

DOUAR

theory, methods, and concepts

C. Thieulot, J. Braun, P. Fullsack

`cedric.thieulot@univ-rennes1.fr`

Univ. Rennes1

Univ. Dalhousie

who?



main developpers

Jean Braun Cedric Thieulot	Géosciences Rennes University of Rennes France
Chris Beaumont Philippe Fullsack	Dalhousie Geodynamics Group Oceanography Department Halifax
Martijn de Kool	Australian National University Australia



contributors

Ritske Huisman	Department of Earth Sciences Bergen University Norway
Likke Geimar	leeds
Frederic Hermann	Caltech

Collaboration supported by the Canadian Institute for Advanced Studies (CIAR).

The physics

Incompressible fluid flow (1)

What follows comes from

Finite Elements and Fast Iterative Solvers

with applications in
incompressible fluid dynamics

**HOWARD ELMAN
DAVID SILVESTER
ANDY WATHEN**

and is intended to be a reminder.

Incompressible fluid flow (2)

0

MODELS OF INCOMPRESSIBLE FLUID FLOW

To set the scene for subsequent chapters, the basic PDE models that are the focus of the book are derived in this chapter. Our presentation is rudimentary. Readers who would like to learn more about the physics underlying fluid flow modelling are advised to consult the books of Acheson [1] or Batchelor [12]. The fundamental principles are conservation of mass and conservation of momentum, or rather, that forces effect a change in momentum as described by Newton's famous Second Law of Motion.

Consider a fluid of density ρ moving in a region of three-dimensional space Ω . Suppose a particular small volume of fluid (imagine a dyed particle or bubble moving in the fluid) is at the position \vec{x} with respect to some fixed coordinates at time t . If in a small interval of time δt this particle moves to position $\vec{x} + \delta\vec{x}$, then the velocity at position \vec{x} and time t is

$$\vec{u} = (u_x, u_y, u_z) := \lim_{\delta t \rightarrow 0} \frac{\delta\vec{x}}{\delta t}.$$

Each of the velocity components u_x , u_y and u_z is a function of the coordinates x , y and z as well as the time t . Inside any particular fixed closed surface ∂D enclosing a volume $D \subset \Omega$, the total mass of fluid is $\int_D \rho d\Omega$, where $d\Omega = dx dy dz$ is the increment of volume. The amount of fluid flowing out of D across ∂D is

$$\int_{\partial D} \rho \vec{u} \cdot \vec{n} dS,$$

where \vec{n} is the unit normal vector to ∂D pointing outwards from D and dS is the increment of surface area. Therefore, since mass is conserved,

the rate of change of mass in D equals
the amount of fluid flowing into D across ∂D .

We express this mathematically as

$$\frac{d}{dt} \int_D \rho d\Omega = - \int_{\partial D} \rho \vec{u} \cdot \vec{n} dS.$$

Employing the Divergence theorem, which says that for a smooth enough vector field \vec{v} and any region R with smooth enough boundary ∂R ,

$$\int_{\partial R} \vec{v} \cdot \vec{n} dS = \int_R \nabla \cdot \vec{v} d\Omega,$$

1

Incompressible fluid flow (3)

and noting that D is fixed, yields

$$\frac{d}{dt} \int_D \rho \, d\Omega + \int_{\partial D} \rho \vec{u} \cdot \vec{n} \, dS = \int_D \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \vec{u}) \, d\Omega = 0,$$

where we have swapped the order of differentiation (with respect to time) and integration (with respect to space) in the first term. Since D is any volume, it follows that

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \vec{u}) = 0 \quad \text{in } \Omega.$$

For an incompressible and homogeneous fluid the density is constant both with respect to time and the spatial coordinates. Hence

$$\nabla \cdot \vec{u} := \frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z} = 0 \quad \text{in } \Omega.$$

In order to express the conservation of momentum, we need to define the acceleration of the fluid. To this end, suppose that the velocity of the small (dyed) volume of fluid at time t is \vec{u} as above and that at $t + \delta t$ it is $\vec{u} + \delta \vec{u}$. Since the velocity depends on both position and time, we explicitly write $\vec{u} = \vec{q}(\vec{x}, t)$ so that

$$\vec{u} + \delta \vec{u} = \vec{q}(\vec{x} + \delta \vec{x}, t + \delta t)$$

or, rearranging slightly,

$$\begin{aligned} \delta \vec{u} &= \vec{q}(\vec{x} + \delta \vec{x}, t + \delta t) - \vec{q}(\vec{x}, t) \\ &= \vec{q}(\vec{x} + \delta \vec{x}, t + \delta t) - \vec{q}(\vec{x}, t + \delta t) + \vec{q}(\vec{x}, t + \delta t) - \vec{q}(\vec{x}, t). \end{aligned}$$

Then, using Taylor series we get

$$\vec{q}(\vec{x} + \delta \vec{x}, t + \delta t) - \vec{q}(\vec{x}, t + \delta t) = (\delta \vec{x} \cdot \nabla) \vec{q}(\vec{x}, t + \delta t) + \mathcal{O}(\|\delta \vec{x}\|^2)$$

and

$$\vec{q}(\vec{x}, t + \delta t) - \vec{q}(\vec{x}, t) = \delta t \frac{\partial}{\partial t} \vec{q}(\vec{x}, t) + \mathcal{O}(\delta t^2).$$

Dividing by δt , taking the limit as $\delta t \rightarrow 0$ and using the definition of $\vec{u} := \delta \vec{x} / \delta t$, we see that the acceleration is given by

$$\frac{d\vec{u}}{dt} = \lim_{\delta t \rightarrow 0} \frac{\delta \vec{u}}{\delta t} = \frac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla) \vec{u}.$$

This derivative, the so-called *convective derivative*,

$$\frac{d(\cdot)}{dt} := \frac{\partial(\cdot)}{\partial t} + (\vec{u} \cdot \nabla)(\cdot),$$

expresses the rate of change of either a scalar quantity (or of each scalar component of a vector quantity) that is "following the fluid". Thus to summarize,

the fluid acceleration is the convective derivative of the velocity. Note that the acceleration is nonlinear in \vec{u} .

For the volume D , which is fixed in space and time, the rate of change of momentum is the product of the mass and the acceleration, or

$$\int_D \rho \left(\frac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla) \vec{u} \right) \, d\Omega.$$

An *ideal fluid* is an incompressible and homogeneous fluid that has no viscosity. For such a fluid, the only forces are due to the pressure, p , and any external *body force*, \vec{f} , such as gravity. Thus, the total force acting on the fluid contained in D are the pressure of the surrounding fluid, and the effect of the body force \vec{f} ,

$$\int_{\partial D} p(-\vec{n}) \, dS + \int_D \rho \vec{f} \, d\Omega.$$

Taking $\vec{v} = p\vec{c}$ in the Divergence theorem with \vec{c} being a constant vector pointing in an arbitrary direction gives

$$\int_D \nabla \cdot (p\vec{c}) \, d\Omega = \int_{\partial D} p\vec{c} \cdot \vec{n} \, dS.$$

However,

$$\nabla \cdot (p\vec{c}) = p \nabla \cdot \vec{c} + \vec{c} \cdot \nabla p = \vec{c} \cdot \nabla p$$

since \vec{c} is constant. Moreover, \vec{c} can be taken outside the integrals since it does not vary in D or on ∂D . This leads to

$$\vec{c} \cdot \left(\int_D \nabla p \, d\Omega - \int_{\partial D} p\vec{n} \, dS \right) = 0.$$

Since \vec{c} points in an arbitrary direction it then follows that

$$\int_D \nabla p \, d\Omega = \int_{\partial D} p\vec{n} \, dS.$$

Newton's Second Law of Motion applied to the fluid in D requires that,

the rate of change of momentum of fluid in D equals the sum of the external forces.

Expressed in mathematical terms, we have that

$$\int_D \rho \left(\frac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla) \vec{u} \right) \, d\Omega = \int_{\partial D} p(-\vec{n}) \, dS + \int_D \rho \vec{f} \, d\Omega.$$

This leads to

$$\int_D \rho \left(\frac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla) \vec{u} \right) + \nabla p - \rho \vec{f} \, d\Omega = 0.$$

Incompressible fluid flow (4)

Then, since D is an arbitrary volume, we obtain the *Euler Equations* for an ideal incompressible fluid,

$$\frac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla) \vec{u} = -\frac{1}{\rho} \nabla p + \vec{f} \quad \text{in } \Omega$$

$$\nabla \cdot \vec{u} = 0 \quad \text{in } \Omega.$$

Written in “longhand” and assuming that $\vec{f} = (g_x, g_y, g_z)^T$, the Euler system is as follows,

$$\frac{\partial u_x}{\partial t} + u_x \frac{\partial u_x}{\partial x} + u_y \frac{\partial u_x}{\partial y} + u_z \frac{\partial u_x}{\partial z} = -\frac{1}{\rho} \frac{\partial p}{\partial x} + g_x$$

$$\frac{\partial u_y}{\partial t} + u_x \frac{\partial u_y}{\partial x} + u_y \frac{\partial u_y}{\partial y} + u_z \frac{\partial u_y}{\partial z} = -\frac{1}{\rho} \frac{\partial p}{\partial y} + g_y$$

$$\frac{\partial u_z}{\partial t} + u_x \frac{\partial u_z}{\partial x} + u_y \frac{\partial u_z}{\partial y} + u_z \frac{\partial u_z}{\partial z} = -\frac{1}{\rho} \frac{\partial p}{\partial z} + g_z$$

$$\frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z} = 0.$$

To fully define a model of a physical problem, appropriate initial and boundary conditions need to be specified. For example, for flow inside a container one must have no flow across the boundary, hence one applies $\vec{u} \cdot \vec{n} = 0$ on $\partial\Omega$.

The Euler equations can be hugely simplified by assuming that the *vorticity*, $\vec{w} := \nabla \times \vec{u}$ is zero. This is called *irrotational flow*, and is a reasonable assumption in a number of important applications, for example, when considering the motion of small amplitude water waves. For irrotational flow, taking any fixed origin O the line integral

$$\phi(P) := \int_O^P \vec{u} \cdot d\vec{x}$$

uniquely defines the value of a *fluid potential* ϕ at the point P . This is a consequence of Stokes’ Theorem: if C is a closed curve in three-dimensional space and S is any surface that forms a “cap” on this curve (think of a soap bubble blown from any particular shape of plastic loop) then

$$\int_C \vec{u} \cdot d\vec{x} = \int_S \nabla \times \vec{u} \cdot \vec{n} \, dS.$$

In other words, the line integral defining $\phi(P)$ is independent of the actual path taken from O to P . Differentiation of this line integral in each coordinate direction then gives

$$\vec{u} = -\nabla\phi.$$

Notice that choosing a different origin O simply changes ϕ by a constant value in space. The value of this constant clearly does not alter \vec{u} even though there may

be a different constant for every time t . (The minus sign is our convention—other authors take a positive sign.) Combining the definition of the potential with the incompressibility condition $\nabla \cdot \vec{u} = 0$ yields *Laplace’s equation*

$$-\nabla^2\phi = 0 \quad \text{in } \Omega.$$

This, in turn, is a particular case of the *Poisson equation*

$$-\nabla^2\phi = f \quad \text{in } \Omega$$

that is the subject of Chapters 1 and 2.

For such *potential flow* problems, if the body forces are conservative so that $\vec{f} = -\nabla\Xi$ for some scalar potential Ξ , then the pressure can be recovered using a classical construction as follows. Substituting the vector identity

$$(\vec{u} \cdot \nabla) \vec{u} = \frac{1}{2} \nabla(\vec{u} \cdot \vec{u}) - \vec{u} \times (\nabla \times \vec{u})$$

into the Euler equations and enforcing the condition that $\nabla \times \vec{u} = 0$ gives

$$\frac{\partial \vec{u}}{\partial t} + \frac{1}{\rho} \nabla p_T = 0 \quad \text{in } \Omega,$$

where $p_T = \frac{1}{2} \rho \vec{u} \cdot \vec{u} + p + \rho\Xi$ is called the *total pressure*. If we rewrite this as

$$\nabla \left(-\frac{\partial \phi}{\partial t} + \frac{p_T}{\rho} \right) = 0 \quad \text{in } \Omega$$

and integrate with respect to the spatial variables, we see that

$$-\frac{\partial \phi}{\partial t} + \frac{p_T}{\rho} = h(t),$$

where h is an arbitrary function of time only. Since $h(t)$ is constant in space, one can consider the value of ϕ to be changed by this constant and so without any loss of generality can take $h(t) = 0$. Thus, to summarize, having computed the potential ϕ by solving Laplace’s equation together with appropriate boundary conditions, the velocity \vec{u} and pressure p can be explicitly computed via $\vec{u} = -\nabla\phi$, and

$$p = p_T - \frac{1}{2} \rho \vec{u} \cdot \vec{u} - \rho\Xi = \rho \left(\frac{\partial \phi}{\partial t} - \frac{1}{2} \vec{u} \cdot \vec{u} - \Xi \right),$$

respectively.

For a “real” viscous fluid, each small volume of fluid is not only acted on by pressure forces (*normal stresses*), but also by *tangential stresses* (also called *shear*

Incompressible fluid flow (5)

stresses). Thus, as in the inviscid case the normal stresses are due to pressure giving rise to a force on the volume D of fluid,

$$\int_{\partial D} p(-\vec{n}) dS,$$

which can be written using the *unit diagonal tensor*

$$\vec{\mathbf{I}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

as

$$\int_{\partial D} -p\vec{\mathbf{I}}\vec{n} dS.$$

(The tensor vector product is like a matrix vector product.) The shear stresses on the other hand can act in any direction at the different points on ∂D so that a full tensor

$$\vec{\mathbf{T}} = \begin{bmatrix} T_{xx} & T_{xy} & T_{xz} \\ T_{yx} & T_{yy} & T_{yz} \\ T_{zx} & T_{zy} & T_{zz} \end{bmatrix}$$

is needed, and the force due to the shear stresses is given by

$$\int_{\partial D} \vec{\mathbf{T}}\vec{n} dS.$$

Applying the Divergence theorem to each row of the shear stress tensor $\vec{\mathbf{T}}$, we arrive at the shear forces

$$\int_D \nabla \cdot \vec{\mathbf{T}} d\Omega,$$

where $\nabla \cdot \vec{\mathbf{T}}$ is the vector

$$\begin{bmatrix} \frac{\partial T_{xx}}{\partial x} + \frac{\partial T_{xy}}{\partial y} + \frac{\partial T_{xz}}{\partial z} \\ \frac{\partial T_{yx}}{\partial x} + \frac{\partial T_{yy}}{\partial y} + \frac{\partial T_{yz}}{\partial z} \\ \frac{\partial T_{zx}}{\partial x} + \frac{\partial T_{zy}}{\partial y} + \frac{\partial T_{zz}}{\partial z} \end{bmatrix}.$$

A *Newtonian fluid* is characterized by the fact that the shear stress tensor is a linear function of the *rate of strain tensor*,

$$\vec{\epsilon} := \frac{1}{2} [\nabla \vec{u} + (\nabla \vec{u})^T] = \frac{1}{2} \begin{bmatrix} 2\frac{\partial u_x}{\partial x} & \frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x} & \frac{\partial u_x}{\partial z} + \frac{\partial u_z}{\partial x} \\ \frac{\partial u_y}{\partial x} + \frac{\partial u_x}{\partial y} & 2\frac{\partial u_y}{\partial y} & \frac{\partial u_y}{\partial z} + \frac{\partial u_z}{\partial y} \\ \frac{\partial u_z}{\partial x} + \frac{\partial u_x}{\partial z} & \frac{\partial u_z}{\partial y} + \frac{\partial u_y}{\partial z} & 2\frac{\partial u_z}{\partial z} \end{bmatrix}.$$

In this book, only such Newtonian fluids are considered. These include most common fluids such as air and water (and hence blood, etc.). Any fluid satisfying a nonlinear stress-strain relationship is called a non-Newtonian fluid. For a list of possibilities, see the book by Joseph [116]. For a Newtonian fluid, we have that

$$\vec{\mathbf{T}} = \mu \vec{\epsilon} + \lambda \text{Trace}(\vec{\epsilon}) \vec{\mathbf{I}},$$

where μ and λ are parameters describing the “stickiness” of the fluid. For an incompressible fluid, the parameter λ is unimportant because $\text{Trace}(\vec{\epsilon}) = \nabla \cdot \vec{u} = 0$. However the *molecular viscosity*, μ , which is a fluid property measuring the resistance of the fluid to shearing, gives rise to the viscous shear forces

$$\mu \nabla \cdot \vec{\epsilon} = \mu \begin{bmatrix} \nabla^2 u_x + \left(\frac{\partial}{\partial x}\right)(\nabla \cdot \vec{u}) \\ \nabla^2 u_y + \left(\frac{\partial}{\partial y}\right)(\nabla \cdot \vec{u}) \\ \nabla^2 u_z + \left(\frac{\partial}{\partial z}\right)(\nabla \cdot \vec{u}) \end{bmatrix} = \mu \begin{bmatrix} \nabla^2 u_x \\ \nabla^2 u_y \\ \nabla^2 u_z \end{bmatrix} = \mu \nabla^2 \vec{u}.$$

Application of Newton's Second Law of Motion in the case of a Newtonian fluid then gives

$$\int_D \rho \left(\frac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla) \vec{u} \right) = \int_{\partial D} -p\vec{\mathbf{I}}\vec{n} dS + \int_D \mu \nabla^2 \vec{u} d\Omega + \int_D \rho \vec{f} d\Omega.$$

We can convert the pressure term into an integral over D exactly as is done in the case of an ideal fluid above. Then, using the fact that D is an arbitrary region in the flow leads to the following generalization of the Euler equations,

$$\frac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla) \vec{u} = -\frac{1}{\rho} \nabla p + \nu \nabla^2 \vec{u} + \vec{f} \quad \text{in } \Omega$$

$$\nabla \cdot \vec{u} = 0 \quad \text{in } \Omega,$$

where $\nu := \mu/\rho$ is called the *kinematic viscosity*.

Incompressible fluid flow (6)

The focus of this book is *steady* flow problems. In this case, the time-derivative term on the left-hand side is zero. Then, absorbing the constant density into the pressure ($p \leftarrow p/\rho$), leads to the “steady-state” *Navier-Stokes equations*,

$$\begin{aligned} -\nu \nabla^2 \vec{u} + \vec{u} \cdot \nabla \vec{u} + \nabla p &= \vec{f} \quad \text{in } \Omega \\ \nabla \cdot \vec{u} &= 0 \quad \text{in } \Omega, \end{aligned}$$

which when combined with boundary conditions forms one of the most general models of incompressible viscous fluid flow. This system is the focus of Chapters 7 and 8.

As observed above, the Navier-Stokes equations are nonlinear. However, simplification can be made in situations where the velocity is small or the flow is tightly confined. In such situations, a good approximation is achieved by dropping the quadratic (nonlinear) term from the Navier-Stokes equations and absorbing the constant ν into the velocity ($\vec{u} \leftarrow \nu \vec{u}$), giving the *Stokes equations*

$$\begin{aligned} -\nabla^2 \vec{u} + \nabla p &= \vec{f} \quad \text{in } \Omega \\ \nabla \cdot \vec{u} &= 0 \quad \text{in } \Omega. \end{aligned}$$

There is little loss of generality if \vec{f} is set to zero. A conservative body force (e.g. that due to gravity when the fluid is supported below) is the gradient of a scalar field, that is, $\vec{f} = -\nabla \Xi$, and thus it can be incorporated into the system by redefining the pressure ($p \leftarrow p + \Xi$). The numerical solution of Stokes equations in this form is described in Chapters 5 and 6.

Another linearization of the Navier-Stokes equations replaces the term $\vec{u} \cdot \nabla \vec{u}$ with $\vec{w} \cdot \nabla \vec{u}$ where \vec{w} is a known vector field (often called the *wind* in this context). If the pressure term is dropped from the momentum equation, the resulting *Convection-Diffusion equation*

$$-\nu \nabla^2 \vec{u} + \vec{w} \cdot \nabla \vec{u} = \vec{f},$$

is an important equation for various reasons. As well as describing many significant physical processes like the transport and diffusion of pollutants, it turns out to be very important to have efficient methods for this equation in order to get good numerical solution strategies for the Navier-Stokes equations. As given here, this is just three uncoupled scalar equations for the separate velocity

components and so is no more or less general than the equation for convection and diffusion of a scalar quantity u with a scalar forcing term f ,

$$-\nu \nabla^2 u + \vec{w} \cdot \nabla u = f \quad \text{in } \Omega.$$

Numerical methods for the solution of this equation together with suitable boundary conditions are described in Chapters 3 and 4.

The Stokes equation

- The Stokes equation system is a fundamental model of viscous flow. The variable \mathbf{v} is a vector-valued function representing the velocity of the fluid, and the scalar p represents the pressure.

$$\begin{aligned}\mu \Delta \mathbf{v} - \nabla p + \rho \mathbf{g} &= \mathbf{0} \\ \nabla \cdot \mathbf{v} &= 0\end{aligned}$$

- ρ is the density of the material
- \mathbf{g} is the gravitational acceleration
- μ is the shear viscosity

- assumptions

- highly viscous material deforming at a sufficiently low speed \rightarrow inertial forces can be neglected (low Reynolds number flow)
- heat is conducted faster than dissipated by viscous flow (low Prandtl number flow),
- fluid is assumed to be incompressible

Re and *Pr*

 *Re*: in fluid mechanics, the Reynolds number is the ratio of inertial forces to viscous forces and consequently it quantifies the relative importance of these two types of forces for given flow conditions. Thus, it is used to identify different flow regimes, such as laminar or turbulent flow.

It is one of the most important dimensionless numbers in fluid dynamics and is used, usually along with other dimensionless numbers, to provide a criterion for determining dynamic similitude. When two geometrically similar flow patterns, in perhaps different fluids with possibly different flowrates, have the same values for the relevant dimensionless numbers, they are said to be dynamically similar.

It is named after Osborne Reynolds (1842-1912), who proposed it in 1883. Typically it is given as follows:

$$Re = \frac{\rho v_s L}{\mu} = \frac{v_s L}{\nu} = \frac{\text{Inertial forces}}{\text{Viscous forces}}$$

where v_s is the mean fluid velocity, L a characteristic length, μ the dynamic fluid viscosity, ν the kinematic fluid viscosity $\nu = \mu/\rho$, and ρ is the fluid density.

 *Pr*: the Prandtl number is a dimensionless number approximating the ratio of momentum diffusivity (viscosity) and thermal diffusivity. It is named after Ludwig Prandtl.

It is defined as:

$$Pr = \frac{\nu}{\alpha} = \frac{\text{viscous diffusion rate}}{\text{thermal diffusion rate}}$$

where ν is the kinematic viscosity, $\nu = \mu/\rho$, and α is the thermal diffusivity, $\alpha = k/(\rho C_p)$.

Typical values for *Pr* are around 0.7 for air and many other gases, around 7 for water, and around 710^{21} for Earth's mantle.

The penalty method

- A slightly compressible material may be modelled by replacing the divergence-free constraint by the relation

$$p = -\lambda \nabla \cdot \mathbf{v}$$

where we have introduced a so-called penalty or compressibility factor λ :

- λ has the dimensions of a viscosity (*Pa.s*)
- λ is commonly taken to be several orders of magnitude larger than the shear viscosity μ

$$\lambda \gg \mu$$

- this ensures a nearly incompressible behaviour for the flow.

- Pressure can be eliminated from the Stokes equations and they become:

$$\mu \Delta \mathbf{v} + \lambda \nabla \nabla \cdot \mathbf{v} + \rho \mathbf{g} = \mathbf{0}$$

- This is called the *penalty method*. The idea was popularised by Hughes et al. in the classic paper *Finite element analysis of incompressible viscous flows by the penalty function formulation* in the Journal of Computational Physics 30, p1-60, 1979.

- The attraction of this approach is that the resulting linear algebra system is symmetric and positive definite. Discretisation of the penalty formulation is tricky however. Typically different quadrature rules must be applied to the μ and the λ terms. If the same quadrature is used for both terms, it implies instability in the incompressible limit.

The heat transport equation

Because rock material properties such as density or viscosity depend on temperature, it is also necessary to compute the temperature within the deforming system.

This is done by solving the energy or heat transport equation which has temperature T as an unknown:

$$\rho c \left(\frac{\partial T}{\partial t} + \underbrace{v \cdot \nabla T}_{adv} \right) = \underbrace{\nabla k \cdot \nabla T}_{conv} + \rho H$$

k is the thermal conductivity,

ρ is the density,

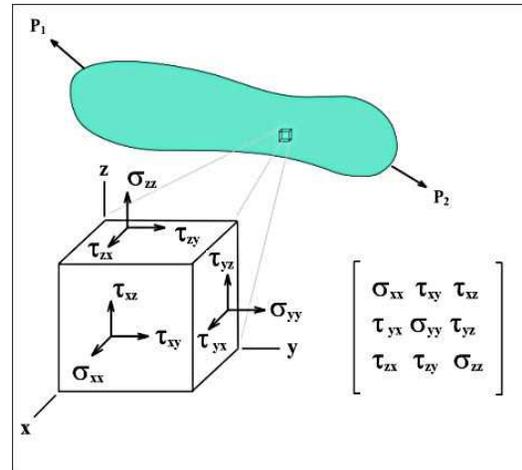
c is the heat capacity and

H is the heat production.

The relative importance of the advective term with respect to the conductive term is measured by the value of the dimensionless Peclet number, $Pe = v_0 L / \kappa$ where v_0 and L are typical velocity and length characterizing the system and $\kappa = k / \rho c$ is the heat diffusivity.

In most active tectonic systems, Pe is large, i.e. comprised between 1 and 100. Thus the advective term, cannot be neglected.

Stress tensor (1)



The state of stress for a three-dimensional point is defined by a matrix containing nine stress components:

$$\boldsymbol{\sigma} = \begin{pmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{pmatrix}$$

Moment equilibrium demands the following relationships on shear stresses:

$$\sigma_{xy} = \sigma_{yx} \quad ; \quad \sigma_{xz} = \sigma_{zx} \quad ; \quad \sigma_{zy} = \sigma_{yz}$$

As a result there are only six independent stress components:

- three normal stresses ($\sigma_{xx}, \sigma_{yy}, \sigma_{zz}$)
- three shear stresses ($\sigma_{xy}, \sigma_{yz}, \sigma_{xz}$)

Stress tensor (2)

● In the absence of body moments, the stress tensor is symmetric and can always be resolved into the sum of two symmetric tensors:

● a mean or hydrostatic stress tensor, involving only pure tension and compression. It is defined as the average of normal stresses in three directions:

$$p = \frac{1}{3}(\sigma_{xx} + \sigma_{yy} + \sigma_{zz}) = \frac{1}{3}Tr[\sigma]$$

● a shear or deviatoric stress tensor, involving only shear stress, defined by

$$s_{ij} = \sigma_{ij} - p\delta_{ij}$$

or,

$$\mathbf{s} = \boldsymbol{\sigma} - p\mathbf{1}$$

● One usually defines the three following quantities:

$$J_1 = \sum_i \sigma_{ii}$$

$$J'_2 = \frac{1}{2} s_{ij} s_{ji}$$

$$J'_3 = \frac{1}{3} \sum_{ijk} s_{ij} s_{jk} s_{ki}$$

Invariants

● Moment equilibrium demands $\sigma_{ij} = \sigma_{ji}$ so that these invariants can be written

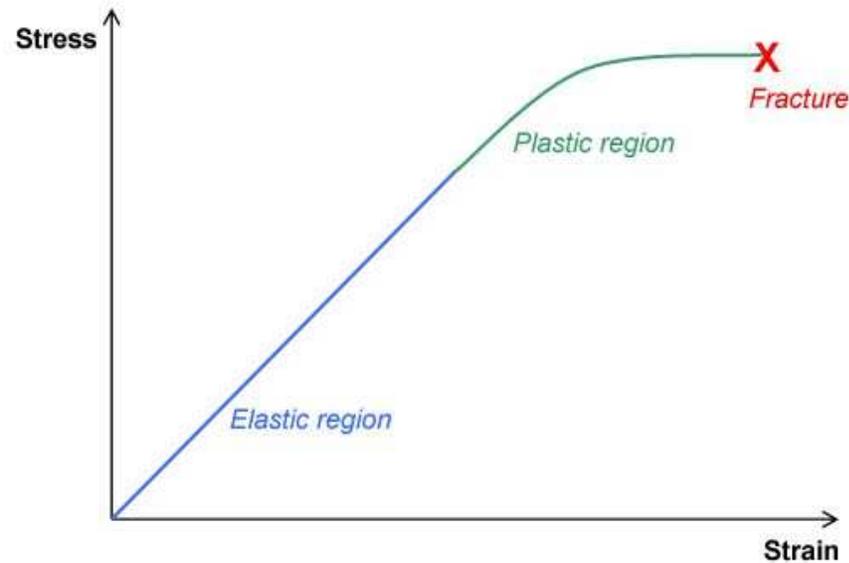
$$\begin{aligned} J_1 &= \sum_i \sigma_{ii} \\ J_2' &= \frac{1}{2} (\sigma_{xx}^2 + \sigma_{yy}^2 + \sigma_{zz}^2) + \sigma_{xy}^2 + \sigma_{yz}^2 + \sigma_{zx}^2 \\ J_3' &= \frac{1}{3} \sigma_{xx} (\sigma_{xx}^2 + 3\sigma_{xy}^2 + 3\sigma_{xz}^2) \\ &\quad + \frac{1}{3} \sigma_{yy} (3\sigma_{xy}^2 + \sigma_{yy}^2 + 3\sigma_{yz}^2) \\ &\quad + \frac{1}{3} \sigma_{zz} (3\sigma_{xz}^2 + 3\sigma_{yz}^2 + \sigma_{zz}^2) \\ &\quad + 2\sigma_{xy}\sigma_{xz}\sigma_{yz} \end{aligned}$$

● One also defines the Lode angle as follows:

$$\theta_l = \frac{1}{3} \sin^{-1} \left(-\frac{3\sqrt{3}}{2} \frac{J_3'}{(J_2')^{3/2}} \right) \quad \theta_l \in \left[-\frac{\pi}{6}; \frac{\pi}{6} \right]$$

Plasticity

- In physics and materials science, plasticity is a property of a material to undergo a non-reversible change of shape in response to an applied force. Plastic deformation occurs under shear stress, as opposed to brittle fractures which occur under normal stress. Examples of plastic materials are clay and mild steel. In engineering, the transition from elastic behavior to plastic behavior is called yield.
- There are several mathematical descriptions of Plasticity. One is deformation theory (see e.g. Hooke's law) where the stress tensor (of order d in d dimensions) is a function of the strain tensor. Although this description is accurate when a small part of matter is subjected to increasing loading (such as strain loading), this theory can not account for irreversibility.
- materials can sustain large plastic deformations without fracture. However, even ductile metals will fracture when the strain becomes large enough - this is as a result of work-hardening of the material, which causes it to become brittle. Heat treatment such as annealing can restore the ductility of a worked piece, so that shaping can continue.



The von Mises yield criterion

- We assume that the plastic flow does not depend on the hydrostatic pressure, so that $F(\boldsymbol{\sigma}) = f(J_2)$.
- The von Mises criterion states that flow occurs only at those points where the second invariant J_2 reaches a certain value depending on the material.
- The yield function can then be written as

$$F = \sqrt{J_2'} - c$$

The Drucker-Prager yield criterion

The von Mises yield criterion is not suitable for modelling the yielding of frictional material as it does not include the effect of mean stress as observed in experiments.

To overcome this limitation, Drucker and Prager (1952) proposed the following revised function for frictional materials:

$$F = \sqrt{J'_2} + \alpha p - k$$

where α and k are material constants. In principal stress space, the Drucker-Prager surface has the form of a circular cone, whilst the von Mises yield surface is an infinitely long cylinder.

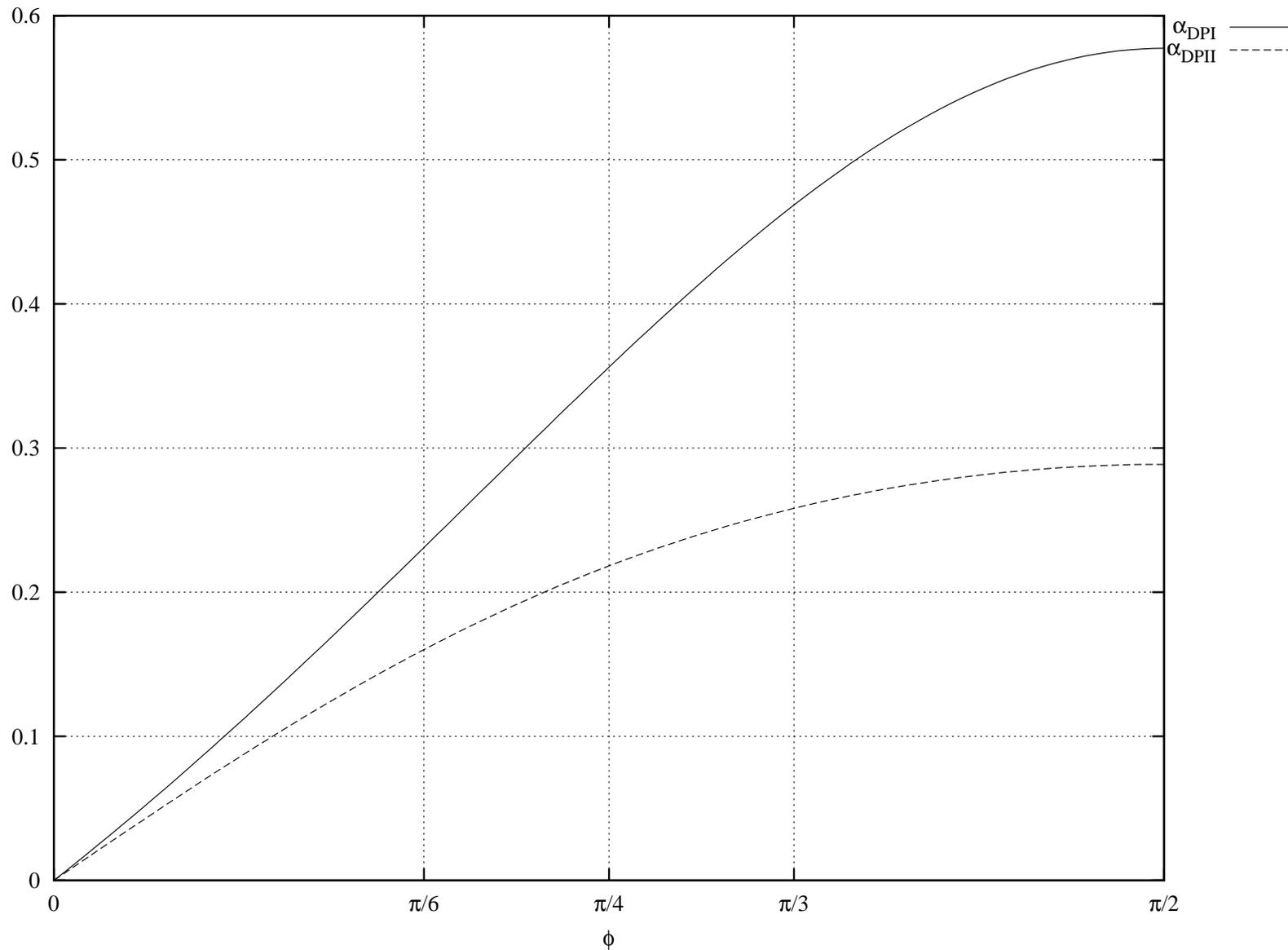
DPI

$$\alpha = \frac{2 \sin \phi}{\sqrt{3}(3 - \sin \phi)}$$
$$k = \frac{6c \cos \phi}{\sqrt{3}(3 - \sin \phi)}$$

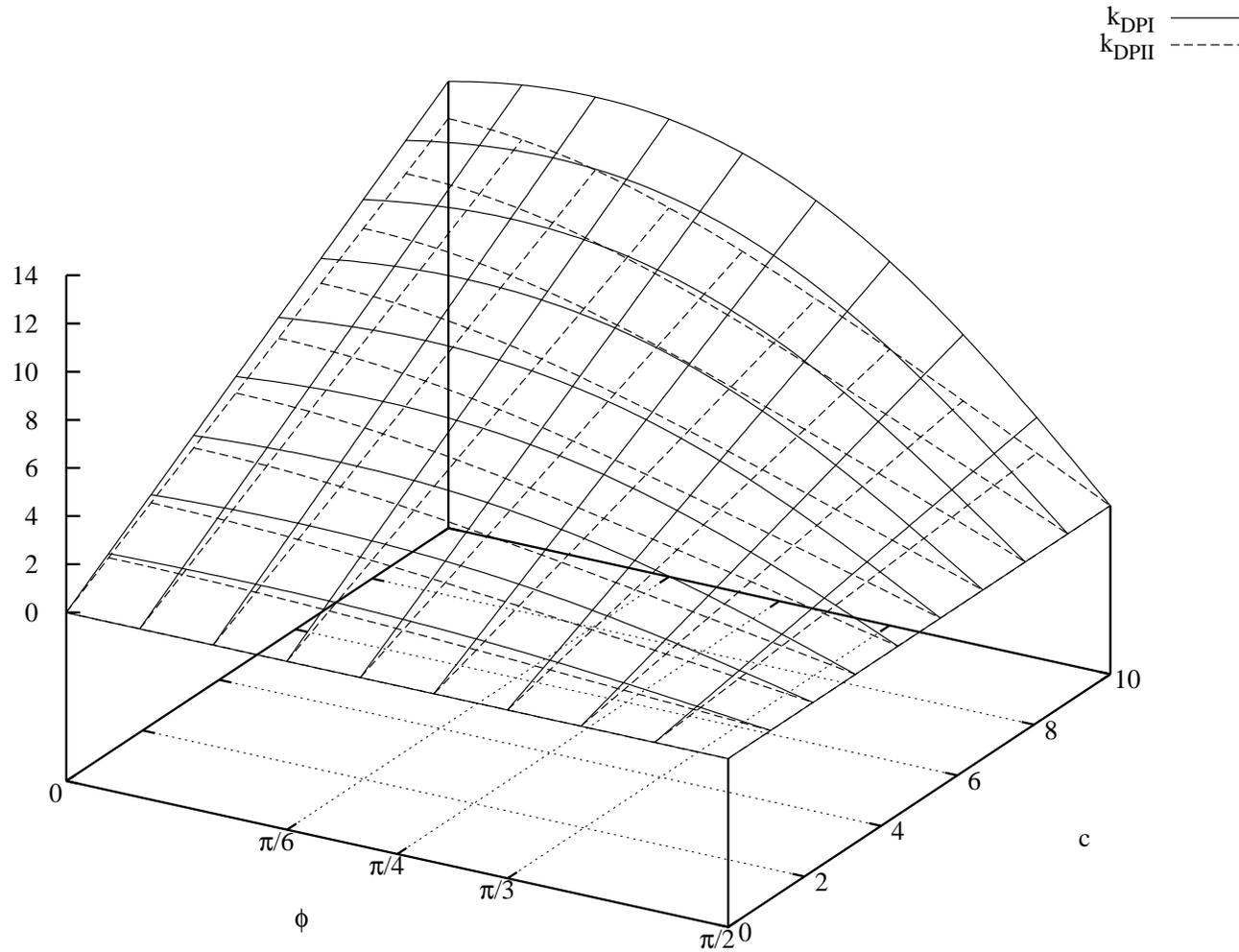
DPII

$$\alpha = \frac{\tan \phi}{\sqrt{9 + 12 \tan^2 \phi}}$$
$$k = \frac{3c}{\sqrt{9 + 12 \tan^2 \phi}}$$

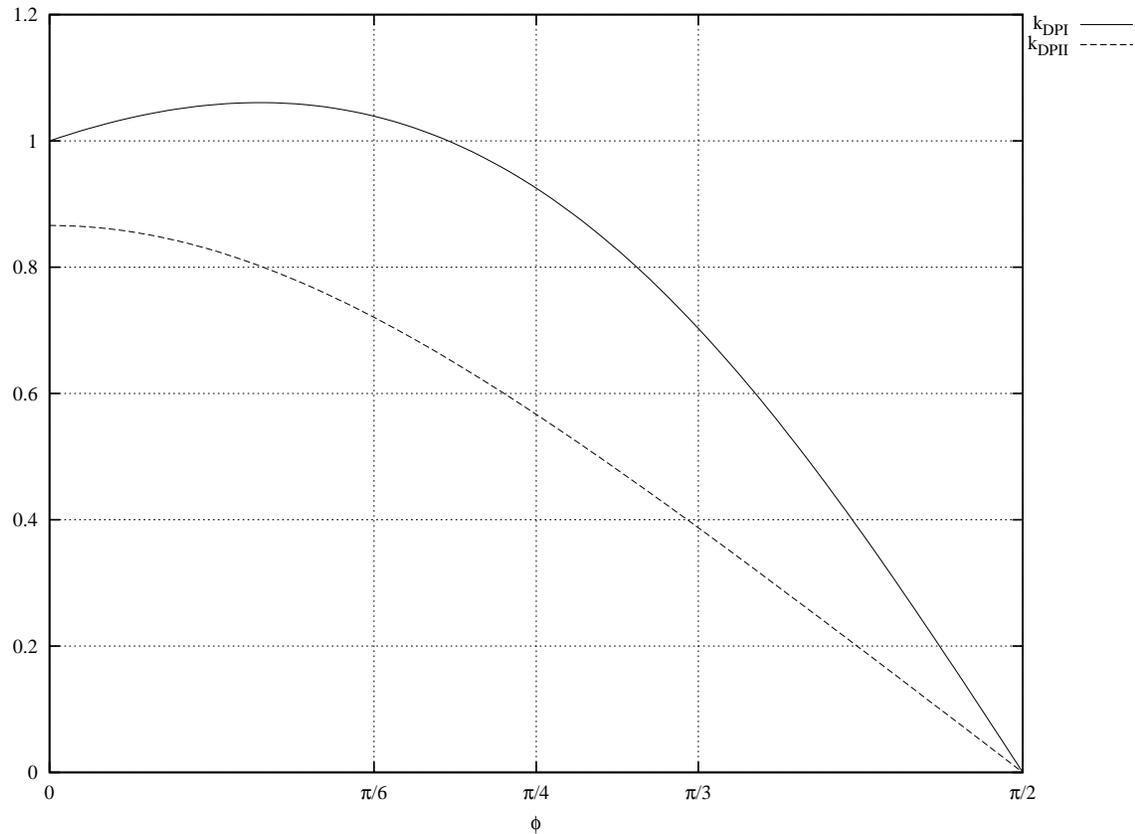
The Drucker-Prager yield criterion



The Drucker-Prager yield criterion



The Drucker-Prager yield criterion



$$c = \frac{\sqrt{3}}{2}$$

The Mohr-Coulomb yield criterion

- The Mohr-Coulomb criterion is commonly used to represent the behaviour of rocks and requires two rheological parameters:
 - ϕ the dimensionless angle of friction
 - c the cohesion that has units of pressure
- In 2D, the yield criterion is in terms of shear stress τ and normal stress σ_n acting on a plane. It suggests that the yielding begins as long as the shear stress and the normal stress satisfy the following equation:

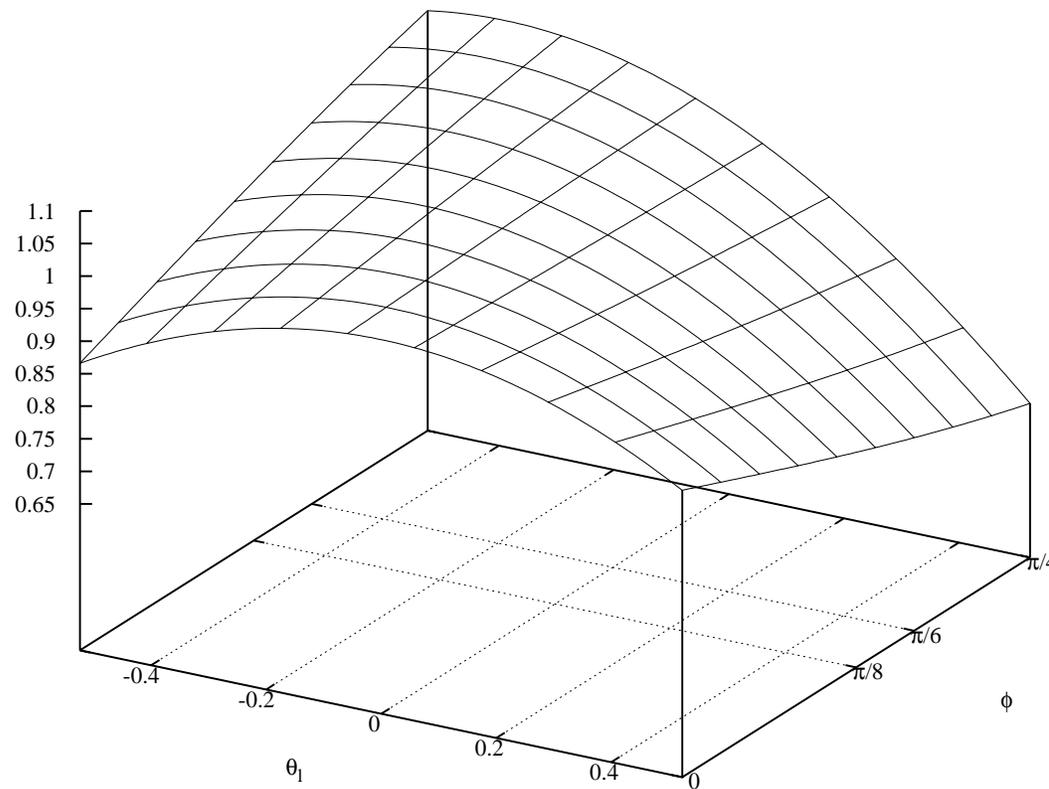
$$|\tau| = c + \sigma_n \tan \phi$$

- in 3D the yield criterion is given by:

$$F = p \sin \phi + \underbrace{\sqrt{J_2'} \left(\cos \theta_l - \frac{1}{\sqrt{3}} \sin \theta_l \sin \phi \right)}_{\zeta(\theta_l, \phi)} - c \cos \phi$$

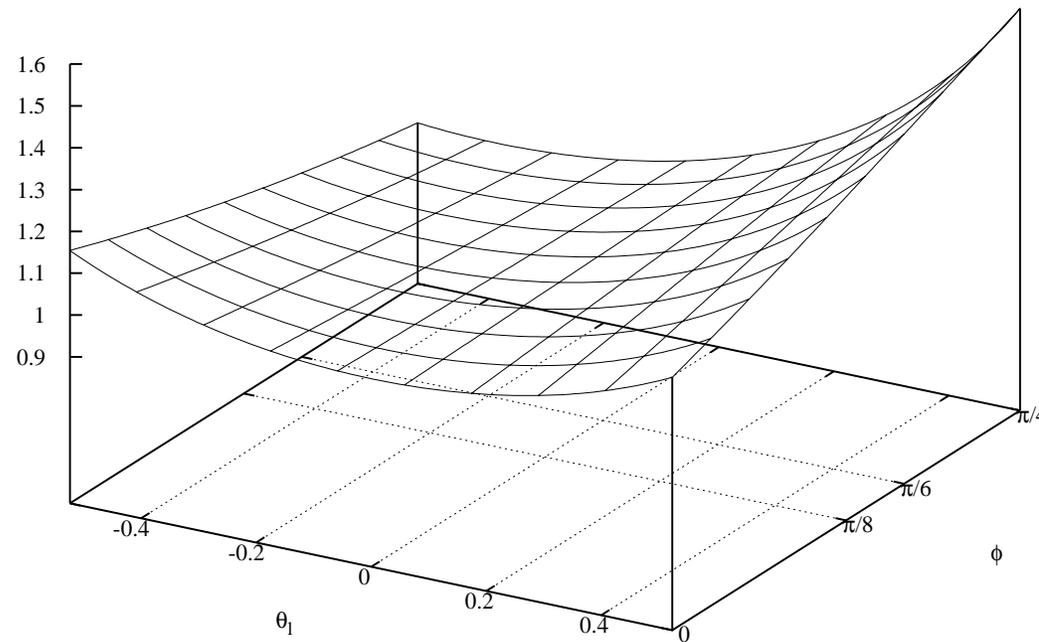
The Mohr-Coulomb yield criterion

$$\zeta(\theta_l, \phi) = \cos \theta_l - \frac{1}{\sqrt{3}} \sin \theta_l \sin \phi \quad \theta_l \in \left[-\frac{\pi}{6} : \frac{\pi}{6}\right] \quad \phi \in \left[0 : \frac{\pi}{4}\right]$$



The Mohr-Coulomb yield criterion

$$m(\theta_l, \phi) = \zeta(\theta_l, \phi) = \frac{1}{\cos \theta_l - \frac{1}{\sqrt{3}} \sin \theta_l \sin \phi} \quad \theta_l \in \left[-\frac{\pi}{6} : \frac{\pi}{6}\right] \quad \phi \in \left[0 : \frac{\pi}{4}\right]$$



The Tresca yield criterion



The yield criterion is given by

$$F = \sqrt{J_2'} \cos \theta_l - c$$

Other yield criteria

- Matsuoka-Nakai (MN) (work in progress)

$$F = \frac{I_1 I_2}{I_3} - (9 + 8 \tan^2 \phi)$$

- Lade-Duncan (LD) (work in progress)

$$F = \frac{I_1^3}{I_3} - k$$

- Griffith-Murrel (GM) (work in progress)

$$F = 4J_2' \cos \theta_l + g(\theta_l) \sqrt{J_2'} - \alpha I_1 - k$$

- Hoek-Brown (HB) (work in progress)

- Cam-Clay (CC) (work in progress)

$$F = J_2' - M^2 (p(p_0 - p))$$

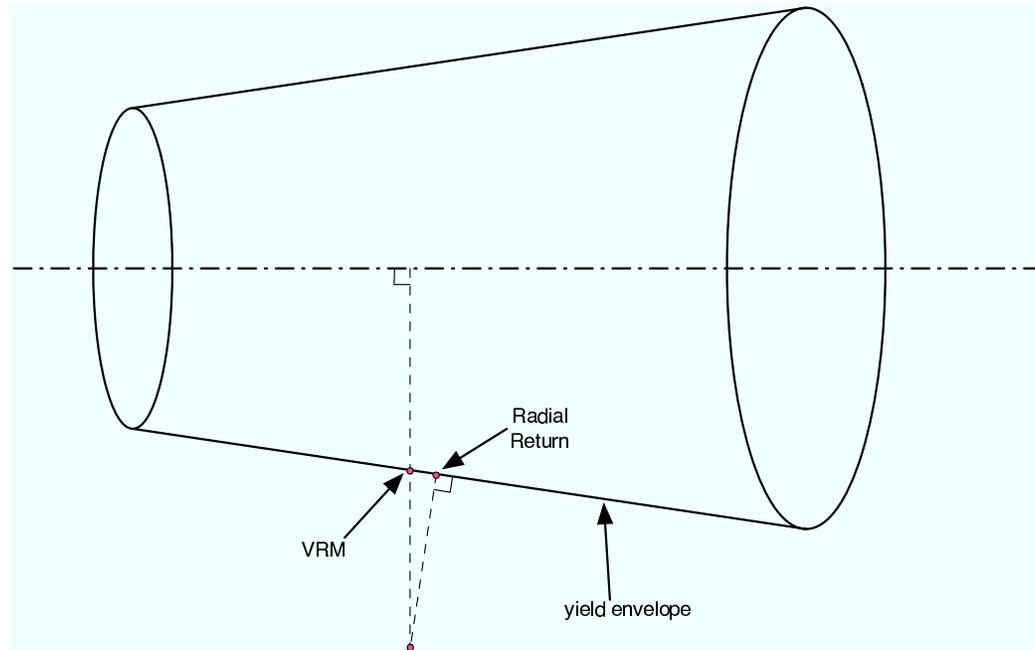
where M is a soil constant, and p_0 a history variable.

the vrm algorithm

$J_2' = \text{second_invariant}(\boldsymbol{\sigma}) = 2\mu \text{ second_invariant}(\dot{\boldsymbol{\epsilon}}) = 2\mu E_2'$
From

$$F(\boldsymbol{\sigma}) = 0$$

a value μ' of the viscosity is computed that by simple rescaling allows for the point to be placed on the yield surface.



Nonlinear viscosity

- At high temperature, rocks deform by creep, a non-linear form of viscous deformation that is commonly approximated by defining a stress or strain rate dependent and thermally activated viscosity:

$$\mu = \mu_0 \dot{\epsilon}^{\left(1 - \frac{1}{n}\right)} \exp\left(-\frac{Q}{nRT}\right)$$

where

- μ_0 is the viscosity of the material at $T = T_0$
- n is the nonlinear exponent
- R is the perfect gas constant
- Q is the activation energy
- T is the temperature
- $\dot{\epsilon}$ is ... ?

material density

- In physics, thermal expansion is the tendency of matter to increase in volume or pressure when heated. For liquids and solids the amount of expansion will normally vary depending on the material's coefficient of thermal expansion.
- The density ρ varies as a function of temperature according to:

$$\rho = \rho_0(1 - \alpha(T - T_0))$$

where

- α is the coefficient of thermal expansion (K^{-1})
- ρ_0 is the value of the density at $T = T_0$

The FEM method

Gaussian quadrature

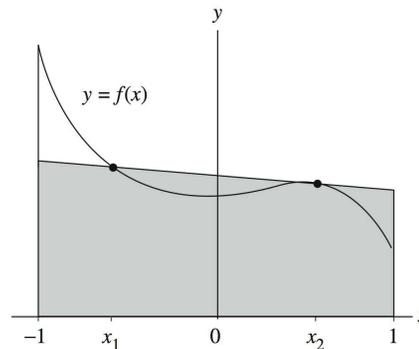
- In numerical analysis, a quadrature rule is an approximation of the definite integral of a function, usually stated as a weighted sum of function values at specified points within the domain of integration.
- An n -point Gaussian quadrature rule, named after Carl Friedrich Gauss, is a quadrature rule constructed to yield an exact result for polynomials of degree $2n - 1$, by a suitable choice of the n points r_i and n weights w_i .
- The domain of integration for such a rule is conventionally taken as $[-1, 1]$, so the rule is stated as

$$\int_{-1}^1 f(r) dr \simeq \sum_{i=1}^n w_i f(r_i)$$

- in three dimensions this becomes

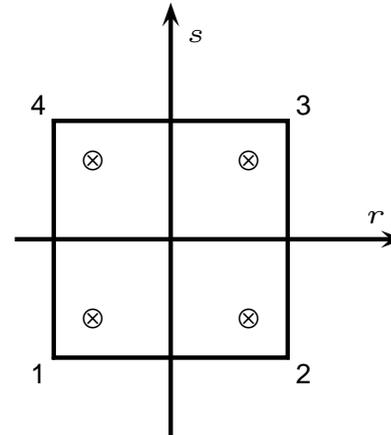
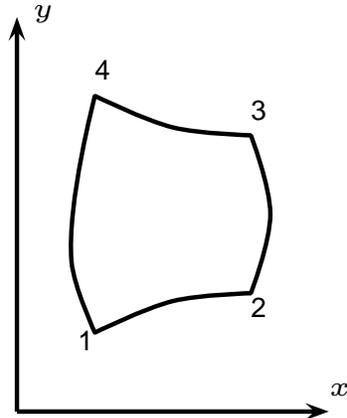
$$\int_{-1}^{+1} \int_{-1}^{+1} \int_{-1}^{+1} f(r, s, t) dr ds dt \simeq \sum_{i=1}^n \sum_{j=1}^n \sum_{k=1}^n w_i w_j w_k f(r_i, s_j, t_k)$$

- in DOUAR we set $n = 2$ so that $r_i = \pm\sqrt{\frac{1}{3}} = \pm 0.57735026919\dots$ and $w_i = 1$



global/local coordinates

- In order to perform the numerical integration described previously, one needs to change from the global coordinate system (x, y, z) to a local coordinate system (r, s, t) .



coordinates change

- When changing from a coordinate system to another coordinate system, one often requires the derivatives $\partial/\partial x$, $\partial/\partial y$, $\partial/\partial z$, and it seems natural to use the chain rule as follows:

$$\frac{\partial}{\partial x} = \frac{\partial r}{\partial x} \frac{\partial}{\partial r} + \frac{\partial s}{\partial x} \frac{\partial}{\partial s} + \frac{\partial t}{\partial x} \frac{\partial}{\partial t}$$

with similar relationships for $\partial/\partial y$ and $\partial/\partial z$, so that one can write

$$\begin{pmatrix} \frac{\partial}{\partial r} \\ \frac{\partial}{\partial s} \\ \frac{\partial}{\partial t} \end{pmatrix} = \begin{pmatrix} \frac{\partial x}{\partial r} & \frac{\partial y}{\partial r} & \frac{\partial z}{\partial r} \\ \frac{\partial x}{\partial s} & \frac{\partial y}{\partial s} & \frac{\partial z}{\partial s} \\ \frac{\partial x}{\partial t} & \frac{\partial y}{\partial t} & \frac{\partial z}{\partial t} \end{pmatrix} \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{pmatrix}$$

- in matrix notation

$$\frac{\partial}{\partial \mathbf{r}} = \mathbf{J} \frac{\partial}{\partial \mathbf{x}}$$

where \mathbf{J} is the Jacobian operator.

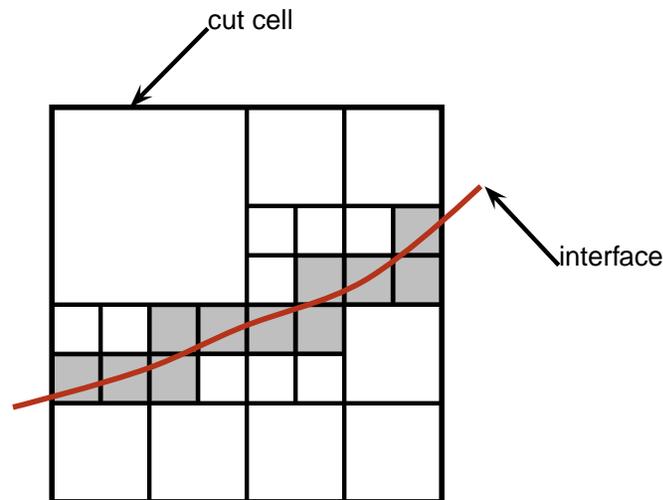
- the volume integration extends over the natural coordinate volume, and the volume differential dV needs also to be written in terms of the natural coordinates. In general we have

$$dV = \det(\mathbf{J}) dr ds dt$$

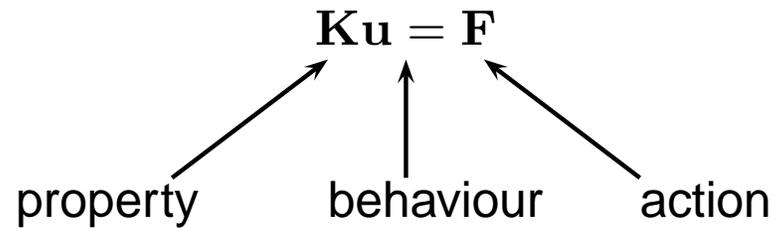
divfem

- From the values of the level set functions, the position of each element with respect to each interface is known as well as possible.
- This information is used to determine the material making up the element, assuming that interfaces are material boundaries.
- When an element is intersected by one or several interfaces, the value of the level set functions at the nodes of the elements are used to compute the part or volume of the element that is in each of the materials.
- These volumes are used to perform the volume integration of the finite element equations.
- To determine the volume that is on the positive side of the interface cutting a given element (the *cut cell*), an octree division of cut cells is performed down to level 3 ($8 \times 8 \times 8$). The level set function is interpolated to the internal nodes and used to determine which part of the volume (positive or negative) each sub-cell belongs.
- The relative positive volumes, α , in the remaining cut cells (the gray cells) are estimated using the following approximate formula and in those cut cells, material properties are averaged.

$$\alpha = \frac{1}{2} \left(\frac{\sum_{i=1}^8 \phi_i}{8} + 1 \right)$$



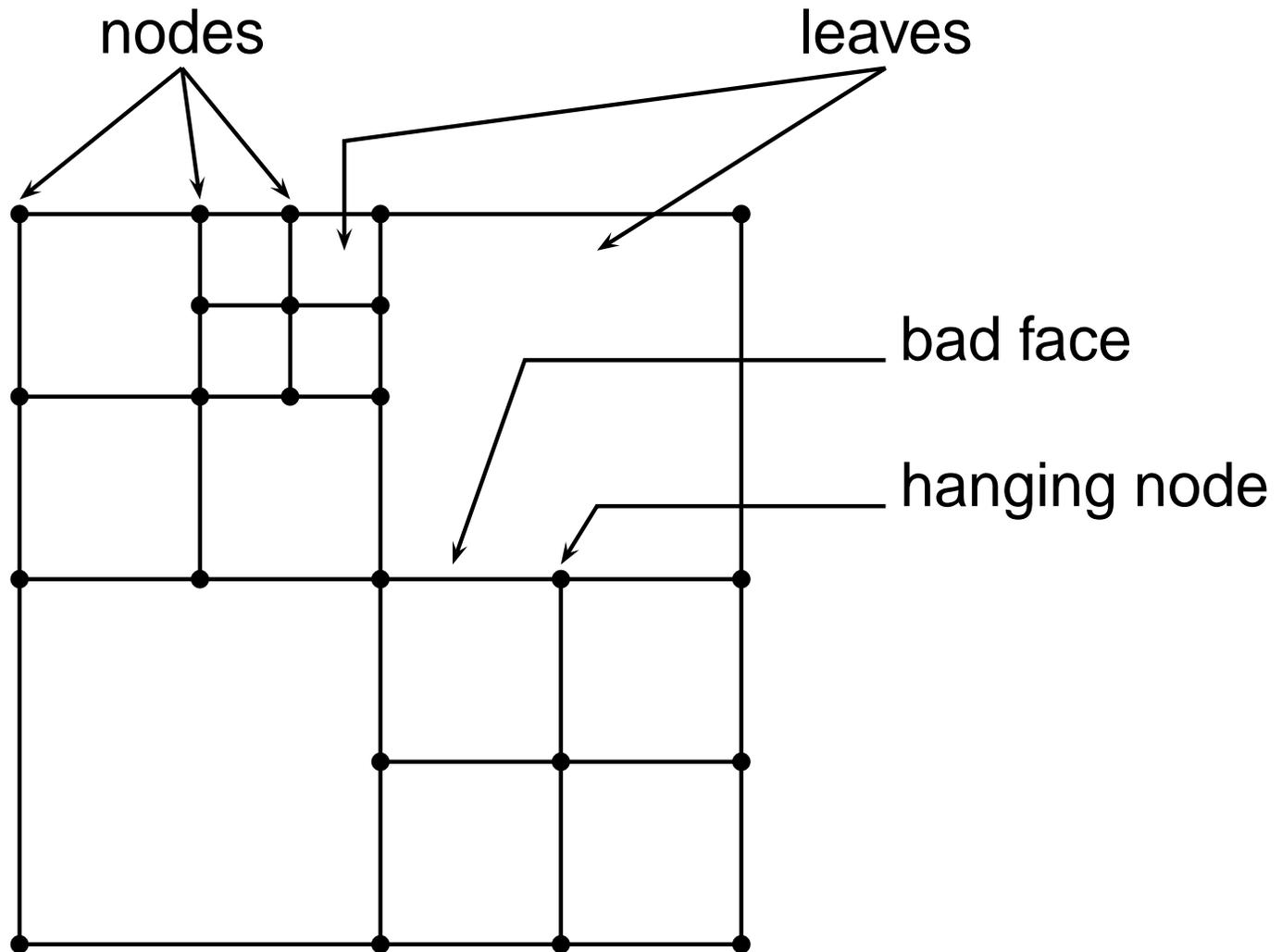
fundamental concept



	property	behaviour	action
elastic	stiffness	displacement	force
thermal	conductivity	temperature	heat source
fluid	viscosity	velocity	body force

Octrees

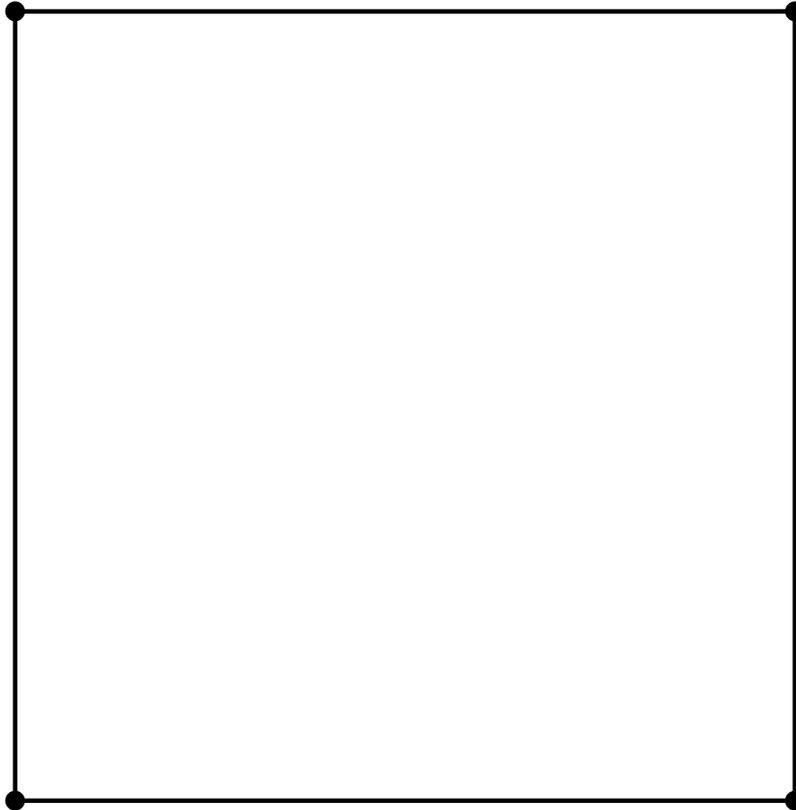
Terminology



Leaf size and levels (1)

level = 0

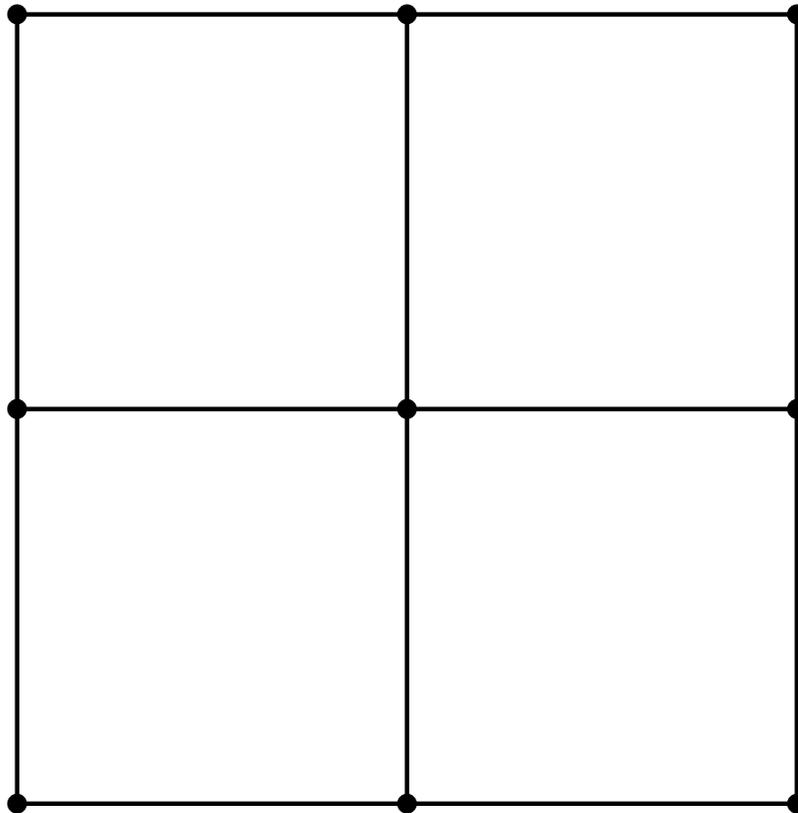
leaf size = $2^0 = 1$



Leaf size and levels (2)

level = 1

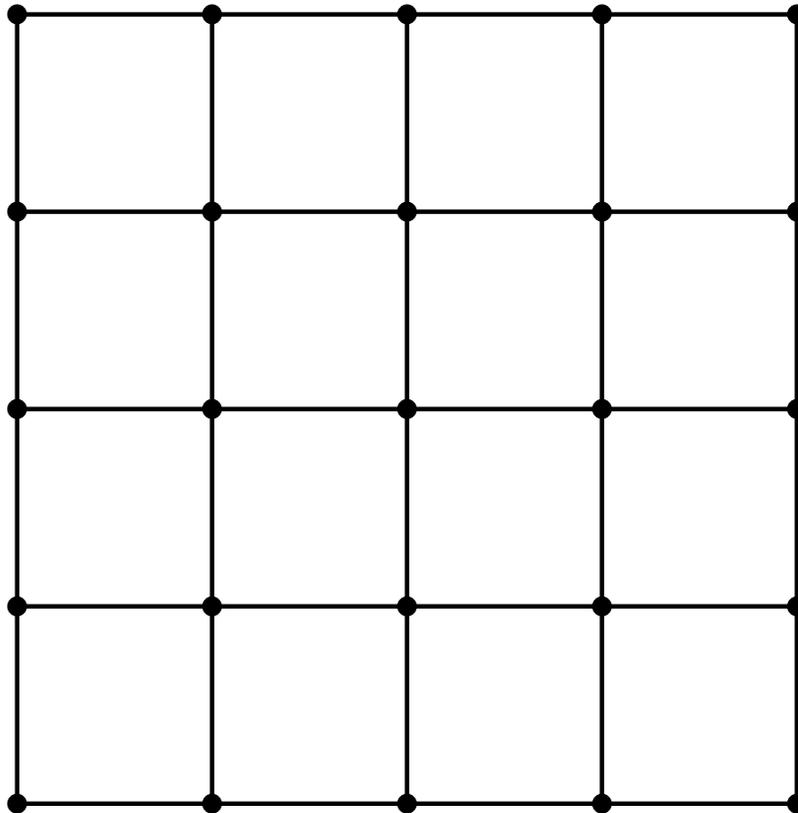
leaf size = $2^{-1} = 0.5$



Leaf size and levels (3)

level = 2

leaf size = $2^{-2} = 0.25$

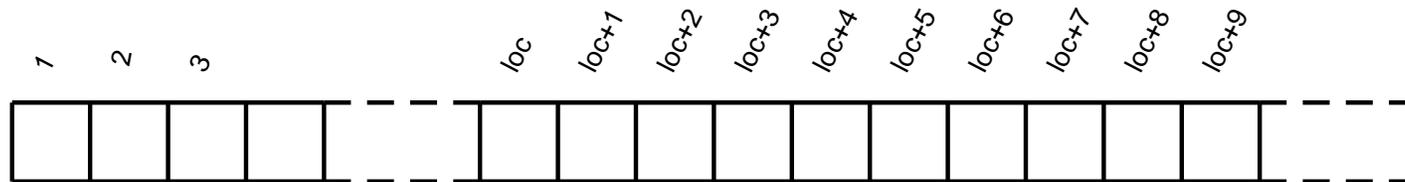


Leaf size and levels (4)

level	leaf size
0	$2^0 = 1$
1	$2^{-1} = 0.5$
2	$2^{-2} = 0.25$
3	$2^{-3} = 0.125$
4	$2^{-4} = 0.0625$
5	$2^{-5} = 0.03125$
6	$2^{-6} = 0.015625$
7	$2^{-7} = 0.0078125$
8	$2^{-8} = 0.00390625$
9	$2^{-9} = 0.001953125$
10	$2^{-10} = 0.0009765625$

Internal structure

- Octrees are very simple and memory-efficient entities that can be built as a single integer array containing, for each cube of the octree, the address in the array of the first of its eight *children cubes*.
- When a cube is not divided, it becomes a leaf to which a name/number is associated and stored in the octree integer array as a negative number (to indicate that it corresponds to a leaf number and not a child's address).



- $octree(1)$ =maximum level (unit cube is level 0)
- $octree(2)$ =number of leaves
- $octree(3)$ =total length of octree

For each cube in the octree (at location loc)

- $octree(loc)$ =level
- $octree(loc+1)$ =address of parent
- $octree(loc+2$ to $loc+9)$ =address of children
(if negative the child is a leaf and the value is the leaf number in the sequence of leaves)

The OctreeBitPlus library



All the following routines are in `/OCTREE/OctreeBitPlus.f90`:

- `octree_init`
- `find_integer_coordinates, find_real_coordinates`
- `octree_create_from_particles`
- `octree_find_leaf`
- `octree_smooth, octree_super_smooth`
- `octree_find_element_level`
- `ioctree_number_of_elements`
- `ioctree_maximum_level, ioctree_size`
- `octree_create_uniform`
- `octree_renumber_nodes`
- `octree_find_node_connectivity`
- `octree_find_bad_faces`
- `octree_interpolate`
- `octree_interpolate_many, octree_interpolate_many_derivative`

octree_init

```
subroutine octree_init (octree , noctree)  
integer noctree , octree (noctree)  
octree(1)=1  
octree(2)=8  
octree(3)=13  
loc=4  
octree(loc)=1  
octree(loc+1)=0  
do k=1,8  
    octree(loc+1+k)=-k  
enddo  
return  
end
```

This routine initialises the octree structure by creating an octree of level 1, containing 8 leaves.

octree_create_from_particles

This routine updates the octree by creating a leaf at points $(x(1:np), y(1:np), z(1:np))$ of level $level(1:np)$. If the leaf (or a cube of smaller level) exists, the routine has no effect on the octree.

```
subroutine octree_create_from_particles (octree , noctree , x , y , z , np , level )
```

```
integer noctree , octree (noctree) , np
```

```
double precision x(np) , y(np) , z(np)
```

```
integer level(np)
```

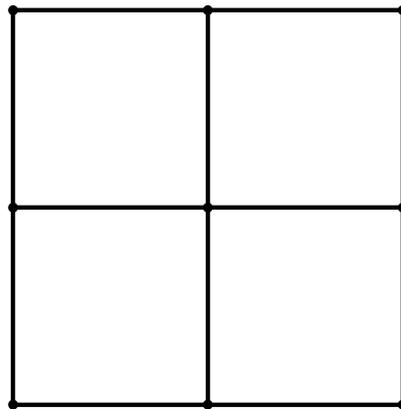
```
[...]
```

```
return
```

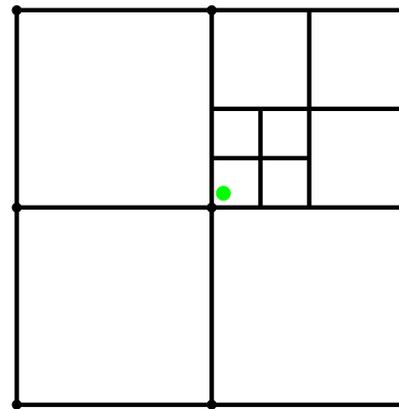
```
end
```

```
call octree_create_from_particles (octree,noctree,x,y,z,1,3)
```

level = 1



level = 3



octree_find_leaf

Given an octree of size noctree, this routine returns the leaf number in which a point (x,y,z) resides, the level of the leaf (0 is unit cube), the location in the octree of the part describing the parent of the leaf (loc), the centroid of the leaf (x0,y0,z0) and its size (dxyz).

subroutine octree_find_leaf (octree , noctree , x , y , z , leaf , level , loc , x0 , y0 , z0 , dxyz)

integer noctree , octree (noctree)

double precision x , y , z , x0 , y0 , z0 , dxyz

integer leaf , level , loc

[...]

return

end

octree_smoothen

This routine smoothens the octree: it ensures that no two adjacent leaves are more than one level apart.

```
subroutine octree_smoothen (octree , noctree)
```

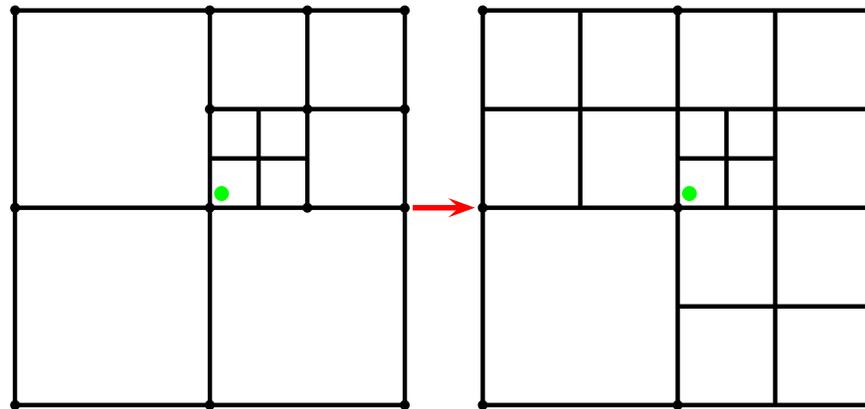
```
integer noctree , octree (noctree)
```

```
[...]
```

```
return
```

```
end
```

call octree_smoothen (octree,noctree)



octree_find_element_level

This routine returns the level of each leaf. The result is returned in the array levs of dimension nleaves

subroutine octree_find_element_level (octree , noctree , levs , nleaves)

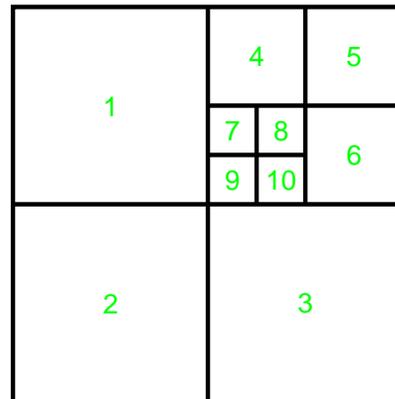
integer noctree , octree (noctree) , nleaves , levs(nleaves)

[...]

return

end

call octree_find_element_level (octree,noctree,levs,nleaves)



nleaves=10

nnode=17

levs(1:10)=(1,1,1,2,2,3,3,3,3)

octree_create_uniform

This routine generates a uniform octree down to level levelt.

```
subroutine octree_create_uniform (octree , noctree , levelt )
```

```
integer noctree , octree (noctree) , levelt
```

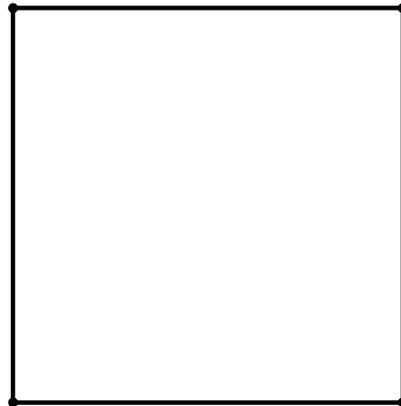
```
[...]
```

```
return
```

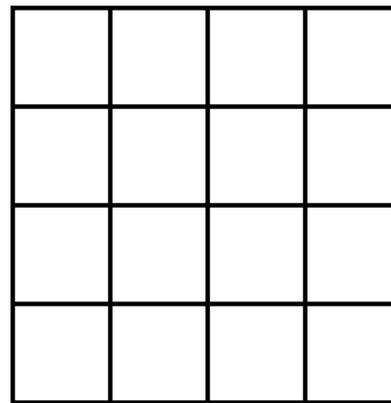
```
end
```

```
call octree_create_uniform (octree,noctree,2)
```

level = 0



level = 2



octree_find_node_connectivity

This routine

- finds the number (na) and locations (xa,ya,za) of the nodes of the octree.
- computes the connectivity array between nodes and leaves (icon).

Icon is dimensioned icon(8,nleaves) and contains the number of the 8 nodes connected by each leaf. When calling this routine, na should have the dimension used to dimension the coordinate arrays in the calling routine. On return na contains the true dimension of these array (ie how many nodes there are in the octree)

subroutine octree_find_node_connectivity (octree , noctree , icon , nleaves , xa , ya , za , na)

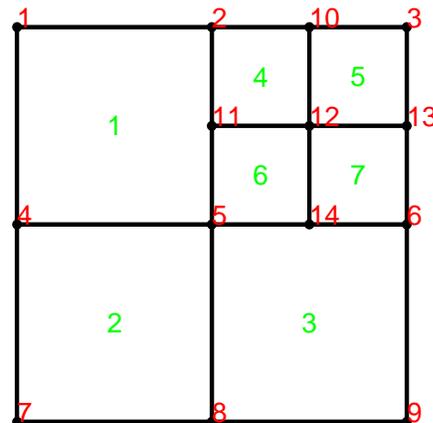
integer noctree , octree (noctree) , nleaves , icon (8 , nleaves) , na

double precision xa (*) , ya (*) , za (*)

[...]

return

end



nleaves=7
nnode=14

icon(1:4,2)=(4,5,7,8)

icon(1:4,7)=(6,12,13,14)

octree_find_bad_faces

this routine returns the bad faces as an array (iface) of 9 nodes per face.
iface is the resulting bad face information iface(9,nface).
nface is the number of bad faces found.

icon is the connectivity array of dimension nelem

subroutine octree_find_bad_faces (octree , noctree , iface , nface , icon , nelem)

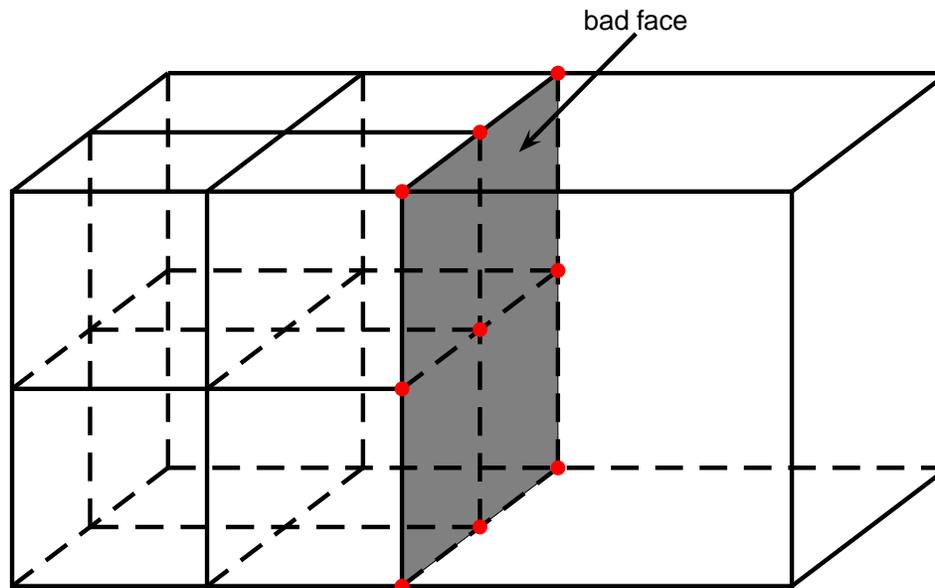
integer noctree , octree (noctree)

integer nface , iface (9 , nface) , nelem , icon (8 , nelem)

[...]

return

end



octree_interpolate

This function returns the value of a field (field) known at the nodes of an octree by trilinear interpolation.

icon is the connectivity matrix, nleaves is the number of leaves in the octree, field is the array of dimension nfield containing the field known at the nodes of the octree and to be interpolated, x,y,z are the location of the point where the field is to be interpolated, f is the resulting interpolated field.

subroutine octree_interpolate (octree , noctree , icon , nleaves , field , nfield , x , y , z , f)

integer noctree , octree (noctree) , nleaves , icon (8 , nleaves)

integer nfield

double precision field (nfield) , x , y , z , f

[...]

return

end

The NN library

The NN library - files (1)

 The library contains five fortran files:

 delaun.f

 subroutine delaun

 del_sub.f

 subroutine visiblelist

 subroutine addpoint

 subroutine insertpoint

 subroutine Triloc_del

 nn.f

 subroutine nn2d_setup

 subroutine nn2D

 subroutine nn2DL

 subroutine build_nv

 subroutine calculate_hulltriangles

 subroutine Triloc

 subroutine ccentres

 subroutine nn2Di

 subroutine nn2Do

 subroutine nn_tri

 subroutine nn2Dr, subroutine nn2Drd

 subroutine nn2Df, subroutine nn2Dfd, subroutine nn2Dfdd

 subroutine first_voronoi, subroutine second_voronoi, subroutine second_voronoi_d

 subroutine circum, subroutine circum_d, subroutine circum_dd

 subroutine second_v_area, subroutine second_v_area_d, subroutine second_v_area_dd

 nnplot.f

 qhullf_dummy.f

The NN library - files (2)

 The library contains six C files:

-  stack.c(c)
 -  stackinit
 -  push
 -  pop
 -  stackempty
 -  stackflush
-  stackpair.c(c)
 -  stackpairinit
 -  pushpair
 -  poppair
 -  stackpairempty
 -  stackpairflush
-  volume.c(c)
 -  cvolume
 -  cvolumef
 -  cvolumeb
 -  cdvda
 -  cdvdaf
 -  cvolumebj

The NN library - bibliography (1)

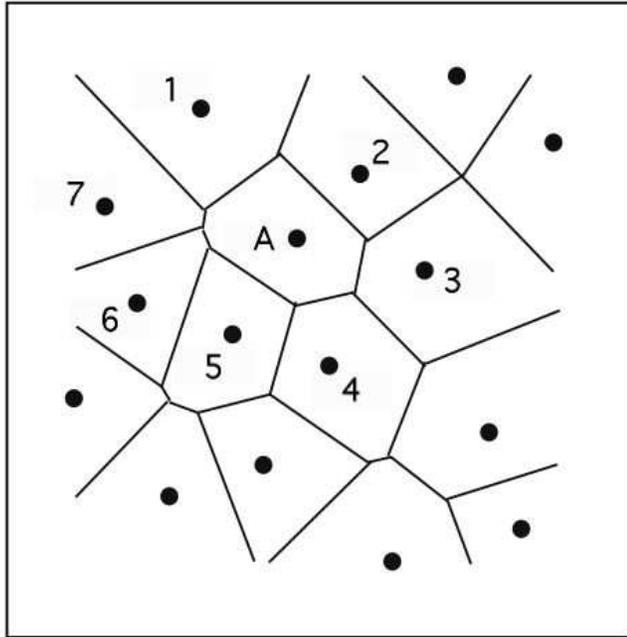
- Efficient parallel inversion using the Neighbourhood Algorithm. Rickwood, P. and Sambridge, M., *Geochem. Geophys. Geosyst*, 7,Q11001, 2006.
- Nonlinear inversion by direct search using the neighbourhood algorithm. Sambridge, M., *Int. Handbook of Earthquake and Engineering Seismology*, 81B, Ch. 85-15, 1635-1637, 2003.
- Constraints on the S-wave Velocity Structure in a Continental Shield From Surface-Wave Data: Comparing Linearized Least-Squares Inversion and the Direct-Search Neighbourhood Algorithm. Snoke, J. A., and Sambridge, M., *J. Geophys. Res.*, 107(B5), 2002.
- A numerical method for solving partial differential equations on highly irregular evolving grids. J.Braun and M.Sambridge, *Nature* 376, 655 - 660 (24 August 2002).
- Seismic Event Location: Nonlinear Inversion Using a Neighbourhood Algorithm. Sambridge, M. S., and Kennett, B. L. N., *Pure. appl. geophys.*, 158, 241-257, 2001.
- Finding acceptable models in nonlinear inverse problems using a neighbourhood algorithm. Sambridge, M., *Inverse Problems*, 17, 387-403, 2001.
- Seismic Source Characterisation using a Neighbourhood Algorithm. Kennett, B. L. N., Marson-Pidgeon, K., and Sambridge, M., *Geophys. Res. Lett.*, 27, No. 20., 3401-3404, 2000.

The NN library - bibliography (2)

- Source depth and Mechanism inversion at teleseismic distances using a neighbourhood algorithm. Marson-Pidgeon, K., Kennett, B. L. N., and Sambridge, M., Bull. seism. Soc. am., 90, 1369-1383, 2000.
- Geophysical Inversion with a Neighbourhood Algorithm -I. Searching a parameter space. Sambridge, M., Geophys. J. Int., 138, 479-494, 1999.
- Geophysical Inversion with a Neighbourhood Algorithm -II. Appraising the ensemble. Sambridge, M., Geophys. J. Int., 138, 727-746, 1999.
- Computational methods for performing Natural Neighbour interpolation in two and three dimensions, Sambridge, M. S., Braun, J., and McQueen, H., Proceedings of the seventh Biennial conference on Computational techniques and applications (CTAC95), Eds. R. L. May and A. K. Easton, 685-692, 1996.
- Geophysical parametrization and interpolation of irregular data using natural neighbours. Sambridge, M. S., Braun, J., and McQueen, H., Geophys. J. Int., 122, 837-857, 1995.
- Dynamical Lagrangian Remeshing (DLR): A new algorithm for solving large strain deformation problems and its application to fault-propagation folding. J. Braun and M. Sambridge, Earth and Planetary Science Letters, Volume 124, Issues 1-4, June 1994, Pages 211-220.

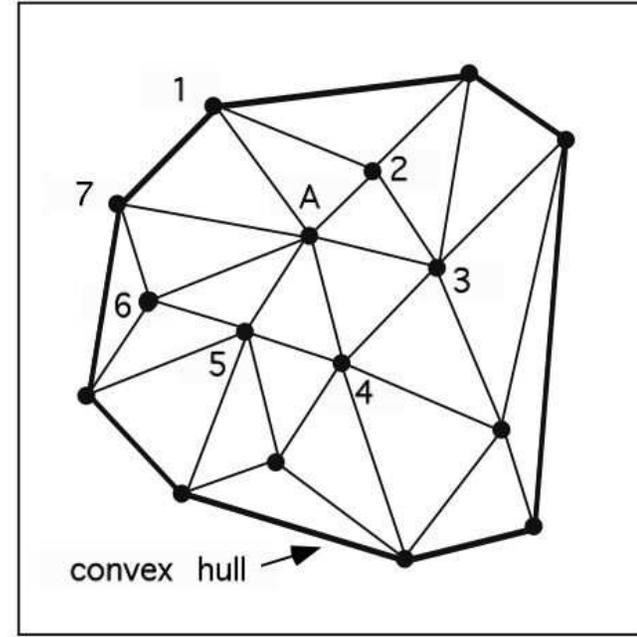
NN - basic principles (1)

a)



Voronoi cells

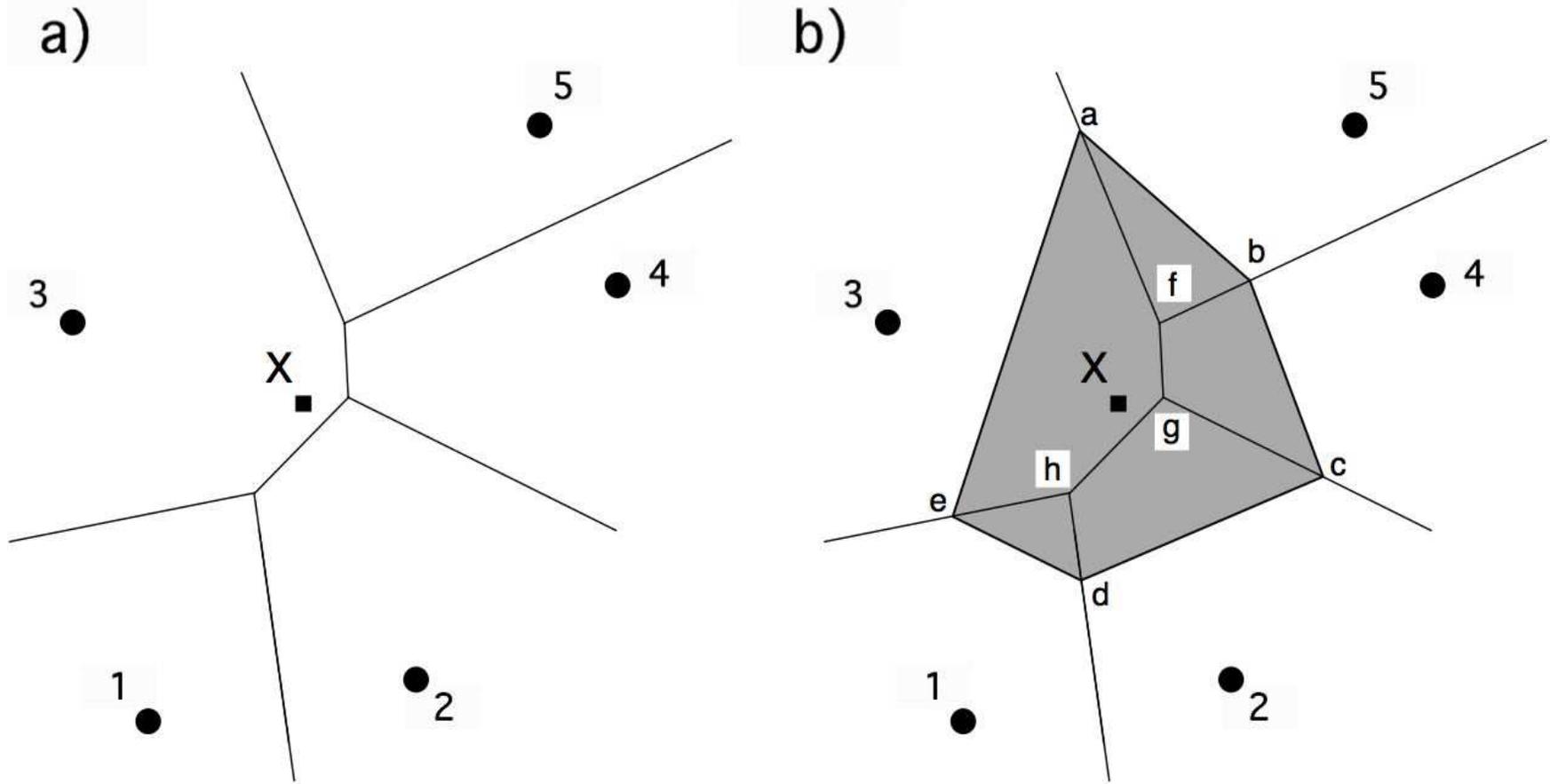
b)



Delaunay Triangulation

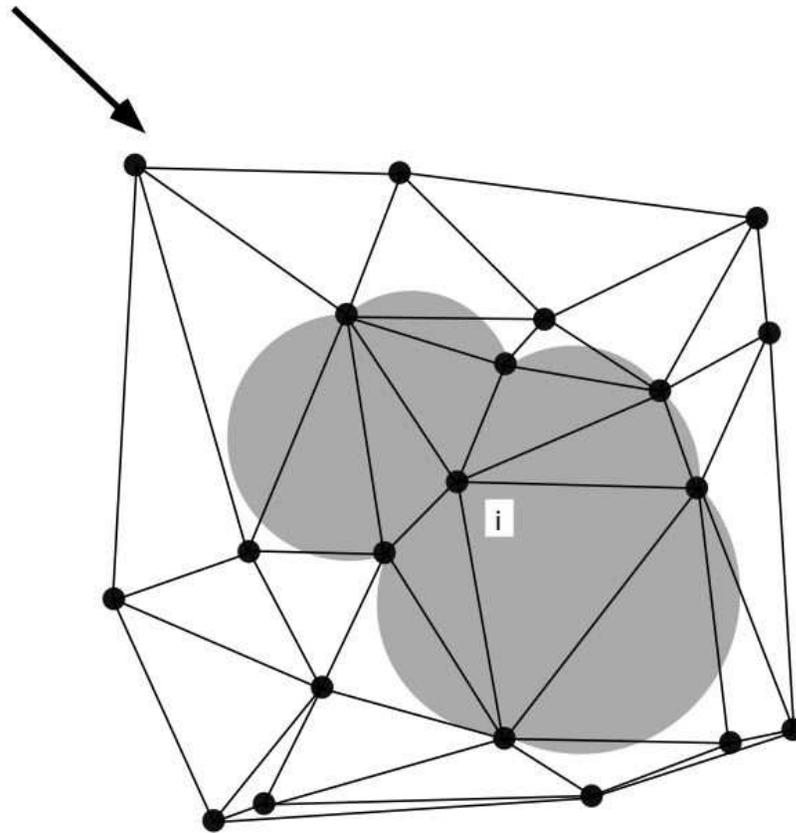
a) The Voronoi diagram for a set of 16 nodes in a plane. b) The corresponding Delaunay triangulation. The thick perimeter line connects the nodes in the convex hull.

NN - basic principles (2)



a) The original Voronoi diagram for 5 neighbouring nodes and an interpolation point X. b) The new Voronoi cell about X (shaded). The overlap of the new Voronoi cell with the original cells forms 5 second-order Voronoi cells between X and its neighbours.

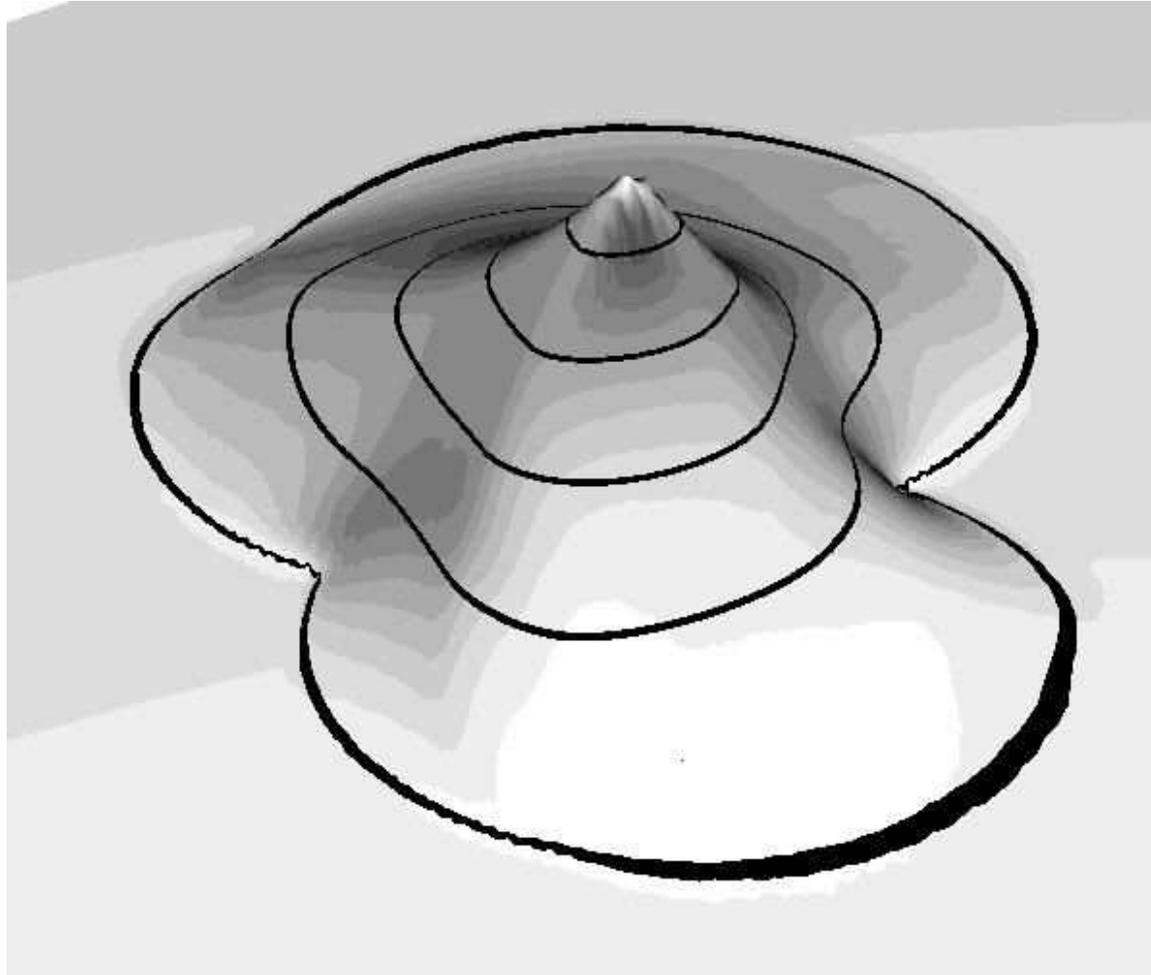
NN - basic principles (3)



Region of influence about node i

The shaded region about node i shows that the area that it can influence in natural-neighbour interpolation.

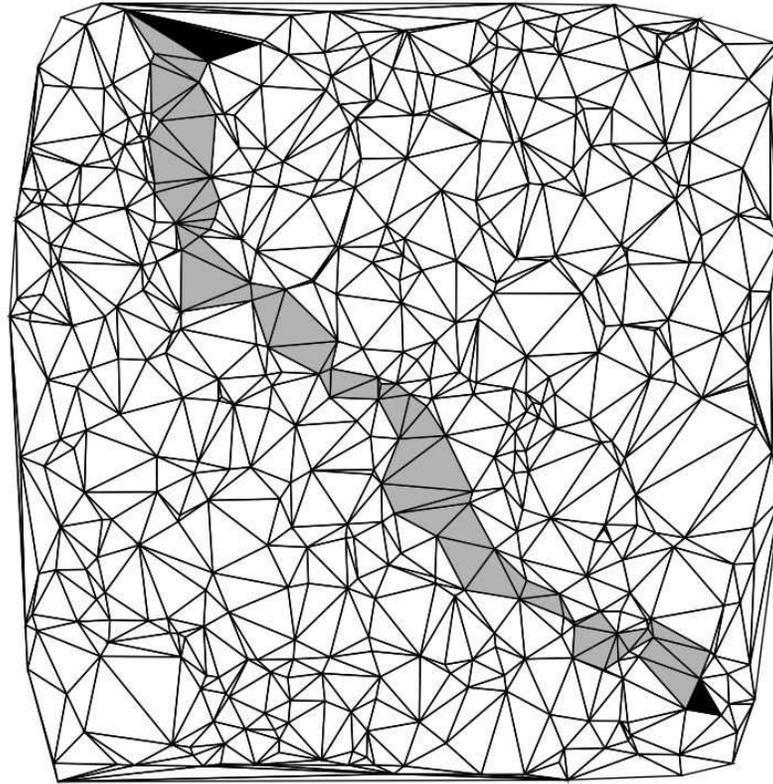
NN - basic principles (4)



A perspective view of the influence surface about node i seen from the direction of the arrow on previous figure. The height of the surface at any point is the value of its natural-neighbour coordinate with respect to node i .

NN - basic principles (5)

Walking triangle algorithm



An example of the path (shaded triangles) taken by the walking triangle algorithm. The initial triangle is in the lower right corner (black) and the final triangle is in the top left corner (black). The nearly direct path taken by the algorithm enables it to locate efficiently a point in any triangle.

The ALE concept

The solver

The code files and routines

build_edge

```
subroutine build_edge (surface,ed,nedge,refine,nadd,naddp, stretch,anglemax,  
nedgepernode,nodenodenummer,nodeedgenumber,nmax, distance_exponent)
```



arguments

- surface is the sheet/surface to be refined
- ed is the computed edge array
- nedge is the number of edges
- refine is the integer array determining the edges to be refined
- stretch is the maximum stretching allowed (set in the input.txt file)
- nadd is the number of edges to be refined
- anglemax is the maximum authorised angle between two normals
- nedgepernode,nodenodenummer and nodenodenummer contain the list of edges that start from each node; for a given node i their number is nedgepernode(i), the edge number in the list of edges is nodeedgenumber(j,i) and the node at the end of the edge is nodenodenummer(j,i) for $j=1,nedgepernode(i)$



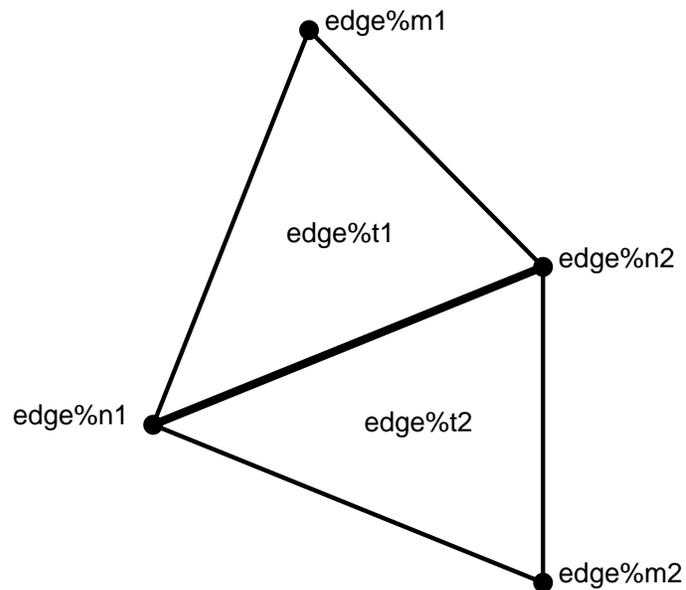
This routine builds an edge array between a set of particles on a surface. It uses the delaunay triangulation and then steps through the triangles to build the edge information. In a second step, the routine checks for refinement and computes an integer array (refine) which contains the list of edges that need to be refined.

It uses the delaunay triangulation and then steps through the triangles to build the edge information. In a second step, the routine checks for refinement and computes a integer array (refine) which contains the list of edges that need to be refined.

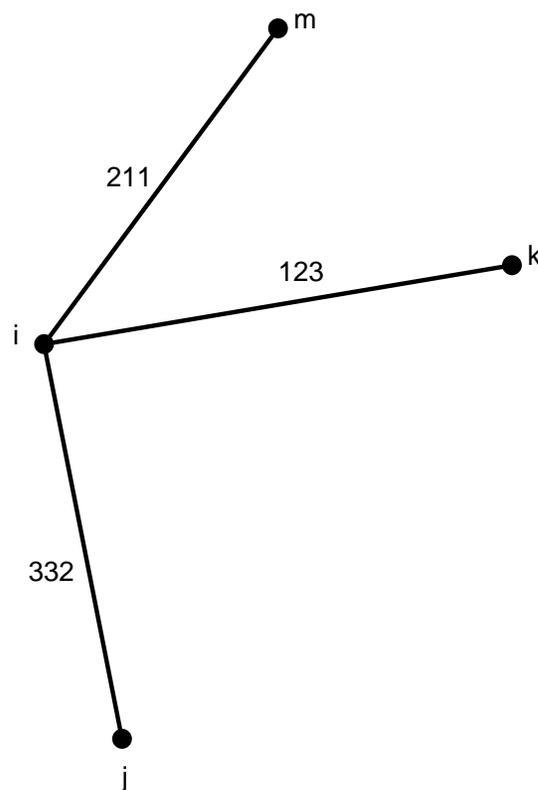
build_edge (2)

the edge derived type:

- it is to store edges in a triangulation
- it is used to update (in a generalized Delaunay sense) the triangulation of the 3D points on the surfaces
- for each edge:
 - $n1, n2$ are the node numbers defining the edge
 - $t1, t2$ are the triangle numbers on either side of the edge going from $n1$ to $n2$, $t1$ is to the left and $t2$ is to the right
 - $m1, m2$ are the node numbers of the two other nodes making $t1$ and $t2$



build_edge (3)



`nedgepernode(i)=3`

`nodeedgenumber(1,i)=211`

`nodeedgenumber(2,i)=123`

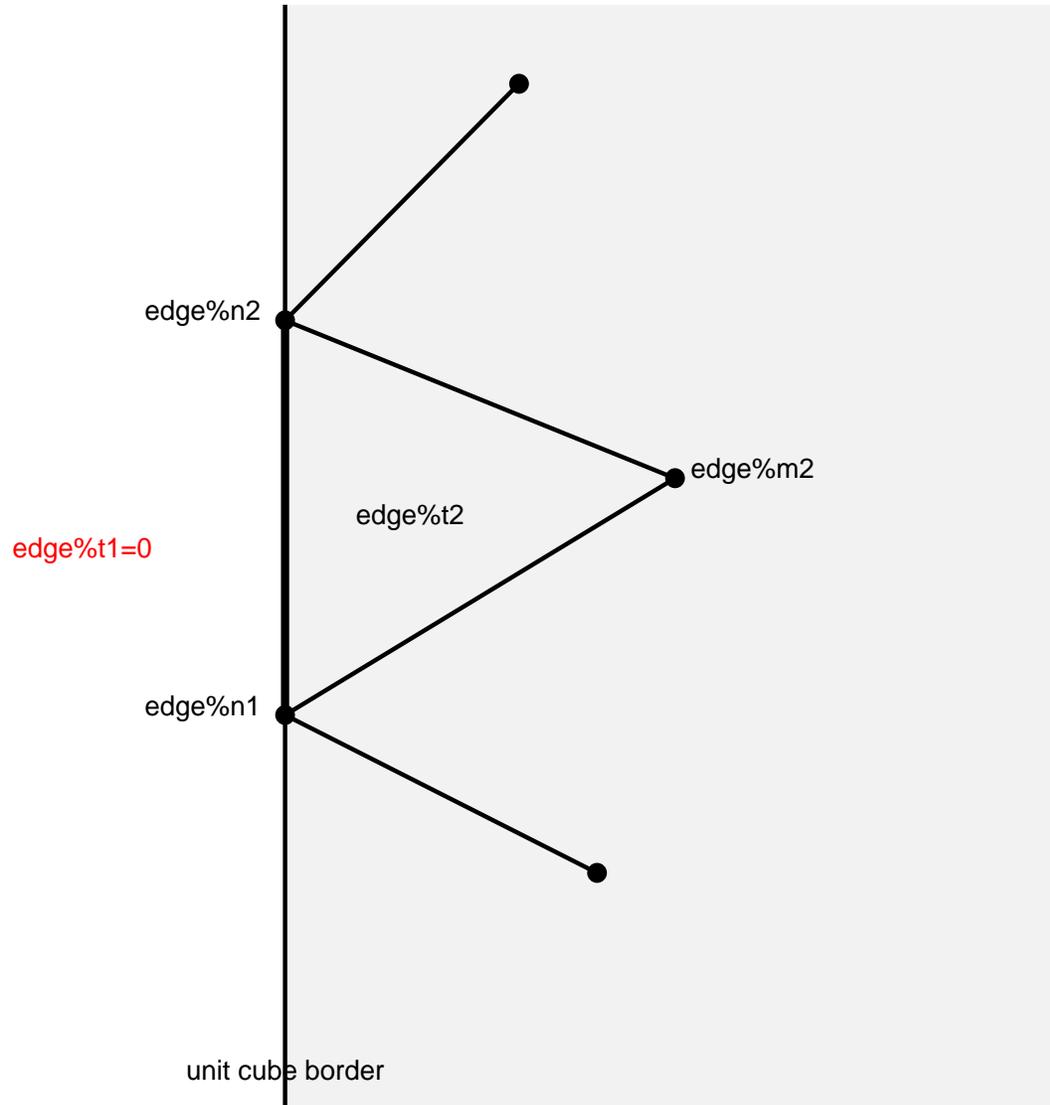
`nodeedgenumber(3,i)=332`

`nodenodenummer(1,i)=m`

`nodenodenummer(2,i)=k`

`nodenodenummer(3,i)=j`

build_edge (4)



build_surface_octree

```
subroutine build_surface_octree (surface,olsf,leveluniform_oct,levelmax_oct,  
                                criterion,anglemax,ismooth)
```

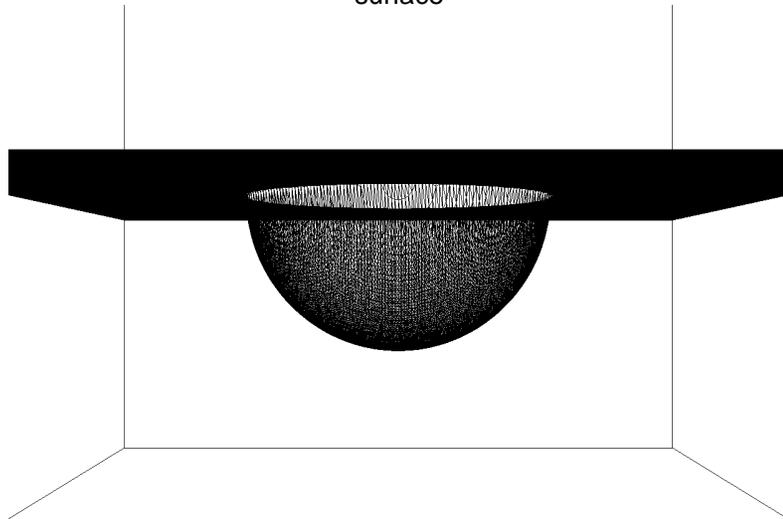
- arguments
 - surface is the surface represented by particles
 - olsf is the octree that we must build (it has already been dimensioned to noctreemax in the main program)
 - leveluniform_oct is the base level for the octree
 - levelmax_oct is the maximum level allowed for leaves of the octree
 - if criterion=1 all triangles of the surface must be entirely comprised in leaf of maximum order. If criterion=2 the octree is discretized to a level computed from the angle made by the normals; when the angle varies between 0 and angle max, the level varies from levelmin to levelmax.
 - ismooth determines whether additional smoothing is to be performed. (ismooth is set in the input.txt file). Additional smoothing imposes that no leaf touches other leaves that are different in size by more than one level.
- This subroutine builds the olsf octree to carry a level set function used to represent a surface. The criterion is used to define the octree in the vicinity of the surface:
 - criterion 1 corresponds to imposing that each leaf that is cut by a triangle of any surface will be refined to levelmax_oct.
 - criterion 2 corresponds to imposing that discretization is proportional to the curvature of the surface; curvature is calculated from the local divergence of the normals.

$$level = \max \left[level, L_u + nint \left(\min \left(\frac{angle}{anglemax}, 1 \right) (L_{max} - L_u) \right) \right]$$

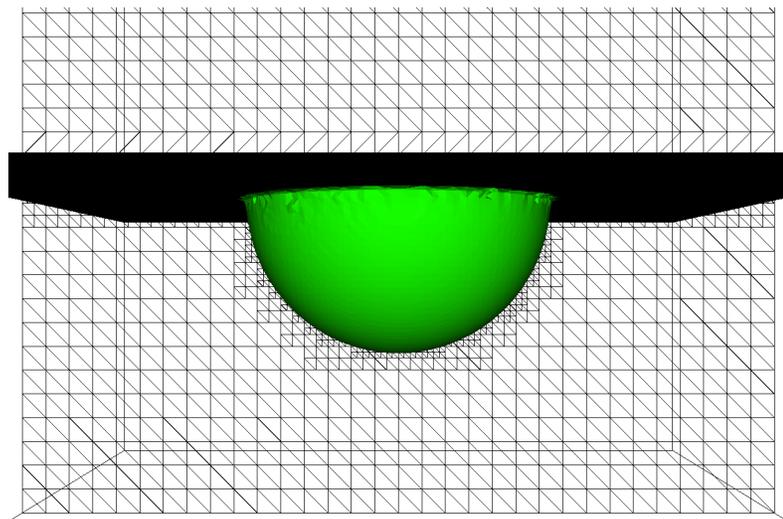
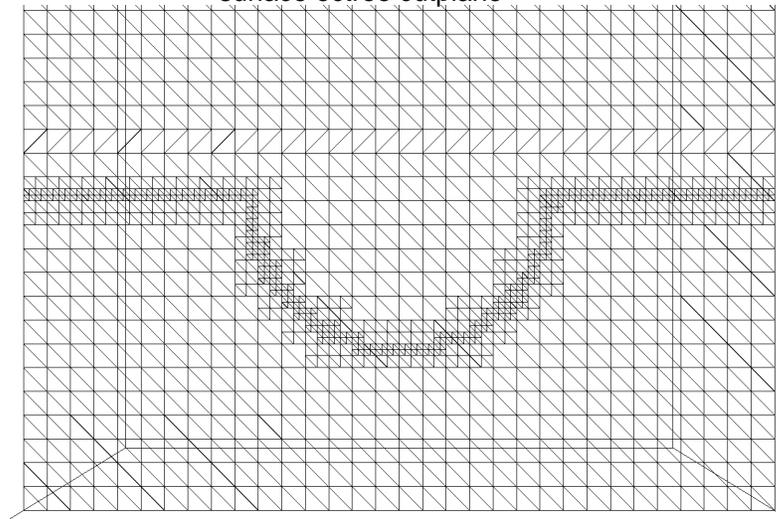
- criterion 3 corresponds to imposing that all leaves that contain at least one particle of any surface is at levelmax_oct.

build_surface_octree: criterion=1

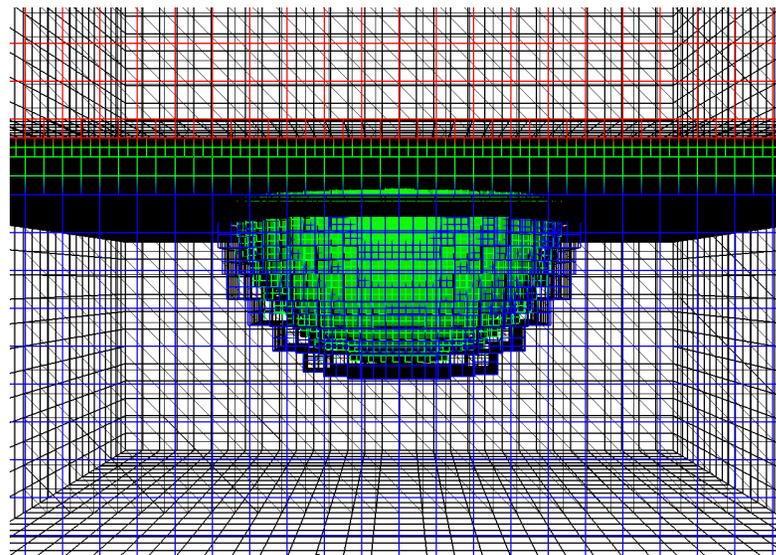
surface



surface octree cutplane



Isf isosurface



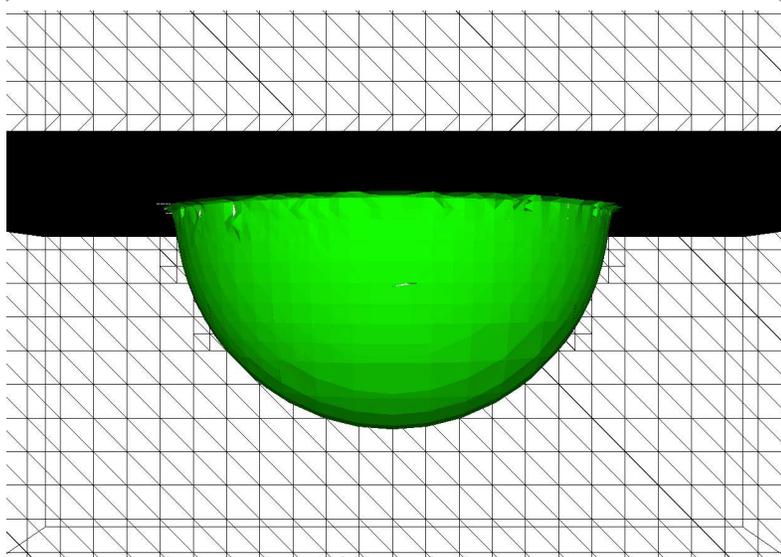
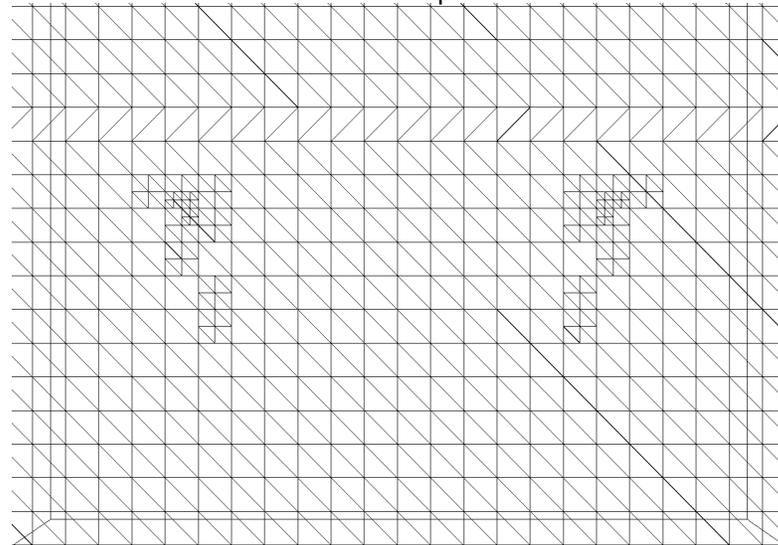
surface octree

build_surface_octree: criterion=2

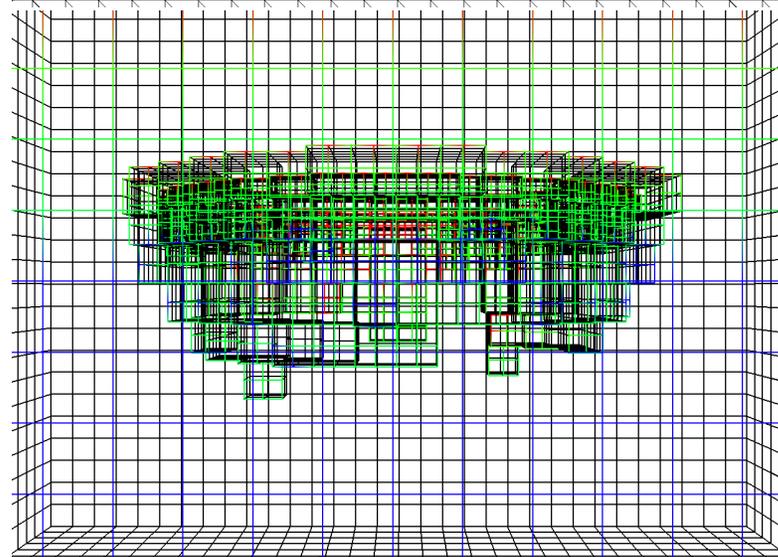
surface



surface octree cutplane



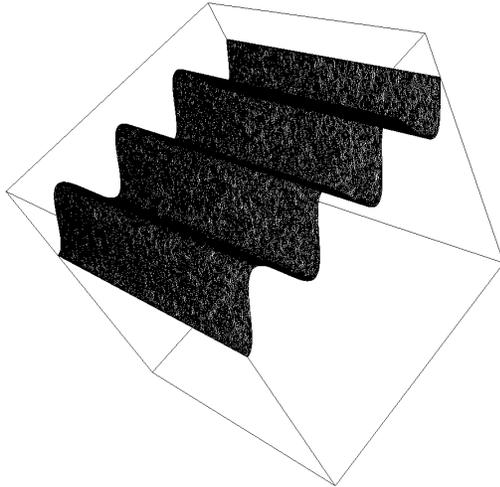
Isf isosurface



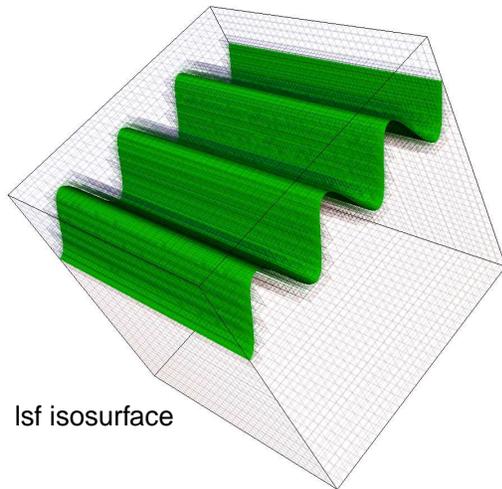
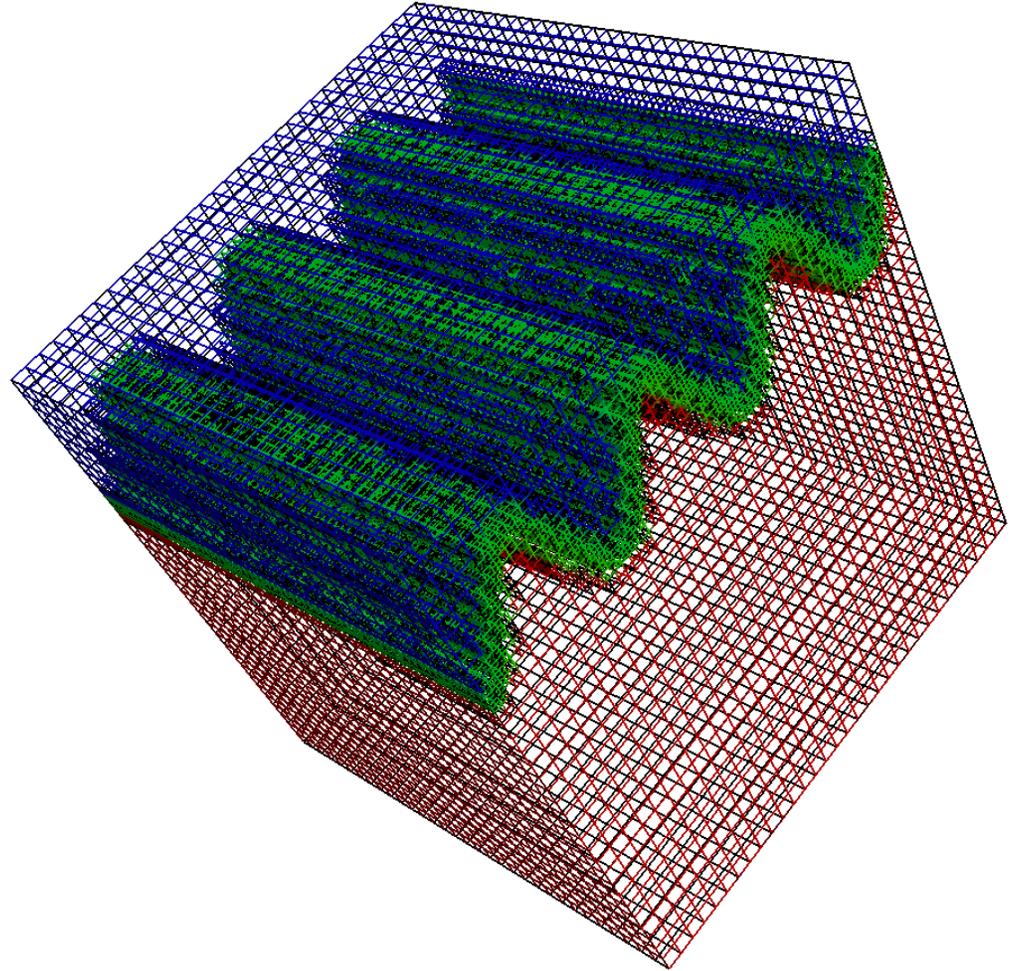
surface octree

build_surface_octree: criterion=3

surface (cosinus)



surface octree



Isf isosurface

build_system.f90

```
subroutine build_system (nleaves, nface, nnode, nz, a, irn, jcn, idg, icon, iface, ndof, mpe, x, y, z, kfix, lsf, nlsf,
mat, nmat, materialn, dt, u, v, w, temp, pressure, strain, rhs, penalty, tempscale, vo,
levelcut, levelapprox, iproc_col, eviscosity, istep, iter, iter_nl, doDoRuRe, forces)
```



arguments

- nleaves
- nface
- nnode
- nz
- a
- irn, jcn, idg
- icon
- iface
- ndof
- mpe
- x, y, z
- kfix
- lsf, nlsf
- mat, nmat
- materialn
- dt
- u, v, w
- temp, pressure, strain
- rhs
- penalty
- tempscale
- vo

calculate_lsf.f90 (1)

```
subroutine calculate_lsf      (lsf,octree_lsf,noctree_lsf,icon_lsf,  
                             nleaves_lsf,na_lsf,xl,yl,zl  
                             icon,nelem,na,levelmax_oct)
```



input

- octree_lsf(noctree_lsf) is the octree on which the lsf is to be built known
- icon_lsf(8,nleaves_lsf) is the node-leaf connectivity of the octree
- xl(na),yl(na),zl(na) are the coordinates of the na particles on the surface
- icon(3,nelem) is the connectivity matrix defining the triangulation
- levelmax_oct is the maximum level of refinement of the octree



output

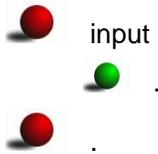
- lsf(na_lsf) is the values of the lsf at the nodes (main output of the routine)



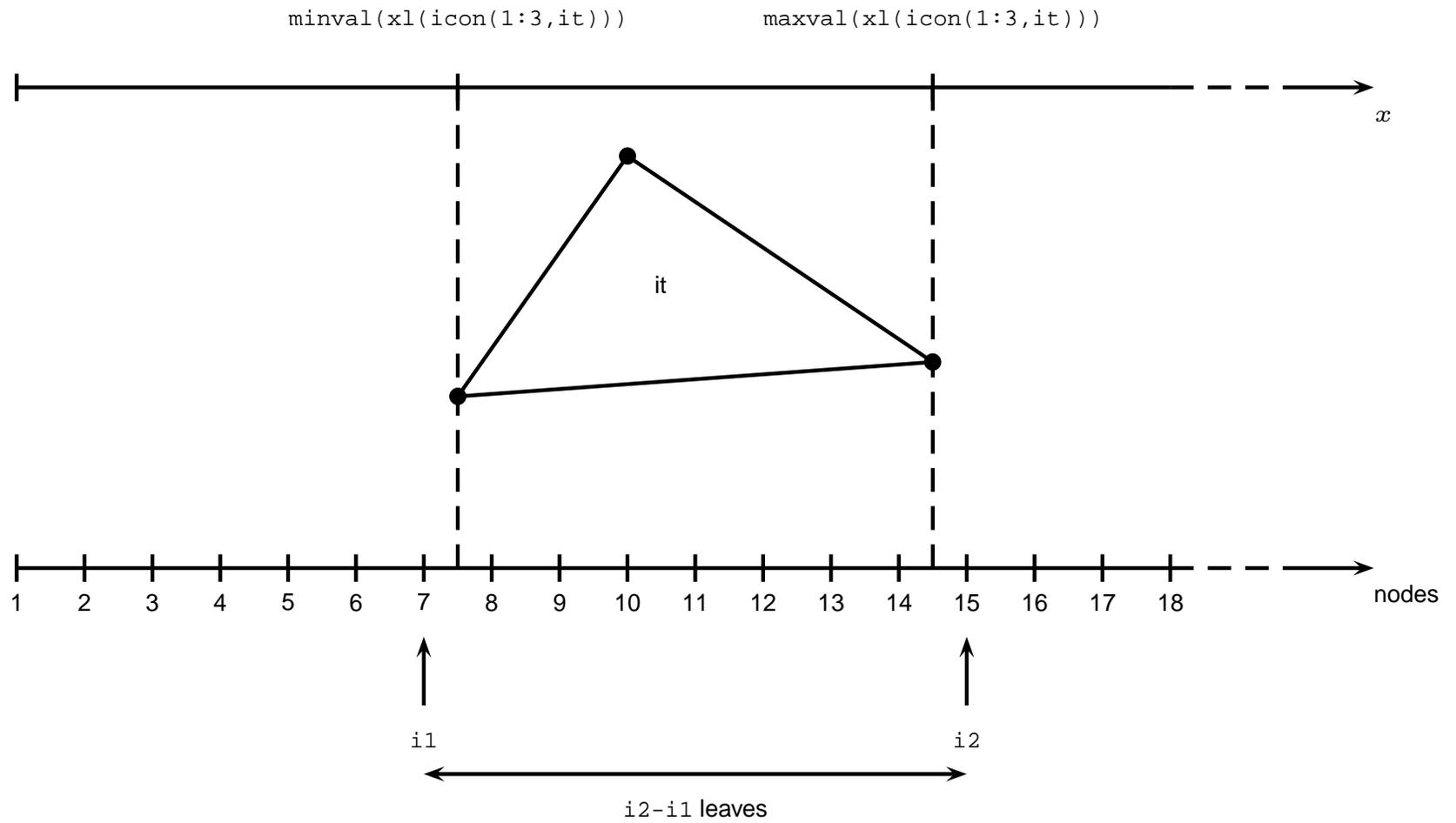
function: this subroutine calculates the lsf (level set function) on an octree from the position of a series of particles connected by a 2D triangulation

calculate_lsf.f90 (2)

```
subroutine distance_to_triangle (x1,y1,z1,x2,y2,z2,x3,y3,z3,  
                                x1n,y1n,z1n,x2n,y2n,z2n,x3n,y3n,z3n,x0,y0,z0,dist)
```



calculate_lsf



$$P_1 = (x_1, y_1, z_1)$$

$$P_2 = (x_2, y_2, z_2)$$

$$P_3 = (x_3, y_3, z_3)$$

$$\mathbf{n}_1 = (x_{1n}, y_{1n}, z_{1n})$$

$$\mathbf{n}_2 = (x_{2n}, y_{2n}, z_{2n})$$

$$\mathbf{n}_3 = (x_{3n}, y_{3n}, z_{3n})$$

barycenter

$$x_C = (x_1 + x_2 + x_3)/3$$

$$y_C = (y_1 + y_2 + y_3)/3$$

$$z_C = (z_1 + z_2 + z_3)/3$$

$$x_c = (x_1 + x_2 + x_3) / 3.d0$$

$$y_c = (y_1 + y_2 + y_3) / 3.d0$$

$$z_c = (z_1 + z_2 + z_3) / 3.d0$$

$$x_n = (y_2 - y_1) * (z_3 - z_1) - (y_3 - y_1) * (z_2 - z_1)$$

$$y_n = (z_2 - z_1) * (x_3 - x_1) - (z_3 - z_1) * (x_2 - x_1)$$

$$z_n = (x_2 - x_1) * (y_3 - y_1) - (x_3 - x_1) * (y_2 - y_1)$$

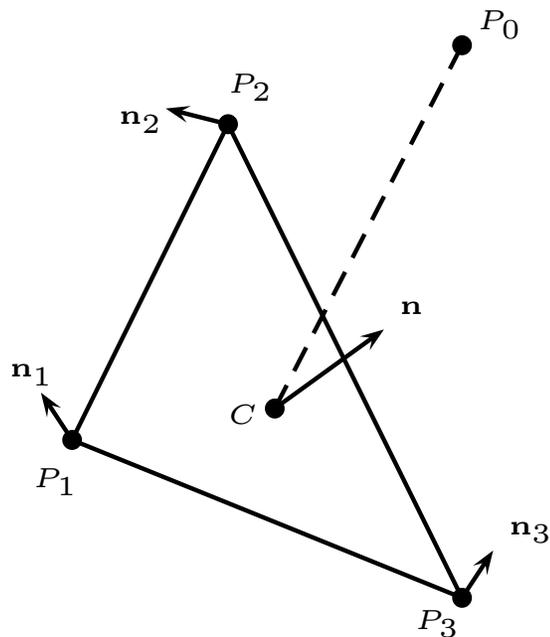
$$xyzn = \text{sqrt}(x_n^2 + y_n^2 + z_n^2)$$

$$x_n = x_n / xyzn$$

$$y_n = y_n / xyzn$$

$$z_n = z_n / xyzn$$

$$\alpha = x_n * (x_0 - x_c) + y_n * (y_0 - y_c) + z_n * (z_0 - z_c)$$

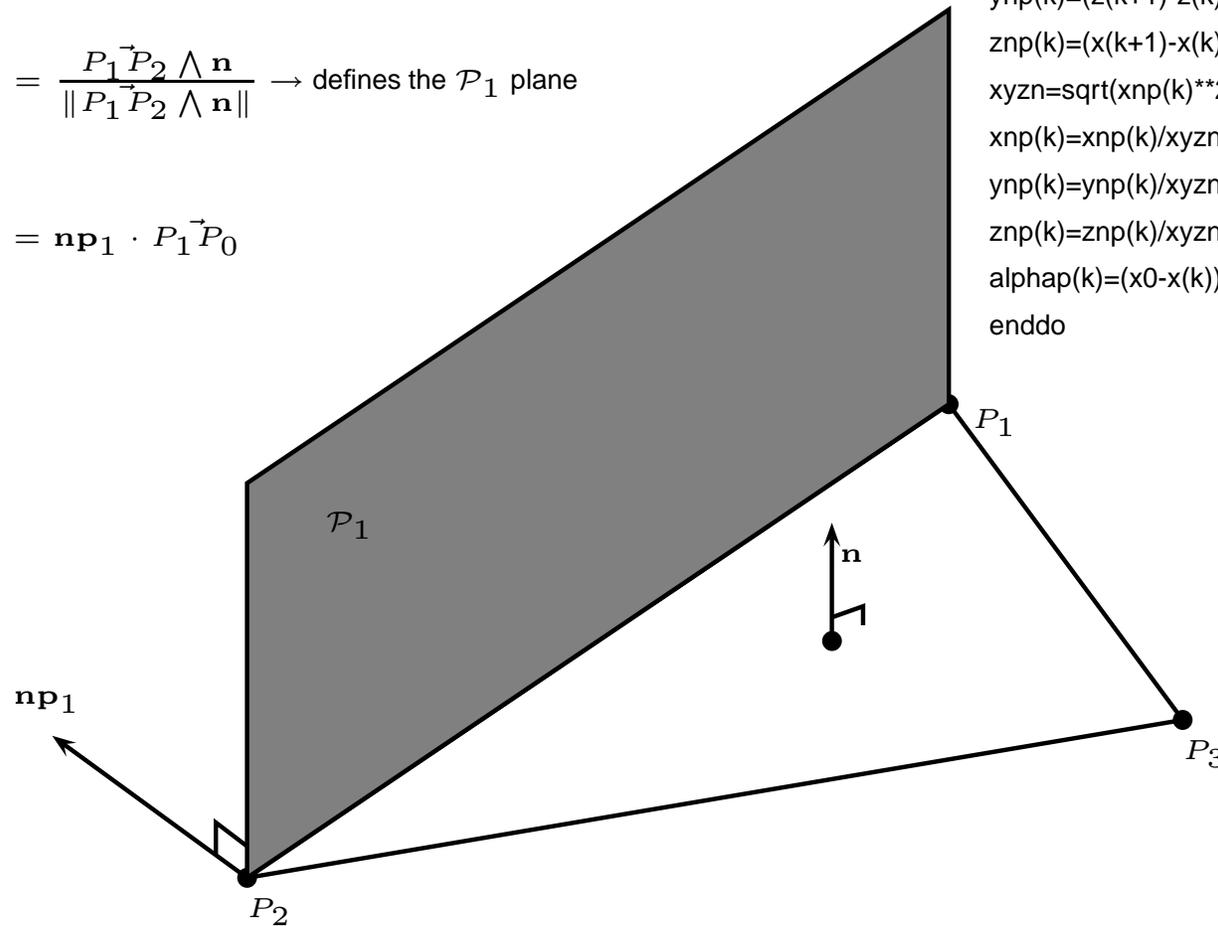


$$\mathbf{n} = \frac{P_1 \vec{P}_2 \wedge P_1 \vec{P}_3}{\|P_1 \vec{P}_2 \wedge P_1 \vec{P}_3\|}$$

$$\alpha = \mathbf{n} \cdot C \vec{P}_0$$

$$\mathbf{np}_1 = \frac{P_1 \vec{P}_2 \wedge \mathbf{n}}{\|P_1 \vec{P}_2 \wedge \mathbf{n}\|} \rightarrow \text{defines the } \mathcal{P}_1 \text{ plane}$$

$$\alpha_{p1} = \mathbf{np}_1 \cdot P_1 \vec{P}_0$$



do k=1,3

xnp(k)=(y(k+1)-y(k))*zn-(z(k+1)-z(k))*yn

ynp(k)=(z(k+1)-z(k))*xn-(x(k+1)-x(k))*zn

znp(k)=(x(k+1)-x(k))*yn-(y(k+1)-y(k))*xn

xyzn=sqrt(xnp(k)**2+ynp(k)**2+znp(k)**2)

xnp(k)=xnp(k)/xyzn

ynp(k)=ynp(k)/xyzn

znp(k)=znp(k)/xyzn

alphap(k)=(x0-x(k))*xnp(k)+(y0-y(k))*ynp(k)+(z0-z(k))*znp(k)

enddo

check_delaunay.f90

```
subroutine check_delaunay (ed,nedge,x,y,z,xn,yn,zn,nnode,icon,nelem,  
nedgepernode,nodenodenummer,nodeedgenumber,nnmax, distance_exponent)
```



arguments

- type (edge) ed(nedge)
- nedge
- x(nnode),y(nnode),z(nnode)
- xn(nnode),yn(nnode),zn(nnode)
- nnode
- icon(3,nelem)
- nelem
- nedgepernode(nnode)
- nodenodenummer(nnmax,nnode)
- nodeedgenumber(nnmax,nnode)
- nnmax
- distance_exponent

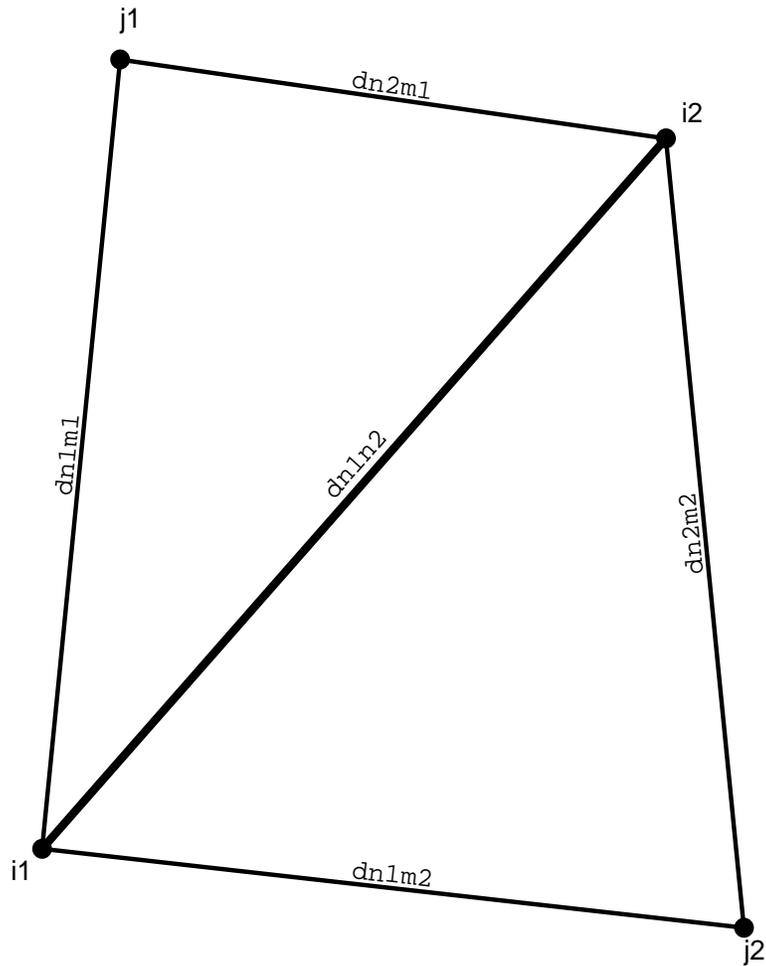


This subroutine calculates the Delaunay triangulation around a set of points located on a surface in three dimensional space. A non Euclidian metrics is used that takes into account the divergence of normals attached to the points such that folding over of the surface is impossible.

The Delaunay triangulation is UPDATED, not reconstructed at each time step.

First the delaunay triangulation is checked along each edge between two adjacent triangles using a generalized in-circle test based on the calculation of distances only; the test is based on an angle. There the test is positive, edge flipping takes place and all the arrays are permuted (icon, edge, and subsidiary arrays)

check_delaunay (1)



compute_convergence_criterion.f90

```
subroutine compute_convergence_criterion (osolve,ov,vo,istep,iter,iter_n1,tol,refine_level,debug,velocity_converged)
```



arguments

- type (octreesolve) osolve
- type (octreev) ov
- type (void) vo
- istep,iter,iter_n1
- tol,maxu,maxv,maxw
- refine_level
- debug
- velocity_converged
- doDoRuRe



This subroutine computes a convergence criterion based on the difference between the velocity field obtained at this iteration (osolve) and the previous velocity field (ov). This is based on the 2-norm.

compute_dhdx_dhdy_dhdz.f90 (1)

```
subroutine compute_dhdx_dhdy_dhdz (mpe, r, s, t, x, y, z, dhdx, dhdy, dhdz, volume)
```

arguments

-  mpe
-  r,s,t
-  x,y,z
-  dhdx,dhdy,dhdz
-  volume

compute_dhdx_dhdy_dhdz.f90 (2)

Given a point (r, s, t) in a leaf (in local coordinates) - it usually is a Gauss integratoin point- the shape functions are computed:

$$\begin{aligned}h_1(r_i, s_i, t_i) &= (1 - r_i) * (1 - s_i) * (1 - t_i)/8 \\h_2(r_i, s_i, t_i) &= (1 + r_i) * (1 - s_i) * (1 - t_i)/8 \\h_3(r_i, s_i, t_i) &= (1 - r_i) * (1 + s_i) * (1 - t_i)/8 \\h_4(r_i, s_i, t_i) &= (1 + r_i) * (1 + s_i) * (1 - t_i)/8 \\h_5(r_i, s_i, t_i) &= (1 - r_i) * (1 - s_i) * (1 + t_i)/8 \\h_6(r_i, s_i, t_i) &= (1 + r_i) * (1 - s_i) * (1 + t_i)/8 \\h_7(r_i, s_i, t_i) &= (1 - r_i) * (1 + s_i) * (1 + t_i)/8 \\h_8(r_i, s_i, t_i) &= (1 + r_i) * (1 + s_i) * (1 + t_i)/8\end{aligned}$$

Their derivatives with respect to r, s, t are then calculated as follows:

$$\begin{aligned}\left. \frac{\partial h_1}{\partial r} \right|_i &= -(1 - s_i) * (1 - t_i)/8 & \dots & \left. \frac{\partial h_8}{\partial r} \right|_i = (1 + s_i) * (1 + t_i)/8 \\ \left. \frac{\partial h_1}{\partial s} \right|_i &= -(1 - r_i) * (1 - t_i)/8 & \dots & \left. \frac{\partial h_8}{\partial s} \right|_i = (1 + r_i) * (1 + t_i)/8 \\ \left. \frac{\partial h_1}{\partial t} \right|_i &= -(1 - r_i) * (1 - s_i)/8 & \dots & \left. \frac{\partial h_8}{\partial t} \right|_i = (1 + r_i) * (1 + s_i)/8\end{aligned}$$

compute_dhdx_dhdy_dhdz.f90 (3)

We then need the Jacobian of the transformation $(x, y, z) \rightarrow (r, s, t)$.

$$\begin{pmatrix} \frac{\partial}{\partial r} \\ \frac{\partial}{\partial s} \\ \frac{\partial}{\partial t} \end{pmatrix} = \underbrace{\begin{pmatrix} \frac{\partial x}{\partial r} & \frac{\partial y}{\partial r} & \frac{\partial z}{\partial r} \\ \frac{\partial x}{\partial s} & \frac{\partial y}{\partial s} & \frac{\partial z}{\partial s} \\ \frac{\partial x}{\partial t} & \frac{\partial y}{\partial t} & \frac{\partial z}{\partial t} \end{pmatrix}}_{\mathbf{J}} \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{pmatrix}$$

Since $x = \sum_{k=1}^8 h_k x_k$, $y = \sum_{k=1}^8 h_k y_k$ and $z = \sum_{k=1}^8 h_k z_k$, we have for instance $\frac{\partial x}{\partial r} = \sum_{k=1}^8 \frac{\partial h_k}{\partial r} x_k$.

One then computes \mathbf{J}^{-1} . \mathbf{J} is a 3×3 matrix, so we simply resort to a simple matrix inversion algorithm. We first compute

volume = $\text{Det}[J] = J_{1,1} J_{2,2} J_{3,3} + J_{1,2} J_{2,3} J_{3,1} + J_{2,1} J_{3,2} J_{1,3} - J_{1,3} J_{2,2} J_{3,1} - J_{1,2} J_{2,1} J_{3,3} - J_{2,3} J_{3,2} J_{1,1}$,

and then \mathbf{J}^{-1} .

Finally,

$$\begin{pmatrix} \frac{\partial h}{\partial x} \\ \frac{\partial h}{\partial y} \\ \frac{\partial h}{\partial z} \end{pmatrix} = \mathbf{J}^{-1} \begin{pmatrix} \frac{\partial h}{\partial r} \\ \frac{\partial h}{\partial s} \\ \frac{\partial h}{\partial t} \end{pmatrix}$$

compute_divergence.f90

```
subroutine compute_divergence (nleaves, nnode, mpe, icon, xnode, ynode, znode, unode, vnode, wnode, voleaf, istep, iter, doDo
```

arguments

-  nleaves:
-  nnode:
-  mpe:
-  icon:
-  xnode,ynode,znode:
-  unode,vnode,znode:
-  voleaf:
-  istep,iter
-  doDoRuRe

 This subroutines computes the elemental divergence of the velocity field in every leaf.

compute_normals.f90

```
subroutine compute_normals (ns,x,y,z,nt,icon,xn,yn,zn)
```



arguments

- ns
- x(ns),y(ns),z(ns)
- nt
- icon(3,nt)
- xn(ns),yn(ns),zn(ns)



given a set of points, and a connectivity array of their triangulation, this routine computes the normal to each point through cross-products: at first one computes the normal to each triangle and add the vector to the three points that make the triangle, then one loops over the points, and normalises the vectors.

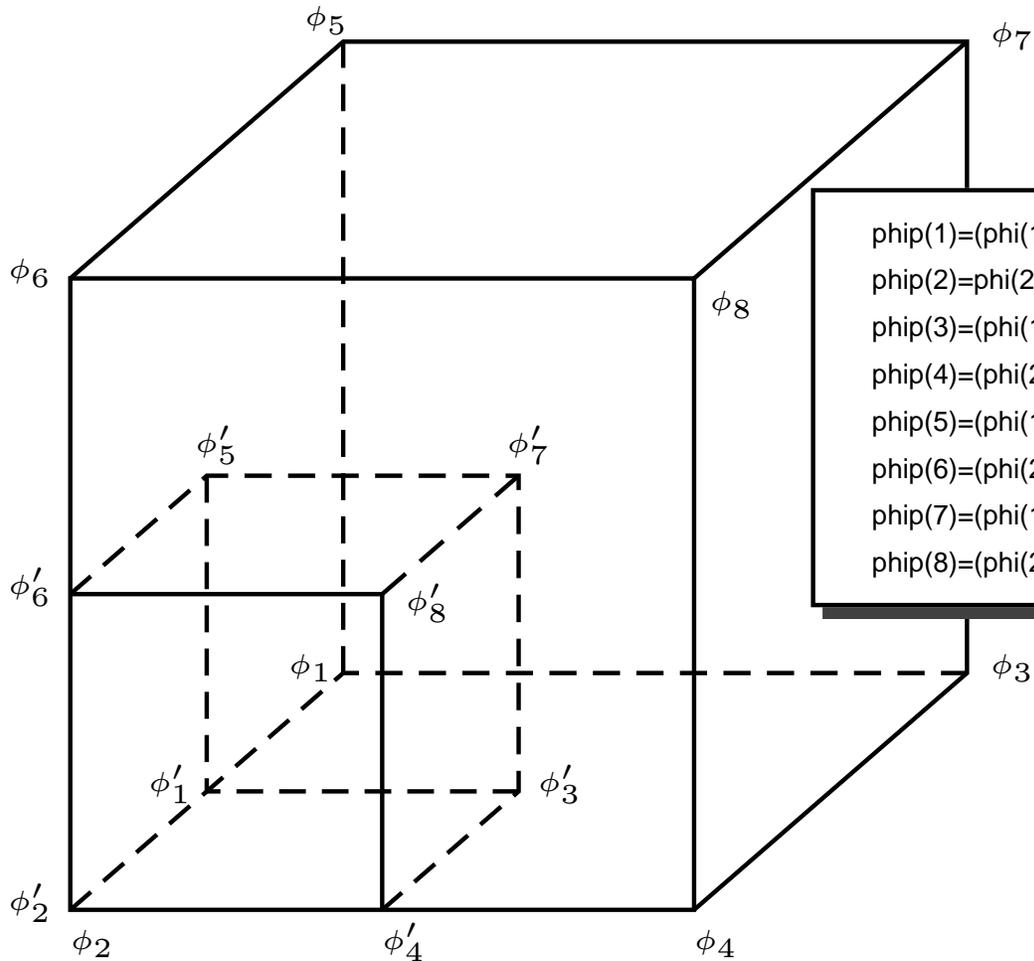
compute_positive_volume.f90

```
subroutine compute_positive_volume (phi,vol,levelmax)
```

- input
 - phi are the lsf values
 - levelmax is the max level for accuracy (power of 2)
- output
 - vol is the returned volume
- function: finds the volume of a cube that is defined by the positive value of a lsf known at the nodes of the cube

```
subroutine volume_lsf (phi,volp,level,levelmax)
```

volume_lsf



```
phip(1)=(phi(1)+phi(2))/2.d0
phip(2)=phi(2)
phip(3)=(phi(1)+phi(2)+phi(3)+phi(4))/4.d0
phip(4)=(phi(2)+phi(4))/2.d0
phip(5)=(phi(1)+phi(2)+phi(5)+phi(6))/4.d0
phip(6)=(phi(2)+phi(6))/2.d0
phip(7)=(phi(1)+phi(2)+phi(3)+phi(4)+phi(5)+phi(6)+phi(7)+phi(8))/8.d0
phip(8)=(phi(2)+phi(4)+phi(6)+phi(8))/4.d0
```

compute_pressure.f90

```
subroutine compute_pressure (nleaves, nface, nnode, icon, ndof, mpe, x, y, z, lsf, nlsf, mat, nmat, materialn,  
u, v, w, temp, pressure, strain, vo, levelcut, levelapprox, octree, noctree)
```

- arguments
 - nleaves
 - nface
 - nnode
 - icon
 - ndof
 - mpe
 - x,y,z
 - lsf,nlsf
 - mat,nmat
 - materialn
 - u,v,w
 - temp,pressure,strain
 - vo
 - levelcut
 - levelapprox
 - octree,noctree

This subroutine computes the elemental pressure in each leaf. It calls `pressure_cut`, which itself calls `make_pressure`. It is based on the same principle as `build_system`, `make_cut` and `make_matrix`.

compute_vol_work.f90

```
subroutine compute_vol_work (osolve,ov,istep,iter,iter_nl)
```

arguments

- osolve
- ov
- istep
- iter
- iter_nl

this routine computes the respective volume and work rate of each material associated to each surface.

The work rate is computed as follows

$$W = \int_V \sigma : \dot{\epsilon} dV$$

$$[\sigma] = M.L.T^{-2}, [\dot{\epsilon}] = T^{-1}, \Rightarrow [W] = ML^2T^{-2}.T^{-1}$$

create_surfaces.f90

```
subroutine create_surf (surface, is, debug)
```

arguments

-  surface
-  is
-  debug

-  This routine creates a surface through a set of points. Each of these points is assigned a normal. The topology of the points can either be based on a regular grid or on a random distribution. The Delaunay triangulation of these points is computed by means of the natural neighbours library. The surface is output in the file '/VTK/surf_XXXX_init.vtk'.

```
subroutine zpoints (ns, x, y, z, surface_type, levelt, sp01, sp02, sp03, sp04, sp05, sp06, sp07, sp08, sp09, sp10)
```

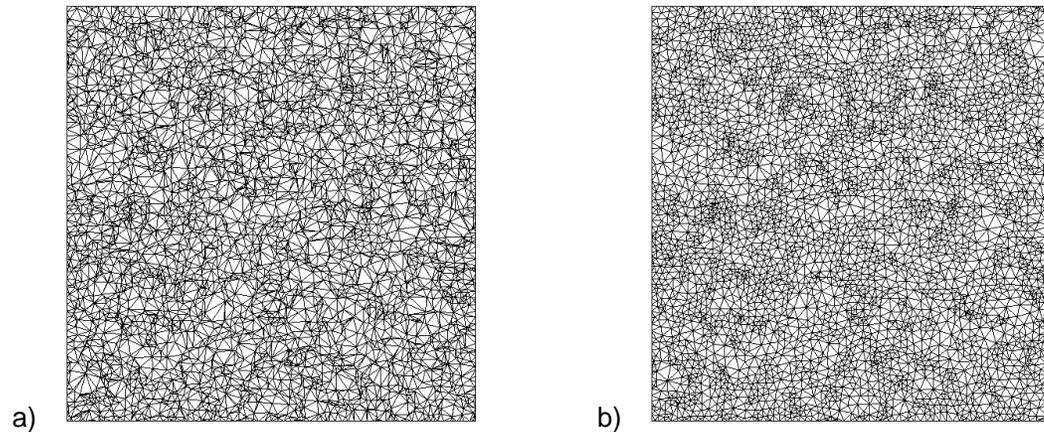
arguments

-  ns is the number of points on the surface
-  x,y,z are the arrays that contain the coordinates of the points
-  surface_type
-  levelt is the level of the surface
-  sp01,sp02,sp03,sp04,sp05,sp06,sp07,sp08,sp09,sp10

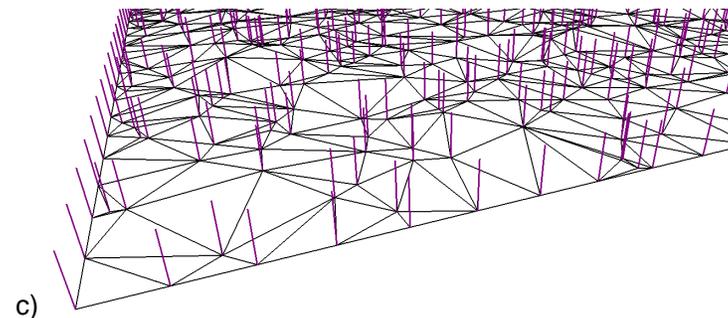
-  This routine computes the z coordinate of the points generated by create_surf according to the type of surface under consideration

create_surfaces.f90 (2)

The convex hull is first built. Then, random points in the unit square are generated (figure a), so that the total number of points is $(2^{level} + 1)^2$. The points are triangulated. The neighbouring list $nn(i)$ is built for each point i . Then the barycenter \mathbf{r}_i^b of the $nb(i)$ neighbours of point i is computed, and once this is done, for every point $\mathbf{r}_i = \mathbf{r}_i^b$ (except for the points on the hull and some others). This insures a certain level of smoothing (figure b).



The triangulation is computed again for these new positions. Having done this, we proceed to compute the normals at the points. In order to do so, each triangle is considered, two of its edges are used to calculate the unit normal to the triangle (by mean of the cross product) that is given (added) to each point forming the triangle. Finally, the normals are (re)-normalised (figure c).

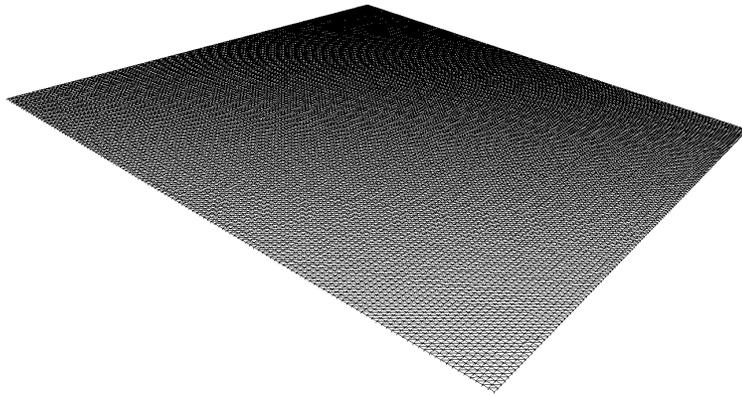


create_surfaces.f90 (3)

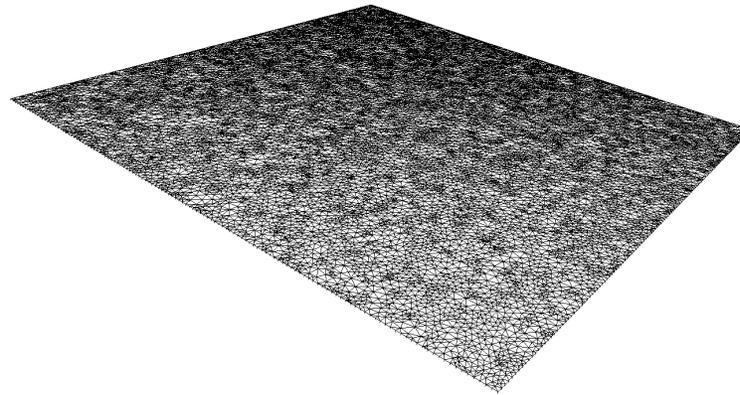
● `surface_type=1` : flat surface

● `sp01` is the z level

`surface%rand=.false.`



`surface%rand=.true.`

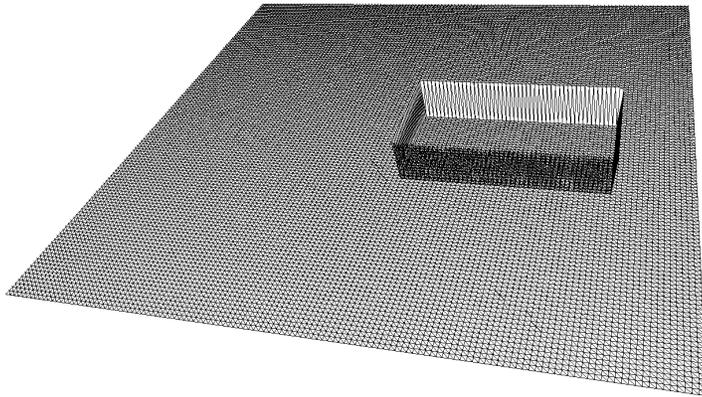


create_surfaces.f90 (4)

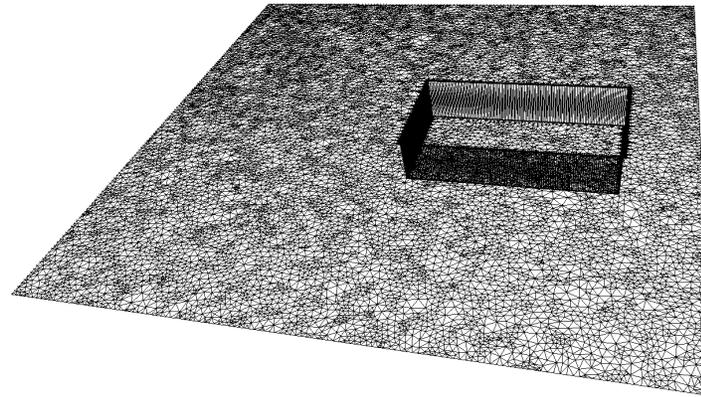
● `surface_type=2` : rectangular emboss

- `sp01` is the z level
- `sp02` and `03` are `x1,x2`
- `sp04` and `05` are `y1,y2`
- `sp06` is the thickness (positive or negative)

`surface%rand=.false.`



`surface%rand=.true.`

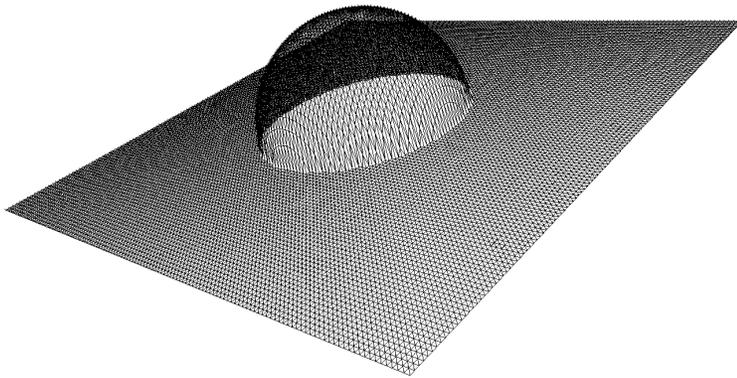


create_surfaces.f90 (5)

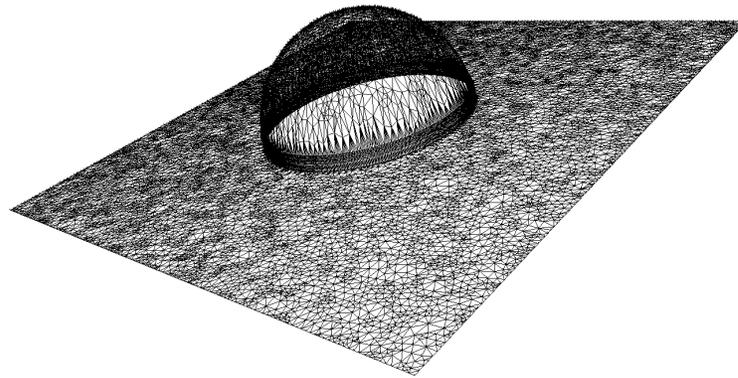
● **surface_type=3** : convex spherical emboss

- sp01 is the z level
- sp02 and 03 are x0,y0
- sp04 is the radius

surface%rand=.false.



surface%rand=.true.

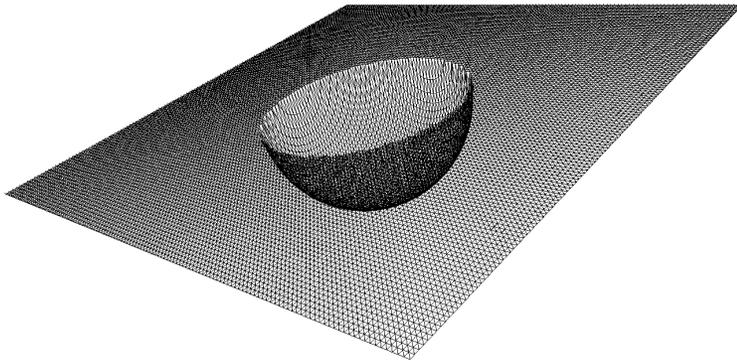


create_surfaces.f90 (6)

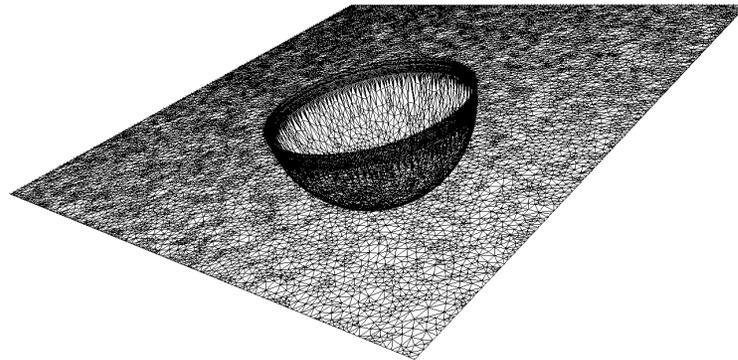
● `surface_type=4` : concave spherical emboss

- `sp01` is the z level
- `sp02` and `03` are `x0,y0`
- `sp04` is the radius

`surface%rand=.false.`



`surface%rand=.true.`

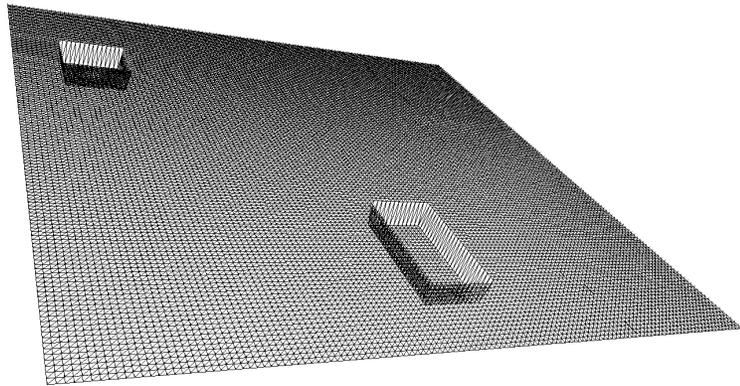


create_surfaces.f90 (7)

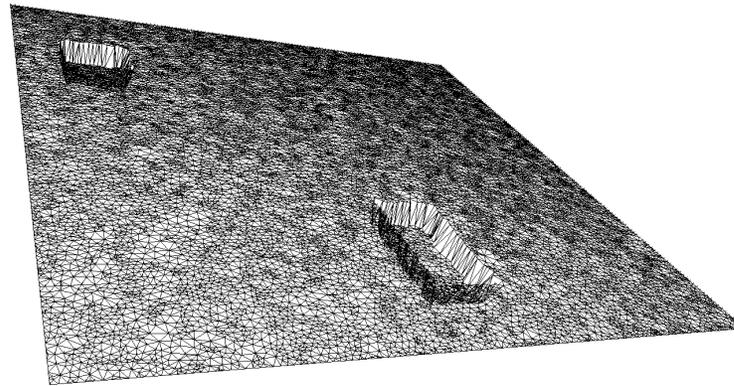
● `surface_type=5` : double rectangular emboss

- `sp01` is the z level
- `sp02` and `03` are `x1,x2`
- `sp04` and `05` are `x3,x4`
- `sp06` and `07` are `y1,y2`
- `sp08` and `09` are `y3,y4`
- `sp10` is the thickness

`surface%rand=.false.`



`surface%rand=.true.`

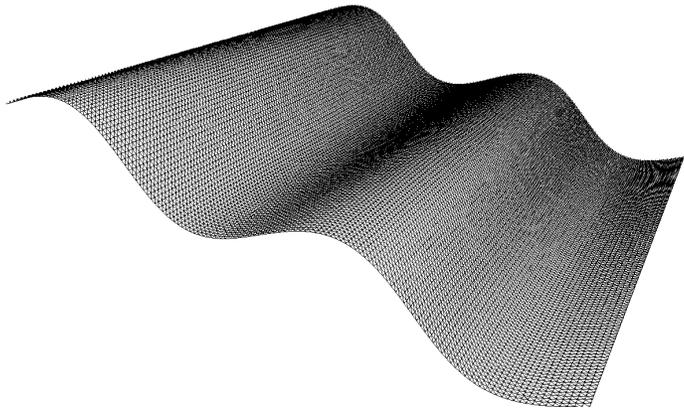


create_surfaces.f90 (8)

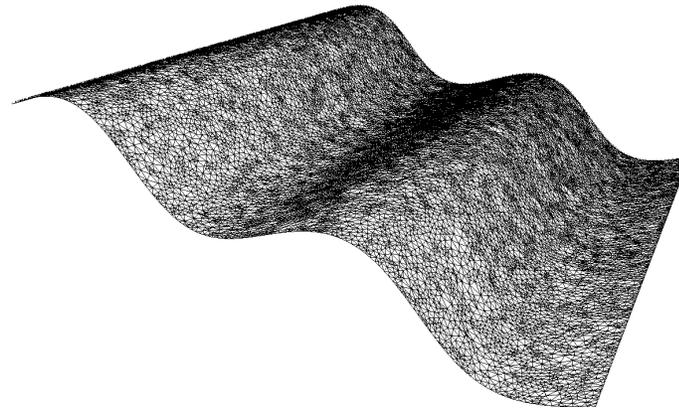
● `surface_type=6` : a sinus

- `sp01` is the z level
- `sp02` is the wavelength
- `sp03` is the amplitude

`surface%rand=.false.`



`surface%rand=.true.`

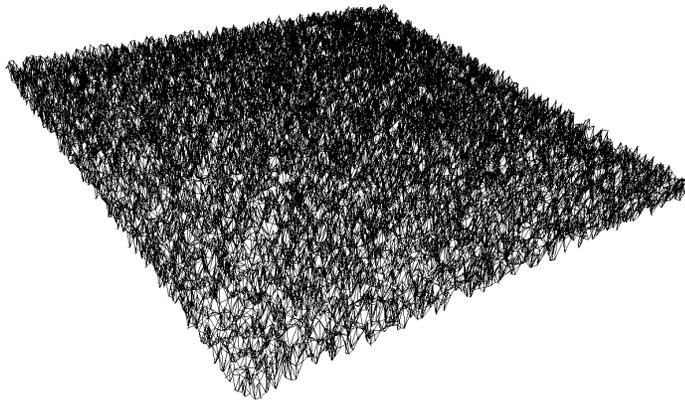


create_surfaces.f90 (9)

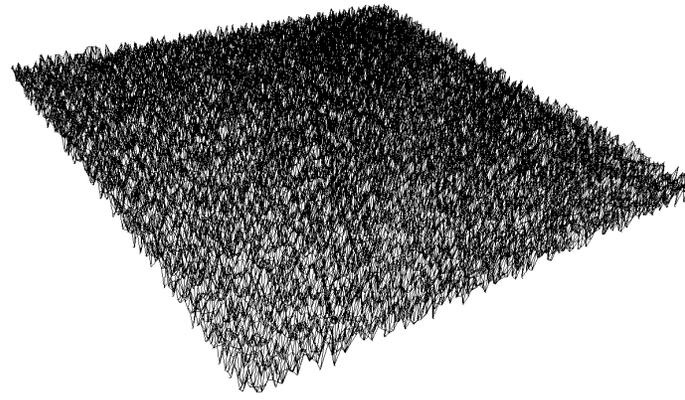
● `surface_type=7` : a noisy surface

- `sp01` is the z level
- `sp02` is the noise amplitude

`surface%rand=.false.`



`surface%rand=.true.`

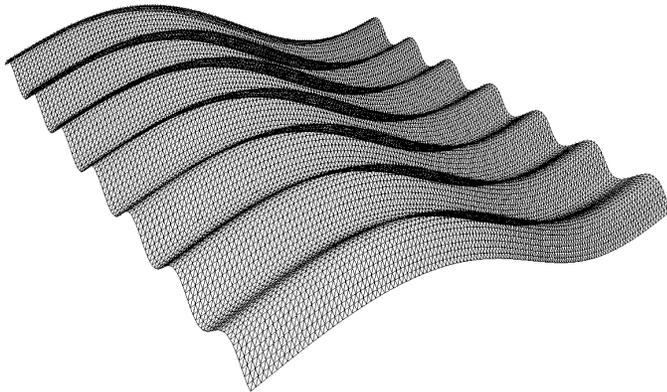


create_surfaces.f90 (10)

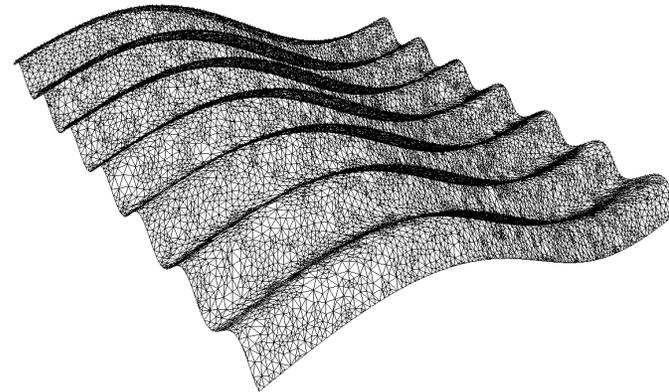
● `surface_type=8` : a double sinus

- `sp01` is the z level
- `sp02` is the x-wavelength
- `sp03` is the x-amplitude
- `sp04` is the y-wavelength
- `sp05` is the y-amplitude

`surface%rand=.false.`



`surface%rand=.true.`

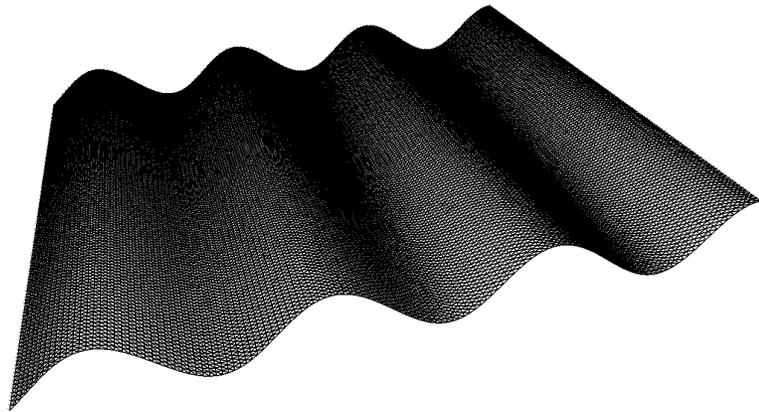


create_surfaces.f90 (11)

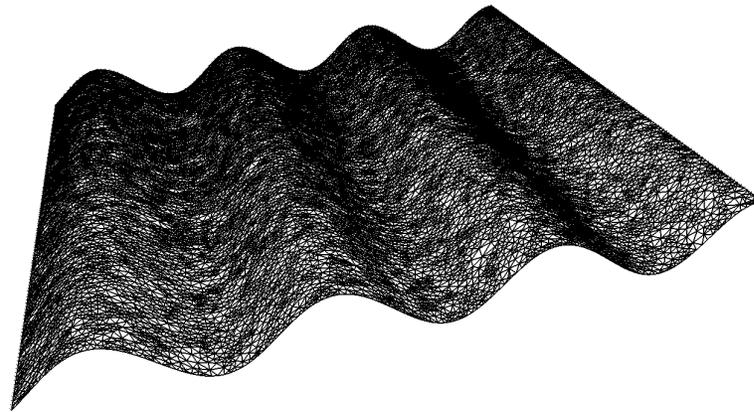
● `surface_type=9` : a cosinus

- `sp01` is the z level
- `sp02` is the wavelength
- `sp03` is the amplitude

`surface%rand=.false.`



`surface%rand=.true.`

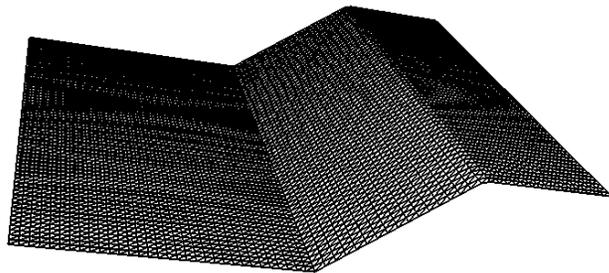


create_surfaces.f90 (12)

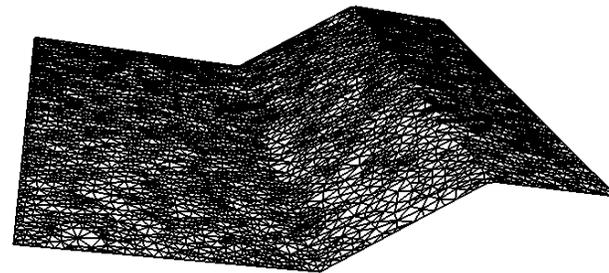
● surface_type=10 : slope

- sp01 is z_0 (base level)
- sp02 is y_0 (position where the slope starts)
- sp03 is ψ (angle of the slope)
- sp04 is δ (maximum thickness of layer)

surface%rand=.false.



surface%rand=.true.



define_bc

```
subroutine define_bc (infile,kfix,u,v,w,x,y,z,nnode,kfixt,temp,vo)
```



arguments

- infile is the a string containing the input.txt
- kfix
- u,v,w are the nodal velocities
- x,y,z are the positions of nodes
- nnode is the number of nodes
- kfixt
- temp is the nodal temperature
- vo is the structure that contains the void information



This subroutine is to be modified by the user to implement the boundary conditions. Here the user should define if the dof is fixed by setting $kfix((inode-1)*3+idof)$ to 1 (node inode and dof idof) and setting the corresponding value of u, v, or w to the set fixed value. Same operation for kfixt and temp but for a single dof. If no input file has been passed as argument to douar, the boundary conditions are those defined in the subroutine. If *input.xxxx* has been passed as argument to douar, then the user should modify accordingly the select case and write its own *define_bc_xxxx.f90*.

define_cloud

```
subroutine define_cloud (cl, irestart, restartfile)
```

- arguments
 - cl is the cloud
 - irestart
 - restartfile is the name of the restartfile if needed
- If irestart=0 this routine allocates and creates the cloud of points present in the system. Otherwise it reads from a user supplied file name the surfaces as they were at the end of a previous run. In this case, since the run output files contain all the octree+lsf+icloud+surface informations, the routine first reads dummy parameters until it gets to the real interesting cloud information.

define_ov

```
subroutine define_ov (ov,noctreemax,leveluniform_oct,irestart,restartfile,ztemp)
```

- arguments
 - ov is the object holding the octree
 - noctreemax is the maximum allowed octree size
 - leveluniform_oct is the minimum/uniform octree level
 - irestart is a flag to decide if this is a restart job or not
 - restartfile is the name of the restart file if it is needed
 - ztemp
- This routine creates a uniform octree that will be used to store the velocity for the next time step or reads it from a restart file.

define_surface

```
subroutine define_surface (surface,ns,irestart,restartfile,total,step,inc,current_time,debug)
```



arguments

- surface
- ns
- irestart
- restartfile
- total, step, inc
- debug



if irestart=0, this routine allocates and creates the ns surfaces present in the system. Otherwise, it reads from a user supplied file name the surfaces as they were at the end of a previous run. In this case, since the run output files contain all the octree+lsf+cloud+surface informations, the routine first reads dummy parameters until it gets to the real interesting surface information .

do_leaf_measurements.f90

```
subroutine compute_e2d_crit (osolve,ov,mpe,refine_criterion,debug,compute_qpgram)
```



arguments

- osolve:
- ov:
- mpe:
- refine_criterion:
- debug:
- compute_qpgram



This subroutine computes the following elemental quantities:

$$\begin{aligned} \text{e2d} &= \sqrt{J'_2(\dot{\epsilon})} \\ \text{e3d} &= \sqrt[3]{J'_3(\dot{\epsilon})} \\ \text{lode} &= \theta_l = \frac{1}{3} \sin^{-1} \left(-\frac{3\sqrt{3}}{2} \frac{J'_3}{(J'_2)^{3/2}} \right) \\ \text{q} &= 2\mu_{\text{eff}} \sqrt{J'_2(\dot{\epsilon})} \\ \text{crit} &= \end{aligned}$$

embed_surface_in_octree.f90

```
subroutine embed_surface_in_octree (osolve,noctreemax,leveluniform_oct,levelmax_oct,criterion,  
anglemax,surface,ismooth,is,ns,debug,istep,iter)
```



arguments

- osolve
- olsf
- noctreemax
- leveluniform_oct
- levelmax_oct
- criterion
- anglemax
- surface
- integer ismooth
- is
- ns
- debug
- istep,iter

erosion.f90

```
subroutine erosion (surface,olsf,is,zerosion)
```

- input
 - is is the number of the surface (0 playing a special role)
 - zerosion is the level of erosion (between 0 and 1)
- input/output
 - surface are the surface/sheet object
 - olsf is a octreelsf object containing the geometry of the current velocity octree/object
- output
- function: it is used to erode the surface 'surface' according to a user supplied algorithm. At the moment it simply 'shaves' the top of the model at a given height.

find_connectivity_dimension.f90

```
subroutine find_connectivity_dimension (nleaves,nface,nnode,vo,icon,iface,ndof,mpe,nz,tpl)
```



arguments

- nleaves: number of leaves
- nface: number of bad faces
- nnode: number of nodes
- vo: object containing the void information
- icon(mpe,nleaves): connectivity matrix between nodes and leaves
- iface(9,nface): bad face matrix
- ndof: number of dofs per node
- mpe: number of nodes per element
- nz: computed total number of nonzero numbers in the global FE matrix
- tpl(vo%nnode*ndof): topology array that is build in this routine; it has the dimension of the total number of dof and contains, per dof, the list of other dofs connected to the dof

find_connectivity.f90

```
subroutine find_connectivity (nleaves, nnode, vo, icon, ndof, mpe, nz, irn, jcn, idg, tpl)
```

- arguments
 - nleaves is the number of leaves/fe
 - nnode is the number of nodes
 - vo is the object containing the void information
 - icon is the connectivity matrix
 - ndof is the number of dofs per node
 - mpe is the number of nodes per element/leaf
 - nz is the total number of nonzero numbers in the global matrix
 - irn and jcn are built in this routine and are needed by MUMPS to locate the nonzero numbers of the global stiffness matrix
 - idg is simply the list of the diagonal elements in the global stiffness matrix once it is unrolled as a single dimension matrix/vector
 - tpl is the topology array that is build in find_connectivity_dimension.f90
- Having the topology dimension, we build three arrays irn, jcn, idg that ! are needed by MUMPS. This is a relatively easy operation once we know tpl.

find_connectivity_local.f90

```
subroutine find_connectivity_local (iprocol,n,idg,idg_loc,tpl,irn,jcn,nz,irn_loc,jcn_loc,nz_loc)
```



arguments

- iproc_col is a flag that determines which column of the matrix is dealt with by which processor
- n is the total ndof
- idg is simply the list of the diagonal elements in the global stiffness matrix once it is unrolled as a single dimension matrix/vector
- idg_loc is the local equivalent
- tpl is the topology array that is build in find_connectivity_dimension
- nz is the total number of nonzero numbers in the global matrix
- irn_loc, jcn_loc and nz_loc are the local equivalent to irn, jcn, nz



This routine is used to find the local (ie for each processor) irn and jcn arrays that are needed by MUMPS to locate the nonzero numbers of the global stiffness matrix once it has been split up between the processors. It also computes the total number of nonzero numbers in each of the sub-matrices (parts of the global matrix distributed to each processor).

find_processors.f90

```
subroutine find_processors ( iproc_col,n,tpl,nz_loc)
```

- arguments
 - iproc_col is a flag that determines which column of the matrix is dealt with by which processor; it is built in this routine.
 - n is the total ndof
 - tpl is the topology array that is build in find_connectivity_dimension
 - nz_loc is the total number of nonzero numbers in the local part of the global matrix; it is also built in this routine.
- ! This is where the division of the dofs between the processors is performed. ! Here we have decided to divide the dofs equally across the processors in an arbitrary way that is decided by the numbering of the nodes. Note that this numbering can be modified and/or optimized by a call to SLOAN's routines now built in the Octree library, but it is not done in the current version as it did not lead to better results.

find_void_nodes.f90

```
subroutine find_void_nodes (nleaves, nnode, nface, icon, iface, mpe, vo, lsf, nlsf, materialn)
```



arguments

- nleaves is the number of leaves/fe
- nface is the number of bad faces
- nnode is the number of nodes
- icon is the element-node connectivity matrix
- iface is the bad face connectivity matrix
- mpe is the number of nodes per elements
- vo is the structure containing the information on where the void is. It is constructed in this routine.
- lsf contains the nodal values of the nlsf level set functions
- materialn contains the material number associated to each lsf



This routine finds all nodes that are completely in the void, i.e. they are not connected by any element to a node that is not in the void. Those nodes are given a vo%node=1 flag; all others have vo%node=0. There are vo%nnode nodes that are not in the void.

It calculates the nodes that are in the fluid vo%influid=.true.

It also calculates the number of elements (vo%nleaves) and bad faces (vo%nface) that are not in the void and corresponding flag arrays vo%leaf and vo%face.

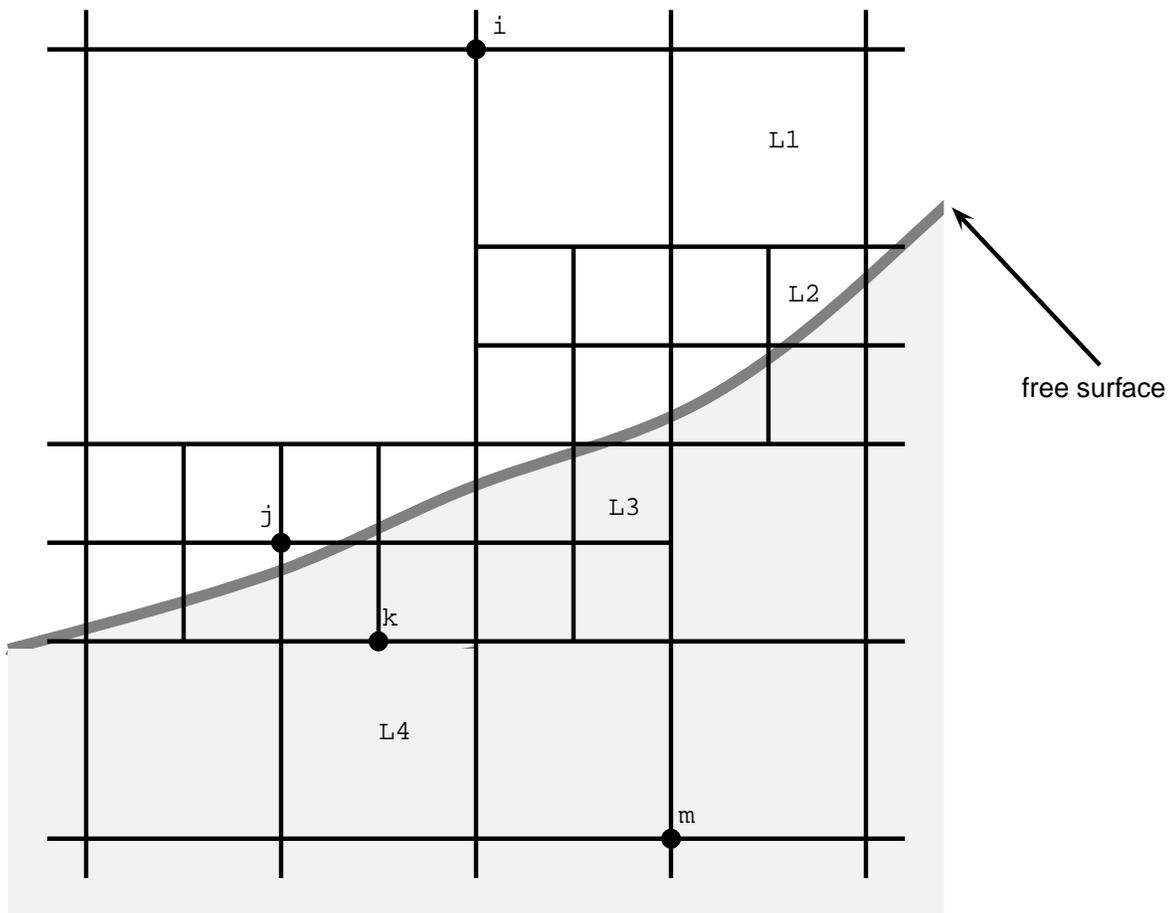
find_void_nodes.f90

The void derived type:

- it is to store information on node, leaves and faces that are in the void
- node=1 for nodes that are completely in the void (they are taken out of the equation set)
- leaf=1 for leaves that are completely in the void
- face=1 for faces that are completely in the void
- nnode is the number of nodes to be solved for (number of nodes not in the void)
- nleaves is the number of active leaves
- nface is the number of active faces
- rtf (restricted to full) is an array that provides the connectivity between the restricted and full node numbers (j=rtf(i) where i is restricted node number (1 to vo%nnode) and j is full node number (1 to nnode))
- ftr (full to restricted) is the complement
- influid is true for nodes that are in the fluid

```
type void
integer,dimension(:),pointer::node,leaf,face,ftr,rtf
logical,dimension(:),pointer::influid
integer nnode,nleaves,nface
end type void
```

find_void_nodes.f90



```
vo%influid(i)=F  
vo%node(i)=1  
vo%influid(j)=F  
vo%node(j)=0  
vo%influid(k)=T  
vo%node(k)=0  
vo%influid(m)=T  
vo%node(m)=0
```

```
vo%leaf(L1)=1  
vo%leaf(L2)=0  
vo%leaf(L3)=0  
vo%leaf(L4)=0
```

improve_osolve.f90

```
subroutine improve_osolve      (osolve,ov,refine_criterion,istep,iter,total,step,  
                               inc,refine_ratio,refine_level,debug,  
                               nboxes,boxes,ismooth,cube_faces,ref_on_faces)
```



arguments

- `ov` is the velocity octree containing the velocity solution
- `osolve` is the octree to be improved
- `refine_ratio` is a parameter read in `input.txt`. When multiplied by the maximum of the criterion previously computed, one obtains the threshold used to determine whether a leaf is to be refined or not.
- `refine_level` is the level at which the `osolve` octree is to be refined.



This routine calculates the second invariant of the strain rate, and a value for the criterion used by the `improve_octree` subroutine, inside each element by using the velocity information at the nodes. Then, the routine improves the `osolve` on which the solution is calculated.

Also, it refines the octree if the user has defined a box (or several boxes) in which the octree is artificially refined to a desired level. `boxes` and `nboxes` are read from the input file.

The routine then refines user supplied rectangular areas of each of the six faces of the cubic simulation domain. The information are stored in `cube_faces` read from the input file.

Finally, the octree is smoothed and super-smoothed if the user has set `ismooth` to 1.

initialize_temperature.f90

```
subroutine initialize_temperature (ov,ztemp)
```

- arguments
 - ov is the velocity octree containing the velocity/temperature solution
 - ztemp
- This routine initializes the temperature field to some basic conductive equilibrium, The temperature should be normalized between 0 and 1. This routine needs to be improved...

interpolate_leaf_quantities_on_nodes.f90

```
subroutine blabla (osolve,ov)
```

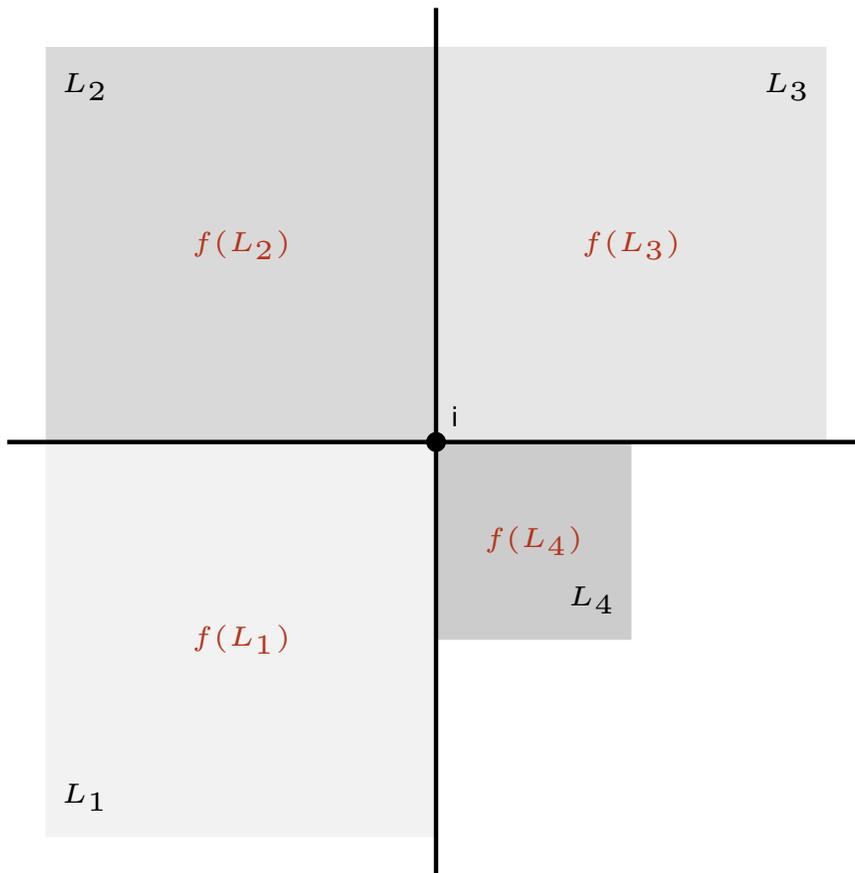
- arguments
 - osolve is the solution octree
 - ov is the velocity octree
- This routine interpolates various quantities between the values known in the leaves of the osolve octree and the nodes of the ov octree. This leads to some level of smoothing.

interpolate_leaf_quantities_on_nodes

$f(L)$: elemental field in leaf L

$V(L)$: volume of leaf L

$$f(i) = \frac{f(L_1)V(L_1)+f(L_2)V(L_2)+f(L_3)V(L_3)+f(L_4)V(L_4)}{V(L_1)+V(L_2)+V(L_3)+V(L_4)}$$



interpolate_ov_on_osolve.f90

```
subroutine blabla (osolve,ov)
```

-  arguments
 -  osolve is the solution octree
 -  ov is the velocity octree
-  This routine transfers the velocity solution from the velocity octree onto the osolve octree. It also performs a transfer of the pressure from the nodes of the ov octree to the leaves of the solution octree.
At the end, the velocity octree is redimensioned to fit the dimension of the solve octree and is thus ready for the next solution step.

make_cut.f90

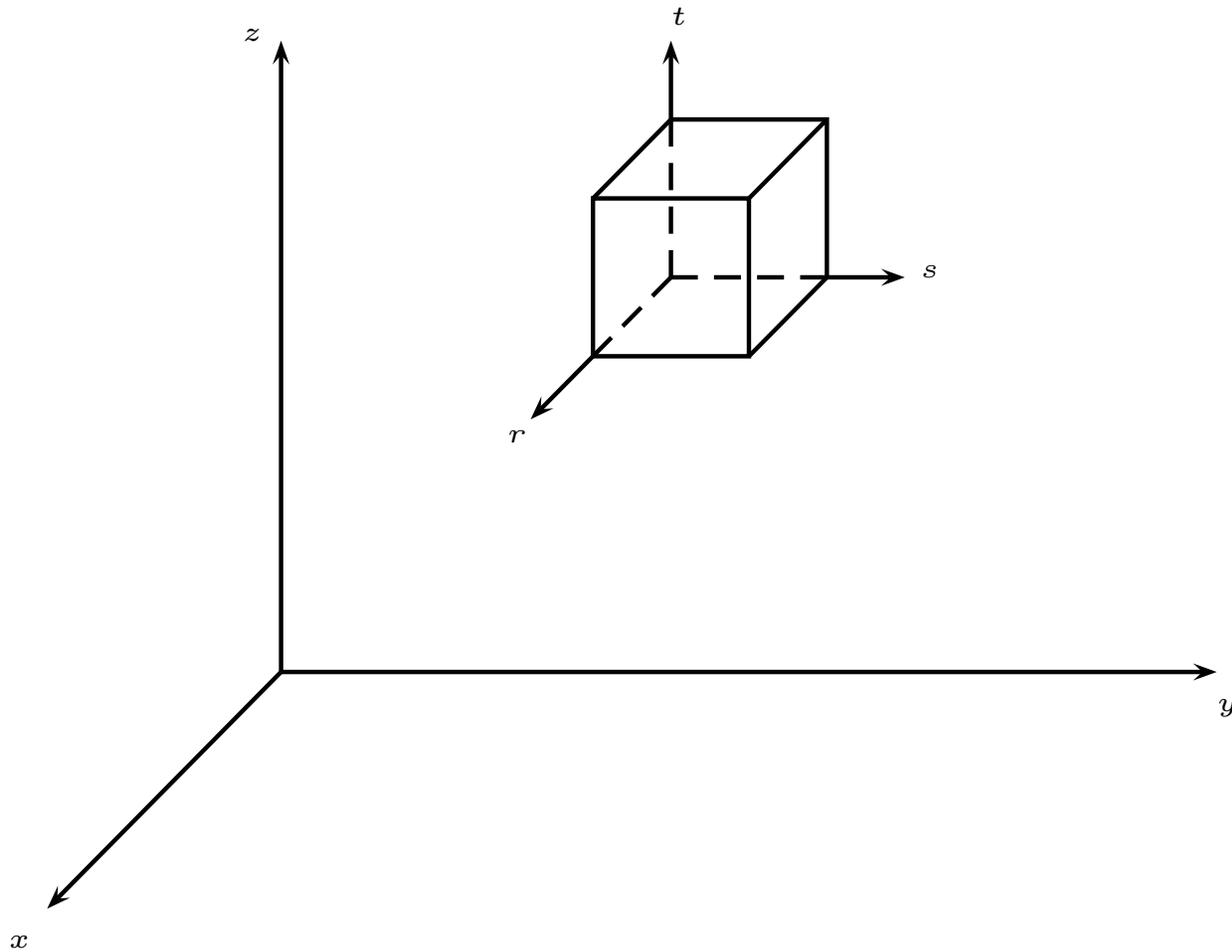
```
recursive subroutine make_cut      (level,levelmax,levelapprox,mpe,ndof,ael,bel,icon,x,y,z,penalty,tempscale,  
                                   kfix,mat,nmat,materialn,dt,u,v,w,temp,pressure,strain,nnode,  
                                   f,lsf,nlsf,r0,s0,t0,rst,icut,ileaves,eviscosity,forces)
```

 arguments

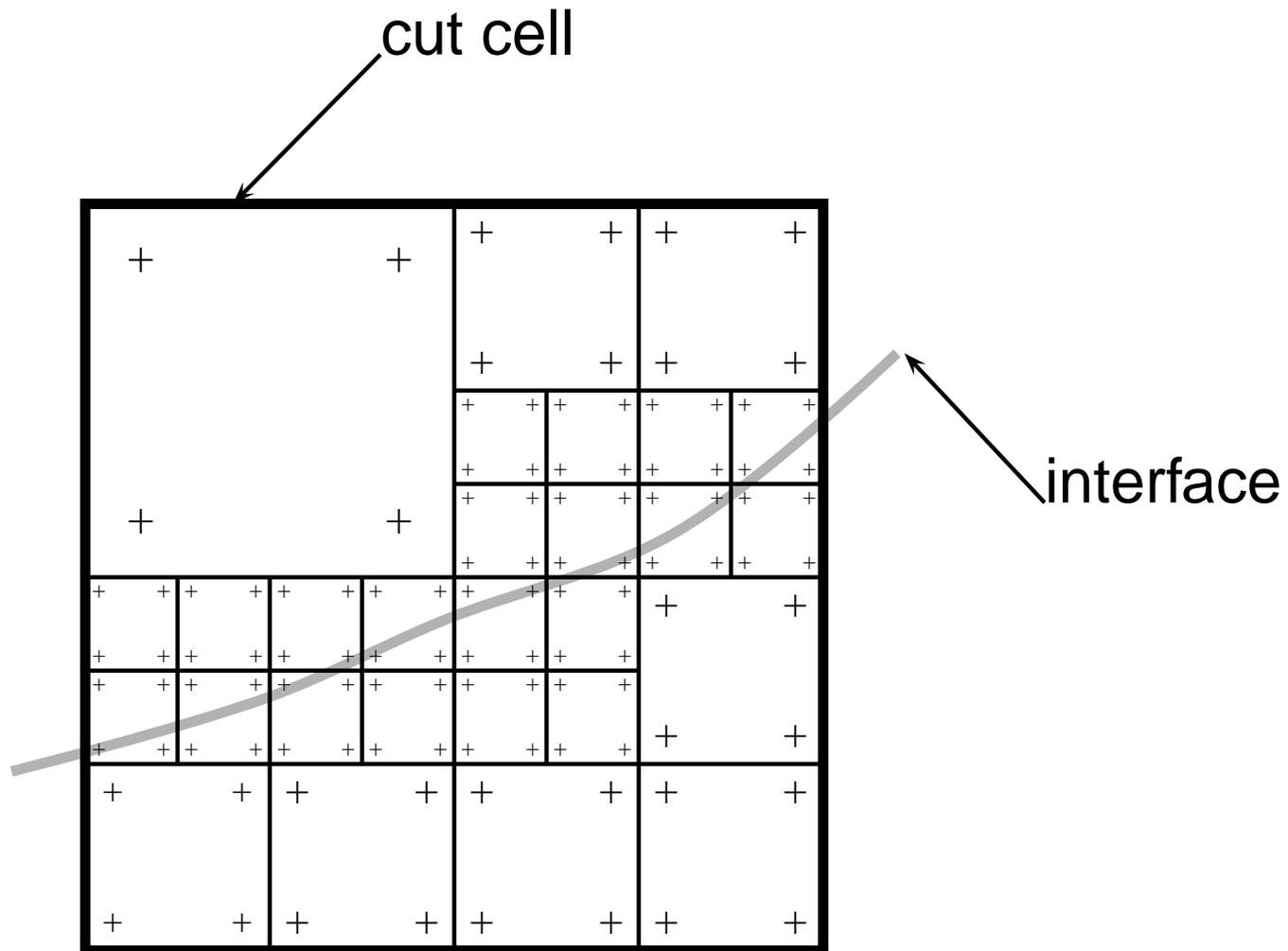
 aaa

 This subroutine is an intermediary routine between build_system and make_matrix to take into account the complex geometry of cut cells. If we are in a non cut cell, make_matrix is called. If we are in a cut cell but at a level that is smaller than levelmax, the cell is further cut and make_cut is recursively called. If we are in a cut cell and level is equal to levelmax, we call make_matrix with material properties that have been interpolated from the various material properties contributing to the cut cell.

make_cut



make_cut



make_matrix.f90

```
subroutine make_matrix (mpe,ndof,ael,bel,icon,xg,yg,zg,penaltyg,temp scale,kfix,viscosity0,density,  
penalty,expon,activationenergy,expansion,diffusivity,heat,plasticity_type,  
plasticity_1st_param,plasticity_2nd_param,dt,unode,vnode,wnode,temp,presure,  
strain,nnode,f,r0,s0,t0,rst,ileaves,eviscosity,forces)
```

- arguments
- aaa
- This routine is called to create the FE matrix and rhs vector for both the Stokes (ndof=3) and Energy equations (ndof=1).

The implemented numerical integration technique is the Gauss-Legendre, or Gauss Quadrature (see [?] p137, [?] p274, [?] p200):

$$\int_{-1}^{+1} \int_{-1}^{+1} \int_{-1}^{+1} F(r, s, t) dr ds dt = \sum_{ijk} \alpha_i \alpha_j \alpha_k F(r_i, s_i, t_i)$$

where the α_i 's are all equal to 1 and the 8 integration points (r_i, s_i, t_i) are given by

- 1 → $(-\sqrt{1/3}, -\sqrt{1/3}, -\sqrt{1/3})$
- 2 → $(\sqrt{1/3}, -\sqrt{1/3}, -\sqrt{1/3})$
- 3 → $(\sqrt{1/3}, \sqrt{1/3}, -\sqrt{1/3})$
- 4 → $(-\sqrt{1/3}, \sqrt{1/3}, -\sqrt{1/3})$
- 5 → $(-\sqrt{1/3}, -\sqrt{1/3}, \sqrt{1/3})$
- 6 → $(\sqrt{1/3}, -\sqrt{1/3}, \sqrt{1/3})$
- 7 → $(\sqrt{1/3}, \sqrt{1/3}, \sqrt{1/3})$
- 8 → $(-\sqrt{1/3}, \sqrt{1/3}, \sqrt{1/3})$

This translates as follows in the code

```
rr(1)=-.577350269189626d0 ; ss(1)=-.577350269189626d0 ; tt(1)=-.577350269189626d0 ; ww(1)=1.d0
rr(2)=.577350269189626d0 ; ss(2)=-.577350269189626d0 ; tt(2)=-.577350269189626d0 ; ww(2)=1.d0
[... ]
rr(7)=.577350269189626d0 ; ss(7)=.577350269189626d0 ; tt(7)=.577350269189626d0 ; ww(7)=1.d0
rr(8)=-.577350269189626d0 ; ss(8)=.577350269189626d0 ; tt(8)=.577350269189626d0 ; ww(8)=1.d0
```

This means that one has to construct $(\mathbf{B}^T \mathbf{C} \mathbf{B} | \mathbf{J}|)$ at every integration point, hence the following loop

```
do iint=1,nint
```

For a given value of iint between 1 and 8, we compute (r_i, s_i, t_i) :

```
r=r0+rst*(rr(iint)+1.d0)/2.d0\  
s=s0+rst*(ss(iint)+1.d0)/2.d0\  
t=t0+rst*(tt(iint)+1.d0)/2.d0\  
w=ww(iint)
```

We then compute the shape functions at the integration point:

$$\begin{aligned}h_1(r_i, s_i, t_i) &= (1 - r_i) * (1 - s_i) * (1 - t_i) / 8 \\h_2(r_i, s_i, t_i) &= (1 + r_i) * (1 - s_i) * (1 - t_i) / 8 \\h_3(r_i, s_i, t_i) &= (1 - r_i) * (1 + s_i) * (1 - t_i) / 8 \\h_4(r_i, s_i, t_i) &= (1 + r_i) * (1 + s_i) * (1 - t_i) / 8 \\h_5(r_i, s_i, t_i) &= (1 - r_i) * (1 - s_i) * (1 + t_i) / 8 \\h_6(r_i, s_i, t_i) &= (1 + r_i) * (1 - s_i) * (1 + t_i) / 8 \\h_7(r_i, s_i, t_i) &= (1 - r_i) * (1 + s_i) * (1 + t_i) / 8 \\h_8(r_i, s_i, t_i) &= (1 + r_i) * (1 + s_i) * (1 + t_i) / 8\end{aligned}$$

$$\begin{aligned}h(1) &= (1. d0 - r) * (1. d0 - s) * (1. d0 - t) / 8. d0 \\h(2) &= (1. d0 + r) * (1. d0 - s) * (1. d0 - t) / 8. d0 \\h(3) &= (1. d0 - r) * (1. d0 + s) * (1. d0 - t) / 8. d0 \\h(4) &= (1. d0 + r) * (1. d0 + s) * (1. d0 - t) / 8. d0 \\h(5) &= (1. d0 - r) * (1. d0 - s) * (1. d0 + t) / 8. d0 \\h(6) &= (1. d0 + r) * (1. d0 - s) * (1. d0 + t) / 8. d0 \\h(7) &= (1. d0 - r) * (1. d0 + s) * (1. d0 + t) / 8. d0 \\h(8) &= (1. d0 + r) * (1. d0 + s) * (1. d0 + t) / 8. d0\end{aligned}$$

and their derivatives

$$\left. \frac{\partial h_1}{\partial r} \right|_i = -(1 - s_i) * (1 - t_i) / 8$$

$$\left. \frac{\partial h_2}{\partial r} \right|_i = (1 - s_i) * (1 - t_i) / 8$$

$$\left. \frac{\partial h_3}{\partial r} \right|_i = -(1 + s_i) * (1 - t_i) / 8$$

$$\left. \frac{\partial h_4}{\partial r} \right|_i = (1 + s_i) * (1 - t_i) / 8$$

$$\left. \frac{\partial h_5}{\partial r} \right|_i = -(1 - s_i) * (1 + t_i) / 8$$

$$\left. \frac{\partial h_6}{\partial r} \right|_i = (1 - s_i) * (1 + t_i) / 8$$

$$\left. \frac{\partial h_7}{\partial r} \right|_i = -(1 + s_i) * (1 + t_i) / 8$$

$$\left. \frac{\partial h_8}{\partial r} \right|_i = (1 + s_i) * (1 + t_i) / 8$$

$$dhdr(1) = -(1 \cdot d0 - s) * (1 \cdot d0 - t) / 8 \cdot d0$$

$$dhdr(2) = (1 \cdot d0 - s) * (1 \cdot d0 - t) / 8 \cdot d0$$

$$dhdr(3) = -(1 \cdot d0 + s) * (1 \cdot d0 - t) / 8 \cdot d0$$

$$dhdr(4) = (1 \cdot d0 + s) * (1 \cdot d0 - t) / 8 \cdot d0$$

$$dhdr(5) = -(1 \cdot d0 - s) * (1 \cdot d0 + t) / 8 \cdot d0$$

$$dhdr(6) = (1 \cdot d0 - s) * (1 \cdot d0 + t) / 8 \cdot d0$$

$$dhdr(7) = -(1 \cdot d0 + s) * (1 \cdot d0 + t) / 8 \cdot d0$$

$$dhdr(8) = (1 \cdot d0 + s) * (1 \cdot d0 + t) / 8 \cdot d0$$

$$\left. \frac{\partial h_1}{\partial s} \right|_i = -(1 - r_i) * (1 - t_i) / 8$$

$$\left. \frac{\partial h_2}{\partial s} \right|_i = -(1 + r_i) * (1 - t_i) / 8$$

$$\left. \frac{\partial h_3}{\partial s} \right|_i = (1 - r_i) * (1 - t_i) / 8$$

$$\left. \frac{\partial h_4}{\partial s} \right|_i = (1 + r_i) * (1 - t_i) / 8$$

$$\left. \frac{\partial h_5}{\partial s} \right|_i = -(1 - r_i) * (1 + t_i) / 8$$

$$\left. \frac{\partial h_6}{\partial s} \right|_i = -(1 + r_i) * (1 + t_i) / 8$$

$$\left. \frac{\partial h_7}{\partial s} \right|_i = -(1 - r_i) * (1 + t_i) / 8$$

$$\left. \frac{\partial h_8}{\partial s} \right|_i = (1 + r_i) * (1 + t_i) / 8$$

$$dhds(1) = -(1 \cdot d0 - r) * (1 \cdot d0 - t) / 8 \cdot d0$$

$$dhds(2) = -(1 \cdot d0 + r) * (1 \cdot d0 - t) / 8 \cdot d0$$

$$dhds(3) = (1 \cdot d0 - r) * (1 \cdot d0 - t) / 8 \cdot d0$$

$$dhds(4) = (1 \cdot d0 + r) * (1 \cdot d0 - t) / 8 \cdot d0$$

$$dhds(5) = -(1 \cdot d0 - r) * (1 \cdot d0 + t) / 8 \cdot d0$$

$$dhds(6) = -(1 \cdot d0 + r) * (1 \cdot d0 + t) / 8 \cdot d0$$

$$dhds(7) = (1 \cdot d0 - r) * (1 \cdot d0 + t) / 8 \cdot d0$$

$$dhds(8) = (1 \cdot d0 + r) * (1 \cdot d0 + t) / 8 \cdot d0$$

$$\left. \frac{\partial h_1}{\partial t} \right|_i = -(1 - r_i) * (1 - s_i) / 8$$

$$\left. \frac{\partial h_2}{\partial t} \right|_i = -(1 + r_i) * (1 - s_i) / 8$$

$$\left. \frac{\partial h_3}{\partial t} \right|_i = -(1 - r_i) * (1 + s_i) / 8$$

$$\left. \frac{\partial h_4}{\partial t} \right|_i = -(1 + r_i) * (1 + s_i) / 8$$

$$\left. \frac{\partial h_5}{\partial t} \right|_i = (1 - r_i) * (1 - s_i) / 8$$

$$\left. \frac{\partial h_6}{\partial t} \right|_i = (1 + r_i) * (1 - s_i) / 8$$

$$\left. \frac{\partial h_7}{\partial t} \right|_i = (1 - r_i) * (1 + s_i) / 8$$

$$\left. \frac{\partial h_8}{\partial t} \right|_i = (1 + r_i) * (1 + s_i) / 8$$

$$dhdt(1) = -(1 \cdot d0 - r) * (1 \cdot d0 - s) / 8 \cdot d0$$

$$dhdt(2) = -(1 \cdot d0 + r) * (1 \cdot d0 - s) / 8 \cdot d0$$

$$dhdt(3) = -(1 \cdot d0 - r) * (1 \cdot d0 + s) / 8 \cdot d0$$

$$dhdt(4) = -(1 \cdot d0 + r) * (1 \cdot d0 + s) / 8 \cdot d0$$

$$dhdt(5) = (1 \cdot d0 - r) * (1 \cdot d0 - s) / 8 \cdot d0$$

$$dhdt(6) = (1 \cdot d0 + r) * (1 \cdot d0 - s) / 8 \cdot d0$$

$$dhdt(7) = (1 \cdot d0 - r) * (1 \cdot d0 + s) / 8 \cdot d0$$

$$dhdt(8) = (1 \cdot d0 + r) * (1 \cdot d0 + s) / 8 \cdot d0$$

We then need the Jacobian of the transformation $(x, y, z) \rightarrow (r, s, t)$.

$$\begin{pmatrix} \frac{\partial}{\partial r} \\ \frac{\partial}{\partial s} \\ \frac{\partial}{\partial t} \end{pmatrix} = \underbrace{\begin{pmatrix} \frac{\partial x}{\partial r} & \frac{\partial y}{\partial r} & \frac{\partial z}{\partial r} \\ \frac{\partial x}{\partial s} & \frac{\partial y}{\partial s} & \frac{\partial z}{\partial s} \\ \frac{\partial x}{\partial t} & \frac{\partial y}{\partial t} & \frac{\partial z}{\partial t} \end{pmatrix}}_{\mathbf{J}} \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{pmatrix}$$

Since $x = \sum_{k=1}^8 h_k x_k$, $y = \sum_{k=1}^8 h_k y_k$ and $z = \sum_{k=1}^8 h_k z_k$, we have for instance $\frac{\partial x}{\partial r} = \sum_{k=1}^8 \frac{\partial h_k}{\partial r} x_k$ so that the computation of the 3×3 Jacobian matrix `jcb` writes:

`jcb=0.`

do `k=1,mpe`

`jcb(1,1)=jcb(1,1)+dhdr(k)*x(k)`

`jcb(1,2)=jcb(1,2)+dhdr(k)*y(k)`

`jcb(1,3)=jcb(1,3)+dhdr(k)*z(k)`

`jcb(2,1)=jcb(2,1)+dhds(k)*x(k)`

`jcb(2,2)=jcb(2,2)+dhds(k)*y(k)`

`jcb(2,3)=jcb(2,3)+dhds(k)*z(k)`

`jcb(3,1)=jcb(3,1)+dhdt(k)*x(k)`

`jcb(3,2)=jcb(3,2)+dhdt(k)*y(k)`

`jcb(3,3)=jcb(3,3)+dhdt(k)*z(k)`

enddo

One then computes \mathbf{J}^{-1} . \mathbf{J} is a 3×3 matrix, so we simply resort to a simple matrix inversion algorithm. We first compute

$$\text{volume} = \text{Det}[\mathbf{J}] = J_{1,1} J_{2,2} J_{3,3} + J_{1,2} J_{2,3} J_{3,1} + J_{2,1} J_{3,2} J_{1,3} - J_{1,3} J_{2,2} J_{3,1} - J_{1,2} J_{2,1} J_{3,3} - J_{2,3} J_{3,2} J_{1,1}$$

$$\begin{aligned} \text{volume} = & \text{jcb}(1,1) * \text{jcb}(2,2) * \text{jcb}(3,3) + \text{jcb}(1,2) * \text{jcb}(2,3) * \text{jcb}(3,1) \ \& \\ & + \text{jcb}(2,1) * \text{jcb}(3,2) * \text{jcb}(1,3) \ \& \\ & - \text{jcb}(1,3) * \text{jcb}(2,2) * \text{jcb}(3,1) - \text{jcb}(1,2) * \text{jcb}(2,1) * \text{jcb}(3,3) \ \& \\ & - \text{jcb}(2,3) * \text{jcb}(3,2) * \text{jcb}(1,1) \end{aligned}$$

and then \mathbf{J}^{-1} is given by:

$$\begin{aligned} \text{jcbi}(1,1) &= (\text{jcb}(2,2) * \text{jcb}(3,3) - \text{jcb}(2,3) * \text{jcb}(3,2)) / \text{volume} \\ \text{jcbi}(2,1) &= (\text{jcb}(2,3) * \text{jcb}(3,1) - \text{jcb}(2,1) * \text{jcb}(3,3)) / \text{volume} \\ \text{jcbi}(3,1) &= (\text{jcb}(2,1) * \text{jcb}(3,2) - \text{jcb}(2,2) * \text{jcb}(3,1)) / \text{volume} \\ \text{jcbi}(1,2) &= (\text{jcb}(1,3) * \text{jcb}(3,2) - \text{jcb}(1,2) * \text{jcb}(3,3)) / \text{volume} \\ \text{jcbi}(2,2) &= (\text{jcb}(1,1) * \text{jcb}(3,3) - \text{jcb}(1,3) * \text{jcb}(3,1)) / \text{volume} \\ \text{jcbi}(3,2) &= (\text{jcb}(1,2) * \text{jcb}(3,1) - \text{jcb}(1,1) * \text{jcb}(3,2)) / \text{volume} \\ \text{jcbi}(1,3) &= (\text{jcb}(1,2) * \text{jcb}(2,3) - \text{jcb}(1,3) * \text{jcb}(2,2)) / \text{volume} \\ \text{jcbi}(2,3) &= (\text{jcb}(1,3) * \text{jcb}(2,1) - \text{jcb}(1,1) * \text{jcb}(2,3)) / \text{volume} \\ \text{jcbi}(3,3) &= (\text{jcb}(1,1) * \text{jcb}(2,2) - \text{jcb}(1,2) * \text{jcb}(2,1)) / \text{volume} \end{aligned}$$

Finally,

$$\begin{pmatrix} \frac{\partial h}{\partial x} \\ \frac{\partial h}{\partial y} \\ \frac{\partial h}{\partial z} \end{pmatrix} = \mathbf{J}^{-1} \begin{pmatrix} \frac{\partial h}{\partial r} \\ \frac{\partial h}{\partial s} \\ \frac{\partial h}{\partial t} \end{pmatrix}$$

is implemented as follows:

do k=1,mpe

dhdx(k)=jcbi(1,1)*dhdr(k)+jcbi(1,2)*dhds(k)+jcbi(1,3)*dhdt(k)

dhdy(k)=jcbi(2,1)*dhdr(k)+jcbi(2,2)*dhds(k)+jcbi(2,3)*dhdt(k)

dhdz(k)=jcbi(3,1)*dhdr(k)+jcbi(3,2)*dhds(k)+jcbi(3,3)*dhdt(k)

enddo

The strain rate tensor is given by $\epsilon = (\nabla \mathbf{v} + (\nabla \mathbf{v})^T)/2$:

$$\begin{aligned} \epsilon_{xx} &= \frac{\partial u}{\partial x} & \epsilon_{xy} &= \frac{1}{2} \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) \\ \epsilon_{yy} &= \frac{\partial v}{\partial y} & \epsilon_{yz} &= \frac{1}{2} \left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y} \right) \\ \epsilon_{zz} &= \frac{\partial w}{\partial z} & \epsilon_{xz} &= \frac{1}{2} \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} \right) \end{aligned}$$

Since $u = \sum_{k=1}^8 h_k u_k$, $v = \sum_{k=1}^8 h_k v_k$ and $z = \sum_{k=1}^8 h_k w_k$, we have for instance $\frac{\partial u}{\partial x} = \sum_{k=1}^8 \frac{\partial h_k}{\partial x} u_k$ and it translates as follows in the code:

```

exx=0.d0
eyy=0.d0
ezz=0.d0
exy=0.d0
eyz=0.d0
ezx=0.d0
do k=1,mpe
  ic=icon(k)
  exx=exx+dhdx(k)*unode(ic)
  eyy=eyy+dhdy(k)*vnode(ic)
  ezz=ezz+dhdz(k)*wnode(ic)
  exy=exy+(dhdx(k)*vnode(ic)+dhdy(k)*unode(ic))/2.d0
  eyz=eyz+(dhdy(k)*wnode(ic)+dhdz(k)*vnode(ic))/2.d0
  ezx=ezx+(dhdz(k)*unode(ic)+dhdx(k)*wnode(ic))/2.d0
enddo

```

We now need to write the **B** matrix. We start from

$$\begin{pmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial u}{\partial y} \\ \frac{\partial u}{\partial z} \end{pmatrix}_i = \begin{pmatrix} \sum_{k=1}^8 \frac{\partial h_k}{\partial x} u_k \\ \sum_{k=1}^8 \frac{\partial h_k}{\partial y} u_k \\ \sum_{k=1}^8 \frac{\partial h_k}{\partial z} u_k \end{pmatrix}_i ; \quad \begin{pmatrix} \frac{\partial v}{\partial x} \\ \frac{\partial v}{\partial y} \\ \frac{\partial v}{\partial z} \end{pmatrix}_i = \begin{pmatrix} \sum_{k=1}^8 \frac{\partial h_k}{\partial x} v_k \\ \sum_{k=1}^8 \frac{\partial h_k}{\partial y} v_k \\ \sum_{k=1}^8 \frac{\partial h_k}{\partial z} v_k \end{pmatrix}_i ; \quad \begin{pmatrix} \frac{\partial w}{\partial x} \\ \frac{\partial w}{\partial y} \\ \frac{\partial w}{\partial z} \end{pmatrix}_i = \begin{pmatrix} \sum_{k=1}^8 \frac{\partial h_k}{\partial x} w_k \\ \sum_{k=1}^8 \frac{\partial h_k}{\partial y} w_k \\ \sum_{k=1}^8 \frac{\partial h_k}{\partial z} w_k \end{pmatrix}_i$$

It is common to work with the following strain-rate vector (mainly in order to remove the 1/2 terms that would appear in the **B** matrix)

$$\begin{pmatrix} \epsilon_{xx} \\ \epsilon_{yy} \\ \epsilon_{zz} \\ 2\epsilon_{xy} \\ 2\epsilon_{xz} \\ 2\epsilon_{yz} \end{pmatrix}_i$$

Let us assume that we evaluate \mathbf{J} at a Gauss-Legendre integration point i :

$$\hat{\epsilon}_i = \begin{pmatrix} \epsilon_{xx} \\ \epsilon_{yy} \\ \epsilon_{zz} \\ 2\epsilon_{xy} \\ 2\epsilon_{xz} \\ 2\epsilon_{yz} \end{pmatrix}_i = \begin{pmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial v}{\partial y} \\ \frac{\partial w}{\partial z} \\ \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \\ \frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} \\ \frac{\partial v}{\partial z} + \frac{\partial w}{\partial y} \end{pmatrix}_i = \begin{pmatrix} \sum_{k=1}^8 \frac{\partial h_k}{\partial x} u_k \\ \sum_{k=1}^8 \frac{\partial h_k}{\partial y} u_k \\ \sum_{k=1}^8 \frac{\partial h_k}{\partial z} u_k \\ \sum_{k=1}^8 \left(\frac{\partial h_k}{\partial y} u_k + \frac{\partial h_k}{\partial x} v_k \right) \\ \sum_{k=1}^8 \left(\frac{\partial h_k}{\partial z} u_k + \frac{\partial h_k}{\partial x} w_k \right) \\ \sum_{k=1}^8 \left(\frac{\partial h_k}{\partial z} v_k + \frac{\partial h_k}{\partial y} w_k \right) \end{pmatrix}_i$$

This can be written as $\epsilon_i = \mathbf{B}_i \hat{\mathbf{u}}$ where \mathbf{B}_i is a $6 \times \text{mpe} \cdot \text{dof}$ matrix and $\hat{\mathbf{u}}$ is a vector of dimensions $\text{mpe} \cdot \text{dof}$:

$$\mathbf{B}_i = \begin{pmatrix} \frac{\partial h_1}{\partial x} & 0 & 0 & \dots & \frac{\partial h_8}{\partial x} & 0 & 0 \\ 0 & \frac{\partial h_1}{\partial y} & 0 & \dots & 0 & \frac{\partial h_8}{\partial y} & 0 \\ 0 & 0 & \frac{\partial h_1}{\partial z} & \dots & 0 & 0 & \frac{\partial h_8}{\partial z} \\ \frac{\partial h_1}{\partial y} & \frac{\partial h_1}{\partial x} & 0 & \dots & \frac{\partial h_8}{\partial y} & \frac{\partial h_8}{\partial x} & 0 \\ \frac{\partial h_1}{\partial z} & 0 & \frac{\partial h_1}{\partial x} & \dots & \frac{\partial h_8}{\partial z} & 0 & \frac{\partial h_8}{\partial x} \\ 0 & \frac{\partial h_1}{\partial z} & \frac{\partial h_1}{\partial y} & \dots & 0 & \frac{\partial h_8}{\partial z} & \frac{\partial h_8}{\partial y} \end{pmatrix}_i \quad \hat{\mathbf{u}} = \begin{pmatrix} u_1 \\ v_1 \\ w_1 \\ u_2 \\ v_2 \\ w_2 \\ \vdots \\ \vdots \\ u_8 \\ v_8 \\ w_8 \end{pmatrix}_i$$

do k=1,mpe

```

k1=(k-1)*3+1 ; k2=(k-1)*3+2 ; k3=(k-1)*3+3
b(k1,1)=dhdx(k) ; b(k2,1)=0.d0 ; b(k3,1)=0.d0
b(k1,2)=0.d0 ; b(k2,2)=dhdy(k) ; b(k3,2)=0.d0
b(k1,3)=0.d0 ; b(k2,3)=0.d0 ; b(k3,3)=dhdz(k)
b(k1,4)=0.d0 ; b(k2,4)=dhdz(k) ; b(k3,4)=dhdy(k)
b(k1,5)=dhdz(k) ; b(k2,5)=0.d0 ; b(k3,5)=dhdx(k)
b(k1,6)=dhdy(k) ; b(k2,6)=dhdx(k) ; b(k3,6)=0.d0

```

enddo

In the case of a more involved (plastic) rheology, the rheological law, or constitutive equation writes

$$\sigma^n = 2\mu_0 e^{Q/RT} \dot{\epsilon}$$

or,

$$\sigma = (2\mu_0)^{1/n} e^{Q/nRT} \dot{\epsilon}^{1/n-1} \dot{\epsilon}$$

The term $\dot{\epsilon}^{1/n-1}$ is simply 'transformed' into a scalar parameter by means of the second invariant of the deviatoric strain-rate tensor:

$$J_2 = \frac{1}{2} \epsilon_{ji} \epsilon_{ij} = \frac{\epsilon_{xx}^2 + \epsilon_{yy}^2 + \epsilon_{zz}^2}{2} + \epsilon_{xy}^2 + \epsilon_{xz}^2 + \epsilon_{yz}^2$$

```
e2d=(exx**2+eyy**2+ezz**2)/2.d0+exy**2+eyz**2+ezx**2
```

```
if (e2d.ne.0.d0) e2d=sqrt(e2d)
```

```
e2dref=1.d0
```

```
viscosity=viscosity0
```

```
if (e2d.ne.0.d0 .and. expon.ne.1.d0) viscosity=viscosity*(e2d/e2dref)**(1.d0/expon-1.d0)
```

We then define η_{eff} as being

$$\eta_{eff} = (2\mu_0)^{1/n} e^{Q/nRT} \dot{\epsilon}^{1/n-1}$$

and the matrix \mathbf{C} is imply given by $\mathbf{C} = 2\eta_{eff} \mathbf{1}$.

In the case of a purely viscous material, the following relationship between the stress tensor and the strain-rate tensor holds:

$$\sigma = 2\eta\dot{\epsilon}$$

where η is a constant viscosity. The matrix \mathbf{C} , defined as $\sigma = \mathbf{C}\dot{\epsilon}$ writes $\mathbf{C} = 2\eta\mathbf{1}$. But since we are working with the strain-rate vector defined hereabove, we must use the following \tilde{C} matrix instead:

$$\tilde{\mathbf{C}} = \begin{pmatrix} 2\eta & 0 & 0 & 0 & 0 & 0 \\ 0 & 2\eta & 0 & 0 & 0 & 0 \\ 0 & 0 & 2\eta & 0 & 0 & 0 \\ 0 & 0 & 0 & \eta & 0 & 0 \\ 0 & 0 & 0 & 0 & \eta & 0 \\ 0 & 0 & 0 & 0 & 0 & \eta \end{pmatrix}$$

d=0.d0

d(1,1)= viscosity *2.d0

d(2,2)= viscosity *2.d0

d(3,3)= viscosity *2.d0

d(4,4)= viscosity

d(5,5)= viscosity

d(6,6)= viscosity

Here one builds the left-hand side element matrix a_{el} of size $mpe \times ndof \times mpe \times ndof$

```
do j=1,mpe*ndof
  do i=1,6
    bd(j,i)=0.d0
    do k=1,6
      bd(j,i)=bd(j,i)+b(j,k)*d(k,i)
    enddo
  enddo
enddo

do j=1,mpe*ndof
  do i=1,mpe*ndof
    do k=1,6
      ael(i,j)=ael(i,j)+b(i,k)*bd(j,k)*volume*w
    enddo
  enddo
enddo
```

The equilibrium equations of the element assemblage corresponding to the nodal point displacement are

$$\mathbf{K} \cdot \mathbf{U} = \mathbf{R}$$

where $\mathbf{R} = \mathbf{R}_B + \mathbf{R}_S - \mathbf{R}_I + \mathbf{R}_C$. The load vector \mathbf{R}_B includes the effect of the element body forces, \mathbf{R}_S the effect of element surface forces, \mathbf{R}_I the effect of element initial stress and \mathbf{R}_C the concentrated loads.

Only \mathbf{R}_B is of interest in our case and we have

$$\mathbf{R}_B = \int_V \mathbf{H}^T \mathbf{f}^B dV$$

where \mathbf{f}^B are the body forces, i.e. $\mathbf{f}^B = (0, 0, \rho g)$. The matrix H is given by $\mathbf{u} = \mathbf{H} \cdot \hat{\mathbf{u}}$. From

$$u = \sum_{k=1}^8 h_k u_k \quad v = \sum_{k=1}^8 h_k v_k \quad w = \sum_{k=1}^8 h_k w_k$$

it follows that \mathbf{H} is a (dof,mpe*dof) array:

$$\begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} h_1 & 0 & 0 & h_2 & 0 & 0 & & & & & h_8 & 0 & 0 \\ 0 & h_1 & 0 & 0 & h_2 & 0 & \dots & & & & 0 & h_8 & 0 \\ 0 & 0 & h_1 & 0 & 0 & h_2 & & & & & 0 & 0 & h_8 \end{pmatrix} \begin{pmatrix} u_1 \\ v_1 \\ w_1 \\ u_2 \\ v_2 \\ w_2 \\ \vdots \\ \vdots \\ u_8 \\ v_8 \\ w_8 \end{pmatrix}$$

$$\mathbf{H}^T \mathbf{f}^B = \begin{pmatrix} h_1 & 0 & 0 \\ 0 & h_1 & 0 \\ 0 & 0 & h_1 \\ h_2 & 0 & 0 \\ 0 & h_2 & 0 \\ 0 & 0 & h_2 \\ \vdots & \vdots & \vdots \\ h_8 & 0 & 0 \\ 0 & h_8 & 0 \\ 0 & 0 & h_8 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ \rho g \end{pmatrix} = \rho g \begin{pmatrix} 0 \\ 0 \\ h_1 \\ 0 \\ 0 \\ h_2 \\ \vdots \\ 0 \\ 0 \\ h_8 \end{pmatrix}$$

One then builds the right-hand side element vector `bel` of length `mpe*ndof`

```
do j=1,mpe
  jj=(j-1)*ndof+ndof
  bel(jj)=bel(jj)+h(j)*volume*w*density
enddo
```

End of the loop on the Gauss-Legendre integration points

```
end do
```

One adds a compressibility term (1 point integration)

$\text{viscomean} = \text{viscomean} / \text{nint}$

$\text{dl} = 0. \text{d0}$

$\text{dl}(1,1) = \text{compressibility} * \text{viscomean}$

$\text{dl}(1,2) = \text{compressibility} * \text{viscomean}$

$\text{dl}(1,3) = \text{compressibility} * \text{viscomean}$

$\text{dl}(2,1) = \text{compressibility} * \text{viscomean}$

$\text{dl}(2,2) = \text{compressibility} * \text{viscomean}$

$\text{dl}(2,3) = \text{compressibility} * \text{viscomean}$

$\text{dl}(3,1) = \text{compressibility} * \text{viscomean}$

$\text{dl}(3,2) = \text{compressibility} * \text{viscomean}$

$\text{dl}(3,3) = \text{compressibility} * \text{viscomean}$

$$\text{dl} = \begin{pmatrix} x & x & x & 0 & 0 & 0 \\ x & x & x & 0 & 0 & 0 \\ x & x & x & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

This unique point of integration is situated in the middle of the cube, and its weight is 1:

$$r = 0. \text{d}0$$

$$s = 0. \text{d}0$$

$$t = 0. \text{d}0$$

$$w = 8. \text{d}0$$

One then computes the derivatives of the shape function:

$$\text{dhdr}(1) = -(1. \text{d}0 - s) * (1. \text{d}0 - t) / 8. \text{d}0$$

...

$$\text{dhds}(1) = -(1. \text{d}0 - r) * (1. \text{d}0 - t) / 8. \text{d}0$$

...

$$\text{dhdt}(1) = -(1. \text{d}0 - r) * (1. \text{d}0 - s) / 8. \text{d}0$$

...

$$\text{dhdt}(8) = (1. \text{d}0 + r) * (1. \text{d}0 + s) / 8. \text{d}0$$

Then, the jacobian and its inverse:

jcb=0.

do k=1,mpe

jcb(1,1)=jcb(1,1)+dhdr(k)*x(k)

jcb(1,2)=jcb(1,2)+dhdr(k)*y(k)

jcb(1,3)=jcb(1,3)+dhdr(k)*z(k)

jcb(2,1)=jcb(2,1)+dhds(k)*x(k)

jcb(2,2)=jcb(2,2)+dhds(k)*y(k)

jcb(2,3)=jcb(2,3)+dhds(k)*z(k)

jcb(3,1)=jcb(3,1)+dhdt(k)*x(k)

jcb(3,2)=jcb(3,2)+dhdt(k)*y(k)

jcb(3,3)=jcb(3,3)+dhdt(k)*z(k)

enddo

volume=jcb(1,1)*jcb(2,2)*jcb(3,3)+jcb(1,2)*jcb(2,3)*jcb(3,1) +jcb(2,1)*jcb(3,2)*jcb(1,3) &
-jcb(1,3)*jcb(2,2)*jcb(3,1) -jcb(1,2)*jcb(2,1)*jcb(3,3) -jcb(2,3)*jcb(3,2)*jcb(1,1)

jcbi(1,1)=(jcb(2,2)*jcb(3,3) -jcb(2,3)*jcb(3,2))/volume

jcbi(2,1)=(jcb(2,3)*jcb(3,1) -jcb(2,1)*jcb(3,3))/volume

jcbi(3,1)=(jcb(2,1)*jcb(3,2) -jcb(2,2)*jcb(3,1))/volume

jcbi(1,2)=(jcb(1,3)*jcb(3,2) -jcb(1,2)*jcb(3,3))/volume

jcbi(2,2)=(jcb(1,1)*jcb(3,3) -jcb(1,3)*jcb(3,1))/volume

jcbi(3,2)=(jcb(1,2)*jcb(3,1) -jcb(1,1)*jcb(3,2))/volume

jcbi(1,3)=(jcb(1,2)*jcb(2,3) -jcb(1,3)*jcb(2,2))/volume

jcbi(2,3)=(jcb(1,3)*jcb(2,1) -jcb(1,1)*jcb(2,3))/volume

jcbi(3,3)=(jcb(1,1)*jcb(2,2) -jcb(1,2)*jcb(2,1))/volume

Then, $\partial h / \partial x$, $\partial h / \partial y$, $\partial h / \partial z$:

```
do k=1,mpe
  dhdx(k)=jcbi(1,1)*dhdr(k)+jcbi(1,2)*dhds(k)+jcbi(1,3)*dhdt(k)
  dhdy(k)=jcbi(2,1)*dhdr(k)+jcbi(2,2)*dhds(k)+jcbi(2,3)*dhdt(k)
  dhdz(k)=jcbi(3,1)*dhdr(k)+jcbi(3,2)*dhds(k)+jcbi(3,3)*dhdt(k)
enddo
```

Then, \mathbf{B}_i :

```
do k=1,mpe
  k1=(k-1)*3+1
  k2=(k-1)*3+2
  k3=(k-1)*3+3
  b(k1,1)=dhdx(k)
  b(k2,1)=0.d0
  ...
  b(k3,6)=0.d0
enddo
```

Finally one computes an additional term to the ael matrix:

```
do j=1,mpe*ndof
  do i=1,6
    bd(j,i)=0.d0
    do k=1,6
      bd(j,i)=bd(j,i)+b(j,k)*dl(k,i)
    enddo
  enddo
enddo

do j=1,mpe*ndof
  do i=1,mpe*ndof
    do k=1,6
      ael(i,j)=ael(i,j)+bd(i,k)*b(j,k)*volume*w
    enddo
  enddo
enddo
```

Add fixed velocity boundary conditions:

```
do ii=1,mpe
  do k=1,ndof
    ij=(icon(ii)-1)*ndof+k
    if (kfix(ij).eq.1) then
      if (k.eq.1) fixt=unode(icon(ii))
      if (k.eq.2) fixt=vnode(icon(ii))
      if (k.eq.3) fixt=wnode(icon(ii))
      i=(ii-1)*ndof+k
      do j=1,mpe*ndof
        bel(j)=bel(j)-ael(j,i)*fixt
        ael(i,j)=0.d0
        ael(j,i)=0.d0
      enddo
      ael(i,i)=1.d0*penaltyg
      bel(i)=fixt*penaltyg
    endif
  enddo
enddo
```

make_pressure.f90

```
subroutine make_pressure (mpe,ndof,icon,xg,yg,zg,viscosity0,penalty,expon,  
unode,vnode,wnode,temp,pressure,strain,nnode,  
r0,s0,t0,rst)
```



arguments

- mpe
- ndof
- icon
- xg,yg,zg
- viscosity0
- penalty
- expon
- unode,vnode,wnode
- temp
- pressure
- strain
- nnode
- r0,s0,t0
- rst
- eviscosity
- q



This routine is a stripped down version of `make_matrix` but is used in the calculation of the pressure. It computes the trace of the strain-rate tensor and multiplies it by the penalty factor to obtain the pressure.

module_colormaps.f90

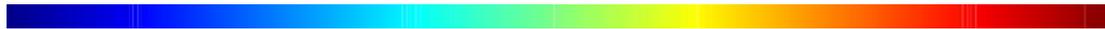
```
red_jet, green_jet, blue_jet (ic,nc)
```

arguments

ic

nc

These functions compute the *jet-rgb* values for a given index comprised between 1 and nc.



```
red_hot, green_hot, blue_hot (ic,nc)
```

arguments

ic

nc

These functions compute the *hot-rgb* values for a given index comprised between 1 and nc.



module_constants.f90

It contains the definition of the following constants :

- pi: π
- Rgas: Perfect Gas Constant $R = 8.31447215 \text{ J.K}^{-1} . \text{mol}^{-1}$
- sqrt2: $\sqrt{2}$
- sqrt3: $\sqrt{3}$

module_definitions.f90

It contains the definition of the derived types used in the code:

- type **edge**: it is used to store edges in a triangulation.
- type **sheet**: it is used to store surfaces tracked by particles.
- type **octreev**: it is used to store velocity/temperature octree.
- type **octreelsf**: it is used to store level set function.
- type **octreesolve**: it is used to store velocity,temperature,lsf,strain,...
- type **material**: it is used to define different materials.
- type **void**: it is used to store information on node, leaves and faces that are in the void.
- type **cloud**: it is used to store volumetric cloud of points.
- type **topology**: it is used to store the matrix topology for the solver.
- type **box**: it is is used to store the geometrical information about a general box that is used to refine an octree.
- type **face**: it is used to store the refinement informations on a given face of the cube
- type **cross_section**: it is used to store all the informations related to the output of cross sections

module_distributions.f90

```
subroutine cumulative_distribution (array_size,array,nb_of_intervals,namefile)
```

- arguments
 - array_size
 - array
 - nb_of_intervals
 - namefile

This subroutine computes the distribution of values contained in the array of size `array_size` passed as argument. The values in the array can be negative and positive. The distribution, computed on `nb_of_intervals` points is written in file `namefile`.

```
subroutine distribution (array_size,array,nb_of_intervals,namefile)
```

- arguments
 - array_size
 - array
 - nb_of_intervals
 - namefile

This subroutine computes the distribution of values contained in the array of size `array_size` passed as argument. The values in the array can be negative and positive. The distribution, computed on `nb_of_intervals` points is written in file `namefile`.

module_DoRuRe.f90

 DoRuRe = Douar Run Report

 module_DoRuRe.f90:

-  io_DoRuRe
-  write_dt_stats
-  write_cloud_stats
-  write_diag_stats
-  write_leaf_stats
-  write_forces_stats
-  write_nelem_stats
-  write_octree_stats
-  write_qpgram
-  write_skyline_stats
-  write_solver_stats
-  write_conv_stats
-  write_maxdeltauvw_stats
-  write_div_stats
-  write_nonlin_stats
-  write_velocity_stats
-  write_gnuplot_surfstats
-  produce_texfiles
-  write_ss
-  timestring

 To produce *report.pdf*:

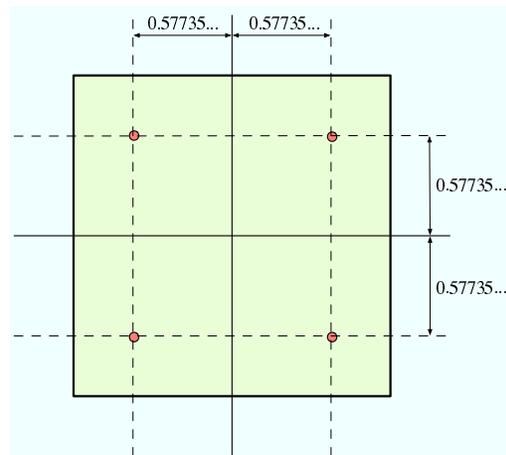
```
> ./DORURE/generate_report
```

module_gauss.f90

This module contains the definition of the ww, rr, ss, tt arrays:

$$\begin{aligned} ww &= (1, 1, 1, 1, 1, 1, 1, 1) \\ rr &= \left(-\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, -\frac{\sqrt{3}}{3}, -\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, -\frac{\sqrt{3}}{3}\right) \\ rr &= \left(-\frac{\sqrt{3}}{3}, -\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, -\frac{\sqrt{3}}{3}, -\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}\right) \\ rr &= \left(-\frac{\sqrt{3}}{3}, -\frac{\sqrt{3}}{3}, -\frac{\sqrt{3}}{3}, -\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}\right) \end{aligned}$$

which are the coordinates and associated weights of the Gauss-Legendre integration points:



module_invariants.f90

```
function trace      (exx,eyy,ezz,exy,eyz,ezx)
```

- arguments
 - exx,eyy,ezz,exy,eyz,ezx: the six components of the symmetric strain-rate tensor
- this function returns the trace of the tensor:

$$\text{trace} = \dot{\epsilon}_{xx} + \dot{\epsilon}_{yy} + \dot{\epsilon}_{zz}$$

```
subroutine deviatoric (exx,eyy,ezz,exy,eyz,ezx)
```

- arguments
 - exx,eyy,ezz,exy,eyz,ezx: the six components of the symmetric strain-rate tensor
- this routine returns the deviatoric part of the strain-rate tensor.

$$\begin{aligned}\dot{\epsilon}_{xx} &= \dot{\epsilon}_{xx} - \frac{1}{3}(\dot{\epsilon}_{xx} + \dot{\epsilon}_{yy} + \dot{\epsilon}_{zz}) \\ \dot{\epsilon}_{yy} &= \dot{\epsilon}_{yy} - \frac{1}{3}(\dot{\epsilon}_{xx} + \dot{\epsilon}_{yy} + \dot{\epsilon}_{zz}) \\ \dot{\epsilon}_{zz} &= \dot{\epsilon}_{zz} - \frac{1}{3}(\dot{\epsilon}_{xx} + \dot{\epsilon}_{yy} + \dot{\epsilon}_{zz})\end{aligned}$$

module_invariants.f90 (2)

```
function second_invariant (exx,eyy,ezz,exy,eyz,ezx)
```

- arguments
 - exx,eyy,ezz,exy,eyz,ezx: the six components of the symmetric strain-rate tensor
- This function returns the second invariant of the symmetric tensor passed as argument:
$$\text{second_invariant} = \frac{1}{2} \sum_{ij} \dot{\epsilon}_{ij} \dot{\epsilon}_{ij} = \frac{1}{2} (\dot{\epsilon}_{xx}^2 + \dot{\epsilon}_{yy}^2 + \dot{\epsilon}_{zz}^2) + \dot{\epsilon}_{xy}^2 + \dot{\epsilon}_{yz}^2 + \dot{\epsilon}_{zx}^2$$

```
function third_invariant (exx,eyy,ezz,exy,eyz,ezx)
```

- arguments
 - exx,eyy,ezz,exy,eyz,ezx: the six components of the symmetric strain-rate tensor
- This function returns the third invariant of the symmetric tensor passed as argument:
$$\begin{aligned} \text{third_invariant} &= \frac{1}{3} \sum_{ijk} \dot{\epsilon}_{ij} \dot{\epsilon}_{jk} \dot{\epsilon}_{ki} = \frac{1}{3} \dot{\epsilon}_{xx} (\dot{\epsilon}_{xx}^2 + 3\dot{\epsilon}_{xy}^2 + 3\dot{\epsilon}_{zx}^2) \\ &+ \frac{1}{3} \dot{\epsilon}_{yy} (3\dot{\epsilon}_{xy}^2 + \dot{\epsilon}_{yy}^2 + 3\dot{\epsilon}_{yz}^2) \\ &+ \frac{1}{3} \dot{\epsilon}_{zz} (3\dot{\epsilon}_{zx}^2 + 3\dot{\epsilon}_{yz}^2 + \dot{\epsilon}_{zz}^2) \\ &+ 2\dot{\epsilon}_{xy} \dot{\epsilon}_{yz} \dot{\epsilon}_{zx} \end{aligned}$$

module_invariants.f90 (3)

```
function lode_angle (J2d,J3d)
```

- arguments
 - J2d,J3d: the second and third invariant of a given tensor
- This function returns the Lode angle associated to the invariants passed as argument. This angle is comprised between $-\pi/6$ and $+\pi/6$.

$$\text{lode_angle} = \frac{1}{3} \sin^{-1} \left(-\frac{3\sqrt{3}}{2} \frac{J'_3}{(J'_2)^{3/2}} \right)$$

module_kernel.f90

```
function kernel      (r,kappah)
```

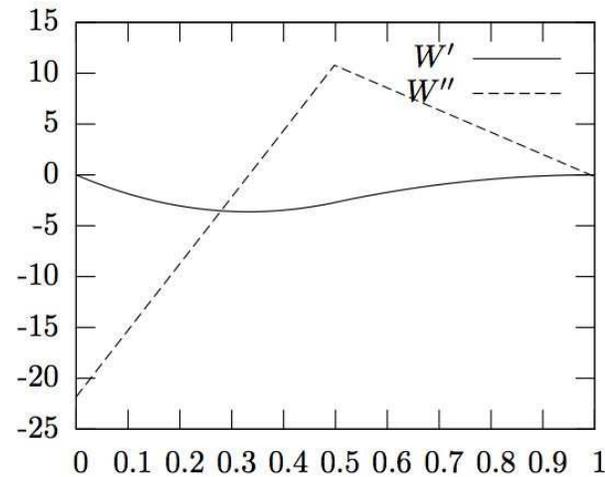
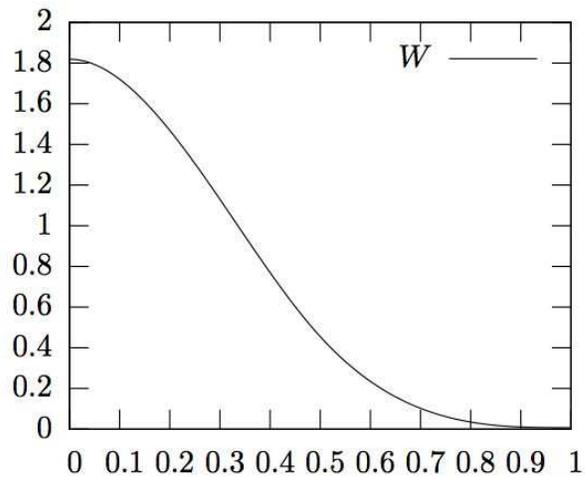
```
function kernel_p    (r,kappah)
```

```
function kernel_pp   (r,kappah)
```

arguments

- r: distance between two particles
- kappah: cutoff radius

these functions return respectively $W(r/\kappa h)$, $W'(r/\kappa h)$, $W''(r/\kappa h)$, where W is the so-called kernel of the interpolation:



module_random.f90

It contains the definition of the *ran* function, from Numerical Recipes

This implements the random number generator of Park and Miller combined with a Marsaglia shift sequence. It returns a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values).

move_cloud.f90

```
subroutine move_cloud (cl,octree,noctree,unode,vnode,wnode,nnode,  
                    icon_octree,nleaves,dt,niter_move)
```



arguments

- cl is the cloud of points to be moved
- octree is the velocity octree
- noctree is its size
- unode,vnode and wnode are the components of the velocity
- icon_octree is the connectivity matrix on the octree
- nleaves is the number of leaves in the octree
- dt is the time step
- levelmax_oct is the maximum level of refinement for the solve octree
- niter_move is the number of iterations used to move the particles (read in input.txt)



This subroutine moves a set of particles from a velocity field known at the nodes of an octree. Here we use a simple method that performs a given (niter_move) number of iterations in a mid-point algorithm.

move_surface.f90

```
subroutine move_surface (surface,surface0,flag0,octree,noctree,unode,vnode,wnode,nnode,icon_octree,nleaves,dt,nl
```



arguments

- surface is the surface on which the particles need to be moved
- surface0 is the surface in its initial geometry
- flag0 is 1 if in midpoint configuration and 0 if at the end of the timestep
- octree is the velocity octree
- noctree is its size
- unode,vnode and wnode are the components of the velocity
- nnode is the number of nodes on the octree
- icon_octree is the connectivity matrix on the octree
- nleaves is the number of leaves in the octree
- dt is the time step
- niter_move is the number of iterations used to move the particles. It is read in input.txt
- normaladvect



This subroutine moves a set of particles on a surface from a velocity field known at the nodes of an octree. Here we use a simple method that performs a given (niter_move) number of iterations in a mid_point algorithm. We also advect the normals by using the velocity gradient computed from the finite element shape function derivatives.

pressure_cut.f90

```
recursive subroutine pressure_cut (level,levelmax,levelapprox,mpe,ndof,icon,x,y,z,mat,nmat,materialn,u,v,w,  
temp,pressure,strain,nnode,lsf,nlsf,r0,s0,t0,rst,icut,ileaves)
```



arguments

- level is current level in cut cell algorithm. It varies between 0 and levelmax.
- levelapprox is used to improve the positive volume calculation by further division
- mpe is number of nodes per element (8)
- ndof is the number of degrees of freedom per node (3)
- icon is connectivity array for the current element
- x,y,z are the global coordinate arrays of length nnode
- mat is the material matrix for the nmat materials
- materialn contains the material number associated to each lsf
- u,v,w is the velocity array (obtained from previous time step or at least containing the proper velocity at the fixed dofs)
- temp, pressure and strain are temperature, pressure and strain
- nnode is number of nodes
- lsf is global array of level set functions defining the surfaces
- nlsf is number of lsfs
- r0,s0,t0 are the bottom left back corner of the part of the element. We are now integrating (in local r,s,t coordinates)
- rst is the size of the part of the element we are integrating
- icut is returned (0 if homogeneous element - 1 if cut element)



This is an intermediary routine between compute_pressure and make_pressure in the same way as make_cut is between build_system and make_matrix.

read_controlling_parameters.f90

```
subroutine read_controlling_parameters (infile, irestart, restartfile, ns, nmat, nboxes, nsections, doDoRuRe)
```

- arguments
 - ns is the number of surfaces in the system
 - irestart
 - restartfile
 - nmat is the number of different material property arrays
 - nboxes is the number of user supplied refinement boxes
 - nsections is the number of cross-sections
 - doDoRuRe
- This subroutine reads the main controlling parameters.

read_input_file.f90

```
subroutine read_input_file (infile,nstep,ns,dt,nmat,material0,mat,  
    leveluniform_oct,levelmax_oct,  
    levelcut,levelapprox,noctreemax,  
    penalty,temp scale,refine_ratio,refine_criterion,  
    octree_refine_ratio,courant,stretch,  
    surface,npmin,npmax,griditer,tol,  
    criterion,anglemax,niter_move,irestart,  
    restartfile,ismooth,boxes,nboxes,sections,nsections,  
    ierosion,zerosion,ref_on_faces,cube_faces,  
    nonlinear_iterations,initial_refine_level,  
    compute_reaction_forces,compute_qpgram,debug,  
    normaladvect,ztemp,visualise_matrix,smoothing_type,  
    renumber_nodes)
```



This routine uses `scanfile` to read in the supplied infile (*input.xxxxx*) all the parameters of the run. Values are read on the master node and then broadcast to the others if necessary.

refine_surface.f90 (1)

```
subroutine refine_surface (surface,ed,nedge,nedgen,refine,nadd,naddp,nedgepernode,  
nodenodenummer,nodeedgenumber,nnmax)
```



arguments

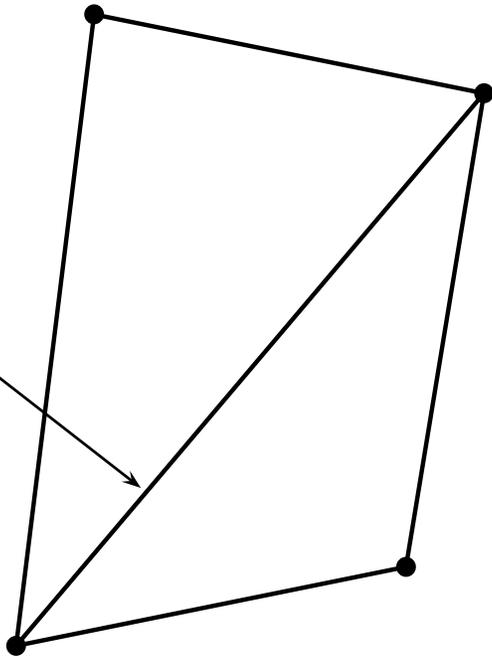
- surface is the sheet/surface to be refined
- ed is the computed edge array
- nedge is the number of edges
- nedgen
- refine
- nadd
- naddp
- nedgepernode
- nodenodenummer
- nnmax



This routine refines a surface of particles based on the value of an integer refine array of length nedge, number of edges in the triangulation connecting the particles.

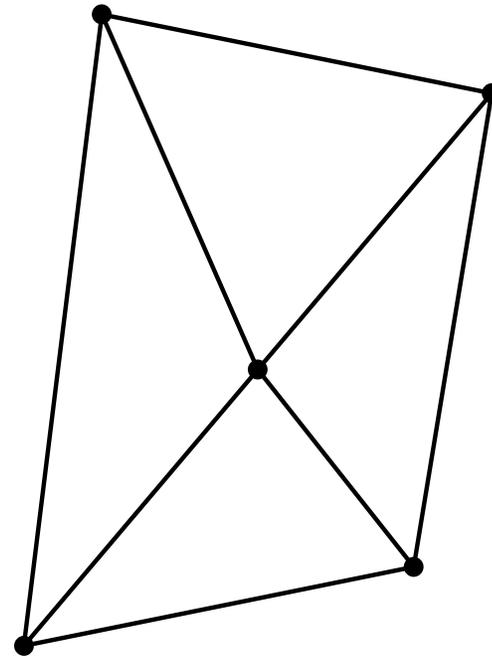
refine_surface.f90 (2)

edge to be refined



surface%nsurface=4
surface%nt=2
nedge=5

nadd=1
naddp=0



nnodemax=surface%nsurface+nadd=4+1=5
nelemmax=surface%nt+(nadd-naddp)*2+naddp=2+(1-0)*2+0=4
nedgesmax=nnodemax+nelemmax-1=5+4-1=8

scanfile.f90

```
subroutine iscanfile (fnme,text,res,ires)
```

- It reads the value of an integer parameter whose name is stored in text from file fnme. The result is stored in res and the flag ires is set to 1 if all went well.

```
subroutine dscanfile (fnme,text,res,ires)
```

- It reads the value of a double precision parameter whose name is stored in text from file fnme. The result is stored in res and the flag ires is set to 1 if all went well.

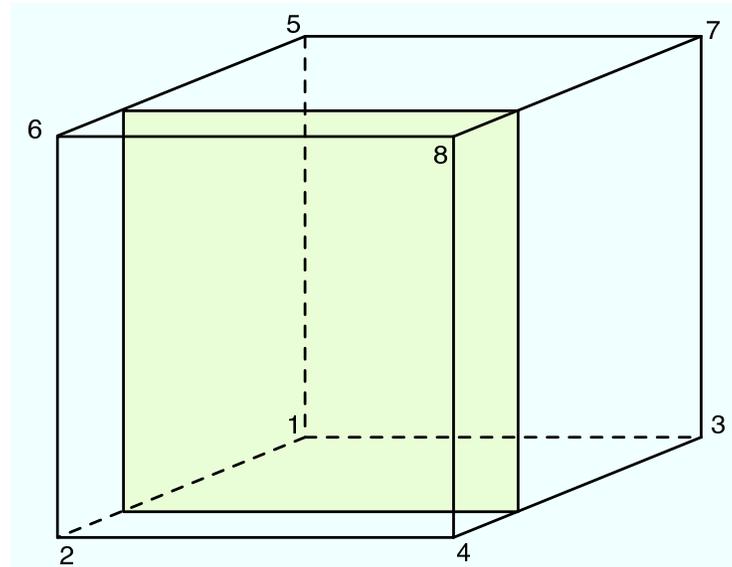
```
subroutine cscanfile (fnme,text,res,ires)
```

- It reads the value of a character string parameter whose name is stored in text from file fnme. The result is stored in res and the flag ires is set to 1 if all went well.

slices.f90

```
subroutine slices (osolve,sections,nsections,istep,iter,inner)
```

- arguments
 - osolve
 - sections
 - nsections
 - istep,iter,inner
- this subroutine outputs cross sections. all parameters have been read in *input.xxxx* and stored in the derived type *cross_section*.



slices.f90 (2)

```
subroutine write_square (iunit, alpha, alpha_offset, beta, beta_offset, dxyz, colour)
```

```
subroutine write_filled_square (iunit, alpha, alpha_offset, beta, beta_offset,
                                dxyz, ratio, flag_colour, colour, colormap, ncolours)
```



arguments

- iunit:
- alpha, beta
- alpha_offset, beta_offset
- dxyz
- ratio
- flag_colour
- colour
- colormap
- ncolours



These routine write in the file associated with unit `iunit` a filled rectangle line at point α, β , offset by $(\alpha_{offset}, \beta_{offset})$ whose size is `dxyz`, aspect ratio is `ratio`. It can be black and white or in colour, depending on the flag `flag_colour`. Its colour is given by `colour`, comprised between 1 and `ncolours`, and taken in `colormap`. The `write_square` routine only draws an empty square.

slices.f90 (3)

```
subroutine write_colormap (iunit, alpha, alpha_offset, beta, beta_offset,  
                          dxyz, ratio, flag_colour, colormap, ncolours)
```

arguments

- iunit:
- alpha, beta
- alpha_offset, beta_offset
- dxyz
- ratio
- flag_colour
- colormap
- ncolours

These routine write in the file associated with unit *iunit* the colormap rectangle at point α , β , offset by $(\alpha_{offset}, \beta_{offset})$ whose size is *dxyz*, aspect ratio is *ratio*. It can be black and white or in colour, depending on the flag *flag_colour*.



slices.f90 (4)

```
subroutine write_line (iunit,alpha,alpha_offset,beta,beta_offset,  
                      valpha,vbeta,colour,flag_colour,colormap)
```

- arguments
 - iunit:
 - alpha, beta
 - alpha_offset, beta_offset
 - valpha,vbeta
 - colour
 - flag_colour
 - colormap
- This routine writes in the file associated to iunit a line starting at point α, β , offset by $(\alpha_offset, \beta_offset)$ whose direction and length is given by the vector (v_α, v_β) .

smooth_pressures.f90 (1)

```
subroutine smooth_pressures (osolve,smoothing_type)
```



arguments



osolve



smoothing_type:



0 : no smoothing



1 : center → nodes → center



2 : center → nodes → center, weighted by neighbouring leaves volumes



3 : regular grid + SPH



4 : SPH



This routine performs the smoothing of the elemental pressure field.

smooth_pressures.f90 (3)

 smoothing_type=1

$$P_A = \frac{1}{4}(P_1 + P_2 + P_{10} + P_i)$$

$$P_B = \frac{1}{4}(P_3 + P_4 + P_5 + P_i)$$

$$P_C = \frac{1}{4}(P_5 + P_6 + P_7 + P_i)$$

$$P_D = \frac{1}{4}(P_7 + P_8 + P_9 + P_i)$$

$$\begin{aligned}\Rightarrow P_i^{\text{new}} &= \frac{1}{4}(P_A + P_B + P_C + P_D) \\ &= \frac{1}{4} \left(P_i + \frac{1}{4}(P_1 + P_2 + P_3 + P_4 + 2P_5 + P_6 + 2P_7 + P_8 + P_9 + P_{10}) \right)\end{aligned}$$

smooth_pressures.f90 (4)

 smoothing_type=2

$$P_A = \frac{P_1 V_1 + P_2 V_2 + P_{10} V_{10} + P_i V_i}{V_1 + V_2 + V_{10} + V_i}$$

$$P_C = \frac{P_5 V_5 + P_6 V_6 + P_7 V_7 + P_i V_i}{V_5 + V_6 + V_7 + V_i}$$

$$P_B = \frac{P_3 V_3 + P_4 V_4 + P_5 V_5 + P_i V_i}{V_3 + V_4 + V_5 + V_i}$$

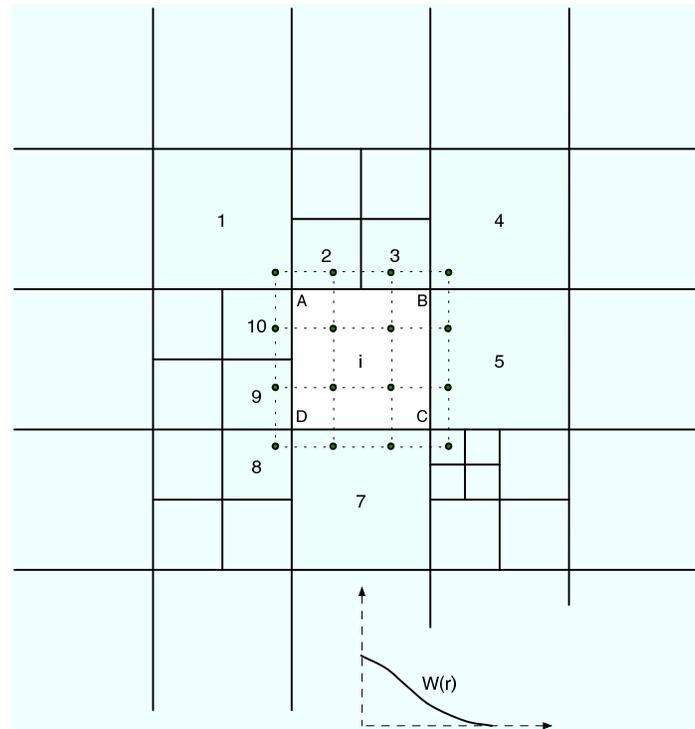
$$P_D = \frac{P_7 V_7 + P_8 V_8 + P_9 V_9 + P_i V_i}{V_3 + V_4 + V_5 + V_i}$$

$$\Rightarrow P_i^{\text{new}} = \frac{1}{4} (P_A + P_B + P_C + P_D)$$

Pb: this lead to different weights for leaves 2 and 3, for instance which is not really logical

smooth_pressures.f90 (5)

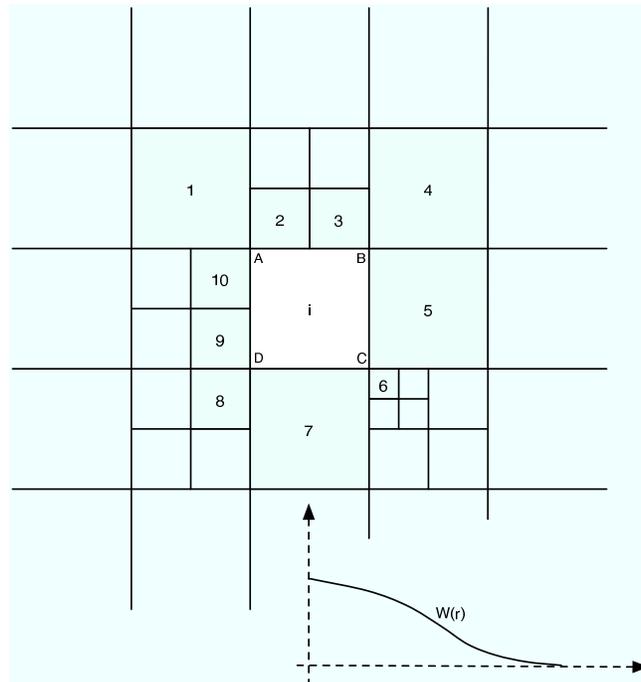
 smoothing_type=3



$$P_i^{\text{new}} = \frac{\sum_{j=i}^{16} W_{ij} P_j}{\sum_{j=1}^{16} W_{ij}}$$

smooth_pressures.f90 (6)

 smoothing_type=4



$$P_i^{\text{new}} = \frac{\sum_{j=i}^{N_i} W_{ij} P_j}{\sum_{j=1}^{N_i} W_{ij}}$$

Pb: this method works rather fine, but since it has not been parallelised yet, the neighbour algorithm is in $n\text{leaves}^2$ so it takes a lot of time to complete...

toolbox.f90

```
subroutine show_time    (total,step,inc,flag,message)
```

- This routine is used to time the main process and track progress of the run by simple output to the main screen.

```
subroutine stop_run    (message)
```

- This subroutine stops an mpi run properly to ensure that no process remains and outputs a message to the screen.

```
subroutine write_error_vtk    (leaf,volume,icon,x,y,z)
```

- This subroutine prints out the 8 nodes numbers and their coordinates of the leaf that has caused a problem. It also outputs error.vtk that simply contains the 8 nodes coordinates, which can be visualised in Mayavi with the Glyph module.

toolbox.f90 (2)

```
subroutine output_octree_olsf      (olsf,surface,is,istep,iter)
```

- This subroutine writes out a triangulated surface and its corresponding octree (+lsf field) in './DEBUG/OLSF/olsf_.....txt', which is to be processed with './DEBUG/OLSF/post' to obtain the corresponding vtk file.

```
subroutine output_surf      (surface,is,string,istep,ref_count)
```

- When using the debugging command in DOUAR, this produces vtk files of the surface passed as argument in './DEBUG/SURFACES/surf_....'

```
subroutine output_osolve_forces  (osolve,nnode,force,istep,iter)
```

- When reaction forces computations are turned on, this routine outputs in './DEBUG/FORCES/osolve_forces_xxxx_xx.txt' the necessary octree and forces data so that './DEBUG/FORCES/post' can process it and create the corresponding vtk file.

```
subroutine output_bc      (nnode,kfix,kfixt,x,y,z,u,v,w,temp)
```

- This subroutine writes in './DEBUG/BC/bc.vtk' the coordinates of the points on which mechanical boundary conditions are imposed and their assigned velocities, and in './DEBUG/BC/bct.vtk' the coordinates of the points on which thermal boundary conditions are imposed.

update_cloud_fields.f90

```
subroutine update_cloud_fields (cl,ov,osolve,dt)
```

- arguments
 - cl is the cloud object
 - ov is the velocity octree
 - osolve is the octree
 - dt is time step
- This routine is used to update the total strain value stored on the particles of the 3D cloud from the strainrate obtained by interpolation and differentiation of the velocity field from the octreev structure.

update_cloud_structure.f90

```
subroutine update_cloud_structure (cl,os,npmin,npmax,ninject,nremove,levelmax_oct)
```



arguments

- cl is the cloud
- os is the octree solve
- npmin is the minimum number of particles in any leaf of the octree solve
- npmax is the maximum number of particles in the smallest leaves of the octree solve
- ninject is number of new particles injected in the cloud
- nremove is number of particles removed from the cloud
- levelmax_oct is maximum level of octree solve



This file contains the operations done on a 3D cloud of particles to check the density of particles, accordingly add or remove particles and transfer the fields carried by the cloud (here the original position of the particles). First it initializes the fields to the current (thus original) position, then we also interpolate the fields (original position) onto the corresponding fields of the octree solve in order for the information to be available during matrix building operation.

visualise_matrix.f90

```
subroutine visualise_matrix (nz,irn,jcn,n,istep,iter)
```

arguments

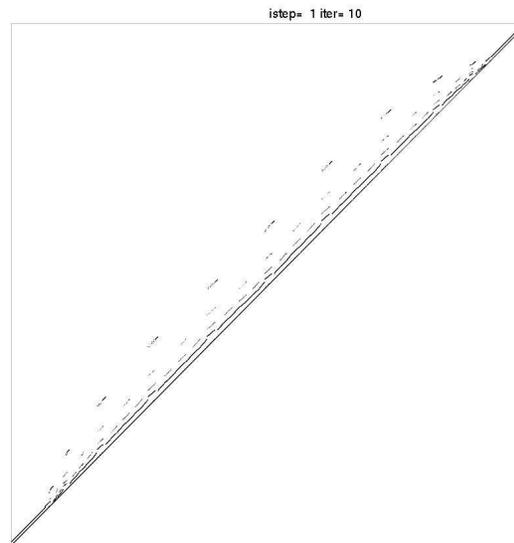
● nz

● irn

● jcn

● istep,iter

● This subroutine outputs a visual representation of the FEM matrix in `./DEBUG/MATRIX/` for each time step and each generated grid.



vrm.f90

```
subroutine vrm (J2d,J3d,viscosity,pressure,plasticity_parameters,plasticity_type,is_plastic)
```

- arguments
 - J2d: $J'_2(\dot{\epsilon})$
 - J3d: $J'_3(\dot{\epsilon})$
 - viscosity
 - pressure
 - plasticity_parameters
 - plasticity_type
 - is_plastic
- This routine computes the rescaled viscosity in the element in the case the failure criterion is positive

vrm.f90 (2)

 Mohr-Coulomb

$$\mu' = \frac{1}{2\sqrt{E'_2}} \frac{c \cos \phi - p \sin \phi}{\zeta(\theta_l, \phi)}$$

 Tresca

$$\mu' = \frac{1}{2\sqrt{E'_2}} \frac{c}{\cos \theta_l}$$

 von Mises

$$\mu' = \frac{1}{2\sqrt{E'_2}} c$$

 Drucker-Prager

$$\mu' = \frac{1}{2\sqrt{E'_2}} (c - \alpha p)$$

with

$$\zeta(\theta_l, \phi) = \cos \theta_l - \frac{1}{\sqrt{3}} \sin \theta_l \sin \phi$$

vrm.f90 (3)

```
subroutine compute_plastic_params (plasticity_parameters,plasticity_type)
```

- arguments
 - plasticity_parameters
 - plasticity_type

This routine computes derived plasticity parameters that will be used a lot and stores them in vacant spaces in plasticity_parameters(8,9):

plasticity parameters	1	2	3	4	5	6	7	8	9
vM	σ_y								
DPI, II	ϕ	c						α	k
DP III	α	k							
MC	ϕ	c						$\sin \phi$	$\cos \phi$
Tr	σ_y								

write_global_output.f90

```
subroutine write_global_output (istep,iter,current_time,osolve,ov,vo,surface,ns,cl,outputtype)
```

- arguments
 - istep
 - iter
 - current_time
 - osolve
 - vo
 - surface
 - ns
 - cl
 - outputtype

This subroutine writes out all the necessary data relative to the run at given istep (and iter if outputtype is set to 'debug' instead of final), for restart and plotting purposes. The files are stored in ./OUT/. In case of a restart, the data file under consideration is accessed by define_surface, define_ov and define_cloud.

Douar Run Reports

DoRuRe (1)

unit	file name	produced in	associated gnuplot script	postscript file
99	screen.txt			
199	debug.txt			
60x	work_0x_stats.dat	compute_vol_work	gnuplot_script_surfstats	work_xx_stats.ps
65x	vol_0x_stats.dat	compute_vol_work	gnuplot_script_surfstats	vol_xx_stats.ps
70x	surface_0x_stats.dat	write_ss	gnuplot_script_surfstats	surface_xx_stats_nt.ps surface_xx_stats_nsurface.ps surface_xx_stats_nedge.ps surface_xx_stats_nadd.ps
804	p_stats.dat	write_leaf_stats.dat	gnuplot_script_e2d_p_q	p_stats.ps
805	q_stats.dat	write_leaf_stats.dat	gnuplot_script_e2d_p_q	q_stats.ps
806	solver_stats.dat	write_solver_stats	gnuplot_script_solver	solver_stats_n.ps solver_stats_nz.ps solver_stats_rinfog3.ps solver_stats_infog3.ps solver_stats_fact_time.ps solver_stats_bcksub_time.ps solver_stats_factvsn solver_stats_factvsnleaves
807	octree_stats.dat	write_octree_stats	gnuplot_script_octree	octree_stats_nleaves.ps octree_stats_nnode.ps octree_stats_nface.ps octree_stats_nnodevsnleaves.ps

DoRuRe (2)

unit	file name	produced in	associated gnuplot script	postscript file
808	level_stats.dat	write_octree_stats	gnuplot_script_levels	level_stats_04.ps level_stats_05.ps level_stats_06.ps level_stats_07.ps level_stats_08.ps level_stats_09.ps level_stats_10.ps
809	frontpage.dat			
810	conv_stats.dat	write_conv_stats	gnuplot_script_conv	conv_stats.ps
811	level_stats_volumes.dat	write_octree_stats	gnuplot_script_levels	level_stats_voltot.ps
812	cloud_stats.dat	write_cloud_stats	gnuplot_script_cloud	cloud_stats_npoints.ps cloud_stats_nremove.ps cloud_stats_ninject.ps cloud_stats_strain.ps cloud_stats_temp.ps cloud_stats_press.ps
813	maxdeltauvw.dat	write_maxdeltauvw_stats	gnuplot_script_max	maxdeltauvw.ps
815	divergence_stats.dat	write_div_stats	gnuplot_script_div	div_stats.ps
816	e2d_stats.dat	write_leaf_stats.dat	gnuplot_script_e2d_p_q	e2d_stats.ps

DoRuRe (3)

unit	file name	produced in	associated gnuplot script	postscript file
817	forces_stats.dat	write_forces_stats.dat	gnuplot_script_forces	forces_x_stats.ps forces_y_stats.ps forces_z_stats.ps
818	gnuplot_script_qpgram	write_qpgram		
819	qpgram_list.tex	write_qpgram		
820	diag1_stats.dat	write_diag_stats	gnuplot_script_diag	diag1_stats.ps
821	diag3_stats.dat	write_diag_stats	gnuplot_script_diag	diag3_stats.ps
822	nelem1_stats.dat	write_nelem_stats	gnuplot_script_nelem	nelem1_stats.ps
823	nelem3_stats.dat	write_nelem_stats	gnuplot_script_nelem	nelem3_stats.ps
824	skyline1_stats.dat	write_skyline_stats	gnuplot_script_skyline	skyline1_stats.ps
825	skyline3_stats.dat	write_skyline_stats	gnuplot_script_skyline	skyline3_stats.ps
826	nonlin_stats	write_nonlin_stats	gnuplot_script_nonlin	nonlin_stats.ps
827	velocity_stats	write_velocity_stats	gnuplot_script_vel	u_stats.ps v_stats.ps w_stats.ps
828	temp_stats.dat	write_temp_stats	gnuplot_script_temp	temp_stats.ps
829	e3d_stats.dat	write_leaf_stats	gnuplot_e2d_p_q	e3d_stats.ps
830	dt_stats.dat	write_dt_stats	gnuplot_script_dt	dt_stats.ps

Compiling and linking

input.txt

input.xxxx

- `debug`: `debug` is a parameter that switches on/off various prints and outputs (the level of printing for error, warning and messages of the solver `cntl(4)` is set to the `debug` value) if 0 : no debugging ; if 1 : same + display of `conv,nleaves,nnode,nsurface` ; if 2 : same + intermediate `write_global_output`.
- `DoRuRe`:
- `compute_qpgram`: this is a flag. If set to 0, it switches off the qp-gram calculations
- `compute_reaction_forces`: this a flag. If set to 0, it switches off the reaction forces calculations
- `irestart`: this is a flag. If set to 0, this is a new run.
- `restartfile`: if `irestart` \neq 0, then it indicates the run should start at step `irestart`, and restart from a previously obtained output file.
- `dt`: this is the time step length (if `dt` is negative, courant condition is used and automatic time stepping is turned on)
- `nstep`: this is the number of time steps
- `courant`: `courant` is only used when `dt` is negative; it determines the size of the time step from the maximum value of the velocity field amplitude. The time step is the product of `courant` by the ratio of the smallest leaf size by the maximum velocity.
- `normaladvect`: this is a flag used to determine which algorithm is to be used to calculate the new geometry of the normals to the surfaces at the nodes on the surfaces. if `normaladvect=1`, the normals are advected using the velocity gradient, if `normaladvect=0`, the normals are re-computed from the geometry of the surface.
- `griditer`: `griditer` is a flag that allows for nonlinear iterations; when positive, a fixed number (`griditer`) of iterations is permitted; when negative, the number of nonlinear iterations is determined by a convergence criterion.
- `octree_refine_ratio`: `octree_refine_ratio` is the threshold value used to determine whether the octree has converged or not. the larger the value, the less stringent the test.
- `nonlinear_iterations`: `nonlinear_iterations` is the maximum number of nonlinear iterations (i.e. the iterations on a given constant grid). If `nonlinear_iterations` is positive, it simply is the number of nonlinear iterations performed for each grid. When negative it indicates an upper bound of nonlinear iterations, but the actual number of nonlinear iterations is determined by a convergence criterion (see the 'tol' parameter).

input.xxxx (2)

- `tol`: `tol` is the relative tolerance used to estimate convergence on the computed velocity field
- `leveluniform_oct`: `leveluniform_oct` is the level of uniform discretization of space; note that a level is a power of two used to divide the unit cube.
- `levelmax_oct`: `levelmax_oct` is maximum level of octree discretization.
- `ismooth`: this is a flag to impose an additional level of smoothing after refinement for the surfaces and strain rate. It ensures that no leaf is flanked by other leaves differing by more than 1 level of refinement. If `ismooth` is 0, no smoothing is performed; if `ismooth` is set to 1, smoothing is performed (default is 1).
- `noctreemax`: `noctreemax` is the maximum size of any octree used in all computations.
- `refine_ratio`: `refine_ratio` is used to determine octree refinement based on a given criterion. All leaves where the criterion is larger than `refine_ratio` times the maximum of this criterion are refined.
- `refine_criterion`: Several criteria exist for the refinement of the `osolve` octree. 1 is the second invariant of the strain-rate tensor; 2 is the second invariant of the deviatoric strain-rate tensor; 3 is $\sqrt{(du/dx)^2+(dv/dy)^2+(dw/dz)^2}$; 4 is $\sqrt{(du/dx)^2+(dv/dy)^2+(dw/dz)^2} * \text{leaf_size}$; any other value sets the criterion to zero and leads to no refinement.
- `initial_refine_level`: `initial_refine_level` is the initial level at which the refinement of the octree will be performed. it has to be smaller than `levelmax_oct`.
- `renumber_nodes`:
- `smoothing_type`:

input.xxxx (3)

- npmin, npmax
- levelcut
- levelapprox
- penalty
- ztemp

input.xxxx (4)

- nmat
- material0
- For each material i
 - density i
 - viscosity i
 - penalty i
 - expon i
 - diffusivity i
 - heat i
 - activation_energy i
 - plasticity_type i
 - plasticity_1st_param i
 - plasticity_2nd_param i
 - plasticity_3rd_param i
 - plasticity_4th_param i
 - plasticity_5th_param i
 - plasticity_6th_param i
 - plasticity_7th_param i
 - plasticity_8th_param i
 - plasticity_9th_param i

input.xxxx (5)

- ns
- for each surface j
 - level $_j$
 - itype $_j$
 - surface_type_ $_j$
 - rand $_j$
 - surface_param_01_ $_j$
 - surface_param_02_ $_j$
 - surface_param_03_ $_j$
 - surface_param_04_ $_j$
 - surface_param_05_ $_j$
 - surface_param_06_ $_j$
 - surface_param_07_ $_j$
 - surface_param_08_ $_j$
 - surface_param_09_ $_j$
 - surface_param_10_ $_j$
 - material $_j$
 - activation_time_ $_j$

input.xxxx (6)

-  niter_move
-  stretch
-  criterion
-  anglemax

input.xxxx (7)

- nboxes
- for each box k
 - box k x0
 - box k x1
 - box k y0
 - box k y1
 - box k z0
 - box k z1
 - box k level
- ref_on_faces
- level_face1, b1,t1,l1,r1
- level_face2, b2,t2,l2,r2
- level_face3, b3,t3,l3,r3
- level_face4, b4,t4,l4,r4
- level_face5, b5,t5,l5,r5
- level_face6, b6,t6,l6,r6

input.xxx (7)

- ierosion=0
- zerosion=.81d0
- visualise_matrix:

input.xxxx (8)

- `nsections`: number of cross-sections to be output in `./XSC/`
- For each cross-section i :
 - `xyz_i`: 1,2, or 3. If equal to one it corresponds to a $x = constant$ plane, etc ...
 - `slice_i`: between 0 and 1. Coordinate of the cross-section plane.
 - `flag_press_i`: yes or no.
 - `flag_e2d_i`: yes or no.
 - `flag_e3d_i`: yes or no
 - `flag_lode_i`: yes or no
 - `flag_crit_i`: yes or no
 - `flag_grid_i`: yes or no
 - `flag_mu_i`: yes or no
 - `flag_u_i`: yes or no
 - `flag_v_i`: yes or no
 - `flag_w_i`: yes or no
 - `flag_q_i`: yes or no
 - `flag_uvw_i`: yes or no
 - `flag_lsf_i`: yes or no
 - `flag_vfield_i`: yes or no
 - `flag_colour_i`: yes or no
 - `flag_plastic_i`: yes or no
 - `scale_i`: size of the output image of the cross-section.
 - `colormap_i`: jet, hot. Type of colormap.
 - `ncolours_i`: 64,128,256, ... number of colours that compose the colormap.

Visualisation

Benchmarks

titre

