{\boldsymbol{u}} \\ \hat{\boldsymbol{p}} \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{b} \\ \boldsymbol{0} \end{Bmatrix}$$

  where the lower-left and upper-right submatrices C_u are indeed jacobians of the incompressibility constraint. The K upper left submatrix will depend on the viscosity and density. The unknowns of the discretized KKT system are indeed the vector of discrete velocities $\hat{\boldsymbol{u}}$, and the vector of discrete pressures $\hat{\boldsymbol{p}}$ as lagrangian multipliers of the constraints.[12]

  Note that if adding also the time dependent term, $\displaystyle\int_\Omega \rho \frac{\partial \boldsymbol{u}}{\partial t} \cdot \boldsymbol{v}$, one has the weak form of the full incompressible Navier-Stokes equation.

[12] Finite elements for velocity and pressure must be to be well matched, otherwise the CFD discretized problem has no proper solution, for instance the shape function polynomials of pressure are often one degree lower respect to the degree of polynomials of velocity.

5. Numerical methods for linear systems

We assume a square matrix $A \in \mathbb{R}^{n \times n}$ and a known vector $\boldsymbol{b} \in \mathbb{R}^n$. The problem we want to solve is

$$A\boldsymbol{x} = \boldsymbol{b}$$

with $\boldsymbol{x}$ unknowns.

This is a problem often encountered in the numerical methods for computational mechanics.

Potential difficulties stem from the following facts:

- the A matrix might have a large size,
- the A matrix might be rank-deficient or badly conditioned,
- in some context the A is never explicitly assembled, because it is known in a factored form or as a sparse data structure, and in an optimal scenario the solver should be able to exploit this data storage type without building the full matrix.

There are different classes of solvers that can be used, each with benefits or drawbacks. Among the most relevant classes:

- Direct methods
- Stationary iterations (fixed point iterations)
- Krylov methods
- Multigrid methods

In the following we briefly list the main solver types belonging to these classes.

5.1 Direct methods

Direct methods provide the exact solution to the linear problem, at least up to the floating-point precision of the machine, in a limited number of steps. Their drawback is that, except some special cases (ex. matrices with banded structure) the number of steps grows superlinearly with the number of unknowns, up to $O(n^3)$ complexity in the worst case for full matrices.

The **Gauss elimination** is one of the oldest direct schemes. It reduces the matrix in a form where the lower left triangle is filled with zeros, so a quick backward elimination procedure can compute $\boldsymbol{x}$. Note that the algorithm requires a pivoting strategy in case it encounters a zero pivot (a zero on the diagonal) because it would cause a division by zero. Pivoting can still fail if the matrix is singular, in detail it fails p steps before reaching the last line if p is the number of zero singular values.

The **LU decomposition** decomposes the matrix into $A = LU$ with upper right triangular matrix U ad lower left triangular matrix L. This done, the

system is easily solved via backward substitutions. This method is quite similar to the Gauss substitution in terms of storage, complexity, CPU time.

The **Cholesky decomposition** decomposes the matrix into $A = LL^T$ with lower left triangular matrix L, this can be seen as a special type of LU decomposition for symmetric positive definite A matrices. Exploiting symmetry, it is a twice as faster as LU.

The **LDLt decomposition** decomposes the matrix into $A = LDL^T$ with lower left triangular matrix L and diagonal D, and it is similar to the Cholesky decomposition. It avoids square roots of Cholesky algorithm, and the matrix can be also negative definite (leading to negatives values in D diagonal, in fact). Pivoting can be used to make the algorithm more stable if pivots are too small, but to avoid breaking symmetry it cannot use generic pivoting, and it must use a diagonal pivoting.

5.1.1 Parallelization and HPC

In general, it is difficult to implement direct methods on *parallel computing architectures* and on High Performance Computing (HPC) at large. There are some noticeable examples of direct parallel solvers, though, and among these we mention PARDISO from the MKL Intel libarary, and MUMPS. Both are able to exploit multicore CPUs to a good extent.

5.2 Stationary iterations

This is the simplest class of iterative methods. Stationary methods, also known as fixed point methods, reach the solution by iterative refinements.

Benefits: their implementation is very simple, they do not need to store the full matrix (they are sparse-matrix friendly), in some flavors they are robust in case of singular matrices, their implementation is most often based on a single computational primitive: (sparse) matrix by vector. Also, they can be easily modified to become *projected* fixed point iterations to solve also nonlinear complementarity problems (ex obstacle problems).

Drawbacks: their convergence is very slow (in some case they can stall and converge to desired tolerance in a too large number of iterations, hence eliding the benefit of the simple implementation). Especially in the case of PDEs arising from FEA, and more markedly FEA with structural elements such as shells, the A matrix might be badly conditioned (bad spectral radius) and this has a huge negative effect on the speed of convergence, up to rendering those methods unusable.

Here is a list of the most famous fixed point iterations for solving $A\boldsymbol{x} = \boldsymbol{b}$

5.2.1 Jacobi method

The **Jacobi method** assumes a splitting $A = D + R$, and obtain the solution by iterating

$$\boldsymbol{x}^{(k+1)} = D^{-1}(\boldsymbol{b} - R\boldsymbol{x}^{(k)})$$

or, in element-wise notation, by iterating

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n.$$

up to $k \leq k_{maxiters}$ or up to reaching tolerance. A sufficient -but not necessary- condition for the method to converge is that A is strictly diagonally dominant. One can see that the Jacobi method is intuitively like solving each linear equation separately for one variable, assuming the other variable fixed, and repeating the process in k loop.

5.2.2 Gauss-Jacobi method

The **Gauss-Jacobi method** assumes a splitting $A = L + U$, where L contains the diagonal and U doesn't, and obtain the solution by iterating

$$\boldsymbol{x}^{(k+1)} = L^{-1}(\boldsymbol{b} - U\boldsymbol{x}^{(k)})$$

There is no need to invert L explicitly, neither to build L or U explicitly, in fact one can exploit the triangular nature of L so that $L^{-1} \ldots$ becomes, componentwise, a sequential backward substitution:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n.$$

From an intuitive point of view, this method is similar to the Jacobi method, i.e solves each linear equation separately for one variable, sequentially in each iteration k, but instead of keeping the other variable fixed as they were known at the last $k-1$ iteration, here it reuses instantly the variables as soon as updated.

5.2.3 SOR successive over-relaxation method

The **SOR successive over-relaxation method** assumes a splitting $A = L + D + U$ with D diagonal, then performs

$$\mathbf{x}_{(k+1)} = (D + \omega L)^{-1}\big(\omega \mathbf{b} - [\omega U + (\omega - 1)D]\mathbf{x}_{(k)}\big) = L_w \mathbf{x}_{(k)} + \mathbf{c},$$

Again, this can be written sequentially to avoid matrix inverse, and we get the sequential form:

$$x_{(k+1)i} = (1-\omega)x_{(k)i} + \frac{\omega}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_{(k+1)j} - \sum_{j>i} a_{ij} x_{(k)j} \right), \quad i = 1, 2, \ldots, n.$$

It is similar to the Gauss-Jacobi method, with the exception that it introduces the relaxation parameter ω. For $\omega = 1$ it is exactly Gauss-Jacobi. For higher ω it can converge faster - but this depends on the spectral radius of the matrix, too high ω will lead to divergence.

5.2.4 Parallelization and HPC

Note that Gauss-Jacobi parallelize very easily (just assign lines of the matrix to different threads and perform the updates: these are independent and they just need a synchronization at each k iteration).

On the other side, SOR and Gauss-Jacobi updates imply cross-dependency of variables, hence cannot be parallelized as in Jacobi otherwise one can run into race conditions.

To avoid this issue, one must develop *red-black partitioning* methods.

5.3 Krylov methods

Krylov methods, also called Krylov subspace methods, are based on the construction of a basis of successive matrix powers times the initial residual. A Krylov subspace is:

$$\mathcal{K}_r(A, \boldsymbol{b}) = \text{span}\,\{\boldsymbol{b}, A\boldsymbol{b}, A^2\boldsymbol{b}, \ldots, A^{r-1}\boldsymbol{b}\}.$$

Building the subspace draws on *matrix by vector* multiplications, and we will see that this computational primitive is the basic building block of all Krylov methods.

Krylov methods find the $\mathcal{K}_r(A, \boldsymbol{b})$ space, and in theory Krylov methods should converge to a maximum number of iterations n with n being the amount of unknowns. In many practical situations, however, they might converge to acceptable precision much faster than this worst case scenario. On the other hand, finite precision of floating point algebra means that the upper limit n could be even broken, and this could happen in unlucky cases with badly conditioned matrices, for example.

For very large scale problems, Krylov methods are competitive respect to direct methods, and the fact that they can work seamlessly with sparse matrices make them a good choice for many PDE solvers.

Parallelization of Krylov methods can be performed at various levels (ex on GPU via CUDA or OpenCL, on multicore using OpenMP, or even on multiple nodes using MPI for extremely large problems) and in general it boils down to parallelizing the matrix x vector operation. Other computational primitives that might be parallelized are the inner products and the vector sums and scaling.

In the following we comment some of the most used Krylov methods, leaving the discussion on others (QMR, TFQMR, QMRCGSTAB, CGS, Barzilai-Borwein, etc) do specialized textbooks.

5.3.1 Steepest descend

This is a very simple method that is not used in practice, but useful for didactical purposes It requires the matrix A to be

- positive definite,
- symmetric.

The method requires iterating through these steps:

$$\begin{aligned}
&\text{repeat}\\
&\quad \boldsymbol{r}_{(k)} = \boldsymbol{b} - A\boldsymbol{x}_{(k)}\\
&\quad \text{if } ||\mathbf{r}_{(k)}|| < \epsilon \quad \text{exit loop}\\
&\quad \alpha_{(k)} = \frac{\boldsymbol{r}_{(k)}^T \boldsymbol{r}_{(k)}}{\boldsymbol{r}_{(k)}^T A \boldsymbol{r}_{(k)}}\\
&\quad \boldsymbol{x}_{(k+1)} = \boldsymbol{x}_{(k)} + \alpha_{(k)} \boldsymbol{r}_{(k)}
\end{aligned} \tag{128}$$

Note that the computational bottleneck is the computational primitive matrix x vector in $A\boldsymbol{r}^{(k)}$.

The matrix can be sparse and there is no need of fill-in elements as the sparsity pattern is not changed (this is a positive feature of all Krylov methods).

The issue of this method is that it converges slowly, as it can be intuitively interpreted by remembering that it corresponds to the gradient descend algorithm for finding the minimum of a function $f(\boldsymbol{x}) = 1/2\boldsymbol{x}^T A\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x} + c$, that can get stuck in a zigzag pattern if the "valley" is skewed and squeezed in a diagonal direction.

5.3.2 Conjugate gradient

This method greatly improves the performance of the steepest descend gradient method. It requires the matrix A to be

- positive definite,
- symmetric.

The pseudocode is:

$$\begin{aligned}
&\mathbf{r}_0 := \mathbf{b} - A\mathbf{x}_0\\
&\mathbf{p}_0 := \mathbf{r}_0\\
&\text{repeat}\\
&\quad \alpha_{(k)} := \frac{\mathbf{r}_{(k)}^{\mathsf{T}} \mathbf{r}_{(k)}}{\mathbf{p}_{(k)}^{\mathsf{T}} \mathbf{A} \mathbf{p}_{(k)}}\\
&\quad \mathbf{x}_{(k+1)} := \mathbf{x}_{(k)} + \alpha_{(k)} \mathbf{p}_{(k)}\\
&\quad \mathbf{r}_{(k+1)} := \mathbf{r}_{(k)} - \alpha_{(k)} A \mathbf{p}_{(k)}\\
&\quad \text{if } ||\mathbf{r}_{(k+1)}|| < \epsilon \quad \text{exit loop}\\
&\quad \beta_{(k)} := \frac{\mathbf{r}_{(k+1)}^{\mathsf{T}} \mathbf{r}_{(k+1)}}{\mathbf{r}_{(k)}^{\mathsf{T}} \mathbf{r}_{(k)}}\\
&\quad \mathbf{p}_{(k+1)} := \mathbf{r}_{(k+1)} + \beta_{(k)} \mathbf{p}_{(k)}
\end{aligned} \tag{129}$$

Also this method requires only one CPU-intensive operation per each iteration, namely the matrix x vector computational primitive.

Note that also the convergence of the conjugate gradient method can be negatively affected by ill-conditioned matrices. In such circumstances, one should revert to using some type of **preconditioner**, that is like an approximate solver of other specie that is interleaved to the Krylov iteration. There is no space here to discuss preconditioners more in detail and how to rewrite the CG iteration in order to accommodate a preconditioner, suffices to say that there are many options (none being optimal in all scenarios) and we name a few: diagonal scaling, ILU (Incomplete LU factorization), block ILU, multigrid, stationary iterations, spectral preconditioners, etc.

5.3.3 MINRES

The Minimum Residual (MINRES) method is similar to the Conjugate Gradient solver but it can work also with singular matrices, in fact it corresponds to finding a minimum norm residual solution to the system of linear equations. It requires the matrix A to be

- symmetric.

Just like all other iterative methods, MINRES is usually paired with a preconditioning method in order to improve convergence.

The MINRES method is very simple to implement, it does not require substantial additions to the CG method, and shows about the same convergence speed in many practical scenarios.

5.3.4 GMRES

The Generalized Minimum Residual (GMRES) method is a more complex Krylov solver that fixes the limitation of the Conjugate Gradient solver, in fact it can work also with non-symmetric matrices. This comes at a cost: the code is more complex (not shown here for conciseness) and it requires the storage of more auxiliary vectors. It requires the matrix A to be

- positive definite.

Just like all other iterative methods, GMRES is usually paired with a preconditioning method in order to improve convergence.

The GMRES method is sometimes restarted after a user-defined number of iterations. Also, anti-stagnation procedures must be put in place in this practical implementation.

5.3.5 BiCGSTAB

The Biconjugate gradients stabilized method (BiCGSTAB) is a refined version of the biconjugate gradient method (BiCG) that exhibits a better convergence. The implementation is simplier than GMRES, but it may exhibit easier stagnation and non monotone irregular convergence. It requires the matrix A to be

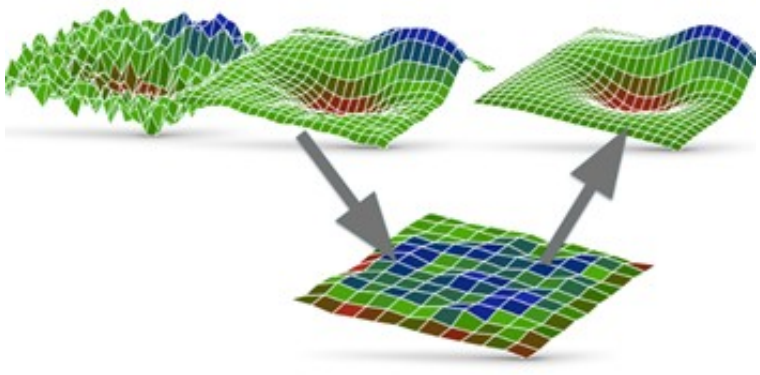

Figure 4: The V cycle in a multigrid method.

- positive definite.

Note that the BiCGSTAB method requires two matrix x vector operations per each iteration. The BiCG method requires one multiplication by an inverse of A, that makes it not practical especially when using sparse matrices.

Although the convergence is irregular and non monotone, it is one of the best among Krylov methods.

5.4 Multigrid methods

(To do...)

- iterate in 'V' or 'W' cycles until tolerance reached,
- solve the problem on different scales, from fine to coarse, then back,
- the solver for each level is a very simple stationary method, often one or two steps of Gauss-Seidell might suffice,
- complex implementation,
- very good performance also on badly conditioned matrices,
- can be used as stand-alone or can be used as preconditioners to other methods (ex Krylov methods).

We distinguish between *geometric multigrid* - requiring structured meshes (rectangular grids) and *algebraic multigrid* - requiring a complex implementation, especially of the smoothing operator.

6. Time integration

Among the PDEs, there are some problems that depend on time. Studying the time evolution of such systems, starting from given initial conditions, is a numerical problem that can be solved via time integration.

There are many different time integration methods (aka time steppers) which fit better some cases or others.

Here we cannot present all of them, but we briefly discuss relevant aspects: the interested reader can find additional details in literature.

6.1 Classes of time-dependent problems

In this section we introduce terminology and classifications for time-dependent problems. Among these, the ODEs and DAEs represent by far the most relevant, and we urge the reader to understand their meaning. Remaining classes presented here, namely DIs, DVIs, MDIs are used only in the more rare cases of problems subject to non-smooth set-valued laws, such as in case of multibody systems with hard frictional contacts or simulation of analog electronic circuits with diods etc.

6.1.1 ODEs

Given a generic state $\boldsymbol{x}$ and time t, an *Ordinary Differential Equation (ODE)* is a system of equations as:

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}\left(\boldsymbol{x}, t\right) \tag{130}$$

together with prescribed initial conditions $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$ [13].

An example: the transient heat equation (99) is an ODE with unknown x as the temperature T, in fact it can be written as

$$\frac{\partial T}{\partial t} = \frac{1}{\rho c_p}\left(k\nabla^2 T + q\right)$$

Solving such ODE means that one can obtain the time evolution of the temperature $T = T(\boldsymbol{x}, t)$ for each point $\boldsymbol{x} \in \Omega$. Initial temperature $\boldsymbol{T}(t_0) = \boldsymbol{T}_0$ must be prescribed as initial conditions.

The expression above is an ODE of *index one*. Generic *index-n* ODEs are in the form

$$\frac{d^n\boldsymbol{x}}{dt^n} = \boldsymbol{f}\left(\frac{d^{(n-1)}\boldsymbol{x}}{dt^{(n-1)}}, \dots \frac{d^2\boldsymbol{x}}{dt^2}, \frac{d\boldsymbol{x}}{dt}, \boldsymbol{x}, t\right) \tag{131}$$

[13] The expression above is the *explicit form* of the ODE. One could express ODEs also in *implicit form* as

$$\boldsymbol{F}\left(\frac{d\boldsymbol{x}}{dt}, \boldsymbol{x}, t\right) = \boldsymbol{0}$$

.

Its interesting to note that all index-n ODE can be converted into an index-one ODE by introducing a larger state with all the derivatives up to $n-1$ as in $\boldsymbol{s} = \left\{ \frac{d^{(n-1)}\boldsymbol{x}}{dt^{(n-1)}}, \ldots, \frac{d^2\boldsymbol{x}}{dt^2}, \frac{d\boldsymbol{x}}{dt}, \boldsymbol{x} \right\}^T$ and rewriting it as:

$$\frac{d\boldsymbol{s}}{dt} = \left\{ \begin{array}{c} \frac{d^n\boldsymbol{x}}{dt^n} \\ \cdots \\ \frac{d^2\boldsymbol{x}}{dt^2} \\ \frac{d\boldsymbol{x}}{dt} \end{array} \right\} = \left\{ \begin{array}{c} \boldsymbol{f}\left(\frac{d^{(n-1)}\boldsymbol{x}}{dt^{(n-1)}}, \ldots \frac{d^2\boldsymbol{x}}{dt^2}, \frac{d\boldsymbol{x}}{dt}, \boldsymbol{x}, t \right) \\ \cdots \\ \frac{d^2\boldsymbol{x}}{dt^2} \\ \frac{d\boldsymbol{x}}{dt} \end{array} \right\}; \quad \frac{d\boldsymbol{s}}{dt} = \boldsymbol{f}_s\left(\boldsymbol{s}, t\right) \quad (132)$$

An example is the index-2 ODE of unconstrained multibody dynamics:

$$M\frac{d\boldsymbol{v}}{dt} = \boldsymbol{f}\left(\boldsymbol{v}, \boldsymbol{x}, t\right) \quad (133)$$

that can be converted into a first-order ODE $\frac{d\boldsymbol{s}}{dt} = \boldsymbol{f}_s\left(\boldsymbol{s}, t\right)$ with $\boldsymbol{s} = \{\boldsymbol{v}, \boldsymbol{x}\}$ and $\frac{d\boldsymbol{s}}{dt} = \{\boldsymbol{a}, \boldsymbol{v}\} = \left\{M^{-1}\boldsymbol{f}(\boldsymbol{v}, \boldsymbol{x}, t), \boldsymbol{v}\right\} = \boldsymbol{f}_s\left(\boldsymbol{s}, t\right)$.

This means that integration methods designed for first-order ODEs can be applied also to second-order ODEs without much work. However, this said, it is better to develop custom integrators for second-order ODEs when possible.

The *Picard-Lindelhöf theorem* states that for $\boldsymbol{f}\left(\boldsymbol{x}, t\right)$ uniformly Lipschitz continuous in $\boldsymbol{x}$ and continuous in t, there is an unique solution $\boldsymbol{x}(t)$ to the ODE for a given initial value. See also Cauchy–Lipschitz theorem.

For infinite-dimensional problems like the transient heat equation and similars, in most cases the ODE is expressed at the discretized level (look the section on weak formulation and discretization) so that one has a finite dimensional problem with a countable set of components in the unknown vector.

6.1.2 DAEs

Given a generic state $\boldsymbol{x}$, and time t, a *Differential Algebraic Equation (DAE)* is a system of equations as:

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}\left(\boldsymbol{x}, \boldsymbol{y}, t\right) \quad (134)$$

$$\boldsymbol{g}\left(\boldsymbol{x}, \boldsymbol{y}, t\right) = \boldsymbol{0} \quad (135)$$

where $\boldsymbol{g}(.)$ is a set of constraint equations, also called *geometric* or *algebraic constraints*, and where $\boldsymbol{y}$ are the *algebraic variables*.

The expression above is the *semi-explicit form* of the DAE. One could express DAEs also in *implicit form* as

$$\boldsymbol{F}\left(\frac{d\boldsymbol{x}}{dt}, \boldsymbol{x}, \boldsymbol{y}, t\right) = \boldsymbol{0} \quad (136)$$

where the jacobian matrix $\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}}$ is singular; in fact if it was singular, one could cast it as a simple ODE $\frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}\left(\boldsymbol{x}, t\right)$.

The DAE *initial conditions* cannot be provided as arbitrarily as in ODEs, because they must be *consistent*, i.e. they must satisfy $\boldsymbol{F}\left(\frac{d\boldsymbol{x}(t_0)}{dt}, \boldsymbol{x}(t_0), \boldsymbol{y}(t_0), t_0\right) = \boldsymbol{0}$.

The *differentiation index* of a DAE, or simply the *index* of a DAE is the number of the differentiations that are required on the constraint equations of the semi-explicit form in order to obtain a pure ODE, after substitutions. An ODE is a DAE of index 0.

After index-reduction to an ODE, one might be tempted to perform the time integration of such ODE with classical integration schemes of ODEs. This might work for some steps but later some error tends to build up in the algebraic equations, giving a *drift-off* phenomena in constraints. This because constraints would be satisfied only at the n-th derivative level (ex. acceleration, and not position) in the reduced ODE. To solve such drifting problem there are different methods. A rough approach is to introduce Baumgarte stabilization terms, like spring dampers that attracts the solution back on the manifold of the original, non differentiated constraints. Another approach is to satisfy constraints in implicit integrations methods specially crafted for DAEs, using Newton-Raphson solvers, and this is the case of HHT, generalized-α and other DAE-oriented implicit solvers.

An example: the incompressible Navier-Stokes equations are a DAE[14] of index 2, where the constraints $\boldsymbol{g}(\boldsymbol{x}, t) = \boldsymbol{0}$ are given by the incompressibility constraint $\nabla \cdot \boldsymbol{u} = 0$.

An example: the index-2 DAE of constrained multibody dynamics

$$M\frac{d\boldsymbol{v}}{dt} - C_q^T\boldsymbol{\lambda} = \boldsymbol{f}\left(\boldsymbol{q}, \boldsymbol{v}, t\right) \tag{137}$$

$$\boldsymbol{C}(\boldsymbol{q}, \boldsymbol{v}, t) = \boldsymbol{0} \tag{138}$$

$$\dot{\boldsymbol{q}} = \Gamma(\boldsymbol{q})\boldsymbol{v} \tag{139}$$

with constraint reactions $\boldsymbol{\lambda}$, configuration $\boldsymbol{q}$, velocities $\boldsymbol{v}$, external and gyroscopic forces $\boldsymbol{f}$. Usually $\boldsymbol{v} = \dot{\boldsymbol{q}}$, but the complication of rotations in 3D might lead to the heterogeneous choice of using, say, quaternions in $\boldsymbol{q}$ and angular velocities in $\boldsymbol{v}$, and this would require the linear operator Γ for $\dot{\boldsymbol{q}} = \Gamma(\boldsymbol{q})\boldsymbol{v}$.

6.1.3 DIs

A Filippov **Differential Inclusion** (DI) can be interpreted as an ODE for problems with discontinuous $\boldsymbol{f}(\boldsymbol{x}, t)$:

$$\frac{d\boldsymbol{x}}{dt} \in \mathcal{F}\boldsymbol{f}(\boldsymbol{x}, t) \quad \mathcal{F}f(\boldsymbol{x}, t) = \bigcap_{\eta>0} \bigcap_{N:\lambda_0(N)=0} \bar{\text{co}}\boldsymbol{f}(\boldsymbol{x} + \eta\boldsymbol{B_1} \setminus N, t) \tag{140}$$

where $\lambda_0(E)$ is a Lebesgue measure on E, B_1 is a origin-centered unit ball, and $\boldsymbol{f}$ is discontinuous in $\boldsymbol{x}$.

[14] Some authors call it a PDAE to underline the (P)artial derivative differential algebraic equation nature, but it is only a matter of terminology.

More in general, a *Differential Inclusion* (DI) is a problem

$$\frac{d\boldsymbol{x}}{dt} \in \mathcal{F}(\boldsymbol{x}, t) \tag{141}$$

where the set-valued function $\mathcal{F}(\boldsymbol{x}, t)$ is closed, bounded and convex and is upper semi-continuous, or equivalently, $\mathcal{F}(\boldsymbol{x}, t)$ has closed graph.

6.1.4 DVIs

We introduce **Differential Variational Inequality** (DVI) problems as:

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}, t) \tag{142a}$$

$$\boldsymbol{u} \in \mathsf{SOL}(\boldsymbol{F}, \mathcal{K}) \tag{142b}$$

where $\mathsf{SOL}(\boldsymbol{F}, \mathcal{K})$ is the (set of) solution to the VI$(\boldsymbol{F}, \mathcal{K})$. This is a special type of DI, too. One can see that a DVI includes DAE as a special case: with n bilateral algebraic constraints one takes $\boldsymbol{F}$ as the vector of algebraic constraint residuals, and uses $\mathcal{K} = \mathbb{R}^n$ so that $\boldsymbol{F} = \boldsymbol{0}$ everywhere by definition VI.

6.1.5 MDIs

A **Measure Differential Inclusion** (MDI) is a generalization of DI (DVI) that also accomodates impulsive events. For second order problems as in mechanics, with $\boldsymbol{v}(t) = d\boldsymbol{q}/dt$ it reads:

$$\frac{d\boldsymbol{v}}{dt} \in \mathcal{K}(\boldsymbol{q}, t) \tag{143}$$

where $\boldsymbol{v}$ is a function of bounded variation and $\mathcal{K}(\boldsymbol{q}, t)$ is a set-valued function with closed graph and closed convex values. The *strong* definition of solution (Moreau) follows the singular measure decomposition of $d\boldsymbol{v} = \boldsymbol{\nu}$ into $\boldsymbol{\nu}_s + \boldsymbol{h}\lambda_0$, with respect to the singular part $\boldsymbol{\nu}_s$ and Lebesgue measure λ_0 for continuous $\boldsymbol{h}(t) \in L^1(a, b)$: then the strong definition of solution is: $\boldsymbol{h}(t) \in \mathcal{K}(t)$ almost all t, and $d\boldsymbol{\nu}_s/|\boldsymbol{\nu}_s|(t) \in \mathcal{K}(t)_\infty$, i.e. the Radon-Nikodym derivative fits the horizon cone. The *weak* definition of solution (Stewart) is:

$$\frac{\int \phi(t) d\boldsymbol{\nu}(dt)}{\int \phi(t) dt} \in \bar{\text{co}} \bigcup_{\tau : \phi(\tau) \neq 0} \mathcal{K}(\tau) \tag{144}$$

for any continuous $\phi(t) : \mathbb{R} \to \mathbb{R}^+$ with compact support.

6.2 Time integration numerical methods

The goal is to provide methods that, given a state $\boldsymbol{x}_n$ at time t_n, can proceed to the next state at time $\boldsymbol{x}_{n+1}$ at time t_{n+1} by advancing with a time step h, with $h = t_n - t_{n+1}$.

Focusing on ODEs only, for the moment, we can introduce some taxonomies of methods.

- We can distinguish between **multistep** and **Runge**[15]**-Kutta** methods. The former depends on a history of previous timesteps in order to extrapolate a new one at the end of the new step, whereas the latter starts from the last timestep information then computes temporary intermediate $\boldsymbol{f}_j(\boldsymbol{x}_j, t)$ values after it, in order to compute the value at the end of the step.

- We can distinguish between **explicit** and **implicit** methods, the former using formulas that do not require any information on the value of state at the end of the step, whereas the latter uses (nonlinear) formulas that depend on the value of state at the end of the step, thus leading to more intricate solution schemes that rely on fixed point iterations or Newton iterations or in general iterative schemes that reach the solution at the end of the time step by iterating multiple times over a nonlinear set of equations. A notable feature of implicit methods is that they can solve problems with large stiffness (steep integrals with sudden rapid oscillations, so to speak) without the need of short time steps, whereas explicit methods require much shorter time steps to avoid divergence - see L-stability etc.

- We can distinguish between **fixed** time step or **variable** time step methods depend on the fact that, in the former case the time step h is assumed constant, whereas in the second case the method can embed some smart heuristics to change the time step h depending on its needs, for example making it shorter when approaching steepy stiff integrals, and making it larger when the time integration can proceed faster with lower CPU overhead. The latter case often involve the ability of un-doing and *repeating the time step* with shorter h when convergence or precision is not satisfactory. Think at this property as a simple add-on feature: almost all fixed step methods can be made of variable time step type by adding the rewinding capability to the original algorithms and adding some heuristics for shrinking/enlarging the time step.

- We mention that someone distinguishes between **stiff** and **non-stiff** methods, meaning the capability of handling problems with high stiffness without incurring in divergence. As this property is specific of implicit methods, often one can consider the class of stiff methods to be synonym of implicit methods. In a broader interpretation, one can use the term stiff also for explicit methods that, once embedded with some variable time step scheme, can attack (moderately) stiff problems by reducing automatically the time step up to avoid divergence.

- We also mention the not so frequent definition of **predictor-corrector** schemes. Rather than being a pure mathematical quality, this is an elaborate trick to help the convergence of sub-iterations of some implicit methods by warmstarting them with the help of another (often cheaper) integrator. In fact, in many predictor-corrector schemes, an explicit method is

[15] It is remarkable how many bright scientist attended the University of Göttingen.

used to roughly estimate the new state working a bit like an extrapolator, and then an implicit integrator uses this estimate as a starting point for the refinement of the solution by running few Neton Raphson (or fixed point) iterations.

The qualities above are orthogonal, in the sense that a specific integration method could be, for example the backward Euler method is of Runge-Kutta type, implicit, usually with fixed time step, and stiff.

We present some relevant integration schemes in the following.

6.2.1 Explicit Runge-Kutta methods

These methods are based on multiple evaluation of the derivative at intermediate time steps, thus subdividing the h time step into invisible temporary sub-steps. The count of substeps is called *stages*. The general formula for an **explicit Runge Kutta** method of s stages is:

$$\begin{cases} \boldsymbol{x}_{n+1} = \boldsymbol{x}_n + h \sum_{i=1}^{s} b_i \boldsymbol{k}_i \\ t_{n+1} = t_n + h \end{cases} \tag{145}$$

with

$$\boldsymbol{k}_1 = \boldsymbol{f}(t_n, \boldsymbol{x}_n), \tag{146}$$
$$\boldsymbol{k}_2 = \boldsymbol{f}(t_n + c_2 h, \boldsymbol{x}_n + h(a_{21}\boldsymbol{k}_1)), \tag{147}$$
$$\boldsymbol{k}_3 = \boldsymbol{f}(t_n + c_3 h, \boldsymbol{x}_n + h(a_{31}\boldsymbol{k}_1 + a_{32}\boldsymbol{k}_2)), \tag{148}$$
$$\vdots \tag{149}$$
$$\boldsymbol{k}_s = \boldsymbol{f}(t_n + c_s h, \boldsymbol{x}_n + h(a_{s1}\boldsymbol{k}_1 + a_{s2}\boldsymbol{k}_2 + \cdots + a_{s,s-1}\boldsymbol{k}_{s-1})). \tag{150}$$

where coefficients are given by a *Butcher tableau*, satisfying $\sum_{j=1}^{i-1} a_{ij} = c_i$ for $i = 2, \ldots, s$.

Among the explicit Runge-kutta methods, the most popular is the 4th order one, with $c_1 = 0, c_2 = 0.5, c_3 = 0.5, c_4 = 1, b_1 = 1/6, b_2 = 1/3, b_3 = 1/3, b_4 = 1/6$. It has a local truncation error on the order of $O(h^5)$.

One can estimate the local error by performing a 5th order RK alongside the 4th order one, but to avoid computing 4+5=9 functions $f()$, special methods such as the Runge-Kutta Fehlberg 4/5th order have been developed, that for the 5th order reuse 4 evaluations of $f()$ of the 4th order plus just one additional $f()$, that is used only for estimating the local error. A similar idea is used for Runge-Kutta 2/3 methods. These are known as ODE45 and ODE23 in Matlab or other packages.

Knowing an estimate of the local truncation error is useful for two reasons: for diagnostic during postprocessing, and/or to tell the integrator that the h step is too large to give acceptable precision and therefore h must be decreased or even re-done, if the integrator is of *variable* time step type.

6.2.2 Explicit Euler

The **explicit Euler** method is a special type of explicit Runge-Kutta method, the simpliest one by the way. It boils down to:

$$\begin{cases} \boldsymbol{x}_{n+1} = \boldsymbol{x}_n + h\boldsymbol{f}(t_n, \boldsymbol{x}_n) \\ t_{n+1} = t_n + h \end{cases} \tag{151}$$

In terms of precision, it is a first-order method, thus it has a local truncation error on the order of $O(h^2)$ and its precision is a bit disappointing (even a simple parabolic ballistic trajectory will be approximated with growing error accumulation).

Moreover, large time steps in stiff problems will easily bring this method to divergence.

The positive note, however, is that the explicit Euler method is very cheap in the computational sense, because it just requires one evaluation of the $\boldsymbol{f}(t_n, \boldsymbol{x}_n)$ term at each time step, given the current state $\boldsymbol{x}_n$. Therefore, one can afford making very short time steps to circumvent its limitation on precision and to avoid the risk of divergence.

6.2.3 Implicit Runge-Kutta methods

The **implicit Runge-Kutta** methods are defined as:

$$\begin{cases} \boldsymbol{x}_{n+1} = \boldsymbol{x}_n + h\sum_{i=1}^{s} b_i \boldsymbol{k}_i \\ t_{n+1} = t_n + h \end{cases} \tag{152}$$

with terms that depend on $\boldsymbol{x}_{n+1}$ as in

$$\boldsymbol{k}_i = \boldsymbol{f}\left(t_n + c_i h,\ \boldsymbol{x}_n + h\sum_{j=1}^{s} a_{ij}\boldsymbol{k}_j\right), \quad i = 1, \ldots, s. \tag{153}$$

6.2.4 Implicit Euler

A very famous sub-case of implicit Runge-Kutta schemes is the **implicit Euler** (also known as backward Euler) method. It is expressed as:

$$\begin{cases} \boldsymbol{x}_{n+1} = \boldsymbol{x}_n + hf(t_n + h,\ \boldsymbol{x}_{n+1}) \\ t_{n+1} = t_n + h \end{cases} \tag{154}$$

Note here a typical complication of all implicit methods: a system of (possibly nonlinear) algebraic equations has to be solved to compute $\boldsymbol{x}_{n+1}$, and this must happen at each time step.

6.2.5 Adams-Bashforth methods

This is an example from the class of multistep methods. As for all multisteps methods, it requires an history of previous time steps.

Recycling the information of previous time steps will help the precision with low computational cost - consider that Runge-Kutta methods achieve accuracy at the cost of computing intermediate time steps that are discarded. However it also leads to some issues, one being the fact that they require some tricks in starting them from scratch, when previous time history is not given.

The Adams-Bashforth methods of order s are explicit linear multistep methods given by the general formula:

$$\begin{cases} \sum_{j=0}^{s} a_j \boldsymbol{x}_{n+j} = h \sum_{j=0}^{s} b_j \boldsymbol{f}(t_{n+j}, \boldsymbol{y}_{n+j}) \\ t_{n+1} = t_n + h \end{cases} \tag{155}$$

where one has $a_s = -1$, $a_j = 0$ for $j < s$, and b_j coefficients are given by a Lagrange polynomial interpolation hence $b_{s-j-1} = \frac{(-1)^j}{j!(s-j-1)!} \int_0^1 \prod_{\substack{i=0 \\ i \neq j}}^{s-1} (u + i)\, du, \qquad \text{for } j = 0, \ldots, s-1$; for example the 4-th order Adams-Bashforth boils down to

$$\begin{cases} \boldsymbol{x}_{n+4} = \boldsymbol{x}_{n+3} + h \left(\frac{55}{24} \boldsymbol{f}(t_{n+3}, y_{n+3}) - \frac{59}{24} \boldsymbol{f}(t_{n+2}, y_{n+2}) + \frac{37}{24} \boldsymbol{f}(t_{n+1}, y_{n+1}) - \frac{9}{24} \boldsymbol{f}(t_n, y_n) \right) \\ t_{n+1} = t_n + h \end{cases} \tag{156}$$

and one can see that it can reuse three previous $\boldsymbol{f}$ informations each time it advances one h step.

Adams-Bashforth methods with s steps have local truncation error on the order of $O(h^{(s+1)})$.

6.2.6 Adams-Moulton methods

This is another class of linear multistep methods, similar to Adams-Bashforth, but this time it is an implicit integration. For example, the 4-th order Adams-Moulton method boils down to:

$$\begin{cases} \boldsymbol{x}_{n+3} = \boldsymbol{x}_{n+2} + h \left(\frac{9}{24} \boldsymbol{f}(t_{n+3}, \boldsymbol{x}_{n+3}) + \frac{19}{24} \boldsymbol{f}(t_{n+2}, \boldsymbol{x}_{n+2}) - \frac{5}{24} \boldsymbol{f}(t_{n+1}, \boldsymbol{x}_{n+1}) + \frac{1}{24} \boldsymbol{f}(t_n, \boldsymbol{x}_n) \right) \\ t_{n+1} = t_n + h \end{cases} \tag{157}$$

and one can see that the $\boldsymbol{f}(t_{n+3}, \boldsymbol{x}_{n+3})$ terms depends on the state at the end of the time step $\boldsymbol{x}_{n+3}$, making it an implicit method (and consequently requiring some complicate machinery like Newton-Raphson iterations to solve for $\boldsymbol{x}_{n+3}$).

Adams-Moulton methods with s steps have order $s+1$ and local truncation error of order $O(h^{(s+2)})$.

6.2.7 Trapezoidal method

The trapezoidal integrator is in implicit integrator, and it can be seen both as a special case of Adams-Moulton integrator with one step, or an implicit Runge-Kutta of 2nd order. After the impicit Euler, it is perhaps the simplest implicit integration scheme. The scheme is:

$$\begin{cases} \boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \frac{1}{2}h\left(\boldsymbol{f}(t_n, \boldsymbol{x}_n) + \boldsymbol{f}(t_{n+1}, \boldsymbol{x}_{n+1}\right) \\ t_{n+1} = t_n + h \end{cases} \tag{158}$$

This method has order 2, local truncation error of order $O(h^{(s+2)})$. The method unstable even if it is unconditionally stable.

Among the implicit methods, it stands out because it does not introduce numerical damping, but this property is less usable than it seems in practical scenarios with strong nonlinearities that can create divergence (the unconditional stability holds in the theoretical ideal case of linear stiffness) so a bit of numerical damping in implicit integrators would be welcome anyway.

6.2.8 BDF methods

These are implicit linear multistep methods, but are a bit simplier and more efficient than the Adams-Moulton methods. The general formula for an order-s BDF scheme is:

$$\begin{cases} \sum_{k=0}^{s} a_k \boldsymbol{x}_{n+k} = h\beta \boldsymbol{f}(t_{n+s}, \boldsymbol{x}_{n+s}) \\ t_{n+1} = t_n + h \end{cases} \tag{159}$$

with properly chosen coefficients β and a_k.

The implicit Euler is also the special case of BDF with order one. We mention here the coefficients for other orders:

- second order BDF:

$$\boldsymbol{x}_{n+2} - \tfrac{4}{3}\boldsymbol{x}_{n+1} + \tfrac{1}{3}\boldsymbol{x}_n = \tfrac{2}{3}h\boldsymbol{f}(t_{n+2}, \boldsymbol{x}_{n+2})$$

- third order BDF:

$$\boldsymbol{x}_{n+3} - \tfrac{18}{11}\boldsymbol{x}_{n+2} + \tfrac{9}{11}\boldsymbol{x}_{n+1} - \tfrac{2}{11}\boldsymbol{x}_n = \tfrac{6}{11}h\boldsymbol{f}(t_{n+3}, \boldsymbol{x}_{n+3})$$

- fourth order BDF:

$$\boldsymbol{x}_{n+4} - \tfrac{48}{25}\boldsymbol{x}_{n+3} + \tfrac{36}{25}\boldsymbol{x}_{n+2} - \tfrac{16}{25}\boldsymbol{x}_{n+1} + \tfrac{3}{25}\boldsymbol{x}_n = \tfrac{12}{25}h\boldsymbol{f}(t_{n+4}, \boldsymbol{x}_{n+4})$$

Note that linear multistep methods with an order greater than 2 cannot be A-stable (see stability).

6.3 Numerical implementation of methods

In the previous section we presented some of the most relevant integration schemes, but in a general setting that assumes an index-1 ODE, without telling how to perform the Newton-Raphson substeps for the implicit methods, and telling nothing about DAEs.

In a practical setting, however, one should better exploit the structure of the problem at hand. For example, in mechanics, it is true that one can turn an order-2 ODE (with accelerations) into an index-1 ODE grouping positions and speeds in a single state $\boldsymbol{x} = (\boldsymbol{q}, \boldsymbol{v})$, however it is better to express the method directly at the level of index-2 ODE with separate equations for speeds and accelerations. This also makes easier to recover the jacobian matrices for the implicit methods that require Newton-Raphson iterations.

Moreover, what can be done in case of DAEs, that is when algebraic constraints are added to ODEs?

In this section we show, for a limited set of integrators, how to implement them at the algorithmical level, also showing some schemes for DAEs.

6.3.1 Trapezoidal, for index-2 ODE

For simplicity, let's start with the ODE case, i.e. also a DAE where no constraints are present.

For an index-2 ODE as used in dynamics in (btw the DAE case will be dealt later), we use state $\boldsymbol{x} = \{\boldsymbol{q}, \boldsymbol{v}\}$, $\dot{\boldsymbol{x}} = \{\boldsymbol{v}, \boldsymbol{a}\}$ so we have:

$$\begin{cases} \boldsymbol{q}^{l+1} - \boldsymbol{q}^l - \left(\dfrac{\boldsymbol{v}^{l+1} + \boldsymbol{v}^l}{2}\right) h = 0 & (160) \\ \boldsymbol{v}^{l+1} - \boldsymbol{v}^l - \left(\dfrac{\boldsymbol{a}^{l+1} + \boldsymbol{a}^l}{2}\right) h = 0 & (161) \\ t^{l+1} = t^l + h & (162) \end{cases}$$

In general one has computable $\boldsymbol{a}^l$ and $\boldsymbol{a}^{l+1}$, where in sake of compactness we write $\boldsymbol{a}^l$ for $\boldsymbol{a}(\boldsymbol{q}^l, \boldsymbol{v}^l, t^l)$, as well as $\boldsymbol{a}^{l+1}$ for $\boldsymbol{a}(\boldsymbol{q}^{l+1}, \boldsymbol{v}^{l+1}, t^{l+1})$. Given that in a mechanical context one splits $\boldsymbol{f} = M\boldsymbol{a}$, one can also write the following (that fits in ODE too) where for brevity we write $\boldsymbol{f}^l$ for $\boldsymbol{f}(\boldsymbol{q}^l, \boldsymbol{v}^l, t^l)$, as well as $\boldsymbol{f}^{l+1}$ for $\boldsymbol{f}(\boldsymbol{q}^{l+1}, \boldsymbol{v}^{l+1}, t^{l+1})$, :

$$\boldsymbol{q}^{l+1} - \boldsymbol{q}^l - \left(\frac{\boldsymbol{v}^{l+1} + \boldsymbol{v}^l}{2}\right) h = 0 \quad (163)$$

$$\left(\boldsymbol{v}^{l+1} - \boldsymbol{v}^l\right)\left(M^{l+1} + M^l\right) - h\boldsymbol{f}^{l+1} - h\boldsymbol{f}^l = 0 \quad (164)$$

Let's call $\boldsymbol{G} = \{\boldsymbol{G}_q, \boldsymbol{G}_v\}$ the residual of the system above. To satisfy $\boldsymbol{G} = 0$, one can use a Newton-Raphson iteration (the subscribt means the iteration number) and solve for the proper $\boldsymbol{q}^{l+1}, \boldsymbol{v}^{l+1}$:

$$\boldsymbol{G}(\boldsymbol{q}^{l+1}, \boldsymbol{v}^{l+1}) = 0 \tag{165}$$

$$\boldsymbol{G}(\boldsymbol{q}_n^{l+1}, \boldsymbol{v}_n^{l+1}) + \left[\begin{array}{cc} \frac{\partial \boldsymbol{G}_q}{\partial \boldsymbol{q}} & \frac{\partial \boldsymbol{G}_q}{\partial \boldsymbol{v}} \\ \frac{\partial \boldsymbol{G}_v}{\partial \boldsymbol{q}} & \frac{\partial \boldsymbol{G}_v}{\partial \boldsymbol{v}} \end{array} \right]_n^{l+1} \left\{ \begin{array}{c} \boldsymbol{q}_{n+1}^{l+1} - \boldsymbol{q}_n^{l+1} \\ \boldsymbol{v}_{n+1}^{l+1} - \boldsymbol{v}_n^{l+1} \end{array} \right\} = 0 \tag{166}$$

also using deltas as a more compact notation for corrections:

$$\boldsymbol{G}(\boldsymbol{q}_n^{l+1}, \boldsymbol{v}_n^{l+1}) + \left[\begin{array}{cc} \frac{\partial \boldsymbol{G}_q}{\partial \boldsymbol{q}} & \frac{\partial \boldsymbol{G}_q}{\partial \boldsymbol{v}} \\ \frac{\partial \boldsymbol{G}_v}{\partial \boldsymbol{q}} & \frac{\partial \boldsymbol{G}_v}{\partial \boldsymbol{v}} \end{array} \right]_n^{l+1} \left\{ \begin{array}{c} \Delta \boldsymbol{q}^{l+1} \\ \Delta \boldsymbol{v}^{l+1} \end{array} \right\} = 0 \tag{167}$$

or, in a even more compact notation for jacobian J, it boils down to a Newton-Raphson step of the type

$$J \left\{ \begin{array}{c} \Delta \boldsymbol{q}^{l+1} \\ \Delta \boldsymbol{v}^{l+1} \end{array} \right\} = -\boldsymbol{G}(\boldsymbol{q}_n^{l+1}, \boldsymbol{v}_n^{l+1}) \tag{168}$$

Now, note that a straight implementation of the above NR step is not efficient since J would be highly sparse and unsymmetric. One can unroll the equations, and from Eq.160 one gets $\Delta \boldsymbol{q}^{l+1} = \frac{h}{2} \Delta \boldsymbol{v}^{l+1}$, that allows [16] to rewrite as:

$$J = \left[\begin{array}{cc} \frac{\partial \boldsymbol{G}_q}{\partial \boldsymbol{q}} & \frac{\partial \boldsymbol{G}_q}{\partial \boldsymbol{v}} \\ \frac{\partial \boldsymbol{G}_v}{\partial \boldsymbol{q}} & \frac{\partial \boldsymbol{G}_v}{\partial \boldsymbol{v}} \end{array} \right]_n^{l+1} = \left[\begin{array}{cc} I & -\frac{h}{2} I \\ -h \nabla_q \boldsymbol{f}^{l+1} & (M^{l+1} + M^l) - h \nabla_v \boldsymbol{f}^{l+1} \end{array} \right] \tag{169}$$

Now one can put that jacobian in Eq.167, develop the equations and arrange them so that one can write the step Eq.167 as three substeps to be repeated in sequence for each Newton iteration, until $\Delta \boldsymbol{q}^{l+1}$ and $\Delta \boldsymbol{v}^{l+1}$ go to zero. In such steps we define $\left(\frac{M^{l+1} + M^l}{2} \right) = \hat{M}$, that is often the constant mass M in many cases. One gets the Newton step procedure:

$$\boxed{\begin{aligned} \left[\hat{M} - \frac{h^2}{4} \nabla_q \boldsymbol{f}^{l+1} - \frac{h}{2} \nabla_v \boldsymbol{f}^{l+1} \right] \Delta \boldsymbol{v}^{l+1} &= \left(\boldsymbol{v}^l - \boldsymbol{v}^{l+1} \right) \hat{M} + \frac{h}{2} \boldsymbol{f}^l + \frac{h}{2} \boldsymbol{f}^{l+1} \\ \boldsymbol{v}_{n+1}^{l+1} &= \boldsymbol{v}_n^{l+1} + \Delta \boldsymbol{v}^{l+1} \\ \boldsymbol{q}^{l+1} &= \boldsymbol{q}^l + \left(\frac{\boldsymbol{v}_{n+1}^{l+1} + \boldsymbol{v}^l}{2} \right) h \end{aligned}}$$

(170)

(171)

(172)

[16] The fact that $\Delta \boldsymbol{q}^{l+1} = \frac{h}{2} \Delta \boldsymbol{v}^{l+1}$ is used in various literature, but it is true only close to the solution, when the residual $\boldsymbol{G}_q$ is close to zero, otherwise it would be $\Delta \boldsymbol{q}^{l+1} = \frac{h}{2} \Delta \boldsymbol{v}^{l+1} - \boldsymbol{G}_q = \frac{h}{2} \Delta \boldsymbol{v}^{l+1} - \boldsymbol{q}^{l+1} + \boldsymbol{q}^l + \left(\frac{\boldsymbol{v}^{l+1} + \boldsymbol{v}^l}{2} \right) h$. However this might overly complicate the NR step that follows.

where only the 1st step Eq.170 is the computationally intensive step. But the matrix is hermitian, at least.

- Note that the residual in Eq.170 is the same residual of Eq.164, except everything is divided by a factor 2 for convenience.
- Note that if the exact expression $\Delta \boldsymbol{q}^{l+1} = \frac{h}{2}\Delta \boldsymbol{v}^{l+1} - \boldsymbol{G}_q$ is used, the term $-\boldsymbol{G}_q$ must modify the residual in Eq.170 by adding $... + \frac{h}{2}\nabla_q \boldsymbol{f}^{l+1}\boldsymbol{G}_q$ that is $... + \frac{h}{2}\nabla_q \boldsymbol{f}^{l+1}\left(\boldsymbol{q}^{l+1} - \boldsymbol{q}^l - \left(\frac{\boldsymbol{v}^{l+1}+\boldsymbol{v}^l}{2}\right)h\right)$.
- Note that $\nabla_q \boldsymbol{f}$ is $-K$, the stiffness matrix in structural elments, and $\nabla_v \boldsymbol{f}$ is $-R$, the damping matrix.

6.3.2 Euler implicit, for index-2 ODE

The Euler implicit method (backward Euler), for a 2nd order ODE, is:

$$\begin{cases} \boldsymbol{q}^{l+1} - \boldsymbol{q}^l - \boldsymbol{v}^{l+1}h = 0 & (173) \\ \boldsymbol{v}^{l+1} - \boldsymbol{v}^l - \boldsymbol{a}^{l+1}h = 0 & (174) \\ t^{l+1} = t^l + h & (175) \end{cases}$$

Again, as with the trapezoidal, this can be solved with a Newton-Raphson iteration. Let's call $\boldsymbol{G} = \{\boldsymbol{G}_q, \boldsymbol{G}_v\}$ the residual of the system above.

$$J \left\{ \begin{array}{c} \Delta \boldsymbol{q}^{l+1} \\ \Delta \boldsymbol{v}^{l+1} \end{array} \right\} = -\boldsymbol{G}(\boldsymbol{q}_n^{l+1}, \boldsymbol{v}_n^{l+1}) \tag{176}$$

where the jacobian, in this case, is:

$$J = \left[\begin{array}{cc} \frac{\partial \boldsymbol{G}_q}{\partial \boldsymbol{q}} & \frac{\partial \boldsymbol{G}_q}{\partial \boldsymbol{v}} \\ \frac{\partial \boldsymbol{G}_v}{\partial \boldsymbol{q}} & \frac{\partial \boldsymbol{G}_v}{\partial \boldsymbol{v}} \end{array} \right]_n^{l+1} = \left[\begin{array}{cc} I & -hI \\ -h\nabla_q \boldsymbol{f}^{l+1} & \hat{M} - h\nabla_v \boldsymbol{f}^{l+1} \end{array} \right] \tag{177}$$

Also in this case we defined $\left(\frac{M^{l+1}+M^l}{2}\right) = \hat{M}$, that is often the constant mass M in many cases. By developing the expression above and by setting $\Delta \boldsymbol{q}^{l+1} = h\Delta \boldsymbol{v}^{l+1}$ one gets the Newton step procedure:

$$\boxed{\begin{array}{ll} \left[\hat{M} - h^2\nabla_q \boldsymbol{f}^{l+1} - h\nabla_v \boldsymbol{f}^{l+1}\right]\Delta \boldsymbol{v}^{l+1} = \left(\boldsymbol{v}^l - \boldsymbol{v}^{l+1}\right)\hat{M} + h\boldsymbol{f}^{l+1} & (178) \\ \boldsymbol{v}_{n+1}^{l+1} = \boldsymbol{v}_n^{l+1} + \Delta \boldsymbol{v}^{l+1} & (179) \\ \boldsymbol{q}^{l+1} = \boldsymbol{q}^l + h\boldsymbol{v}_{n+1}^{l+1} & (180) \end{array}}$$

Some remarks here.

- In Eq.180 one could use $\Delta \boldsymbol{q}^{l+1} = h\Delta \boldsymbol{v}^{l+1}$ and rather write $\boldsymbol{q}_{n+1}^{l+1} = \boldsymbol{q}_n^{l+1} + h\Delta \boldsymbol{v}_{n+1}^{l+1}$, but this is questionable.

- The single first step of this NR process is exactly the Euler semi-implicit DVI in Chrono::Engine (not considering the constraints and contacts, here) because one can express it with unknowns $\boldsymbol{v}_{n+1}^{l+1}$ rather than deltas, so it can be also:

$$\left[\hat{M} - h^2 \nabla_q \boldsymbol{f}^{l+1} - h \nabla_v \boldsymbol{f}^{l+1}\right] \boldsymbol{v}_{n+1}^{l+1} = \left[\hat{M} - h^2 \nabla_q \boldsymbol{f}^{l+1} - h \nabla_v \boldsymbol{f}^{l+1}\right] \boldsymbol{v}_{n}^{l+1} + \\ + \left(\boldsymbol{v}^{l} - \boldsymbol{v}^{l+1}\right) \hat{M} + h \boldsymbol{f}^{l+1} \tag{181}$$

$$\boldsymbol{q}^{l+1} = \boldsymbol{q}^{l} + \boldsymbol{v}_{n+1}^{l+1} h \tag{182}$$

  and then, if at the first step we set $\boldsymbol{v}_{0}^{l+1} = \boldsymbol{v}^{l}$, and assuming $\boldsymbol{f}^{l+1} \approx \boldsymbol{f}^{l}$, we simply get:

$$\left[\hat{M} - h^2 \nabla_q \boldsymbol{f}^{l+1} - h \nabla_v \boldsymbol{f}^{l+1}\right] \boldsymbol{v}^{l+1} = \left[\hat{M} - h^2 \nabla_q \boldsymbol{f}^{l+1} - h \nabla_v \boldsymbol{f}^{l+1}\right] \boldsymbol{v}^{l} + h \boldsymbol{f}^{l} \tag{183}$$

$$\boldsymbol{q}^{l+1} = \boldsymbol{q}^{l} + \boldsymbol{v}_{n+1}^{l+1} h \tag{184}$$

  (just by removing the stiffness and damping matrices $\nabla_q \boldsymbol{f}^{l+1}$, $\nabla_v \boldsymbol{f}^{l+1}$ if there are no finite elements, and by adding jacobians to satisfy bilateral constraints, one turns Eq.183 into the same problem solved by the DVI timestepper of Chrono::Engine if without unilateral constraints)

6.3.3 Euler implicit, for DAE

Let's consider an index-2 DAE as in constrainted multibody mechanics, see (139).

There are different options for the DAE solution, for instance use constraints in DAE implicit form $F()$ to solve with Newton-Raphson, or do not solve them monolithically with DAE but just do projections on the manifold at each Newton iteration, etc. Here we enforce constraints (at the end of the time step) using a Newton-Raphson iteration, together with the Euler first order approximation used in the constraint-less case.

Add constraints $\boldsymbol{C}(\boldsymbol{q}, t) = \boldsymbol{0}$. For brevity, denote

$$C_q = \nabla_q \boldsymbol{C}^T = \frac{\partial \boldsymbol{C}}{\partial \boldsymbol{q}} \tag{185}$$

$$C_t = \nabla_t \boldsymbol{C}^T = \frac{\partial \boldsymbol{C}}{\partial t} \tag{186}$$

Also, in sake of compactness we write $\boldsymbol{C}^l$ for $\boldsymbol{C}(\boldsymbol{q}^l, t^l)$, as well as $\boldsymbol{C}^{l+1}$ for $\boldsymbol{C}(\boldsymbol{q}^{l+1}, t^{l+1})$.

The Euler implicit method (backward Euler), for DAE with constraints satisfied at the end of timestep, is:

$$\begin{cases} \boldsymbol{q}^{l+1} - \boldsymbol{q}^{l} - \boldsymbol{v}^{l+1}h = 0 & (187) \\ \hat{M}(\boldsymbol{v}^{l+1} - \boldsymbol{v}^{l}) - h\boldsymbol{f}^{l+1} - hC_q^T\boldsymbol{\lambda}^{l+1} = 0 & (188) \\ \boldsymbol{C}(\boldsymbol{q}^{l+1}, t^{l+1}) = \boldsymbol{0} & (189) \\ t^{l+1} = t^{l} + h & (190) \end{cases}$$

Let's call $\boldsymbol{G} = \{\boldsymbol{G}_q, \boldsymbol{G}_v, \boldsymbol{G}_c\}$ the residual of the system above. Solving with a Newton-Raphson iteration:

$$\begin{bmatrix} I & -hI & 0 \\ -h\nabla_q \boldsymbol{f}^{l+1} & \hat{M} - h\nabla_v \boldsymbol{f}^{l+1} & -hC_q^T \\ C_q & 0 & 0 \end{bmatrix} \begin{Bmatrix} \Delta\boldsymbol{q}^{l+1} \\ \Delta\boldsymbol{v}^{l+1} \\ \Delta\boldsymbol{\lambda}^{l+1} \end{Bmatrix} = -\boldsymbol{G} \tag{191}$$

By developing the expression above and by setting $\Delta\boldsymbol{q}^{l+1} = \frac{h}{2}\Delta\boldsymbol{v}^{l+1}$ (see footnote in previous page) one gets the 'unrolled' Newton step procedure, where the hard part is the 1st step, the classical saddle-point problem:

$$\begin{bmatrix} \left[\hat{M} - h^2\nabla_q \boldsymbol{f}^{l+1} - h\nabla_v \boldsymbol{f}^{l+1}\right] & C_q^T \\ C_q & 0 \end{bmatrix} \begin{Bmatrix} \Delta\boldsymbol{v}^{l+1} \\ -h\Delta\boldsymbol{\lambda}^{l+1} \end{Bmatrix} = \begin{Bmatrix} \left(\boldsymbol{v}^{l} - \boldsymbol{v}^{l+1}\right)\hat{M} + h\boldsymbol{f}^{l+1} + hC_q^T\boldsymbol{\lambda}^{l+1} \\ \frac{\boldsymbol{C}^{l+1}}{h} \end{Bmatrix} \tag{192}$$

$$\boldsymbol{v}_{n+1}^{l+1} = \boldsymbol{v}_{n}^{l+1} + \Delta\boldsymbol{v}^{l+1} \tag{193}$$

$$\boldsymbol{\lambda}_{n+1}^{l+1} = \boldsymbol{\lambda}_{n}^{l+1} + \Delta\boldsymbol{\lambda}^{l+1} \tag{194}$$

$$\boldsymbol{q}^{l+1} = \boldsymbol{q}^{l} + h\boldsymbol{v}_{n+1}^{l+1} \tag{195}$$

6.3.4 Trapezoidal, for DAE

A possibility is to add to Eqs. 163 and 164 the following constraint that requires positions to be satisfied at the end of the step:

$$\boldsymbol{C}(\boldsymbol{q}^{l+1}, t^{l+1}) = \boldsymbol{0}$$

and add reaction forces $\boldsymbol{\lambda}$ too, as $\boldsymbol{f} + \nabla_v \boldsymbol{C}\boldsymbol{\lambda} = M\boldsymbol{a}$. This leads to the following trapezoidal rule for DAE:

$$\left[\begin{array}{r} \boldsymbol{q}^{l+1} - \boldsymbol{q}^{l} - \left(\dfrac{\boldsymbol{v}^{l+1} + \boldsymbol{v}^{l}}{2}\right) h = 0 \\ \left(\boldsymbol{v}^{l+1} - \boldsymbol{v}^{l}\right)\left(M^{l+1} + M^{l}\right) - h\boldsymbol{f}^{l+1} - h\boldsymbol{f}^{l} - h\nabla_v \boldsymbol{C}^{l}\boldsymbol{\lambda}^{l} - h\nabla_v \boldsymbol{C}^{l+1}\boldsymbol{\lambda}^{l+1} = 0 \\ \boldsymbol{C}(\boldsymbol{q}^{l+1}, t^{l+1}) = \boldsymbol{0} \\ t^{l+1} = t^{l} + h \end{array}\right. \tag{196}$$

(197)

(198)

(199)

This done, one can express a Newton-Raphson process to compute the unknowns $\boldsymbol{v}^{l+1}$, $\boldsymbol{v}^{l+1}$, $\boldsymbol{\lambda}^{l+1}$ that give a zero residual $\boldsymbol{G}$ for the three equations above:

$$\left[\begin{array}{ccc} I & -\frac{h}{2}I & 0 \\ -\frac{h}{2}\nabla_q \boldsymbol{f}^{l+1} & \hat{M} - \frac{h}{2}\nabla_v \boldsymbol{f}^{l+1} & -\frac{h}{2}C_q^{l+1,T} \\ C_q^{l+1} & 0 & 0 \end{array}\right] \left\{\begin{array}{c} \Delta\boldsymbol{q}^{l+1} \\ \Delta\boldsymbol{v}^{l+1} \\ \Delta\boldsymbol{\lambda}^{l+1} \end{array}\right\} = -\boldsymbol{G} \tag{200}$$

This can be unrolled to work with a more friendly linear system with hermitian matrix and two following uncoupled steps:

$$\left[\begin{array}{cc} \left[\hat{M} - \frac{h^2}{4}\nabla_q \boldsymbol{f}^{l+1} - \frac{h}{2}\nabla_v \boldsymbol{f}^{l+1}\right] & C_q^{l+1,T} \\ C_q^{l+1} & 0 \end{array}\right] \left\{\begin{array}{c} \Delta\boldsymbol{v}^{l+1} \\ -\frac{h}{2}\Delta\boldsymbol{\lambda}^{l+1} \end{array}\right\} = \left\{\begin{array}{c} \left(\boldsymbol{v}^{l} - \boldsymbol{v}^{l+1}\right)\hat{M} + \frac{h}{2}\left(\boldsymbol{f}^{l} + \boldsymbol{f}^{l+1} + C_q^{l,T}\boldsymbol{\lambda}^{l} + C_q^{l+1,T}\boldsymbol{\lambda}^{l+1}\right) \\ \frac{1}{h}\boldsymbol{C}^{l+1} \end{array}\right\} \tag{201}$$

$$\boldsymbol{v}_{n+1}^{l+1} = \boldsymbol{v}_{n}^{l+1} + \Delta\boldsymbol{v}^{l+1} \tag{202}$$

$$\boldsymbol{\lambda}_{n+1}^{l+1} = \boldsymbol{\lambda}_{n}^{l+1} + \Delta\boldsymbol{\lambda}^{l+1} \tag{203}$$

$$\boldsymbol{q}^{l+1} = \boldsymbol{q}^{l} + \frac{h}{2}\left(\boldsymbol{v}^{l} + \boldsymbol{v}_{n+1}^{l+1}\right) \tag{204}$$

6.3.5 Newmark, for ODE and DAE

The Newmark family of second-order integrators is very popular in the FEM community.

One has

$$\left[\begin{array}{r} \boldsymbol{q}^{l+1} - \boldsymbol{q}^{l} - h\boldsymbol{v}^{l} - \dfrac{h^2}{2}\left[(1-2\beta)\boldsymbol{a}^{l} + 2\beta\boldsymbol{a}^{l+1}\right] = 0 \\ \boldsymbol{v}^{l+1} - \boldsymbol{v}^{l} - h\left[(1-\gamma)\boldsymbol{a}^{l} + \gamma\boldsymbol{a}^{l+1}\right] = 0 \\ M\boldsymbol{a}^{l+1} + (C_q^{T}\boldsymbol{\lambda} - \boldsymbol{f})^{l+1} = 0 \\ t^{l+1} = t^{l} + h \end{array}\right. \tag{205}$$

(206)

(207)

(208)

Different values of the γ and *beta* parameters provide different behaviors.

- The γ parameter usually ranges in the $[1/2, 1]$ interval.
- For $\gamma = 1/2$, no numerical damping.
- For $\gamma > 1/2$, numerical damping increases.
- The β parameter usually ranges in the $[0, 1]$ interval.
- For $\beta = 1/4, \gamma = 1/2$ one gets the constant acceleration method.
- For $\beta = 1/6, \gamma = 1/2$ one gets the linear acceleration method.
- The method is second order accurate only for $\gamma = 1/2$.

For this type of method, considering constraints being enforced in the DAE, the Newton iteration is represented again by a linear problem, this time in accelerations:

$$\boxed{\begin{aligned}
&\begin{bmatrix} H & \overline{C}_q^T \\ \overline{C}_q & 0 \end{bmatrix} \begin{Bmatrix} \Delta \boldsymbol{a}^{l+1} \\ \Delta \boldsymbol{\lambda}^{l+1} \end{Bmatrix} = \\
&\begin{Bmatrix} M\boldsymbol{a}^{l+1} + (C_q^T \boldsymbol{\lambda} - \boldsymbol{f})^{l+1} \\ \frac{1}{\beta h^2} \boldsymbol{C}^{l+1} \end{Bmatrix} && (209)\\
&\boldsymbol{a}_{n+1}^{l+1} = \boldsymbol{a}_n^{l+1} + \Delta \boldsymbol{a}^{l+1} && (210)\\
&\boldsymbol{\lambda}_{n+1}^{l+1} = \boldsymbol{\lambda}_n^{l+1} + \Delta \boldsymbol{\lambda}^{l+1} && (211)\\
&\boldsymbol{v}^{l+1} = \boldsymbol{v}^l + h\left[(1-\gamma)\boldsymbol{a}^l + \gamma \boldsymbol{a}^{l+1}\right] && (212)\\
&\boldsymbol{q}^{l+1} = \boldsymbol{q}^l + h\boldsymbol{v}^l + \frac{h^2}{2}\left[(1-2\beta)\boldsymbol{a}^l + 2\beta \boldsymbol{a}^{l+1}\right] && (213)
\end{aligned}}$$

where

$$H = \left[M - \gamma h \nabla_v \boldsymbol{f}^{l+1} - \beta h^2 \nabla_q \boldsymbol{f}^{l+1} + \beta h^2 \left[(M\boldsymbol{a})_q + ({C_q}^T \boldsymbol{\lambda})_q\right]\right]$$

6.3.6 HHT, for DAE

The HHT integrator is a generalization of the Newmark family of second-order integrators, and provides a control on the numerical dissipation yet retaining a second order accuracy as the trapezoidal method.

One has

$$\left[\begin{aligned}
\boldsymbol{q}^{l+1} - \boldsymbol{q}^l - h\boldsymbol{v}^l - \frac{h^2}{2}\left[(1-2\beta)\boldsymbol{a}^l + 2\beta \boldsymbol{a}^{l+1}\right] &= 0 && (214)\\
\boldsymbol{v}^{l+1} - \boldsymbol{v}^l - h\left[(1-\gamma)\boldsymbol{a}^l + \gamma \boldsymbol{a}^{l+1}\right] &= 0 && (215)\\
M\boldsymbol{a}^{l+1} + (1+\alpha)(C_q^T \boldsymbol{\lambda} - \boldsymbol{f})^{l+1} - \alpha(C_q^T \boldsymbol{\lambda} - \boldsymbol{f})^l &= 0 && (216)\\
t^{l+1} = t^l + h && (217)
\end{aligned}\right.$$

The HHT method provides the A-stability and order provided that $\alpha \in [-\frac{1}{3}, 0]$ and

$$\gamma = \frac{1-2\alpha}{2} \quad \beta = \frac{(1-\alpha)^2}{4} \tag{218}$$

The closer to 0 is α, the less damping has the method, where for $\alpha = 0$ one has precisely the trapezoidal method.

Looking at the work in [**?**], one sees that the Newton iteration is represented again by a linear problem with unknown accelerations and constraint forces:

$$\boxed{\begin{aligned}
&\begin{bmatrix} H & \overline{C}_q^T \\ \overline{C}_q & 0 \end{bmatrix} \begin{Bmatrix} \Delta \boldsymbol{a}^{l+1} \\ \Delta \boldsymbol{\lambda}^{l+1} \end{Bmatrix} = \\
&\begin{Bmatrix} \frac{1}{1+\alpha}(M\boldsymbol{a}^{l+1}) + (C_q^T \boldsymbol{\lambda} - \boldsymbol{f})^{l+1} - \frac{\alpha}{1+\alpha}(C_q^T \boldsymbol{\lambda} - \boldsymbol{f})^l \\ \frac{1}{\beta h^2} \boldsymbol{C}^{l+1} \end{Bmatrix} && (219) \\
&\boldsymbol{a}_{n+1}^{l+1} = \boldsymbol{a}_n^{l+1} + \Delta \boldsymbol{a}^{l+1} && (220) \\
&\boldsymbol{\lambda}_{n+1}^{l+1} = \boldsymbol{\lambda}_n^{l+1} + \Delta \boldsymbol{\lambda}^{l+1} && (221) \\
&\boldsymbol{v}^{l+1} = \boldsymbol{v}^l + h\left[(1-\gamma)\boldsymbol{a}^l + \gamma \boldsymbol{a}^{l+1}\right] && (222) \\
&\boldsymbol{q}^{l+1} = \boldsymbol{q}^l + h\boldsymbol{v}^l + \frac{h^2}{2}\left[(1-2\beta)\boldsymbol{a}^l + 2\beta \boldsymbol{a}^{l+1}\right] && (223)
\end{aligned}}$$

where

$$H = \left[M - \gamma h \nabla_v \boldsymbol{f}^{l+1} - \beta h^2 \nabla_q \boldsymbol{f}^{l+1} + \beta h^2 \left[(M\boldsymbol{a})_q + ({C_q}^T \boldsymbol{\lambda})_q\right]\right]$$

Note the term $\beta h^2 \left[(M\boldsymbol{a})_q + ({C_q}^T \boldsymbol{\lambda})_q\right]$, could be omitted for faster performance, at the risk of lower Newton convergence.

6.3.7 Generalized-α, for DAE

The generalized-α integrator is an evolution of the Newmark and HHT methods, where a single parameter ρ_∞ can be used to define the numerical damping. Just like the HHT integrator, it is a second-order implicit integrator that provides a control on the numerical dissipation yet retaining a second order accuracy as the trapezoidal method.

We recall the second barrier of the Dahlquist theorem: there are no explicit A-stable and linear multistep methods, and the implicit ones have order of convergence at most 2. The trapezoidal rule has the smallest error constant amongst the A-stable linear multistep methods of order 2. The HHT and the generalized-α methods are among the few timesteppers of practical interest that feature the second order of convergence, and that can solve constrained DAEs. In fact the trapezoidal method is confined mostly to ODEs since it gives oscillatory unstable reactions in constraints if used in DAEs.

In generalized-α one has

$$\begin{cases} \boldsymbol{q}^{l+1} - \boldsymbol{q}^l - h\boldsymbol{v}^l - \frac{h^2}{2}\left[(1-2\beta)\boldsymbol{a}^l + 2\beta\boldsymbol{a}^{l+1}\right] = 0 & (224) \\ \boldsymbol{v}^{l+1} - \boldsymbol{v}^l - h\left[(1-\gamma)\boldsymbol{a}^l + \gamma\boldsymbol{a}^{l+1}\right] = 0 & (225) \\ (1-\alpha_m)\boldsymbol{a}^{l+1} + \alpha_m\boldsymbol{a}^l = (1-\alpha_f)\boldsymbol{a}*^{l+1} + \alpha_f\boldsymbol{a}*^l & (226) \\ M\boldsymbol{a}*^{l+1} + (C_q^T\boldsymbol{\lambda} - \boldsymbol{f})^{l+1} = 0 & (227) \\ t^{l+1} = t^l + h & (228) \end{cases}$$

The coefficients in the generalized-α method are automatically set as following, once the spectral value ρ_∞ is set in the $[0, 1]$ range:

$$\alpha_m = \frac{2\rho_\infty - 1}{1 + \rho_\infty} \quad (229)$$

$$\alpha_f = \frac{\rho_\infty}{1 + \rho_\infty} \quad (230)$$

$$\beta = \frac{1}{4}\left(\gamma + \frac{1}{2}\right)^2 \quad (231)$$

$$\gamma = \frac{1}{2} + \alpha_f - \alpha_m \quad (232)$$

Note that for $\rho_\infty = 0$ one has the maximum asymptotic dissipation, and the method introduce the maximum numerical damping (i.e. whatever frequency is damped in a single step h).

One can see that the HHT method is like the generalized-α with $\alpha_m = 0$, $\alpha_f = \alpha$. Indeed they behave in a similar way, and they also are the only implicit methods of practical interest that feature second-order accuracy and that can solve constrained DAEs.

On the other side, the limit of $\rho_\infty = 1$ has no numerical dissipation, just like in the trapezoidal method. Differently from the trapezoidal method, though, this integrator behaves well with constrains and does not lead to oscillatory reactions in constraints.

Of course it is always convenient to introduce some artificial numerical dissipation even if the system, at the physical level, is not dissipative at all. This because, even with simplectic integrators, numerical issues might lead to increasing hamiltonian after many oscillations - something that in the long run is more likely to cause divergence and bad artifacts.

Usually, an intermediate $\rho_\infty \in [0, 1]$ value is used. This helps the user to discard unnecessary high frequency oscillations that are of little interest, ant that would just hamper the performance of the solver. Setting a proper value of ρ_∞ in general, non-linear cases, is often an heuristic process. One can start with a near zero value and raise it by repeating the same test simulation, until unnecessary high frequency oscillations start to appear.

The Newton iteration is represented again by a linear problem. As with the HHT case, the iteration can be expressed at the acceleration level, at the velocity level, at the configuration (i.e. position) level. Here we report the iteration with position (increments) as unknowns:

$$\boxed{\begin{array}{l}\left[\begin{array}{cc} H & \overline{C}_q^T \\ \overline{C}_q & 0 \end{array}\right]\left\{\begin{array}{c} \Delta \boldsymbol{q}^{l+1} \\ \Delta \boldsymbol{\lambda}^{l+1} \end{array}\right\} = \\ \left\{\begin{array}{c} \boldsymbol{r}^q \\ \boldsymbol{r}^\lambda \end{array}\right\} \end{array}} \tag{233}$$

$$\boldsymbol{q}_{n+1}^{l+1} = \boldsymbol{q}_n^{l+1} + \Delta \boldsymbol{q}^{l+1} \tag{234}$$

$$\boldsymbol{v}_{n+1}^{l+1} = \boldsymbol{v}_n^{l+1} + \gamma' \Delta \boldsymbol{q}^{l+1} \tag{235}$$

$$\boldsymbol{a}*_{n+1}^{l+1} = \boldsymbol{a}*_n^{l+1} + \beta' \Delta \boldsymbol{q}^{l+1} \tag{236}$$

$$\boldsymbol{\lambda}_{n+1}^{l+1} = \boldsymbol{\lambda}_n^{l+1} + \Delta \boldsymbol{\lambda}^{l+1} \tag{237}$$

where

$$H = \left[\nabla_q \left[M\boldsymbol{a}^{l+1} - \boldsymbol{f}^{l+1} + C_q^T \boldsymbol{\lambda}^{l+1}\right] - \gamma' \nabla_v \boldsymbol{f}^{l+1} + \beta' M \boldsymbol{a} -\right]$$

and

$$\beta' = \frac{1-\alpha_m}{h^2 \beta (1-\alpha_f)}$$

$$\gamma' = \frac{\gamma}{h\beta}$$

.

After the iteration has converged, one updates

$$\boldsymbol{a}^{l+1} = \boldsymbol{a}^l + \frac{1-\alpha_f}{1-\alpha_m} \boldsymbol{a}*^{l+1}$$

and similarly, before starting the iteration, $\boldsymbol{q}_n^{l+1}$, $\boldsymbol{v}_n^{l+1}$, $\boldsymbol{a}_n^{l+1}$ are initialized with proper formulas.

Note that the term $\nabla_q \left[M\boldsymbol{a}^{l+1} - \boldsymbol{f}^{l+1} + C_q^T \boldsymbol{\lambda}^{l+1}\right]$ could be simplified, approximating to $-\nabla_q \boldsymbol{f}^{l+1}$ for higher performance, at the risk of a worse convergence in the Newton loop.

Note that, although the method is unconditionally stable, this theoretical result holds for linearly stiff problems. This means that the method might still diverge if too large h is used in real cases that feature marked nonlinear geometric behaviour or highly nonlinear material properties.

Also, the Newton iteration is not guaranteed to converge in highly nonlinear cases. For instance, when dealing with buckling and near-bifurcation situations, some continuation or globalization strategies in the Newton computation are required, otherwise smaller timesteps must be used anyway.

6.4 Abstracting a common solver architecture

Looking at equations in boxed frames for all the DAE time-steppers, one can see that all methods share a common feature: at each time step there is one (or more) linear system to solve that has a saddle-point structure, usually (but

not always) with unknowns about velocities and reactions, always with such structure:

$$\begin{bmatrix} H & D^t \\ D & 0 \end{bmatrix} \begin{Bmatrix} \boldsymbol{x} \\ \boldsymbol{\lambda} \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{a} \\ \boldsymbol{b} \end{Bmatrix} \tag{238}$$

All implicit integrators therefore will interface to the physical system via two procedures: one that can provide the H jacobians (usually from mass, stiffness and damping matrices), and one that can provide the $\boldsymbol{a}$ terms (usually from internal+external forces). If constraints are used too, also D constraint jacobians and $\boldsymbol{b}$ terms should be fetched.

6.5 Accuracy, convergence, stability

In general, shorter time steps allow integration schemes to achieve a better *accuracy*. Also, short time steps are used to avoid *divergence*, that is the behavior that amplifies errors leading to solutions that explode to infinite values after one or few diverging oscillations, where divergence is more likely to happen if dealing with stiff equations. Some methods are less prone to divergence even if dealing with stiff equations, that is, they exploit better qualities of *stability*, and there are methods that are *unconditionally stable*, that is, their stability is guaranteed regardless of the integration time step. How these concepts expressed in a formal way?

We refer the reader for more details on this subject, here we just recall basic terminology and concepts.

- The **global error** is the error accumulated after n integration steps, that is, the error at the end of a sequence of time steps as the difference between the exact value $x(t_n)$ and the computed value x_n.

  $$\epsilon_n = x(t_n) - x_n$$

  Note that in most cases there is no analytical solution $x(t)$ even for moderately complex ODEs and DAEs, hence the exact error value cannot be computed. In lucky scenarios, maybe one can just estimate upper bounds.

- The **local truncation error** of a method is the error committed over a single step when moving to the approximated x_{n+1}, respect to the exact $x(t_{n+1})$

  $$\tau_{n+1} = (x(t_{n+1}) - x(t_n)) - (x_{n+1} - x_n)$$

  Note that for all methods presented so far, one can express x_{n+1} as a function $x_{n+1} = L(x_n, x_{n-1}, x_{n-2}, \ldots, h)$ and therefore

  $$\tau_{n+1} = L(x_n, x_{n-1}, x_{n-2}, \ldots, h) - x(t_{n+1})$$

- A method is said **consistent** if

  $$\lim_{h \to 0} \frac{L(x_n, x_{n-1}, x_{n-2}, \ldots, h)}{h} = 0$$

- A method has **order** p if

$$\tau_{n+1} = O(h^{p+1}) \quad \text{as} \quad h \to 0$$

- A method is said **convergent** if x_n approaches the true solution $x(t_n)$ as the time step is refined to zero: $\lim_{h\to 0} x_n = x(t_n)$, or equivalently

$$\lim_{h\to 0} \max_{0<n<N} \|\epsilon_n\| = 0$$

- A method is said to be **stable** if there exists an integration time step $h_0 > 0$ so that for any $h \in [0, h_0]$, a finite variation of the state at time t_n leads only a non-increasing variation of the state at a successive time. Consistency and convergence are necessary and sufficient conditions for stability. See also *zero-stability*.

- A system is said to be **stiff** if stability requirements, rather than those of accuracy, constrain the step length to be small. Stiff systems, in a mechanical intuitive sense, are those that exhibit sudden changes of values in the state vector, as in jumps, sudden oscillations, quick decays.

  The big issue of stiff systems is that, in common engineering scenarios, those quick phenomena are often interleaved by calm and smooth time intervals, so if one picks a short time step to be stable across the quick phenomena, it also wastes computational time during the smooth events where a long time step woudl have guaranteed stability anyways. Variable-time-stepping schemes can fix this to some extent, but not always; a more radical solution is to use implicit methods that exhibits *absolute* stability -see later- and that, for this reason, can overcome stiff phenomena even with large time steps.

- In order to study the stability of methods, one can restrict the attention to a special case of ODE, that is the *linearly stiff* differential equation

$$\dot{x} = \lambda x \quad \text{for } \lambda \in \mathbb{C}_- \,,\; x_0 \in \mathbb{R}$$

  for whom, by the way, one already knows the analytical solution $x(t) = x_0 e^{\lambda t}$. This can be stable only if $\mathrm{Re}\lambda < 0$ otherwise $x(t)$ will grow to ∞ anyway, also in the analytical solution regardless of the numerical scheme, that is why we restrict λ to be a complex but excluding the positive real plane: $\lambda \in \mathbb{C}_-$, thus decaying from x_0 and converging as $\lim_{t\to 0} x(t) = 0$. Finally, one can intuitively see that the larger $|\lambda|$, the faster the decay, thus the stiffer the equation.

- If we can express an integration method, applied to the linearly stiff test equation above, as a product $x_{n+1} = \Phi(h\lambda)x_n$, the term $\Phi(h\lambda)$ is called **stability function**, and one can see that the series converges $\lim_{n\to\infty} x_n = 0$ only for $|\Phi(h\lambda)| < 1$. For the values $h\lambda$ where it holds, the method is called **linearly stable**.

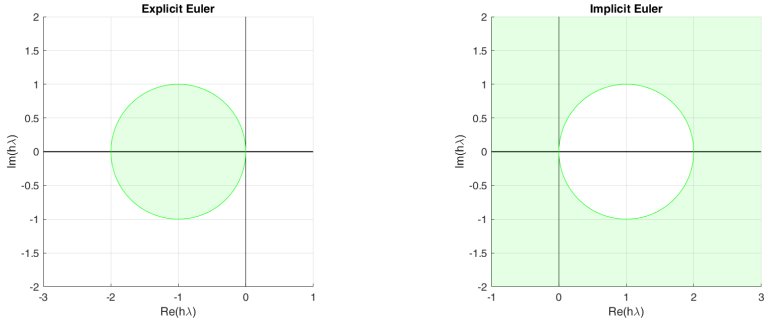

Figure 5: Stability region of explicit Euler (left) and stability region of implicit Euler (right)

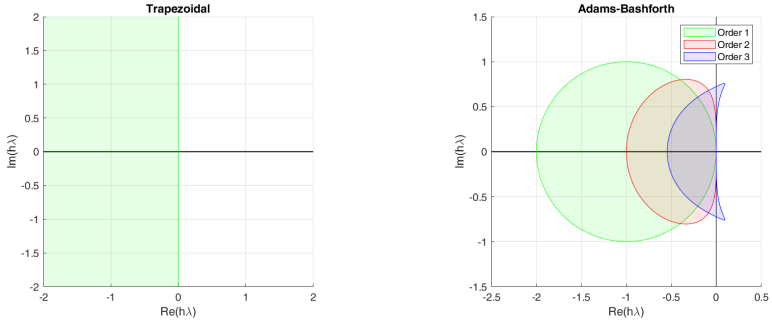

Figure 6: Stability region of trapezoidal method (left) and stability region of Adams-Bashforth (right)

- For a given method, one can plot the region(s) where the linearly stable property holds, as a function of the $h\lambda$ value in the complex plane, obtaining areas shown in Fig.7. Those region where the stability function is $|\Phi(h\lambda)| < 1$, is called **stability region** $\mathcal{R}_A$of the method

$$\mathcal{R}_A = \{h\lambda \,:\, |\Phi(h\lambda)| < 1\}$$

- All methods of practical interest contain the origin $0, 0$ of the plane, that is called **zero-stability**. Note that, for bounded stability regions like circles etc., the larger the h value, or the larger the λ value, the farther we go from the origin and the easier we go out of the stability region. This explains why the need of using short h if meeting large stiffness λ.

- Methods whose stability region covers at least the entire half plane of negative real values, are called *absolutely stable*, hence the property called **A-stability**.

$$\mathrm{A-stability} \;\Leftrightarrow\; \mathcal{R}_A \supseteq \{h\lambda \,:\, \mathrm{Re}\{h\lambda\} < 0\}$$

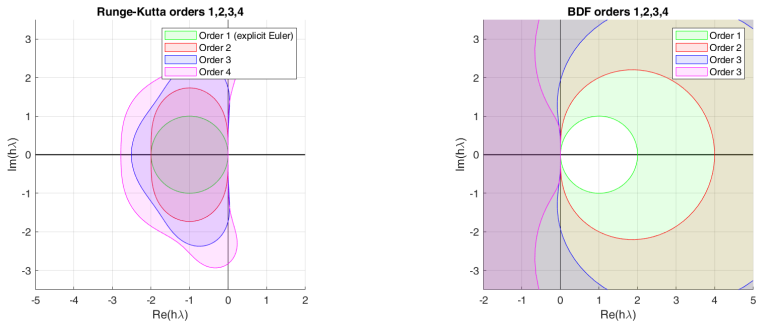

Figure 7: Stability region of explicit Runge-Kutta (left) and stability region of implicit BDF (right)

Those method converges to zero with the test function regardless of how much stiffness λ or how large is h. Example: the implicit Euler, the trapezoidal method. Note: not all implicit methods are A-stable, for example BDF of order greater than 2 cover a large portion of the left half plane, but not entirely.

- Theorem (the **second Dahlquist barrier**). An A-stable multistep method must be of order $r \leq 2$. [17]

- A more relaxed requirement it the **A(α)-stability**, where the stability region must include a cone in the left semiplane, with aperture 2α:

$$\mathrm{A}\alpha\text{−stability} \Leftrightarrow \mathcal{R}_A \supseteq \{h\lambda : -\alpha < \pi - \arg(h\lambda) < \alpha\}$$

- A more stringent requirement is the **L-stability**, that requires the method to be A-stable plus $\lim_{h\lambda\to-\infty} |\Phi(h\lambda)| = 0$. For instance, the trapezoidal method is A-stable but not L-stable because its $|\Phi(h\lambda)| \to 1$ when $\lambda \to -\infty$[18]. The implicit Euler is L-stable.

- From the stronger to the more relaxed property:

$$\mathrm{L} - \text{stability} \Rightarrow \mathrm{A} - \text{stability} \Rightarrow \mathrm{A}\alpha\text{−stability} \to \mathrm{A}_0\text{−stability}$$

From a practical perspective, methods that stay on the left of this scale are more and more robust (in the computer science sense, that is, they

[17] This means that, despite the efforts, no one would ever invent a method that has absolute stability and integration order greater than 2. If one needs an high order because he is concerned with extreme precision, as it happens in celestial mechanics when studying trajectories, probably he better give up with the requirement of absolute stability (luckily enough, in those cases there are no problems of stiff integration).

[18] In practical scenarios, this can be seen as the known issue of the trapezoidal method: it dampens decays and harmonics like other implicit methods, but the damping is less and less noticeable for increasing frequencies.

cope better with unknown and unpredictable systems). For instance, the implicit Euler, although a first order and not so accurate integrator, is often used in VR, videogames and real-time applications because of its L-stability, that make it able to handle sudden stiff situations without exploding into divergence.

- Remember that the stability classes introduced so far are relative to a linearly-stiff test function, but in real scenarios the right hand side can be highly non-linear, therefore it is not guaranteed that even a L-stable method will work all the times. For example, divergence or breakdown of the method could happen for many reasons such as missed convergence in Newton-Raphson iterations, resonance and aliasing with rheonomic terms, finite floating point precision, etc. For instance, if simulating contacts with stiff penalty functions, also L-stable methods could diverge or give unusable results. This said, anyway, L-stable and A-stable method are much less likely to have problems than others.

- Despite the superior stability properties, implicit methods have some drawbacks. First, they introduce some numerical damping, that is more and more visible the larger the h time step. For example, the implicit Euler for DAEs presented in these pages will approach a static analysis for $h \to \infty$ - look at the formulas. This can be a positive side effect (this artificial numerical damping can be welcome to make the simulation more stable and robust), but if not needed, few implicit methods allow to adjust it, one notable case is the generalized-α method.

- Another drawback of implicit methods: they require a complex implementation leading to at least 3-4 Newton-Raphson iterations at each time step (hence the need of solving 3-4 linear systems per time step). Explicit integrators on the other hand do not need any linear system to be solved, so they require much less CPU time per each time step. Therefore, implicit integrators pay back if you use large time steps thus making good use of their stability properties. Otherwise, if you must use extremely short time steps anyway, for example for geometric reasons such as when studying high-speed collisions with thin objects, maybe that explicit integrators would be more efficient (as they would be stable anyway with such short time steps).