Lecture Notes for MVComp 1

in Winter Semester 2019/2020

Fundamentals of Simulation Methods

originally by Volker Springel (MPA)

modified and extended by Cornelis Dullemond (ZAH),
Philipp Girichidis (AIP), Frauke Gräter (HITS),
Rüdiger Pakmor (MPA), Christoph Pfrommer (AIP),
Friedrich Röpke (HITS/ZAH), and Ralf Klessen (ZAH)

February 11, 2021

MPA:
Max-Planck-Institut für Astrophysik (MPA)
Karl-Schwarzschild-Str. 1
85748 Garching

ZAH:
Zentrum für Astronomie der Universität Heidelberg
Institut für Theoretische Astrophysik
Albert Ueberle Str. 2
69120 Heidelberg

HITS:
Heidelberg Institute for Theoretical Studies
Schloss-Wolfsbrunnenweg 35
69118 Heidelberg

AIP:
Leibniz-Institut für Astrophysik Potsdam
An der Sternwarte 16
14482 Potsdam

Contents

13 Finite Element methods and the Galerkin formalism

13.1 Introduction

In an earlier chapter we discussed how a linear boundary value problem can be numerically solved by converting it into a matrix equation (i.e. a set of coupled linear equations). The method we used was to set up a grid in space, and formulate the differential operators (e.g. the Laplace operator) as finite differences. As an example we solved the Poisson equation for gravity, and set the potential at the boundary to zero.

In multiple dimensions the boundaries of such a regular grid are rectangular if a cartesian grid is used. For a spherical coordinate system the boundaries are either parts of a sphere or parts of a cone. For the example of gravity this is not a problem, because we anyway put the boundaries as far as possible away from the regions of interest.

However, there are numerous boundary value problems in physics and engineering for which the shape of the boundary is much more complex. For example: the flow of air over an airplane, or the strain in a solid object when subject to external stresses. For such boundary value problems we need flexible grids, or "unstructured grids". It then becomes much more difficult to express the partial differential equations (PDEs) in discrete form on such a non-rectangular grid.

In this chapter we will discuss methods for handling such problems. We will focus on the family of *elliptic equations* of which the Poisson equation is the simplest example. The problem is formulated on a volume V with a closed surface $S = \partial V$. The boundary conditions are specified on ∂V. The volume V and its surface ∂V can have arbitrarily complex shape. For instance, it could be that of a concrete bridge. The volume V is then the space filled by the concrete, and ∂V is the surface of the concrete.

A *Finite Element Method* (FEM) is a method of computation in which the elliptic equations are expressed on an irregular (unstructured) grid sampling the volume V. Each grid cell can have arbitrary shape. If we choose them to be cubes, then we are back at a regular grid. But if we choose them to be tetrahedron (in 3-D) or triangles (in 2-D) we are much more flexible. We can even have different shape types within the same irregular grid. The only condition is that the cells completely fill the volume V. These grid cells are the "finite elements" of the finite element method. If we apply FEM to structural analysis of, e.g., concrete buildings, these "elements" can be regarded as "pieces of concrete" with certain material properties. But in general these "elements" are better viewed as irregularly shaped grid cells.

Given the arbitrary shape of these grid cells it is no longer obvious how to express the elliptic equations in the form of finite differences (which is necessary to numerically solve the equations). For the regular grid we could use our intuition: $\nabla^2\phi = (\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} - 4\phi_{i,j})/dx^2$. This is no longer possible for such irregular grids. A rigorous way to derive the discrete version of the elliptic equations on a finite element grid is the *Galerkin approach*. Most FEM methods are based on this approach.

The Galerkin method is, however, a more general method to discretize a set of partial differential equations. It can also be used outside of the FEM context.

13.2 Galerkin method of discretizing a PDE

13.2.1 Basic method

Consider a linear elliptic partial differential equation

$$L[y(\mathrm{x})] = f(\mathrm{x}) \tag{13.1}$$

where y is a scalar function in a 3-D space, x is the coordinate vector in that space, $L[]$ is a linear differential operator (for instance the Poisson operator $L[y] = \nabla^2 y$) and f is a given scalar function. The equation is to be solved for $y(\boldsymbol{x})$ within a bounded volume V with surface ∂V. On the surface ∂V boundary conditions are specified, either of the Neumann or Dirichlet type. The simplest version of this equation is:

$$\nabla^2 y(\mathrm{x}) = f(\mathrm{x}) \tag{13.2}$$

which is the Poisson equation.

The method of Galerkin is to devise a set of convenient functions $\phi_i(\boldsymbol{x})$ (for $i = 1, \cdots, n$, where n is the finite number of functions you choose), and find the linear combination of these functions that best approximates the solution. The choice of functions $\phi_i(\boldsymbol{x})$ is quite arbitrary: you can choose whatever you like. But they have to be linearly independent. And if we have $y = 0$ as a boundary condition, these functions have to become zero at the boundary ∂V. We then construct the function $y(\mathrm{x})$ as a linear sum:

$$y(\mathrm{x}) = \sum_{i=1}^{n} \alpha_i\,\phi_i(\mathrm{x}) \tag{13.3}$$

where α_i are numerical constants. The task is to find the values of α_i for which $y(\mathbf{x})$ is closest to the solution of the PDE.

To quantify what is "closest to the solution of the PDE", we define what is called the *residual*:

$$R[y(\mathbf{x})] = L[y(\mathbf{x})] - f(\mathbf{x}) \tag{13.4}$$

If $y(\mathbf{x})$ is the true solution, the residual is zero everywhere. But if $y(\mathbf{x})$ is given by the linear sum of a finite set of basis functions $\phi_i(\mathbf{x})$, then there will likely remain a non-zero residual.

The method of Galerkin determines the values of α_i by demanding that the residual function $R[y(\mathbf{x})]$ should be linearly *orthogonal* to each and every basis function $\phi_i(\mathbf{x})$, with respect to the L^2 inner product. As a reminder: the L^2 inner product between two functions $f(\mathbf{x})$ and $g(\mathbf{x})$ is defined on the volume V as

$$\langle f(\mathbf{x}), g(\mathbf{x})\rangle = \int_V f(\mathbf{x})g(\mathbf{x})\, d^3x \tag{13.5}$$

So the Galerkin method demands

$$\int_V \phi_i(\mathbf{x})R[y(\mathbf{x})]\, d^3x = 0 \quad \forall i \in [1,n] \tag{13.6}$$

and the task is to find the values of α_i that ensures this.

What is the reason for demanding this orthogonality? If we regard our set of basis functions $\phi_i(\mathbf{x})$ to span a linear subspace of the full linear space of functions on V, then by demanding that the residual is orthogonal to that subspace, we essentially demand that the *projection* of the residual onto that subspace is zero. In other words: within the (finite-dimensional) subspace, the residual is zero, even though it may still be non-zero in other dimensions. This is the best we can do with a finite set of basis functions, and the resulting values of α_i thus leads to the best approximation of the solution, given this set of basis functions.

So let us see how we can determine the values of α_i. We insert the expression for $R[y(\mathbf{x})]$ into the orthogonality condition,

$$\int_V \phi_i(\mathbf{x})\left(L[y(\mathbf{x})] - f(\mathbf{x})\right)\, d^3x = 0 \quad \forall i \in [1,n] \tag{13.7}$$

and then replace $y(\mathbf{x})$ by its expansion into the basis functions $\phi_i(\mathbf{x})$

$$\int_V \phi_i(\mathbf{x})\left(\sum_{k=1}^{n} \alpha_k\, L[\phi_k(\mathbf{x})] - f(\mathbf{x})\right) d^3x = 0 \quad \forall i \in [1,n] \tag{13.8}$$

We obtain

$$\sum_{k=1}^{n} \alpha_k \int_V \phi_i(\mathbf{x})\, L[\phi_k(\mathbf{x})]\, d^3x - \int_V \phi_i(\mathbf{x})\, f(\mathbf{x})\, d^3x = 0 \quad \forall i \in [1,n] \tag{13.9}$$

If we have chosen the functions $\phi_i(\mathbf{x})$ wisely, then we will be able to *analytically* compute the above integrals. This is the crucial step! We do not want to have to numerically have to compute these integrals, because then we are again confronted with the question which method of numerical integration to use. Therefore, in practice we choose $\phi_i(\mathbf{x})$ to be simple functions for which we can easily calculate the result of the operator $L[]$ and which can be easily integrated. For instance: polynomials, or piecewise polynomials, or sine/cosine waves. We will discuss a piecewise linear set of basis functions in the section on FEM.

13.2.2 Advantage of the weak form: simpler basis functions

In principle we have now already reached our goal. Let us analytically compute the integrals and name these values in the following way:

$$M_{ik} = \int_V \phi_i(\mathbf{x})\, L[\phi_k(\mathbf{x})]\, d^3x \tag{13.10}$$

$$b_i = \int_V \phi_i(\mathbf{x})\, f(\mathbf{x})\, d^3x \tag{13.11}$$

the above equation then becomes

$$\sum_{k=1}^{n} M_{ik}\, \alpha_k = b_i \quad \forall i \in [1, n] \tag{13.12}$$

which we recognize as a matrix equation of the type

$$M \cdot \alpha = b \tag{13.13}$$

where M is a matrix. Both M and b are given, and we solve for α. We have learned in earlier chapters how to deal with such equations. If we choose our basis functions $\phi_k(\mathbf{x})$ cleverly, then we can assure that M is a *sparse* matrix. The solution of this problem will then more easily scale to huge n.

In principle we are now done. But we can do better. So far we must choose our basis functions $\phi_i(\mathbf{x})$ to be differentiable according to the operator $L[]$. Suppose that $L = \nabla^2$, this means that the functions $\phi_i(\mathbf{x})$ have to be twice differentiable: they must have finite first *and* second derivatives. This is in principle not a problem, but it is more convenient to allow, for instance, piecewise linear functions such as a pyramid. These are first order differentiable, but have diverging second derivatives (they are delta functions at the edges/kinks of these functions). With a trick of partial integration we can fix this.

Suppose we have $L[] = \nabla^2$. The matrix elements M_{ik} are then

$$M_{ik} = \int_V \phi_i(\mathbf{x})\, \nabla^2 \phi_k(\mathbf{x})\, d^3x \tag{13.14}$$

Using partial integration we can rewrite this as

$$M_{ik} = \int_{\partial V} \phi_i(\mathbf{x})\,\nabla\phi_k(\mathbf{x})\cdot\boldsymbol{n}\,dS - \int_V \nabla\phi_i(\mathbf{x})\cdot\nabla\phi_k(\mathbf{x})\,d^3x \tag{13.15}$$

where $\mathbf{n}$ is, as usual, the normal vector on the surface ∂V. Since we have chosen $\phi_i(\mathbf{x}) = 0$ on ∂V, the surface integral vanishes, and we obtain

$$M_{ik} = -\int_V \nabla\phi_i(\mathbf{x})\cdot\nabla\phi_k(\mathbf{x})\,d^3x \tag{13.16}$$

We see that we can compute the matrix elements M_{ik} also for basis functions that are only singly differentiable. To achieve this, we have made use of the fact that M_{ik} does not depend on $\nabla^2\phi_k(\mathbf{x})$, but on its integral.

We may recognize this from a similar phenomenon in hyperbolic PDEs such as the equations of hydrodynamics. In the partial differential equations of hydrodynamics we find the divergence operator $\nabla\cdot$ appearing in each conservation equation. That implies that the functions have to be differentiable. However, when we integrate the equation over a volume V (which could be a grid cell, for instance), we find that the integral form of the equations is also valid if the solutions have jumps (i.e. are not differentiable). In other words: the integral form of the equations allows one degree less smooth solutions or basis functions than the differential form.

In both elliptic PDEs and hyperbolic PDEs the integral form of the equations is called the *weak* form, while the PDE-form is the *strong* form of the equations. The Galerkin method casts the equations into weak form, and thus allows one degree less smooth basis functions to be used than if the strong form was used. This opens the door for *piecewise* polynomial basis functions to be used: functions that are polynomial on finite small areas, but have kinks between two such neighboring small areas. The simplest kind are piecewise linear basis functions such as, in 1-D, the triangle (which we will discuss in the FEM method).

13.2.3 The concept of a "test function"

In the literature on the Galerkin method you often read about a mysterious "test function" $w(\mathbf{x})$ that is used to derive the Galerkin equations. Starting from the PDE itself, the PDE is multiplied by this test function and then integrated over V:

$$\int_V w(\mathbf{x})\,(L[y(\mathbf{x})] - f(\mathbf{x}))\,d^3x = 0 \tag{13.17}$$

This equation then has to hold for all test functions $w(\mathbf{x})$. This is the formal way to cast the PDE into a weak form.

The next step is then to expand $y(\mathbf{x})$ into the basis functions $\phi_k(\mathbf{x})$ as before. If we have n basis functions $\phi_k(\mathbf{x})$, then we cannot demand that the above weak equation holds for *all* possible test functions $w(\mathbf{x})$. But we can choose a set of n well-chosen (but in the end arbitrary) test functions $w_i(\mathbf{x})$. This will lead to n equations (one for each test functions $w_i(\mathbf{x})$) for n unknown values α_k, and we again arrive at a matrix equation of the form $M \cdot \alpha = b$, as before.

If we choose $w_i(\mathbf{x}) = \phi_i(\mathbf{x})$, then we are back at the Galerkin method we discussed above. But we can also choose other kinds of functions. For instance we can set $w_i(\mathbf{x}) = \delta(\mathbf{x}-\mathbf{x}_i)$, where $\mathbf{x}_i$ are a set of chosen grid points. This choice demands that the residual vanishes at those grid points.

13.2.4 Example in 1-D

Let us taken the example of the Poisson equation in 1-D:

$$\frac{\partial^2 y(x)}{\partial x^2} = f(x) \tag{13.18}$$

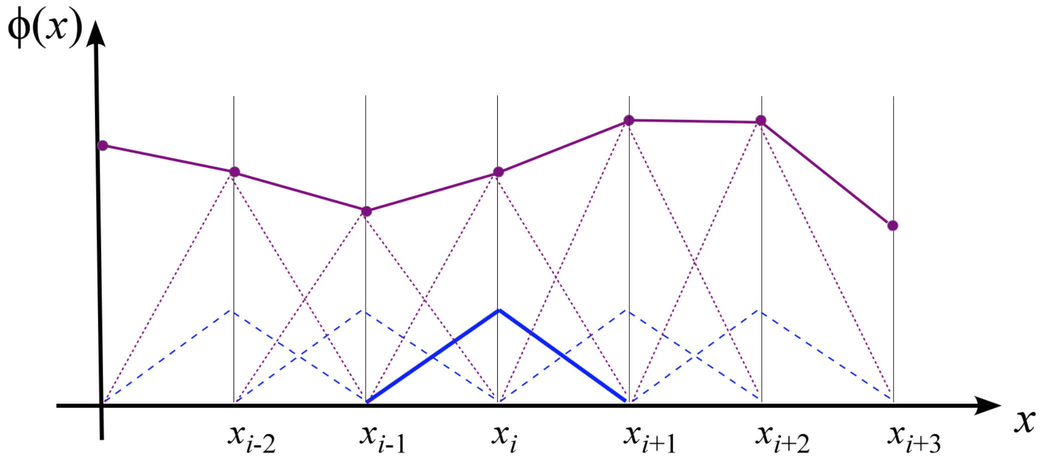
Let us define special "grid points" x_i:

$$x_i = x_0 + i\,\Delta x \tag{13.19}$$

Now choose as basis functions $\phi_i(x)$:

$$\phi_i(x) = \begin{cases} \frac{x-x_{i-1}}{\Delta x} & \text{for} \quad x \in [x_{i-1}, x_i] \\ \frac{x_{i+1}-x}{\Delta x} & \text{for} \quad x \in [x_i, x_{i+1}] \\ 0 & \quad\text{otherwise} \end{cases} \tag{13.20}$$

The construction of $y(x)$ from the linear sum of these basis functions then looks like this:



The matrix elements M_{ik} can then be computed from the inner products of the derivatives of these functions:

$$M_{ik} = -\int \frac{\partial\phi_i(x)}{\partial x}\frac{\partial\phi_k(x)}{\partial x}dx \tag{13.21}$$

These derivatives are:

$$\frac{\partial\phi_i(x)}{\partial x} = \begin{cases} \frac{1}{\Delta x} & \text{for} \quad x \in [x_{i-1}, x_i] \\ -\frac{1}{\Delta x} & \text{for} \quad x \in [x_i, x_{i+1}] \\ 0 & \quad \text{otherwise} \end{cases} \tag{13.22}$$

This leads to

$$M_{ik} = \begin{cases} -\frac{2}{\Delta x} & \text{for} \quad i = k \\ \frac{1}{\Delta x} & \text{for} \quad i = k \pm 1 \\ 0 & \quad \text{otherwise} \end{cases} \tag{13.23}$$

Apart from a factor Δx This is exactly the same matrix as we found when we discretized the Poisson equation on a regular grid! It shows that the Galerkin method using the simple triangular piecewise linear basis functions, when applied to a regular grid, reproduces what we have earlier derived. Now it is important to recall why we did all this effort: we now have a method, the Galerkin method, that is much more general than the regular grid discretization method. This brings us to the topic of Finite Elements.

13.3 Finite Element Method

Let us recall what the basic idea of the FEM method was: to divide the volume V into a discrete set of "cells" called "elements", and to express our equations on these elements. To derive the discrete version of the equations on these elements, we make use of the method of Galerkin of the previous section. The idea is to chose the basis functions ϕ_i such that they describe the interaction between the cells.

13.3.1 1-D Example

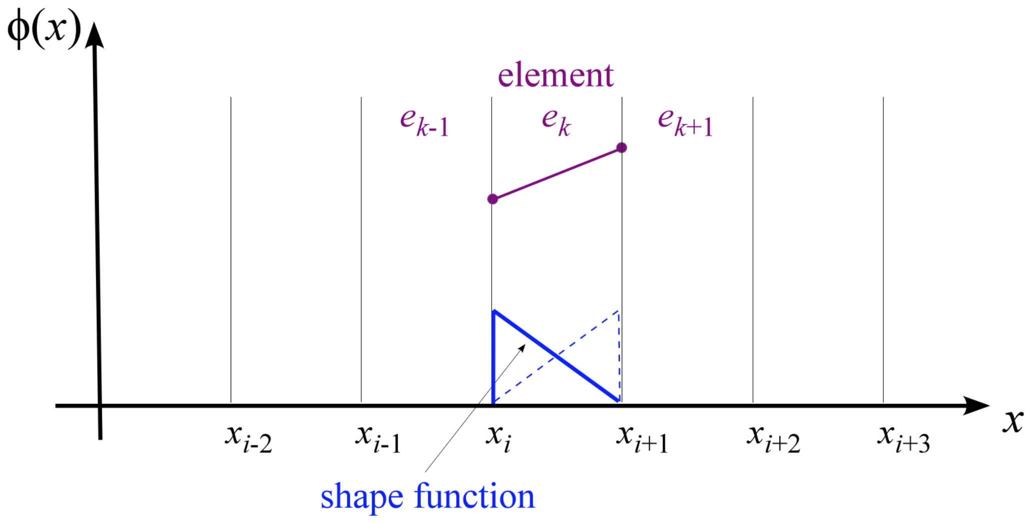
Let us first restrict ourselves to 1-D, so as to keep it easier to visualize. We again choose for simplicity a regular grid $x_i = x_0 + i\,\Delta x$. These grid points should be viewed as the cell walls of the cells, where cell k is the cell between x_i and x_{i+1}. We call this cell an "element". Any linear function between x_i and x_{i+1} (i.e. any linear function *within* the element k) can be regarded as the superposition of two *shape functions* $s_{k,L}(x)$ and

$s_{k,R}(x)$:

$$s_{k,L}(x) = \begin{cases} (x_{i+1} - x)/\Delta x & \text{for} \quad x \in [x_i, x_{i+1}] \\ 0 & \text{otherwise} \end{cases} \tag{13.24}$$

$$s_{k,R}(x) = \begin{cases} (x - x_i)/\Delta x & \text{for} \quad x \in [x_i, x_{i+1}] \\ 0 & \text{otherwise} \end{cases} \tag{13.25}$$

These two shape functions for each cell look like this:



where the solid blue line shows $s_{k,L}(x)$ and the dashed blue line shows $s_{k,R}(x)$. For each element, these two shape functions determine what the element looks like.

Next we need to "glue" the elements together, so that they can interact with one another. If we glue element k to its left to element $k-1$, then we need to combine shape function $s_{k,L}(x)$ with shape function $s_{k-1,R}(x)$ into a single basis function $\phi_i(x)$. This function has to be continuous, and so the value of $s_{k,L}(x = x_i)$ has to be equal to the value of $s_{k-1,R}(x = x_i)$. In our case this happens to be the case. If not, then we should adapt the shape functions accordingly. The result is:

$$\phi_i(x) = s_{i,L}(x) + s_{i-1,R}(x) \tag{13.26}$$

The basis function $\phi_i(x)$ is therefore the function that glues the two elements to each other. We are then back at the triangular basis functions we used above.

In this 1-D example this is not yet very useful, because this is identical to the simplest form of the Galerkin method on a 1-D grid, and that was identical to the simple discretization of the 1-D Poisson equation. But now we have the tools to go to 2-D and 3-D and use irregularly shaped grids.

13.3.2 2-D Example

Now let us go to 2-D. Let us use the simplest possible finite elements: triangles. We can "triangulate" the "volume" V (in 2-D this is a surface of course) out to the edge ∂V. Since the triangular elements always have straight lines, it will not be possible to perfectly match the edge ∂V, but with sufficiently many small triangles it should be good enough.

Each triangle k is described by 3 nodes (corners): $\mathbf{x}_{k,l}$ with $l = 1, 2, 3$. We can describe each point x inside this triangle by two coordinates ξ and η:

$$\mathbf{x} = \mathbf{x}_1 + \xi\,(\mathbf{x}_2 - \mathbf{x}_1) + \eta\,(\mathbf{x}_3 - \mathbf{x}_1) \tag{13.27}$$

Any linear function $g(\xi, \eta)$ on this triangle can then be written as

$$g(\xi, \eta) = g_1 + g_2\xi + g_3\eta \tag{13.28}$$

If we now want to introduce linear shape functions similar to those we introduced in the 1-D case, we may want to have these functions being 1 at one of the nodes, and 0 at the others. For the triangle these shape functions will then be

$$S_{k,1}(\xi, \eta) = 1 - \xi - \eta \tag{13.29}$$

$$S_{k,2}(\xi, \eta) = \xi \tag{13.30}$$

$$S_{k,3}(\xi, \eta) = \eta \tag{13.31}$$

The basis functions $\phi_i(\boldsymbol{x})$ will then be the sum of the shape functions co-joined at vertex $\mathbf{x}_i$. These basis functions will thus link $\mathbf{x}_i$ to all neighboring vertices. The finite elements are thus joined through their vertices.

13.3.3 3-D Example

We can continue to higher dimensions using simplices. A 3-D simplex is a tetrahedron. This has 4 vertices, 4 faces and 6 ribbons. Like in the 2-D case we can define shape functions that are 1 at one of the vertices and zero at the other ones. Everything goes analogously. We will now have three variables ξ, η and λ and four shape functions.

13.4 Structural mechanics

So far we have discussed only a very simple set of elliptic equations. In engineering we often encounter a more complicated problem: how to compute the deformation of a solid object when it is put under stress. For a more in-depth discussion, see e.g. `http://web.mit.edu/16.20/homepage/`.

13.4.1 Formulation

Let us describe the deformation of a solid object filling a volume V with surface ∂V. Deformation is that any point $\mathbf{x}$ in that object is displaced by a small displacement vector $\delta\mathbf{x}$. The new physical location $\mathbf{X}$ is then

$$\mathbf{X} = \mathbf{x} + \delta\mathbf{x} \tag{13.32}$$

The displacement can be a function of $\mathbf{x}$ itself. We thus have:

$$\mathbf{X}(\mathbf{x}) = \mathbf{x} + \delta\mathbf{x}(\mathbf{x}) \tag{13.33}$$

If we write $\mathbf{x}$ out into components (x, y, z) we get:

$$\begin{aligned} X(x,y,z) &= x + \delta x(x,y,z) && (13.34)\\ Y(x,y,z) &= y + \delta y(x,y,z) && (13.35)\\ Z(x,y,z) &= z + \delta z(x,y,z) && (13.36) \end{aligned}$$

Not all displacements lead to deformation. For instance,

$$\begin{aligned} \delta x(x,y,z) &= \delta x && (13.37)\\ \delta y(x,y,z) &= \delta y && (13.38)\\ \delta z(x,y,z) &= \delta z && (13.39) \end{aligned}$$

(i.e. a constant displacement vector) is merely a translation, and

$$\begin{aligned} \delta x(x,y,z) &= x\cos\alpha + y\sin\alpha - x && (13.40)\\ \delta y(x,y,z) &= -x\sin\alpha + y\cos\alpha - y && (13.41)\\ \delta z(x,y,z) &= 0 && (13.42) \end{aligned}$$

is simply a rotation.

However, we can define the *strain* tensor

$$\epsilon_{kl} = \frac{1}{2}\left(\nabla_k \delta x_l + \nabla_l \delta x_k\right) \tag{13.43}$$

If any component of this tensor is non-zero, $\epsilon_{kl} \neq 0$, then the displacements describe a deformation of the object rather than just a translation or rotation. The strain tensor can be decomposed into a diagonal part and an off-diagonal part. In 2-D this would be:

$$\epsilon_{kl} = \begin{pmatrix} \nabla_x \delta x & \frac{1}{2}(\nabla_x \delta y + \nabla_y \delta x) \\ \frac{1}{2}(\nabla_x \delta y + \nabla_y \delta x) & \nabla_y \delta y \end{pmatrix} \tag{13.44}$$

The diagonal components describe the *stretching* (or when negative: *compression*), while the off-diagonal components describe the *shear*.

Strain leads to *stress*. Stress is a description of the forces acting inside the solid as a result of the strain. Stress can also be described as a second rank tensor: σ_{ik}. In general, for small displacements (i.e. in the linear regime), we can write the relation between stress and strain as

$$\sigma_{ij} = \sum_{kl} C_{ijkl}\epsilon_{kl} \tag{13.45}$$

where C_{ijkl} is called the *stiffness tensor*. This is the general linear form of the *stress-strain relation*. As long as we are in the linear regime, stress and strain are linearly proportional to each other, as the above linear stress-strain relation implies. We are then in the *elastic regime*, and forces are reversible. No entropy is generated.

But for very strong displacements, the material may get over-strained, and cracks may appear inside the material. Then the stress will cease to linearly increase with increasing strain. Instead, it will increase sub-linearly. We are then in the *plastic regime*. The plastic regime is not reversible. Entropy is generated, because the material is irreversibly damaged.

If we continue to increase the strain in spite of being in the plastic regime, the cracks eventually will become so large that the stress drops to zero and the material breaks into two or more pieces.

For planning the construction of some building, bridge, airplane or car, the aim is to keep the stresses under all circumstances smaller than the plastic regime at all locations in the volume V. To test this, for a multitude of different external forces or externally imposed strains, we need FEM.

13.4.2 Material properties

Solid materials can have very complex properties. For instance, if they consist of fibers, then their properties are non-isotropic. The same is true for crystals. The rank-4 tensor C_{ijkl} can contain this full complexity.

Let us, however, focus here on simple isotropic materials. If we stretch or compress such materials, the stress inside the material will then be along the same direction. For instance if we focus on the 1-direction we can write

$$\epsilon_{11} = \frac{1}{E}\sigma_{11} \tag{13.46}$$

where E is called *Young's modulus* and is a material property. This is called *Hooke's law*. In reaction to a stretch in x (i.e. 1-) direction, the material wants to shrink (in general) in the other directions:

$$\epsilon_{22} = \epsilon_{33} = -\nu\epsilon_{11} = -\frac{\nu}{E}\sigma_{11} \tag{13.47}$$

The parameter ν is the *Poisson ratio* and is mostly between 0 and 0.5. Some materials have $\nu < 0$, but it can never be smaller than -1. Material is incompressible when $\nu = 0.5$. In general one can thus write:

$$\epsilon_{11} = +\frac{1}{E}\sigma_{11} - \frac{\nu}{E}\sigma_{22} - \frac{\nu}{E}\sigma_{33} \tag{13.48}$$

$$\epsilon_{22} = -\frac{\nu}{E}\sigma_{11} + \frac{1}{E}\sigma_{22} - \frac{\nu}{E}\sigma_{33} \tag{13.49}$$

$$\epsilon_{33} = -\frac{\nu}{E}\sigma_{11} - \frac{\nu}{E}\sigma_{22} + \frac{1}{E}\sigma_{33} \tag{13.50}$$

For the shear stress-strain relation one has simpler expressions

$$\epsilon_{12} = \mu\sigma_{12} \qquad \epsilon_{13} = \mu\sigma_{13} \qquad \epsilon_{23} = \mu\sigma_{23} \tag{13.51}$$

where μ is the *shear modulus*. For perfectly elastic materials we have

$$\mu = \frac{E}{2(1+\nu)} \tag{13.52}$$

In matrix notation we can write the full stress-strain relation for such simple isotropic materials as

$$\begin{pmatrix} \epsilon_{11} \\ \epsilon_{22} \\ \epsilon_{33} \\ \epsilon_{12} \\ \epsilon_{13} \\ \epsilon_{23} \end{pmatrix} = \begin{pmatrix} \frac{1}{E} & -\frac{\nu}{E} & -\frac{\nu}{E} & 0 & 0 & 0 \\ -\frac{\nu}{E} & \frac{1}{E} & -\frac{\nu}{E} & 0 & 0 & 0 \\ -\frac{\nu}{E} & -\frac{\nu}{E} & \frac{1}{E} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{\mu} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{\mu} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{\mu} \end{pmatrix} \begin{pmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{12} \\ \sigma_{13} \\ \sigma_{23} \end{pmatrix} \tag{13.53}$$

13.4.3 The static field equations

Typically what we want to solve in structural mechanics is how a solid object reacts to a set of external forces and/or boundary conditions. Forces could be, e.g. gravity (a body force). A boundary condition could be, for instance, that a bridge is fixed in position on both ends, with the horizontal position on one end kept free (to assimilate to expansion/contraction due to temperature fluctuations).

Let us first write down the equation of force equilibrium:

$$\nabla_i\sigma_{ij}(\mathbf{x}) = -f_j(\mathbf{x}) \tag{13.54}$$

where we use Einstein's summation convention on the index i (which we will continue to use henceforth in this chapter). The f_j is the body force, for instance the gravity force (in newton per m^3). If we insert the generalized stress-strain relation, and assume that the stiffness tensor is constant in space (for convenience) we obtain

$$C_{ijkl}\nabla_i\epsilon_{kl}(\mathbf{x}) = -f_j(\mathbf{x}) \tag{13.55}$$

If we insert the expression for ϵ_{kl} we obtain

$$\frac{1}{2}C_{ijkl}\nabla_i\left(\nabla_k\delta x_l(\mathbf{x})+\nabla_l\delta x_k(\mathbf{x})\right)=-f_j(\mathbf{x}) \tag{13.56}$$

Making use of one of the symmetries of the stiffness tensor ($C_{ijkl}=C_{ijlk}$) this simplifies to

$$C_{ijkl}\nabla_i\nabla_k\delta x_l(\mathbf{x})=-f_j(\mathbf{x}) \tag{13.57}$$

We thus have a coupled set of three ($j=1,2,3$) second-order partial differential equations for three unknown functions $\delta x_1(\mathbf{x})$, $\delta x_2(\mathbf{x})$, $\delta x_3(\mathbf{x})$. While it is clearly more complex than the Poisson-type equations we discussed previously, it is also clear that the type of equation is similar: to the left of the $=$ sign we have a double-derivative operator and on the right we have a given function.

As a boundary condition on ∂V there are many possibilities. One could say that $\delta\mathbf{x}=0$ at the boundaries, which is the same as saying that the object is "glued" to some external rigid object. Or we can let the surface free, i.e. there is no external force acting on the surface. This amounts to demanding $n_i\sigma_{ij}=0$, where n_i is the normal vector. Or a combination of both, for instance by demanding that the *shape* of ∂V does not change, i.e. $\mathbf{n}\cdot\delta\mathbf{x}=0$.

The solution methods for these equations using FEM will now be the same as we discussed for the simple Poisson equation.

Note that the field equations can be generalized to become time-dependent dynamic equations. This would allow us to compute, for instance, vibrational eigenmodes of constructions, and see if they remain stable under the influence of wind or earthquakes.

13.5 Simplified structural mechanics FEM model: The truss model

A full-fledged 3-D FEM model of a structural mechanics problem can be somewhat computationally demanding. But for constructions that are made of trusses, a simpler version of FEM can be used: the truss model.

A truss can be regarded mathematically as a 1-D piece of material of a certain length L, with a certain *stiffness*. This 1-D "finite element" can be used within a 3-D space by connecting multiple such trusses at their end-points. Such points of connection (joints) we will call *vertices* or *nodes*. By connecting several such trusses together you can construct 2-D and 3-D objects and test their deformation under external forces.

What we will discuss in this section is called the *direct stiffness method* (DSM).

13.5.1 DSM: A 1-D finite element in 1-D

Let us consider a 1-D rod in 1-D space. The rod has length L, radius r and Young's modulus E. We model the rod by first defining two nodes: Node A having $X_A^{(0)} = 0$ and node B having $X_B^{(0)} = L$ (the superscript (0) means: before the deformation). We now stretch (or squeeze) this rod in the x-direction (the only direction we have, for now). So we impose a displacement of node A, which we denote by δx_A and a displacement of node B, which we denote by δx_B. So we have

$$\begin{aligned} X_A &= X_A^{(0)} + \delta x_A && (13.58) \\ X_B &= X_B^{(0)} + \delta x_B && (13.59) \end{aligned}$$

We can now ask ourselves: Which external force f_A^x do we have to impose on node A and external force f_B^x do we have to impose on node B such that we can achieve the displacements δx_A and δx_B? To answer this equation, let us consider first the case $\delta x_B = 0$ and vary δx_A. If $\delta x_A < 0$, i.e. if we stretch the rod by pulling on A while keeping B fixed, then this can only be done if we indeed impose a force $f_A^x = K\delta x_A$ on node A, where the stiffness coefficient K is given by

$$K = E\pi r^2/L \tag{13.60}$$

But we have to impose the opposite force on node B, otherwise the rod would experience a netto force and start accelerating and fly away. So we have $f_B^x = -K\delta x_A$. A similar logic can be used for the case when we keep $\delta x_A = 0$ and put $\delta x_B \neq 0$. All of this can be expressed in matrix form:

$$\begin{pmatrix} f_A^x \\ f_B^x \end{pmatrix} = \begin{pmatrix} K_{AA} & K_{AB} \\ K_{BA} & K_{BB} \end{pmatrix} \begin{pmatrix} \delta x_A \\ \delta x_B \end{pmatrix} = K \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} \delta x_A \\ \delta x_B \end{pmatrix} \tag{13.61}$$

The matrix is called the stiffness matrix.

So for any set of displacements we now can calculate the required forces to achieve those. But usually we encounter the opposite problem: We impose a set of forces and boundary conditions and wish to compute the displacements. In other words: We wish to solve the matrix equation given by

$$\begin{pmatrix} K & -K \\ -K & K \end{pmatrix} \begin{pmatrix} \delta x_A \\ \delta x_B \end{pmatrix} = \begin{pmatrix} f_A^x \\ f_B^x \end{pmatrix} \tag{13.62}$$

for the displacements $(\delta x_A, \delta x_B)$, for a given force vector (f_A^x, f_B^x). However, we see that the stiffness matrix is degenerate: the first row equals minus the second row. That means that we cannot find δx_A and δx_B for a given set of forces f_A^x and f_B^x. Physically this is logical: We know that we have translation invariance, so setting the forces cannot fully determine δx_A and δx_B. Clearly our system is ill determined.

The trick is to replace one of the two equations with a "boundary condition". For instance, let us insist that $\delta x_A = 0$. To do this, let us replace the first row of the above matrix equation with the equation $\delta x_A = 0$. The matrix equation then becomes:

$$\begin{pmatrix} 1 & 0 \\ -K & K \end{pmatrix} \begin{pmatrix} \delta x_A \\ \delta x_B \end{pmatrix} = \begin{pmatrix} 0 \\ f_B^x \end{pmatrix} \tag{13.63}$$

This equation is solvable to $(\delta x_A, \delta x_B)$. Once we have this solution, we can compute the actual vectors:

$$X_A = X_A^{(0)} + \delta x_A \qquad\qquad X_B = X_B^{(0)} + \delta x_B \tag{13.64}$$

This is now the solution we seek.

13.5.2 DSM: A 1-D finite element in 2-D

We can construct very complex 2-D structures using just rods. But before we do this, we must understand how we can place a 1-D rod into a 2-D space. Instead of just X_A and X_B we now have X_A, Y_A, X_B and Y_B, and likewise for the original node positions $X_A^{(0)}$, ${}^{(0)}Y_A$, $X_B^{(0)}$ and $Y_B^{(0)}$. The same holds for the displacements: Instead of just δx_A and δx_B we now have δx_A, δy_A, δx_B and δy_B. Likewise we have the forces f_A^x, f_A^y, f_B^x and f_B^y. The rod will now have some angle θ and L given by

$$L = \sqrt{(X_B - X_A)^2 + (Y_B - Y_A)^2} \tag{13.65}$$

$$\cos\theta = \frac{X_B - X_A}{L} \tag{13.66}$$

$$\sin\theta = \frac{Y_B - Y_A}{L} \tag{13.67}$$

We do not know X_A, Y_A, X_B and Y_B beforehand. We only know $X_A^{(0)}$, $Y_A^{(0)}$, $X_B^{(0)}$ and $Y_B^{(0)}$ beforehand. We will therefore approximate L and θ to be given by:

$$L = \sqrt{\left(X_B^{(0)} - X_A^{(0)}\right)^2 + \left(Y_B^{(0)} - Y_A^{(0)}\right)^2} \tag{13.68}$$

$$\cos\theta = \frac{X_B^{(0)} - X_A^{(0)}}{L} \tag{13.69}$$

$$\sin\theta = \frac{Y_B^{(0)} - Y_A^{(0)}}{L} \tag{13.70}$$

If the displacements are small enough, then this is OK. If not, then we must iterate (using the newly obtained X_A, Y_A, X_B and Y_B and insert them back into the equations for L and θ).

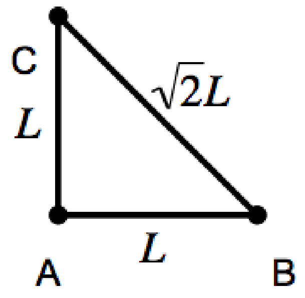


Figure 13.1: The geometry of our simple triangle constructed from three rods.

Now you can verify that the resulting matrix equation becomes

$$\begin{pmatrix} f_A^x \\ f_A^y \\ f_B^x \\ f_B^y \end{pmatrix} = K \begin{pmatrix} \cos^2\theta & \cos\theta\,\sin\theta & -\cos^2\theta & -\cos\theta\,\sin\theta \\ \cos\theta\,\sin\theta & \sin^2\theta & -\cos\theta\,\sin\theta & -\sin^2\theta \\ -\cos^2\theta & -\cos\theta\,\sin\theta & \cos^2\theta & \cos\theta\,\sin\theta \\ -\cos\theta\,\sin\theta & -\sin^2\theta & \cos\theta\,\sin\theta & \sin^2\theta \end{pmatrix} \begin{pmatrix} \delta x_A \\ \delta y_A \\ \delta x_B \\ \delta y_B \end{pmatrix} \tag{13.71}$$

Note that the matrix consists of submatrices

$$\begin{pmatrix} K\cos^2\theta & K\cos\theta\,\sin\theta \\ K\cos\theta\,\sin\theta & K\sin^2\theta \end{pmatrix} \tag{13.72}$$

placed at four positions: the AA position (top left 2×2 matrix), the AB position (top right 2×2 matrix with a minus sign), the BA position (bottom left 2×2 matrix with a minus sign) and finally the BB position (bottom right 2×2 matrix).

13.5.3 DSM: Constructing a triangle with three rods

Using the above model of a 1-D rod in 2-D space above we can now start constructing a simple triangle in 2-D space from three individual rods and compute the stiffness matrix for that triangle. See Fig.13.1 for the geometry. The displacement vector now have 6 components, and so does the force vector. The stiffness matrix now has 6×6 components.

The way we can compute this stiffness matrix in a structured way in our computer program is to do the following: First make a 6×6 matrix and put all elements to 0. Now write a subroutine that connects node i with node j by placing the submatrices Eq. (13.72) with proper sign at the correct position in the matrix. We then call this subroutine for $(i = 1, j = 2)$, for $(i = 1, j = 3)$ and for $(i = 2, j = 3)$. In this way all three rods have been inserted. We leave it as an exercise for the reader to show that we obtain, *for this particular example*:

$$\begin{pmatrix} f_A^x \\ f_A^y \\ f_B^x \\ f_B^y \\ f_C^x \\ f_C^y \end{pmatrix} = K_0 \begin{pmatrix} 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 \\ -1 & 0 & 1+\frac{1}{2\sqrt{2}} & -\frac{1}{2\sqrt{2}} & -\frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} \\ 0 & 0 & -\frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & -\frac{1}{2\sqrt{2}} \\ 0 & 0 & -\frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & -\frac{1}{2\sqrt{2}} \\ 0 & -1 & \frac{1}{2\sqrt{2}} & -\frac{1}{2\sqrt{2}} & -\frac{1}{2\sqrt{2}} & 1+\frac{1}{2\sqrt{2}} \end{pmatrix} \begin{pmatrix} \delta x_A \\ \delta y_A \\ \delta x_B \\ \delta y_B \\ \delta x_C \\ \delta y_C \end{pmatrix} \tag{13.73}$$

with again $K_0 = E\pi r^2/L_0$ with L_0 defined in Fig.13.1. Note that for the horizontal rod we have $K = K_0$. For the vertical rod as well. But for the diagonal rod we have $K = K_0/\sqrt{2}$ because it is longer.

The result can be interpreted in two ways: in one sense we have constructed a small object from three rods. We could add more rods and create more complex objects. In another sense we have created a model for a 2-D triangular finite element. However, this model is just one model, and is not a generally valid model for triangular finite elements.

13.5.4 DSM: Solving for the shape of a construction made out of rods

Let us now demonstrate how we can use the above stiffness matrix formulation to solve an actual problem. Let us assume that we install the triangle against a wall in the way depicted in Fig.13.2. Point A is fixed, point C is allowed to move up and down, but not left or right and point B is completely free. A force f_{ext} is applied downward on point B. We now wish to solve for the displacements of these points. The stiffness equation becomes:

$$K_0 \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1+\frac{1}{2\sqrt{2}} & -\frac{1}{2\sqrt{2}} & -\frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} \\ 0 & 0 & -\frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & -\frac{1}{2\sqrt{2}} \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & -1 & \frac{1}{2\sqrt{2}} & -\frac{1}{2\sqrt{2}} & -\frac{1}{2\sqrt{2}} & 1+\frac{1}{2\sqrt{2}} \end{pmatrix} \begin{pmatrix} \delta x_A \\ \delta y_A \\ \delta x_B \\ \delta y_B \\ \delta x_C \\ \delta y_C \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ -f_{\text{ext}} \\ 0 \\ 0 \end{pmatrix} \tag{13.74}$$

where we have replaced rows 1,2 and 5 of the stiffness matrix of Eq.(13.73) with the condition that $\delta x_A = 0$, $\delta x_A = 0$ and $\delta x_C = 0$, which are the boundary conditions of the problem. Note that such boundary conditions are essential, because otherwise the matrix has zero determinant and the problem is ill-determined. By replacing these three lines with the boundary conditions the problem now becomes solvable. The

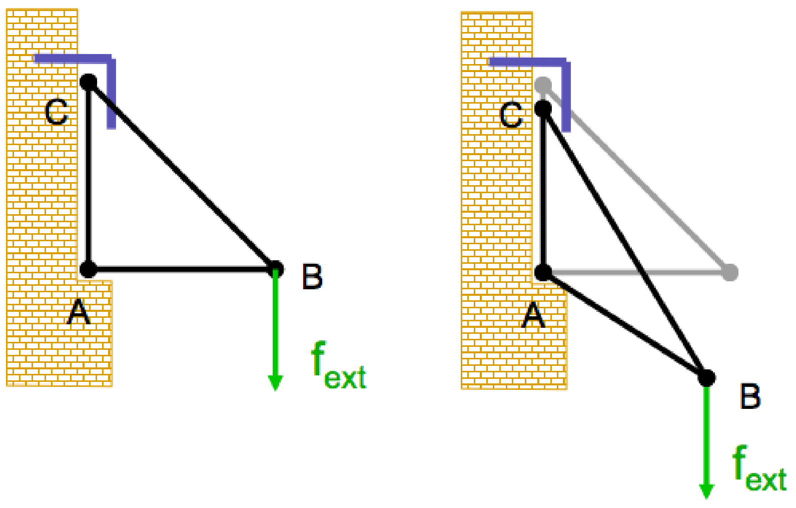


Figure 13.2: The static problem we wish to solve. Left: before application of force. Right: after application of force.

solution is:

$$\delta x_A = 0 \tag{13.75}$$
$$\delta y_A = 0 \tag{13.76}$$
$$\delta x_B = -\frac{f_{\text{ext}}}{K} \tag{13.77}$$
$$\delta y_B = -2(1+\sqrt{2})\frac{f_{\text{ext}}}{K} \tag{13.78}$$
$$\delta x_C = 0 \tag{13.79}$$
$$\delta y_C = -\frac{f_{\text{ext}}}{K} \tag{13.80}$$

13.5.5 DSM: Example of a bridge

Let us now construct something more complex. If we programmed the subroutine that adds a connection between node i and node j well, then this is now fairly simple. For instance, let us make a bridge out of N horizontal segments of length L and a support structure of height H. For this we need $2N+1$ nodes. Their initial X-coordinates are:

$$X_i^{(0)} = \begin{cases} (i-1)*L/2 & \text{for} \quad i = 1,3,5,\cdots,2N+1 \\ (i-0.5)*L/2 & \text{for} \quad i = 2,4,6,\cdots,2N \end{cases} \tag{13.81}$$

and their initial Y-coordinates are:

$$Y_i^{(0)} = \begin{cases} 0 & \text{for} \quad i = 1,3,5,\cdots,2N+1 \\ H & \text{for} \quad i = 2,4,6,\cdots,2N \end{cases} \tag{13.82}$$

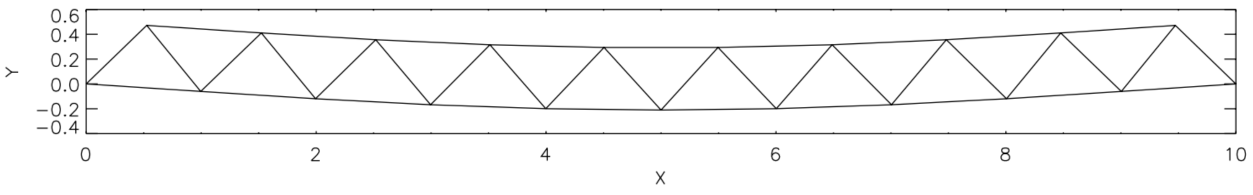


Figure 13.3: The example of the bridge. Here all the nodes are connected by the rods connecting them.

Now we must connect these with $4N - 1$ rods. The starting nodes of the rods are:

$$i_{\text{start}} = 1, 1, 2, 2, 3, 3, 4, \cdots \tag{13.83}$$

and the ending nodes are:

$$i_{\text{end}} = 2, 3, 3, 4, 4, 5, 5, \cdots \tag{13.84}$$

Now the forces: Let us impose a vertical force $-f$ on all nodes except the first and the last one. The force vector then becomes $(0, 0, 0, -f, 0, -f, \cdots, 0, -f, 0, 0)$.

Now we need to impose boundary conditions to eliminate three degrees of freedom: Two translational ones and one rotational one. Let us fix the location of the first node. We thus replace the first two lines of the matrix with $(1, 0, 0, \cdots, 0)$ and $(0, 1, 0, \cdots, 0)$ respectively, and since we want these displacements to be 0, we set the first two elements of the right-hand-side to 0 (which they already are, incidently). We can also fix the location of the last node. Since we need only 1 more boundary condition we could simply fix only the y-coordinate (preventing rotation): We thus replace the last line of the matrix with $(0, 0, \cdots, 0, 1)$ and the last element of the right-hand-side also to 0 (which it already is). We can also, if we want, impose *more than* 3 boundary conditions! For instance, we can fix *both* the x- and y- displacements of the last node to 0. This means replacing the last two rows of the matrix with $(0, 0, \cdots, 1, 0)$ and $(0, 0, \cdots, 0, 1)$ respectively.

An example of a bridge of this kind is shown in Fig. 13.3.

13.6 Discontinuous Galerkin methods

A very powerful FEM approach is to solve a PDE system in a so-called weak formulation in which one gives up on requiring continuity of the solution across elements. In this form, one obtaines discontinuous Galerkin (DG) schemes that can be applied, for example, to the non-linear hyperbolic PDE system of the Euler equations. A particular advantage of DG is a systematic path to derive high-order methods, which is difficult to obtain in classic finite volume methods, where already 3rd order begins to become unwieldy. Also, DG approaches have been shown to be more accurate than finite volume schemes at equivalent computational cost, suggesting that they are also more efficient.

Here we shall use the Euler equations to introduce such a state-of-the-art DG formulation (based on **?**). We recall that the Euler equations are a system of hyperbolic partial differential equations and can be written in compact form as

$$\frac{\partial \boldsymbol{u}}{\partial t} + \sum_{\alpha=1}^{3} \frac{\partial \boldsymbol{f}_\alpha}{\partial x_\alpha} = 0, \tag{13.85}$$

with the state vector

$$\boldsymbol{u} = \begin{pmatrix} \rho \\ \rho \boldsymbol{v} \\ \rho e \end{pmatrix} = \begin{pmatrix} \rho \\ \rho \boldsymbol{v} \\ \rho u + \frac{1}{2}\rho \boldsymbol{v}^2 \end{pmatrix}, \tag{13.86}$$

and the flux vectors

$$\boldsymbol{f}_1 = \begin{pmatrix} \rho v_1 \\ \rho v_1^2 + p \\ \rho v_1 v_2 \\ \rho v_1 v_3 \\ (\rho e + p) v_1 \end{pmatrix} \quad \boldsymbol{f}_2 = \begin{pmatrix} \rho v_2 \\ \rho v_1 v_2 \\ \rho v_2^2 + p \\ \rho v_2 v_3 \\ (\rho e + p) v_2 \end{pmatrix} \quad \boldsymbol{f}_3 = \begin{pmatrix} \rho v_3 \\ \rho v_1 v_3 \\ \rho v_2 v_3 \\ \rho v_3^2 + p \\ (\rho e + p) v_3 \end{pmatrix}. \tag{13.87}$$

The unknown quantities are density ρ, velocity $\boldsymbol{v}$, pressure p, and total energy per unit mass e. The latter can be expressed in terms of the internal energy per unit mass u and the kinetic energy of the fluid, $e = u + \frac{1}{2}\boldsymbol{v}^2$. For an ideal gas, the system is closed with the equation of state

$$p = \rho u(\gamma - 1), \tag{13.88}$$

where γ denotes the adiabatic index.

13.6.1 Solution representation

Lets assume that we partition the domain into elements consisting of non-overlapping cubical cells, which may correspond to a Cartesian mesh of constant resolution, or to an adaptively refined nested grid. Moreover, we follow the approach of a classical modal DG scheme, where the solution in the interior of cell K is given by a linear combination of $N(k)$ orthogonal and normalized basis functions ϕ_l^K:

$$\boldsymbol{u}^K(\boldsymbol{x}, t) = \sum_{l=1}^{N(k)} \boldsymbol{w}_l^K(t)\phi_l^K(\boldsymbol{x}). \tag{13.89}$$

In this way, the dependence on time and space of the solution is split into time-dependent weights, and basis functions which are constant in time. Consequently, the state of a cell is completely characterized by the $N(k)$ weight vectors $\boldsymbol{w}_j^K(t)$.

The above equation can be solved for the weights by multiplying with the corresponding basis function ϕ_j^K and integrating over the cell volume. Using the orthogonality and normalization of the basis functions yields

$$\boldsymbol{w}_j^K = \frac{1}{|K|} \int_K \boldsymbol{u}^K \phi_j^K \,\mathrm{d}V, \quad j = 1, \ldots, N(k), \tag{13.90}$$

where $|K|$ is the volume of the cell. The first basis function is chosen to be $\phi_1 = 1$ and hence the weight $\boldsymbol{w}_1^K$ is the cell average of the state vector $\boldsymbol{u}^K$. The higher order moments of the state vector are described by weights $\boldsymbol{w}_j^K$ with $j \geq 2$.

The basis functions can be defined on a cube in terms of scaled variables $\boldsymbol{\xi}$,

$$\phi_l(\boldsymbol{\xi}) : [-1, 1]^3 \to \mathbb{R}. \tag{13.91}$$

The transformation between coordinates $\boldsymbol{\xi}$ in the cell frame of reference and coordinates $\boldsymbol{x}$ in the laboratory frame of reference is

$$\boldsymbol{\xi} = \frac{2}{\Delta x^K}(\boldsymbol{x} - \boldsymbol{x}^K), \tag{13.92}$$

where Δx^K and $\boldsymbol{x}^K$ are the edge length and cell centre of cell K, respectively. One potential basis is obtained by constructing a set of three-dimensional polynomial basis functions with a maximum degree of k as products of one-dimensional scaled Legendre polynomials $\tilde{P}$:

$$\{\phi_l(\boldsymbol{\xi})\}_{l=1}^{N(k)} = \left\{ \tilde{P}_u(\xi_1)\tilde{P}_v(\xi_2)\tilde{P}_w(\xi_3) | u, v, w \in \mathbb{N}_0 \wedge u + v + w \leq k \right\}. \tag{13.93}$$

The first few Legendre polynomials are

$$\begin{array}{ll} P_0(\xi) = 1 & P_1(\xi) = \xi \\ P_2(\xi) = \dfrac{1}{2}(3\xi^2 - 1) & P_3(\xi) = \dfrac{1}{2}(5\xi^3 - 3\xi) \\ P_4(\xi) = \dfrac{1}{8}(35\xi^4 - 30\xi^2 + 3) & P_5(\xi) = \dfrac{1}{8}(63\xi^5 - 70\xi^3 + 15\xi). \end{array} \tag{13.94}$$

and are obtained as solutions of Legendre's differential equation:

$$\frac{\mathrm{d}}{\mathrm{d}\xi}\left[(1 - \xi^2)\frac{\mathrm{d}}{\mathrm{d}\xi}P_n(\xi)\right] + n(n+1)P_n(\xi) = 0, \quad n \in \mathbb{N}_0. \tag{13.95}$$

They are pairwise orthogonal to each other. Moreover, we define scaled polynomials as

$$\tilde{P}(\xi)_n = \sqrt{2n+1}P(\xi)_n, \tag{13.96}$$

such that

$$\int_{-1}^{1} \tilde{P}_i(\xi)\tilde{P}_j(\xi)\,\mathrm{d}\xi = \begin{cases} 0 & \text{if } i \neq j \\ 2 & \text{if } i = j. \end{cases} \tag{13.97}$$

The number of basis functions for polynomials with a maximum degree of k is

$$N(k) = \sum_{u=0}^{k}\sum_{v=0}^{k-u}\sum_{w=0}^{k-u-v} 1 = \frac{1}{6}(k+1)(k+2)(k+3). \tag{13.98}$$

We note that when polynomials with a maximum degree of k are used, a scheme with spatial order $p = k + 1$ is obtained. For example, piece-wiese linear basis functions ($k = 1$) lead to a scheme which is of second order in space.

13.6.2 Initial conditions

Given initial conditions $\boldsymbol{u}(\boldsymbol{x}, t = 0) = \boldsymbol{u}(\boldsymbol{x}, 0)$, we have to provide an initial state for the DG scheme which is consistent with the solution representation. To this end, the initial conditions can be expressed by means of the polynomial basis on cell K, which will then be

$$\boldsymbol{u}^K(\boldsymbol{x}, 0) = \sum_{l=1}^{N(k)} \boldsymbol{w}_l^K(0)\phi_l^K(\boldsymbol{x}). \tag{13.99}$$

If the initial conditions at hand are polynomials with degree $\leq k$, this representation preserves the exact initial conditions, otherwise equation (13.99) is an approximation to the given initial conditions. Otherwise, optimal initial weights can be obtained by performing an L^2-projection,

$$\min_{\left\{w_{l,i}^K(0)\right\}_l} \int_K \left(u_i^K(\boldsymbol{x}, 0) - u_i(\boldsymbol{x}, 0)\right)^2 \,\mathrm{d}V, \tag{13.100}$$

where $i = 1, \ldots, 5$ enumerates the conserved variables. The projection above leads to the integral

$$\boldsymbol{w}_j^K(0) = \frac{1}{|K|}\int_K \boldsymbol{u}(\boldsymbol{x}, 0)\phi_j^K(\boldsymbol{x})\,\mathrm{d}V, \quad j = 1, \ldots, N(k), \tag{13.101}$$

which can be transformed to the reference frame of the cell, viz.

$$\boldsymbol{w}_j^K(0) = \frac{1}{8}\int_{[-1,1]^3} \boldsymbol{u}(\boldsymbol{\xi}, 0)\phi_j(\boldsymbol{\xi})\,\mathrm{d}\boldsymbol{\xi}, \quad j = 1, \ldots, N(k). \tag{13.102}$$

This integral can be computed numerically by means of tensor product Gauss-Legendre quadrature (hereafter called Gaussian quadrature) with $(k + 1)^3$ nodes:

$$\boldsymbol{w}_j^K(0) \approx \frac{1}{8}\sum_{q=1}^{(k+1)^3} \boldsymbol{u}(\boldsymbol{\xi}_q^{\mathrm{3D}}, 0)\phi_j(\boldsymbol{\xi}_q^{\mathrm{3D}})\omega_q^{\mathrm{3D}}, \quad j = 1, \ldots, N(k). \tag{13.103}$$

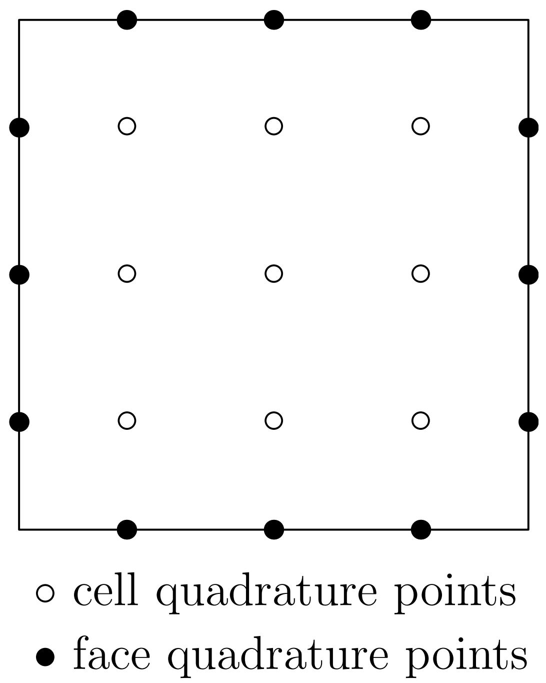


Figure 13.4: In the DG scheme, a surface and a volume integral has to be computed numerically for every cell, see equation (13.108). These integrals can be done by means of Gauss-Legendre quadrature. This example shows the nodes for a third order DG method (with second order polynomials) when used in a two-dimensional configuration. The black nodes indicate the positions where the surface integral is evaluated, which involves a numerical flux calculation with a Riemann solver. The white nodes are used for numerically estimating the volume integral.

Here, $\boldsymbol{\xi}_q^{\mathrm{3D}}$ is the position of the quadrature node q in the cell frame of reference, and ω_q^{3D} denotes the corresponding quadrature weight.

13.6.3 Gauss-Legendre quadrature

In the technique of Gaussian quadrature, the numerical integration of a function $f : [-1,+1] \to \mathbb{R}$ is approximated with a Gaussian quadrature rule involving n points and a weighted sum,

$$\int_{-1}^{+1} f(\xi)\,\mathrm{d}\xi \approx \sum_{q=1}^{n} f(\xi_q^{\mathrm{1D}})\omega_q^{\mathrm{1D}}. \tag{13.104}$$

Here, $\xi_q^{\mathrm{1D}} \in (-1,+1)$ are the Gaussian quadrature nodes and ω_q^{1D} are the corresponding weights. To integrate a 2D function $f : [-1,+1]^2 \to \mathbb{R}$ the tensor product of the n Gauss points can be used, viz.

$$\int_{-1}^{+1}\int_{-1}^{+1} f(\xi_1,\xi_2)\,\mathrm{d}\xi_1\mathrm{d}\xi_2$$
$$\approx \sum_{q=1}^{n}\sum_{r=1}^{n} f(\xi_{1,q}^{1\mathrm{D}},\xi_{2,r}^{1\mathrm{D}})\omega_q^{1\mathrm{D}}\omega_r^{1\mathrm{D}} = \sum_{q=1}^{n^2} f(\boldsymbol{\xi}_q^{2\mathrm{D}})\omega_q^{2\mathrm{D}}. \tag{13.105}$$

The n-point Gaussian quadrature rule is exact for polynomials of degree up to $2n-1$ when the one-dimensional nodes are given as the roots of the Legendre polynomial $P_n(\xi)$. These roots need to be found numerically, e.g. by means of the Newton-Raphson method, and can then be tabulated. The corresponding weights are then calculated as

$$\omega_q = \frac{2}{(1-\xi_q^2)P_n'(\xi_q)^2}, \quad q=1,\ldots,n. \tag{13.106}$$

13.6.4 Evolution equation for the weights

In order to derive the DG scheme on a cell K, the Euler equations for a polynomial state vector $\boldsymbol{u}^K$ are multiplied by the basis function ϕ_j^K and integrated over the cell volume,

$$\int_K \left[\frac{\partial \boldsymbol{u}^K}{\partial t} + \sum_{\alpha=1}^{3}\frac{\partial \boldsymbol{f}_\alpha}{\partial x_\alpha}\right]\phi_j^K\,\mathrm{d}V = 0, \tag{13.107}$$

yielding a weak formulation of the PDE. Integration by parts of the flux divergence term and a subsequent application of Gauss' theorem leads to

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_K \boldsymbol{u}^K\phi_j^K\,\mathrm{d}V - \sum_{\alpha=1}^{3}\int_K \boldsymbol{f}_\alpha\frac{\partial\phi_j^K}{\partial x_\alpha}\,\mathrm{d}V + \sum_{\alpha=1}^{3}\int_{\partial K}\boldsymbol{f}_\alpha n_\alpha\phi_j^K\,\mathrm{d}S = 0, \tag{13.108}$$

where $\hat{n} = (n_1,n_2,n_3)^T$ denotes the outward pointing unit normal vector of the surface ∂K. In the following, we discuss each of the terms separately.

According to equation (13.90) the first term is simply the time variation of the weights,

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_K \boldsymbol{u}^K\phi_j^K\,\mathrm{d}V = |K|\frac{\mathrm{d}\boldsymbol{w}_j^K}{\mathrm{d}t}. \tag{13.109}$$

The second and third terms are discretized by transforming the integrals to the cell frame and applying Gaussian quadrature (Fig. 13.4). With this procedure the second term becomes

$$\begin{aligned}&\textstyle\sum_{\alpha=1}^{3}\int_K \boldsymbol{f}_\alpha\left(\boldsymbol{u}^K(\boldsymbol{x},t)\right)\frac{\partial\phi_j^K(\boldsymbol{x})}{\partial x_\alpha}\,\mathrm{d}V\\ &= \textstyle\frac{(\Delta x^K)^2}{4}\sum_{\alpha=1}^{3}\int_{[-1,1]^3}\boldsymbol{f}_\alpha\left(\boldsymbol{u}^K(\boldsymbol{\xi},t)\right)\frac{\partial\phi_j(\boldsymbol{\xi})}{\partial\xi_\alpha}\,\mathrm{d}\boldsymbol{\xi}\\ &\approx \textstyle\frac{(\Delta x^K)^2}{4}\sum_{\alpha=1}^{3}\sum_{q=1}^{(k+1)^3}\boldsymbol{f}_\alpha\left(\boldsymbol{u}^K(\boldsymbol{\xi}_q^{3\mathrm{D}},t)\right)\left.\frac{\partial\phi_j(\boldsymbol{\xi})}{\partial\xi_\alpha}\right|_{\boldsymbol{\xi}_q^{3\mathrm{D}}}\omega_q^{3\mathrm{D}}.\end{aligned} \tag{13.110}$$

Note that the transformation of the derivative $\partial/\partial x_\alpha$ gives a factor of 2, see equation (13.92). The volume integral is computed by Gaussian quadrature with $k+1$ nodes per dimension. These nodes allow the exact integration of polynomials up to degree $2k+1$.

While the flux functions $\boldsymbol{f}_\alpha$ in the above expression can be evaluated analytically, this is not the case for the fluxes in the last term of the evolution equation (13.108). This is because the solution is *discontinuous* across cell interfaces. We hence have to introduce a numerical flux function $\bar{\boldsymbol{f}}\left(\boldsymbol{u}^{K-},\boldsymbol{u}^{K+},\hat{n}\right)$, which in general depends on both states left and right of the interface and on the normal vector. With this numerical flux, the third term in equation (13.108) takes the form

$$\begin{aligned}
&\textstyle\sum_{\alpha=1}^{3}\int_{\partial K}\boldsymbol{f}_\alpha n_\alpha(\boldsymbol{x})\phi_j^K(\boldsymbol{x})\,\mathrm{d}S \\
&= \tfrac{(\Delta x^K)^2}{4}\textstyle\int_{\partial[-1,1]^3}\bar{\boldsymbol{f}}\left(\boldsymbol{u}^{K-}(\boldsymbol{\xi},t),\boldsymbol{u}^{K+}(\boldsymbol{\xi},t),\hat{n}(\boldsymbol{\xi})\right)\phi_j(\boldsymbol{\xi})\,\mathrm{d}S_{\boldsymbol{\xi}} \\
&\approx \tfrac{(\Delta x^K)^2}{4}\textstyle\sum_{A\in\partial[-1,1]^3}\sum_{q=1}^{(k+1)^2}\bar{\boldsymbol{f}}\left(\boldsymbol{u}^{K-}(\boldsymbol{\xi}_{q,A}^{\mathrm{2D}},t),\boldsymbol{u}^{K+}(\boldsymbol{\xi}_{q,A}^{\mathrm{2D}},t),\hat{n}\right)\phi_j(\boldsymbol{\xi}_{q,A}^{\mathrm{2D}})\omega_q^{\mathrm{2D}}.
\end{aligned} \tag{13.111}$$

Here for each interface of the normalized cell a two-dimensional Gaussian quadrature rule with $(k+1)^2$ nodes is applied. The numerical flux across each node can be calculated with a one-dimensional Riemann solver, as in ordinary Godunov schemes.

We have now discussed each term of the basic equation (13.108) and arrived at a spatial discretization of the Euler equations of the form

$$\frac{\mathrm{d}\boldsymbol{w}_j^K}{\mathrm{d}t}+\boldsymbol{R}_K=0,\quad j=1,\ldots,N(k), \tag{13.112}$$

which represents a system of coupled ordinary differential equations. What is left to do is to intergrate them forward in time with a suitable integration scheme. Normally Runge-Kutta schemes are used for this, which can also be systematically extended to higher order so that the order of the time integration matches that of the spatial discretization. Additionally, problems involving strong shock waves usually require so-called limiting schemes that damp oscillatory parts of the oscillations should they be excited by true discontinuities in the physical solution. The limiting may involve reducing or setting to zero the higher order expansion coefficients in elements directly adjacent to a detected discontinuity.

13.6.5 Efficiency of DG schemes

In the above, the order p of the scheme can be chosen relatively freely, unlike in common finite volume approaches. If one has the goal to compute a more accurate numerical solution for a given problem, one has in principle two options. One either chooses a finer grid with a smaller spacing h (so called h-refinement) or one increases the order of the scheme (so called p-refinement). Note that in both cases, the number of degrees of freedom (which are the total number of weight coefficients that one evolves in time) goes up. An interesting question is then whether it is better to make the grid finer, or to go to higher order, where “better” typically means lower computational cost, i.e. less CPU time.

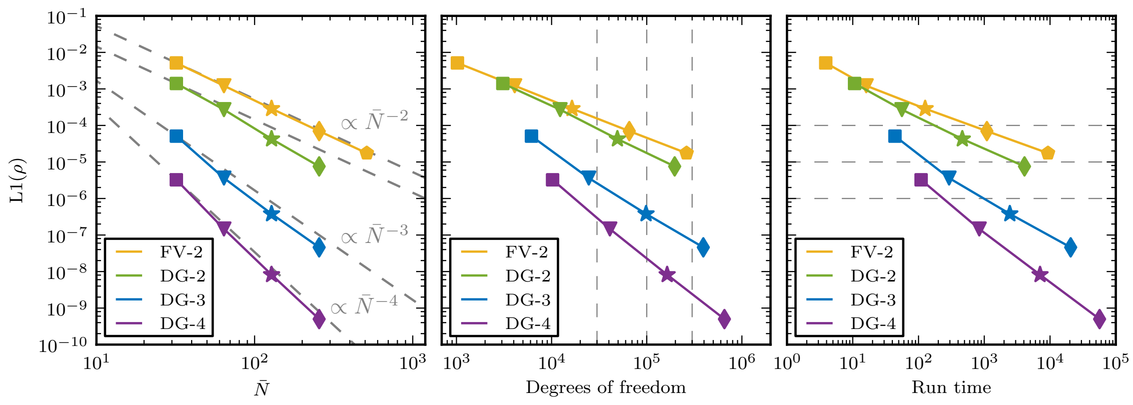


Figure 13.5: Performance of DG for the isentropic vortex flow (**?**). *Left panel:* L1 error norm as a function of linear resolution for a two-dimensional isentropic vortex test. Each data point corresponds to a simulation, different colours indicate the different methods. The convergence rate is as expected (dashed line) or slightly better for all schemes. *Middle panel:* The same simulation errors as a function of degrees of freedom (DOF), which is an indicator for the memory requirements. *Right panel:* L1 error norm versus the measured run time of the simulations. The second order FV implementation (FV-2) and the second order DG (DG-2) realization are approximately equally efficient in this test, i.e. a given precision can be obtained with a similar computational cost. In comparison, the higher order methods can easily be faster by more than an order of magnitude for this smooth problem. This illustrates the fact that an increase of order (p-refinement) of the numerical scheme can be remarkably more efficient than a simple increase of grid resolution (h-refinement).

A general answer to this question cannot be given as it is problem-dependent. However, one can systematically study the situation for a well defined problem. One such target problem is a stationary, isentropic vortex flow, the so-called Yee vortex. Here the analytic solution is known (which is identical to the initial conditions due to the stationarity), so that one can compute an error norm for a numerical solution by calculating the integral

$$\mathrm{L1} = \frac{1}{V}\int_V |\rho'(x,y) - \rho^0(x,y)|\,\mathrm{d}V,. \tag{13.113}$$

with the density solution polynomial $\rho'(x,y)$ and the analytic expression $\rho^0(x,y)$ of the initial conditions.

In Figure 13.5 results are showns where this problem was simulated in 2D for some time for 2nd, 3rd, and 4th order DG, and compared to solutions obtained with a 2nd order finite volume approach. The error norm is then shown as a function of the number of elements used per dimension, demonstrating the different convergence order of the schemes. Also shown are is the error as a function of the number of degrees used, and finally as a function of the CPU time. It is clearly seen that in this case the 4th order DG scheme wins, i.e. it provides for a given investment of CPU time the highest accuracy.

13.7 Literature

- Carlos A. Felippa "Introduction to Finite Element Methods". `http://kis.tu.kielce.pl/mo/COLORADO_FEM/colorado/IFEM.Ch00.pdf` (and later chapters, just replace 00 with 01 etc).

14 Parallelization techniques

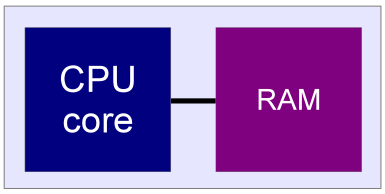
Modern computer architectures offer ever more computational power that we ideally would like to use to their full extent for scientific purposes, in particular for simulations in physics. However, unlike in the past, the speed of individual compute cores, which we may view as serial computers, has recently hardly grown any more (in stark contrast to the evolution a few years back). Instead, the number of cores on large supercomputers has started to increase exponentially. Even on laptops and cell-phones, multi-core computers have become the norm rather than the exception.

However, most algorithms and computer languages are constructed around the concepts of a serial computer, in which a stream of operations is executed sequentially. This is also how we typically think when we write computer code. In order to exploit the power available in modern computers, one needs to change this approach and adopt parallel computing techniques. Due to the large variety of computer hardware, and the many different technical concepts for devising parallel programs, we can only scratch the surface here and make a few basic remarks about parallelization, and some basic techniques that are currently in wide use for it. The interested student is encouraged to read more about this in books and/or in online resources.

14.1 Hardware overview

Let's start first with a schematic overview over some of the main characteristics of current computer architectures.

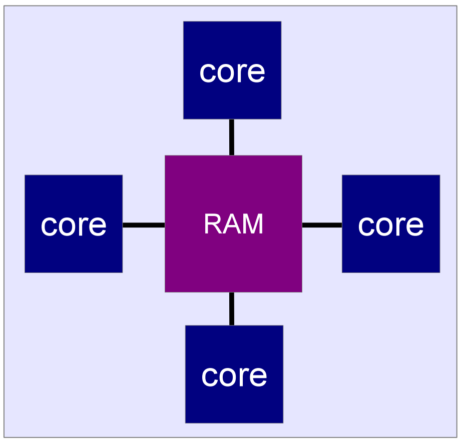
14.1.1 Serial computer



The traditional model of a computer consists of a central processing unit (CPU) capable of executing a sequential stream of load, store, and compute operations, with the data stored in a random access memory (RAM). Branches and jumps in this stream are possible too, but at any given time, only one operation is carried out. The operating system may still provide the illusion that several programs are executed concurrently, but in this case this is reached by time slicing the single compute resource.

Most computer languages are built around this model; they can be viewed as a means to create the stream of serial operations in a convenient way. One can in principle also by-pass the computer language and write down the machine instructions directly (assembler programming), but fortunately, modern compilers make this unnecessary in scientific applications (except perhaps in very special circumstances where extreme performance tuning is desired).

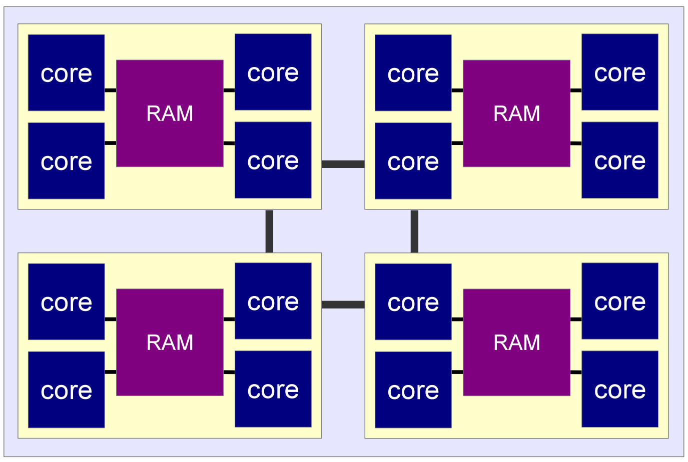
14.1.2 Multi-core nodes



It is possible to attach multiple CPUs to the same RAM, and, especially in recent times, computer vendors have started to add multiple cores to individual CPUs. On each CPU and each core of a CPU, different programs can be executed concurrently, allowing real parallel computations. In machines with uniform memory access, the individual cores can access the memory with the same speed, at least in principle. In this case the distinction between a CPU and a core can become confusing (and is in fact superfluos at some level), because it is ambiguous whether "CPU" refers to a single core, or all the cores on the same die of silicon (recommendation: it's usually best to simple speak about cores to avoid any confusion).

14.1.3 Multi-socket compute nodes

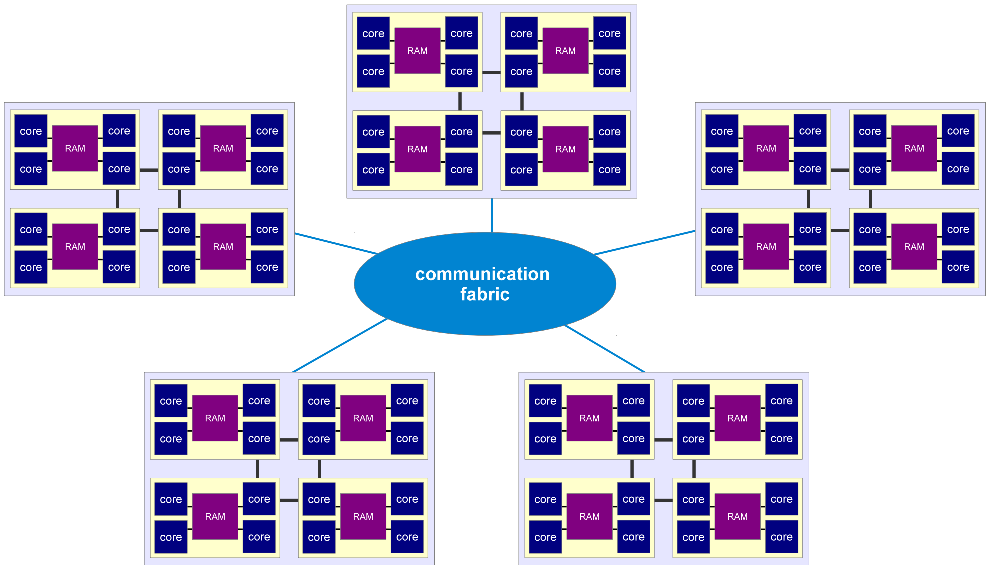
Most powerful compute servers feature a so-called NUMA (non-uniform memory access) architecture these days. Here the full main memory is accessible by all cores, but not all parts of it with the same speed. The compute nodes usually feature individual multi-core CPUs, each with a dedicated memory bank attached. Read and write operations to this part of physical memory are fastest, while accessing the other memory banks is typically noticeably slower and often involves going through special, high-bandwidth interprocessor bus systems.



In such machines, maximum compute performance is only reached when the data that is worked on by a core resides on the "right" memory bank. Fortunately, the operating system will normally try to help with this by satisfying memory requests out of the closest part of physical memory, if possible.

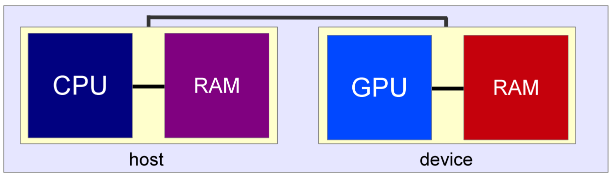
14.1.4 Compute clusters

Very powerful supercomputers used in the field of high-performance computing (HPC) can be formed by connecting many compute nodes through a fast communication network. This can be standard gigabit ethernet in some cases, but usually much faster (and more expensive) communication networks such as infiniband are employed. The leading supercomputers in the world are of this type, and currently reach several 10^5 cores in total.



14.1.5 Device computing

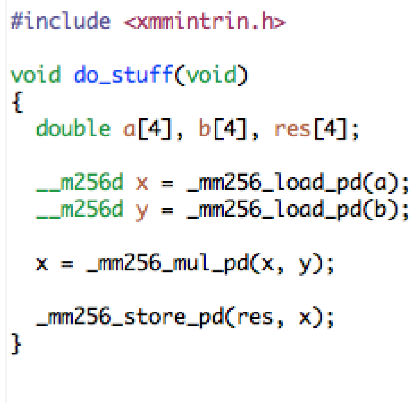
A comparatively new trend is to augment classical compute nodes with special accelerator cards that are particularly tuned to floating point calculations. These cards have much simpler, less flexible compute cores, but the transistors saved on implementing chip complexity can be spent on building more powerful compute engines that can execute many floating point operations in parallel. Graphics processing units (GPUs) have been originally developed with such a design just for the vector operations necessary to render graphics, but now their streaming processors can also be used for general purpose calculations. For certain applications, GPUs can be much faster than ordinary CPUs, but programming them is harder.



In so-called hybrid compute nodes, one has one or several ordinary CPUs coupled to one or several GPUs, or other accelerator cards such as the new Intel Phi. Of course, these hybrid nodes can be clustered again with a fabric to form powerful supercomputers. In fact, the fastest machines in the world are presently of this type.

14.1.6 Vector cores

Another hardware aspect that should not be overlooked is that single compute cores are actually increasingly capable to carry out so-called vector instructions. Here a single instruction (such as addition, multiplication, etc.) is applied to multiple data elements (a vector). This is also a form of parallelization, allowing the calculation throughput to be raised significantly. Below is an example that calculates $x = a \cdot b$ element by element for 4-vectors a and b. This can be programmed explicitly through *intrinsics* in C, which are basically individual machine instructions hidden as macros or function calls within C. (Usually, one does not do this manually though, but rather hopes the compiler emits such instructions somehow automatically.)



```
#include <xmmintrin.h>

void do_stuff(void)
{
  double a[4], b[4], res[4];

  __m256d x = _mm256_load_pd(a);
  __m256d y = _mm256_load_pd(b);

  x = _mm256_mul_pd(x, y);

  _mm256_store_pd(res, x);
}
```

The newest generation of the x86 processors from Intel/AMD features so-called SSE/AVX instructions that operator on vectors of up to 256 bits. This means that 4 double-precision or 8 single precision operations can be executed with a single such instruction, roughly in the same speed that an ordinary double or single-precision operation takes. So if these instructions can be used in an optimum way, one achieves a speed-up by a factor of up to 4 or 8, respectively. On the Intel Phi chips, the vector length has already doubled again and is now 512 bits, hence allowing another factor of 2 in the performance. Likely, we will see even larger vector lengths in the near future.

14.1.7 Hyperthreading

A general problem in exploiting modern computer hardare to its full capacity is that accessing main memory is very much slower than doing a single floating point operation in a compute core (moving data also costs more energy than doing a floating point calculation, which is becoming an ever more important point too). As a result, a compute core typically spends a large fraction of its cycles waiting for data to arrive from memory.

The idea of hyperthreading as implemented in CPUs by Intel and IBM (Power architecture) is to use this wait time by letting the core do some useful work in the meantime. This is achieved by "overloading" the compute core with several execution streams. But instead of letting the operating system toggle between their execution, the hardware itself can switch very rapidly between these different "hyperthreads". Even though there is still a considerable overhead in changing the execution context from one thread to another, this strategy can still lead to a substantial net increase in the calculational throughput on the core. Effectively, to the operating system and user it appears as if there are more cores (so called virtual cores) than there really are physical cores. For example, the IBM CPU on the Bluegene/Q machine has 16 physical cores with 4-fold hyperthreading, yielding 64 virtual cores. One may then start 64 threads in the user application. Compared with just starting 16 threads, one will then not get four times the performance, but still perhaps 1.8 times the performance or so.

14.2 Ahmdahl's law

Before we discuss some elementary parallelization techniques, it is worthwhile to point out a fundamental limit to the parallel speed-up that may be reached for a given program. We define the speed up here as the ratio of the total execution time without parallelization (i.e. when the calculation is done in serial) to the total execution time obtained when the parallelization is enabled.

Suppose we have a program that we have successfully parallelized. In practice, this parallelization is never going to be fully perfect. Normally there are parts of the calculation that remain serial, either for algorithmic reasons, due to technical limitations, or we considered them unimportant enough that we have not bothered to parallelize those too. Let us call the *residual serial fraction* f_s, i.e. this is the fraction of the execution time spent in the corresponding code parts when the program is executed in ordinary serial mode.

Then Ahmdahl's law gives the maximum parallel speed up as

$$\text{maximum parallel speed up} = \frac{1}{f_s}. \tag{14.1}$$

This is simply because in the most optimistic case we can assume that our parallelization effort has been perfect, so that the time spent in the parallel parts approaches zero for a sufficiently large number of cores. The serial time remains unaffected by this, however, and does not shrink at all. The lesson is a bit sobering: Achieving large parallel speed-ups, say beyond a factor of 100 or so, also requires extremely tiny residual serial fractions. This is sometimes very hard to reach in practice, and for certain problems, it may even be impossible.

14.3 Shared memory parallelization with OpenMP

- Shared memory parallelization can be used to distribute a computational workload on the multiple available compute cores that have access to the same memory, which is where the data resides.
- UNIX processes are *isolated* from each other – they usually have their own protected memory, preventing simple joint work on the same memory space (data exchange requires the use of files, sockets, or special devices such as /dev/shm). But, a process may be split up into multiple execution paths, called *threads*. Such threads share all the resources of the parent process (memory, files, etc.), and they are the ideal vehicle for efficient shared memory parallelization.
- Threads can be created and destroyed manually, e.g. with the pthreads-libary of the POSIX standard. This is a bit cumbersome in practice. Here is an example how this can look in practice:

```
#include <pthread.h>

void do_stuff(void)
{
  pthread_attr_t attr;
  pthread_t mythread;
  int i, threadid = 1;

  pthread_attr_init(&attr);
  pthread_create(mythreads, &attr, evaluate, &threadid);

  for(i = 0; i < 100; i++)
    some_expensive_calculation(i);
}


void *evaluate(void *p)
{
  int i;

  for(i = 100; i < 200; i++)
    some_expensive_calculation(i);
}
```

- A simpler approach is to use the OpenMP standard, which is a language/compiler extension for C/C++ and Fortran. It allows the programmer to give simple hints to the compiler, instructing it which parts can be executed in parallel sections. OpenMP then automatically deals with the thread creation and destruction, and completely hides this nuisance from the programmer. As a result, it becomes possible to parallelize a code with minimal modifications, and the modified program can still be compiled and executed without OpenMP as a serial code. Here is how the example from above would like like in OpenMP.

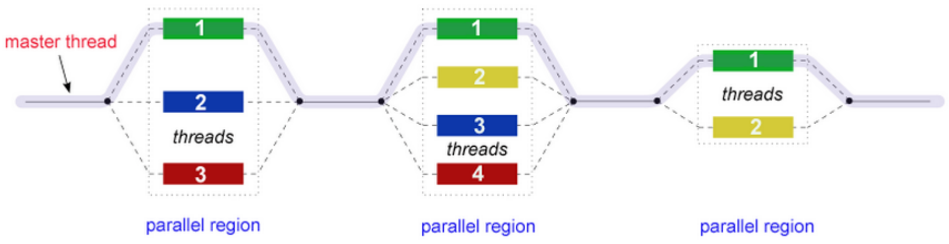
```
#include <omp.h>

void do_stuff(void)
{
  int i;

#pragma omp parallel for
  for(i = 0; i < 200; i++)
    some_expensive_calculation(i);
}
```

14.3.1 OpenMP's "fork-join-model"

The central idea of OpenMP is to let the programmer identify sections in a code that can be executed in parallel. Whenever such a section is encounter, the program execution is split into a number of threads that work in a team in parallel on the work of the section. Often, this work is a simple loop whose iterations are distributed evenly on the team, but also more general parallel sections are possible. At the end of the parallel section, the threads join again onto the master thread, the team is dissolved, and serial execution resumes until the next parallel section starts.



Normally, the number of threads used in each parallel section is constant, but this can also be changed through calls of the OpenMP runtime library functions, as in the above example.

14.3.2 Loop-level parallelism with OpenMP

The simplest and most important use of OpenMP is to parallelize a simple loop. Suppose for example we want to parallelize the loop:

```
for(i = 0; i < 200; i++)
  some_expensive_calculation(i);
```

This can be simply done by placing a special directive for the compiler in front of the loop:

```
#pragma omp parallel for
   for(i = 0; i < 200; i++)
     some_expensive_calculation(i);
```

That's basically all. The OpenMP compiler will then automatically wake up all available threads at the beginning of the loop (the "fork"), it will then distribute the loop iterations evenly onto the different threads, and they are then executed concurrently. Finally, once all loop iterations have completed, the threads are put to sleep again, and only the master thread continues in serial fashion. Note that this will only work correctly if there are *no dependencies* between the different loop iterations, or in other words, the order in which they are carried out needs to be unimportant. If everything goes well, the loop is then executed faster by a factor close to the number of threads.

In order for this to work in practice, one has to do a few additional things:

- The code has to be compiled with an OpenMP capable compiler. This feature often needs to be enabled with a special switch, e.g. with gcc,

  ```
  gcc -fopenmp ...
  ```

  needs to be used.

- For some more advanced OpenMP features accessible through calls of OpenMP-library functions, one should include the OpenMP header file

  ```
  #include <omp.h>
  ```

- In order to set the number of threads that are used, one should set the `OMP_NUM_THREADS` environment variable before the program is started. Depending on the shell that is used (here bash is assumed), this can be done for example through

  ```
  export OMP_NUM_THREADS=8
  ```

  in which case 8 threads would be allocated by OpenMP. Normally one should then also have at least eight (virtual) cores available. The `omp_get_num_threads()` function call can be used inside a program to check how many threads are available.

14.3.3 Race conditions

When OpenMP is used, one can easily create hideous bugs if different threads may modify the same variable at the same time. Which thread wins the "race" and gets to modify a variable first is essentially undetermined in OpenMP (because the exact timings on a compute core will vary stochastically due to "timing noise" originating in interruptions from the operating system), so that subsequent executions will seemingly not produce deterministic behaviour.

Example: Double loop

A simple example is the following double-loop:

```
#pragma omp parallel for
   for(i = 0; i < N; i++)
     {
       for(j = 0; j < N; j++)
         {
            .
            .
            do_stuff();
            .
            .
         }
      }
```

Here the simple OpenMP directive in the outer loop will instruct the `i`-loop to be split up. However, there is only one variable for `j`, *shared* by all the threads. They are hence not able to carry out the inner loop independent from each other! What is needed here is that each thread gets its own copy of `j`, so that the inner loop can be executed independently. This can be simply achieved by adding a `private` clause to the OpenMP directive of the outer loop, like this

```
#pragma omp parallel for private(j)
   for(i = 0; i < N; i++)
     {
       for(j = 0; j < N; j++)
         {
            .
            .
            do_stuff();
            .
            .
         }
      }
```

Example: Reduction

Another common problem are reductions such as done in this example:

```
   int count = 0;
```

```
#pragma omp parallel for
   for(i = 0; i < N; i++)
     {
       for(j = 0; j < N; j++)
         {
            .
            .
            if(complicated_calculation(i) > 0)
              count++;
            .
            .
         }
      }
```

Here the loop is nicely parallelized by OpenMP, but we may nevertheless sometimes get an incorrect result for `count`. This is because the increment of this variable is not really carried out as a single instruction. It basically involves a read from memory, an addition of 1, and a write back. If now two threads happen to arrive at this statement at essentially the same time, they will both read `count`, increment it, and then write it back. But in this case the variable will end up being incremented only by one unit and not by two, because one of the threads is ignorant of the change of `count` by the other and overwrites it!

There are different solutions to this problem. One is to serialize the increment of `count` by putting a so-called lock around it. This can be done by enclosing it with a

```
   #pragma omp critical
     {
       count++;
     }
```

construct. But this can cost substantial performance: If one or several threads arrive at the statement at the same time, they have to wait and do the operation one after the other.

A better solution would be to have private variables for `count` for each thread, and only at the end of the parallel section add up the different copies to get the global sum. OpenMP can generate the required code automatically, all that is needed is to add the clause `reduction(+:count)` to the directive for parallelizing the loop:

```
   int count = 0;
```

```
#pragma omp parallel for reduction(+:count)
   for(i = 0; i < N; i++)
     {
       for(j = 0; j < N; j++)
         {
            .
            .
            if(complicated_calculation(i) > 0)
              count++;
            .
            .
         }
      }
```

A more detailed description of the OpenMP standard can for example be found in the very good online tutorial `https://computing.llnl.gov/tutorials/openMP`, or in various textbooks.

14.4 Distributed memory parallelization with MPI

To use multiple nodes in compute clusters, OpenMP is not sufficient. Here one either has to use special languages (like UPC, Co-Array Fortran, etc.) which are popular among theoretical computer scientists (but hardly anyone else), or one turns to the "Message Passing Interface" (MPI). MPI has become the de-facto standard for programming large-scale simulation code.

MPI offers library functions for exchanging messages between different processes running on the same or different compute nodes. The compute nodes do not necessarily have to be physically close, in principle they can also be loosely connected over the internet (although for tightly coupled problems the message latency makes this unattractive). Most of the time, the same program is executed on all compute cores (SPMD, "single program multiple data"), but they operate on different data such that the computational task is put onto many shoulders and a parallel speed up is achieved. Since the MPI processes are isolated from each other, all data exchanges necessary for the computations have to be explicitly programmed – this makes this approach much harder than, e.g., OpenMP. Often substantial program modifications and algorithmic changes are needed for MPI.

Once a program has been parallelized with MPI, it may also be augmented with OpenMP. Such hybrid parallel code may then be executed in different ways on a cluster. For example, if one has two compute nodes with 8 cores each, one could run the program with 16 MPI tasks, or with 2 MPI tasks that each using 8 OpenMP threads, or with 4

MPI tasks and 4 OpenMP threads each. It would not make sense to use 1 MPI task and 16 OpenMP threads, however – then only one of the two compute nodes could be used.

14.4.1 General structure of an MPI program

A template of a simple MPI program in C would look as follows:

```
#include <mpi.h>

int main(int argc, char **argv)
{
   MPI_Init(&argc, &argv);
     .
     .
   /* now we can send/receive message to other MPI ranks */
     .
     .
   MPI_Finalize();
}
```

- To compile this program, one would normally use a compiler wrapper, for example `mpicc` instead of `cc`, which simply sets a pathname correctly such that the MPI header files and MPI library files are found by the compiler.
- For executing the MPI program, one would normally use a start-up program such as `mpirun` or `mpiexec`. For example

  ```
  mpirun -np 8 ./mycalc
  ```

  could be used to launch 8 instances of the program `mycalc`.

If a normal serial program is augmented by `MPI_Init` in the above fashion, and if it is started multiple times with `mpirun -np X`, it will simply do multiple times exactly the same thing as the corresponding serial program. To change this behavior and achieve non-trivial parallelism, the execution paths in each copy of the program need to be somewhat different. This is normally achieved by making it explicitly depend on the *rank* of the MPI task. All the N processes of an MPI program form a so-called communicator, and they are labelled with a unique rank-id $0, 1, 2, \ldots, N-1$. MPI processes can then send and receive message from different ranks using these IDs to identify each other.

The first thing an MPI program normally does is therefore to find out how many MPI processes there are in the “world”, and what the rank of the program itself is. This is done with the function calls

```
  int NTask, ThisTask;

  MPI_Comm_size(MPI_COMM_WORLD, &NTask);
  MPI_Comm_rank(MPI_COMM_WORLD, &ThisTask);
```

The returned integers `NTask` and `ThisTask` then contain the number of MPI tasks and the rank of the current one, respectively.

14.4.2 A simple point to point message

With this information in hand, we can then exchange simple messages between two different MPI ranks. For example, a send of a message from rank 5 to rank 7 could be programmed like this:

```
int data[50], result[50]
.
.
if(ThisTask == 5)
  MPI_Send(data, 50, MPI_INT, 7,     /* buffer, size, type, destination */
           12345, MPI_COMM_WORLD);  /* message tag, communicator id    */

if(ThisTask == 7)
  MPI_Recv(result, 50, MPI_INT, 5,    /* receive buffer, size, type, send
           12345, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  /* tag, comm, stat
.
.
```

Here one sees the general structure of most send/recv calls, which always decompose a message into an "envelope" and the "data". The envelope describes the rank-id of sender/receiver, the size and type of the message, and a user-specified tag (this is the '12345' here), which can be used to distinguish messages of the same length.

Through the if-statements that depend on the local MPI rank, different execution paths for sender and receiver are achieved in this example. Note that if something goes wrong here, for example an MPI rank posts a receive but the matching send does not occur, it is simple to create deadlocks in a program. Here one or several of the MPI tasks get stuck in waiting in vain for messages that are not sent.

It is also possible to make MPI communications non-blocking and achieve asynchronous communication in this way. The MPI-2 standard even contains some calls for one-sided communication operations that do not always require direct involvement of both the sending and receiving sides.

14.4.3 Collective communications

The MPI standard knowns a large number of functions that can be used to conveniently make use of commonly encountered communication patterns. For example, there are calls for *broadcasts* which send the same data to all other MPI tasks in the same communicator. There are also *gather* and *scatter* operations that collect data elements from all tasks, or distribute them as disjoint sets to the other tasks. Finally, there are *reduction* function that allow one to conveniently calculate sums, minima, maxima, etc., over variables held by all MPI tasks in a communicator.

A detailed description of all these possibilities is way passed the scope of these brief lecture notes. Please check out some of the online resources (for example `https://computing.ll` or a text book is you want more information about this.

References

Bauer, A., Springel, V. (2012), *Subsonic turbulence in smoothed particle hydrodynamics and moving-mesh simulations*, MNRAS, 423, 2558

Eckart, C. (1960), *Variation Principles of Hydrodynamics*, Physics of Fluids, 3, 421

Gingold, R. A., Monaghan, J. J. (1977), *Smoothed particle hydrodynamics - Theory and application to non-spherical stars*, MNRAS, 181, 375

Lucy, L. B. (1977), *A numerical approach to the testing of the fission hypothesis*, AJ, 82, 1013

Monaghan, J. J. (1992), *Smoothed particle hydrodynamics*, ARA&A, 30, 543

Price, D. J. (2012), *Smoothed Particle Hydrodynamics: Things I Wish My Mother Taught Me*, in *Advances in Computational Astrophysics: Methods, Tools, and Outcome*, edited by R. Capuzzo-Dolcetta, M. Limongi, A. Tornambè, volume 453 of *Astronomical Society of the Pacific Conference Series*, 249

Springel, V. (2010), *Smoothed Particle Hydrodynamics in Astrophysics*, ARA&A, 48, 391

Springel, V., Hernquist, L. (2002), *Cosmological smoothed particle hydrodynamics simulations: the entropy equation*, MNRAS, 333, 649