The Finite Element Method in Geodynamics

C. Thieulot

November 16, 2018

Contents

1 Introduction

practical hands-on approach

as little as possible jargon

no mathematical proof

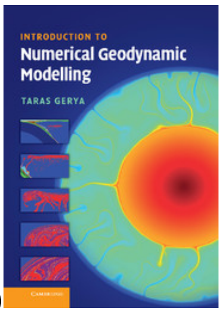
no optimised codes (readability over efficiency). avoiding as much as possible to have to look elsewhere. very sequential, so unavoidable repetitions (jacobian, shape functions)

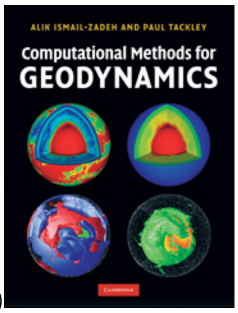
FE is one of several methods.

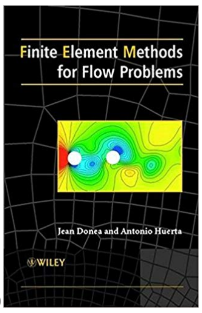
1.1 Acknowledgments

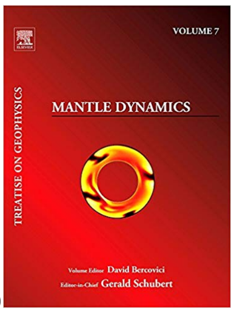
Jean Braun, Philippe Fullsack, Arie van den Berg. Lukas van de Wiel. Robert Myhill. Menno, Anne Too many BSc and MSc students to name indivisually, although Job Mos did produce the very first version of fieldstone as part of his MSc thesis. The ASPECT team in general and Wolfgang Bangerth in particular.

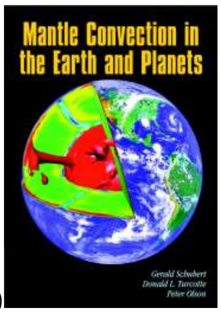
1.2 Essential literature

a)


b)

ALIK ISMAIL-ZADEH AND PAUL TACKLEY
Computational Methods for GEODYNAMICS

c)

Finite Element Methods for Flow Problems
Jean Donea and Antonio Huerta
WILEY

d)

VOLUME 7
TREATISE ON GEOPHYSICS
MANTLE DYNAMICS
Volume Editor David Bercovici
Editor-in-Chief Gerald Schubert

e)


1.3 Installation

```
python3.6 -m pip install --user numpy scipy matplotlib
```

2 The physical equations of Fluid Dynamics

[from aspect manual] We focus on the system of equations in a $d = 2$- or $d = 3$-dimensional domain Ω that describes the motion of a highly viscous fluid driven by differences in the gravitational force due to a density that depends on the temperature. In the following, we largely follow the exposition of this material in Schubert, Turcotte and Olson [31].

Specifically, we consider the following set of equations for velocity $\mathbf{u}$, pressure p and temperature T:

$$-\nabla \cdot \left[2\eta \left(\dot{\varepsilon}(\boldsymbol{v}) - \frac{1}{3}(\nabla \cdot \boldsymbol{v})\mathbf{1}\right)\right] + \nabla p = \rho \boldsymbol{g} \qquad \text{in } \Omega, \tag{1}$$

$$\nabla \cdot (\rho \boldsymbol{v}) = 0 \qquad \text{in } \Omega, \tag{2}$$

$$\begin{aligned}\rho C_p \left(\frac{\partial T}{\partial t} + \boldsymbol{v} \cdot \nabla T\right) - \nabla \cdot k \nabla T = \rho H& \\ + 2\eta \left(\dot{\varepsilon}(\boldsymbol{v}) - \frac{1}{3}(\nabla \cdot \boldsymbol{v})\mathbf{1}\right) : \left(\dot{\varepsilon}(\boldsymbol{v}) - \frac{1}{3}(\nabla \cdot \boldsymbol{v})\mathbf{1}\right)& \\ + \alpha T\,(\boldsymbol{v} \cdot \nabla p)& \\ + \rho T \Delta S \left(\frac{\partial X}{\partial t} + \boldsymbol{v} \cdot \nabla X\right)& \qquad \text{in } \Omega,\end{aligned} \tag{3}$$

where $\dot{\varepsilon}(\mathbf{u}) = \frac{1}{2}(\nabla \mathbf{u} + \nabla \mathbf{u}^T)$ is the symmetric gradient of the velocity (often called the *strain rate*).

In this set of equations, (4) and (5) represent the compressible Stokes equations in which $\mathbf{v} = \mathbf{v}(\mathbf{x}, t)$ is the velocity field and $p = p(\mathbf{x}, t)$ the pressure field. Both fields depend on space $\mathbf{x}$ and time t. Fluid flow is driven by the gravity force that acts on the fluid and that is proportional to both the density of the fluid and the strength of the gravitational pull.

Coupled to this Stokes system is equation (6) for the temperature field $T = T(\mathbf{x}, t)$ that contains heat conduction terms as well as advection with the flow velocity $\mathbf{v}$. The right hand side terms of this equation correspond to

- internal heat production for example due to radioactive decay;
- friction heating;
- adiabatic compression of material;
- phase change.

The last term of the temperature equation corresponds to the latent heat generated or consumed in the process of phase change of material. In what follows we will not assume that no phase change takes place so that we disregard this term altogether.

2.1 the Boussinesq approximation: an Incompressible flow

[from aspect manual] The Boussinesq approximation assumes that the density can be considered constant in all occurrences in the equations with the exception of the buoyancy term on the right hand side of (4). The primary result of this assumption is that the continuity equation (5) will now read

$$\boldsymbol{\nabla} \cdot \boldsymbol{v} = 0$$

This implies that the strain rate tensor is deviatoric. Under the Boussinesq approximation, the equations are much simplified:

$$-\nabla \cdot [2\eta \dot{\varepsilon}(\boldsymbol{v})] + \nabla p = \rho \boldsymbol{g} \qquad \text{in } \Omega, \tag{4}$$

$$\nabla \cdot (\rho \boldsymbol{v}) = 0 \qquad \text{in } \Omega, \tag{5}$$

$$\rho_0 C_p \left(\frac{\partial T}{\partial t} + \boldsymbol{v} \cdot \nabla T\right) - \nabla \cdot k \nabla T = \rho H \qquad \text{in } \Omega \tag{6}$$

Note that all terms on the rhs of the temperature equations have disappeared, with the exception of the source term.

These are the equations which are used and implemented in all the codes, unless otherwise indicated.

2.1.1 The penalty approach

In order to impose the incompressibility constraint, two widely used procedures are available, namely the Lagrange multiplier method and the penalty method [1, 17]. The latter is implemented in ELEFANT, which allows for the elimination of the pressure variable from the momentum equation (resulting in a reduction of the matrix size).

Mathematical details on the origin and validity of the penalty approach applied to the Stokes problem can for instance be found in [7], [28] or [16].

The penalty formulation of the mass conservation equation is based on a relaxation of the incompressibility constraint and writes

$$\boldsymbol{\nabla} \cdot \boldsymbol{v} + \frac{p}{\lambda} = 0 \tag{7}$$

where λ is the penalty parameter, that can be interpreted (and has the same dimension) as a bulk viscosity. It is equivalent to say that the material is weakly compressible. It can be shown that if one chooses λ to be a sufficiently large number, the continuity equation $\boldsymbol{\nabla} \cdot \boldsymbol{v} = 0$ will be approximately satisfied in the finite element solution. The value of λ is often recommended to be 6 to 7 orders of magnitude larger than the shear viscosity [11, 18].

Equation (7) can be used to eliminate the pressure in Eq. (**??**) so that the mass and momentum conservation equations fuse to become :

$$\boldsymbol{\nabla} \cdot (2\eta\dot{\varepsilon}(\boldsymbol{v})) + \lambda \boldsymbol{\nabla}(\boldsymbol{\nabla} \cdot \boldsymbol{v}) = \rho \boldsymbol{g} = 0 \tag{8}$$

[23] have established the equivalence for incompressible problems between the reduced integration of the penalty term and a mixed Finite Element approach if the pressure nodes coincide with the integration points of the reduced rule.

In the end, the elimination of the pressure unknown in the Stokes equations replaces the original saddle-point Stokes problem [2] by an elliptical problem, which leads to a symmetric positive definite (SPD) FEM matrix. This is the major benefit of the penalized approach over the full indefinite solver with the velocity-pressure variables. Indeed, the SPD character of the matrix lends itself to efficient solving stragegies and is less memory-demanding since it is sufficient to store only the upper half of the matrix including the diagonal [15] . ToDo: list codes which use this approach.

3 The Finite Element Method

3.1 aaa

3.2 Solving procedures

3.2.1 the whole matrix at once

3.2.2 the pressure Schur complement appraoch

4 Additional techniques

4.1 The method of manufactured solutions

4.2 Sparse storage

4.3 Mesh generation

4.4 The value of the timestep

4.5 Tracking materials

4.6 Visco-Plasticity

4.7 Picard and Newton

4.8 The choice of solvers

4.9 The SUPG formulation for the energy equation

4.10 Tracking materials and/or interfaces

4.11 Dealing with a free surface

so much to do ...
impose bc on el matrix
Q2Q1
Q3Q2
3D - Burstedde benchmark
full compressible
total energy calculations
constraints
Q1Q1-stab
non-linear rheologies (punch, two layer brick spmw16, tosn15)
Picard vs Newton
markers
Schur complement approach
periodic boundary conditions
open boundary conditions
export to vtu
free surface
SUPG
produce fastest version possible for convection box
zaleski disk advection
all kinds of interesting benchmarks
Busse convection pb, compare with aspect
cvi !!!
pure elastic
including phase changes (w. R. Myhill)
derivatives on nodes
Nusselt
aBBT matrix as whole to solver?
discontinuous galerkin
formatting of code style
navier-stokes ? (LUKAS)
pressure smoothing
compute strainrate in middle of element or at quad point for punch?
GEO1442 code
GEO1442 indenter setup in plane ?
in/out flow on sides for lith modelling

Problems to solve:

colorscale
velocity arrows
better yet simple matrix storage ?

5 `fieldstone`: simple analytical solution

From [10]. In order to illustrate the behavior of selected mixed finite elements in the solution of stationary Stokes flow, we consider a two-dimensional problem in the square domain $\Omega = [0,1] \times [0,1]$, which possesses a closed-form analytical solution. The problem consists of determining the velocity field $\boldsymbol{v} = (u, v)$ and the pressure p such that

$$-\nu \Delta \boldsymbol{v} + \boldsymbol{\nabla} p = \boldsymbol{b} \qquad \text{in } \Omega$$

$$\boldsymbol{\nabla} \cdot \boldsymbol{v} = 0 \qquad \text{in } \Omega$$

$$\boldsymbol{v} = \boldsymbol{0} \qquad \text{on } \Gamma$$

where the fluid viscosity is taken as $\nu = 1$. The components of the body force $\boldsymbol{b}$ are prescribed as

$$\begin{aligned}
b_x &= (12-24y)x^4 + (-24+48y)x^3 + (-48y+72y^2-48y^3+12)x^2 \\
&\quad +(-2+24y-72y^2+48y^3)x + 1 - 4y + 12y^2 - 8y^3 \\
b_y &= (8-48y+48y^2)x^3 + (-12+72y-72y^2)x^2 \\
&\quad +(4-24y+48y^2-48y^3+24y^4)x - 12y^2 + 24y^3 - 12y^4
\end{aligned}$$

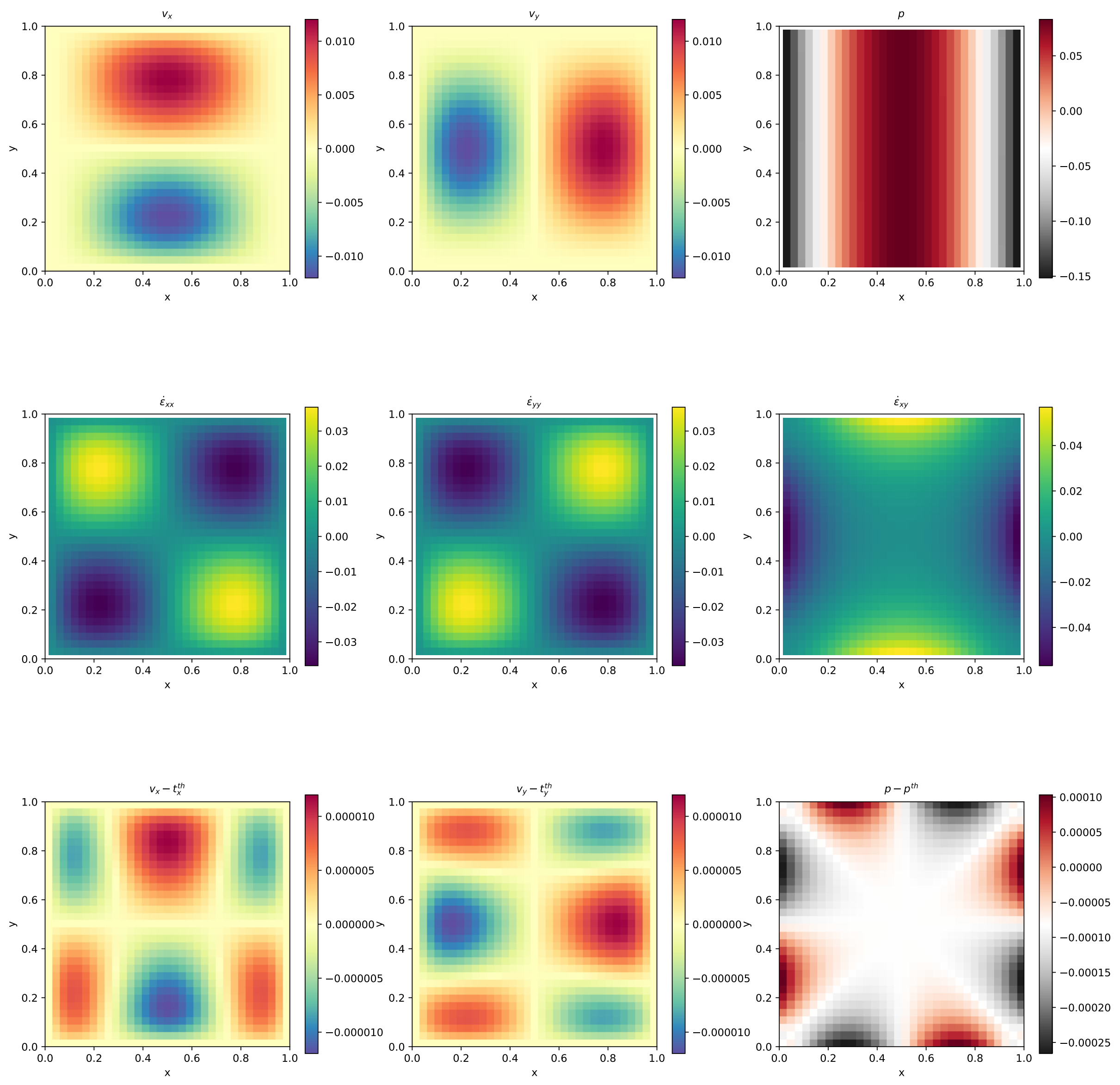
With this prescribed body force, the exact solution is

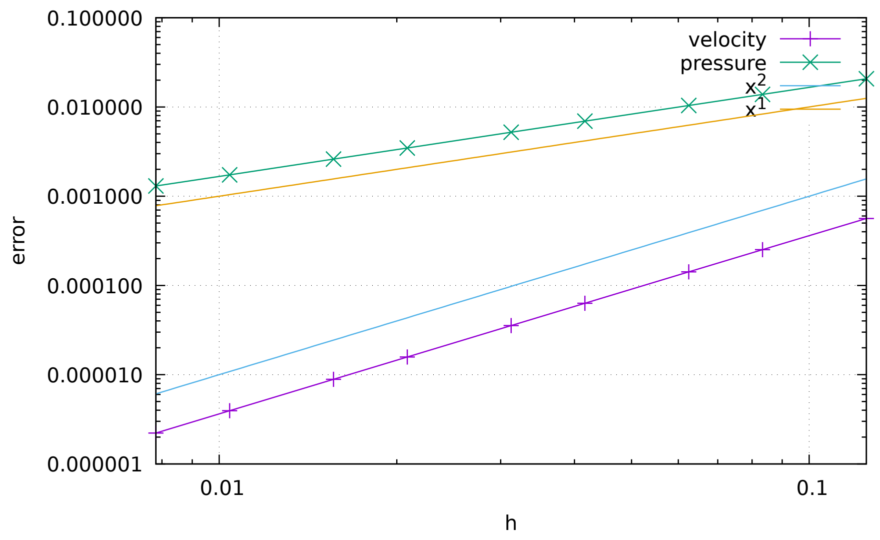
$$\begin{aligned}
u(x,y) &= x^2(1-x)^2(2y-6y^2+4y^3) \\
v(x,y) &= -y^2(1-y)^2(2x-6x^2+4x^3) \\
p(x,y) &= x(1-x) - 1/6
\end{aligned}$$

Note that the pressure obeys $\int_\Omega p \, d\Omega = 0$

features

- $Q_1 \times P_0$ element
- incompressible flow
- penalty formulation
- Dirichlet boundary conditions (no-slip)
- direct solver
- isothermal
- isoviscous
- analytical solution





Quadratic convergence for velocity error, linear convergence for pressure error, as expected.

ToDo:
pressure normalisation?
different cmat, a la schmalholz
To go further:

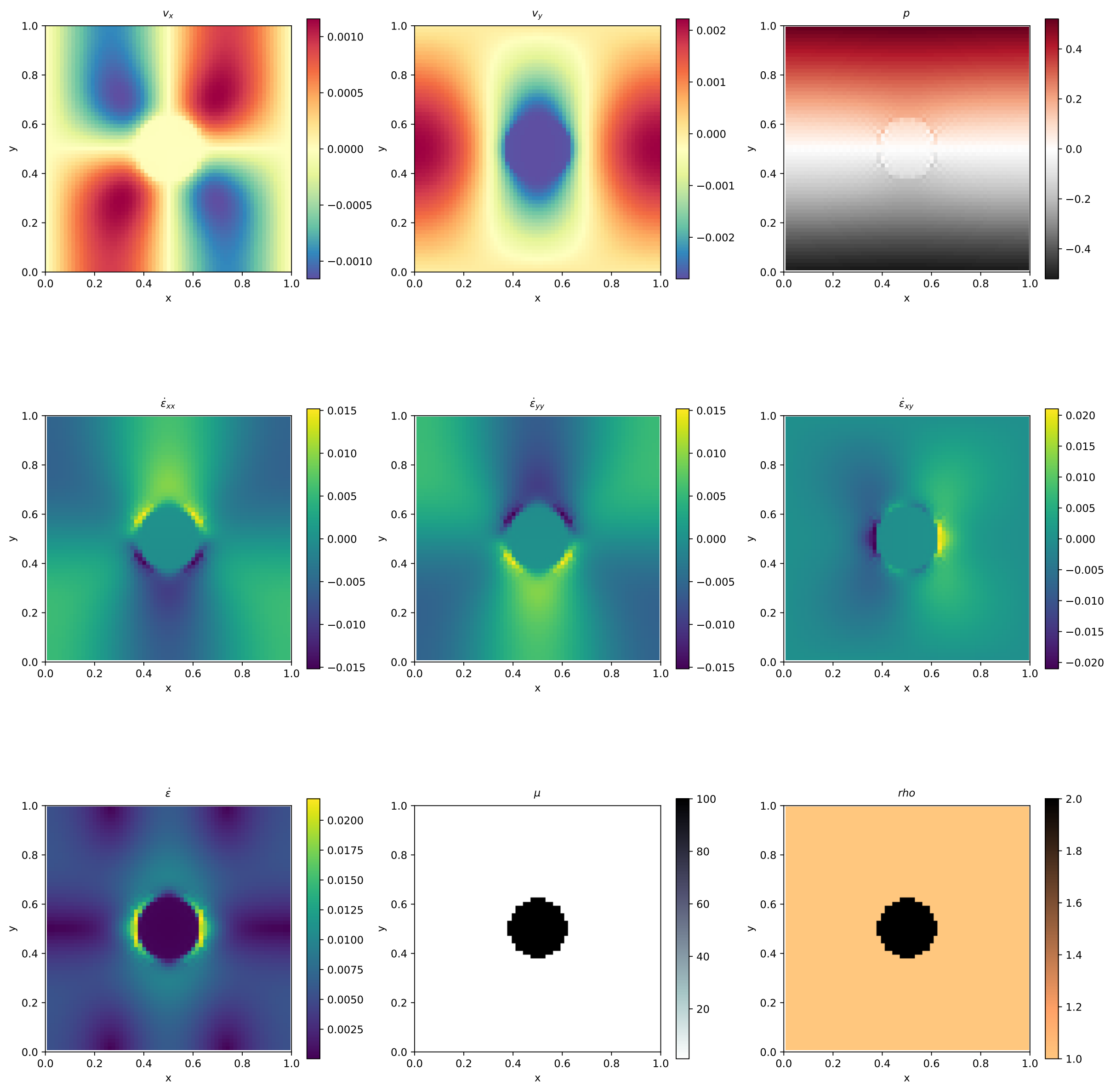
1. make your own analytical solution

6 `fieldstone`: Stokes sphere

Viscosity and density directly computed at the quadrature points.

features

- $Q_1 \times P_0$ element
- incompressible flow
- penalty formulation
- Dirichlet boundary conditions (free-slip)
- direct solver
- isothermal
- non-isoviscous
- buoyancy-driven flow



7 `fieldstone`: Convection in a 2D box

This benchmark deals with the 2-D thermal convection of a fluid of infinite Prandtl number in a rectangular closed cell. In what follows, I carry out the case 1a, 1b, and 1c experiments as shown in [3]: steady convection with constant viscosity in a square box.

The temperature is fixed to zero on top and to ΔT at the bottom, with reflecting symmetry at the sidewalls (i.e. $\partial_x T = 0$) and there are no internal heat sources. Free-slip conditions are implemented on all boundaries.

The Rayleigh number is given by

$$Ra = \frac{\alpha g_y \Delta T h^3}{\kappa \nu} = \frac{\alpha g_y \Delta T h^3 \rho^2 c_p}{k \mu} \tag{9}$$

In what follows, I use the following parameter values: $L_x = L_y = 1$,$\rho_0 = c_P = k = \mu = 1$, $T_0 = 0$, $\alpha = 10^{-2}$, $g = 10^2 Ra$ and I run the model with $Ra = 10^4, 10^5$ and 10^6.

The initial temperature field is given by

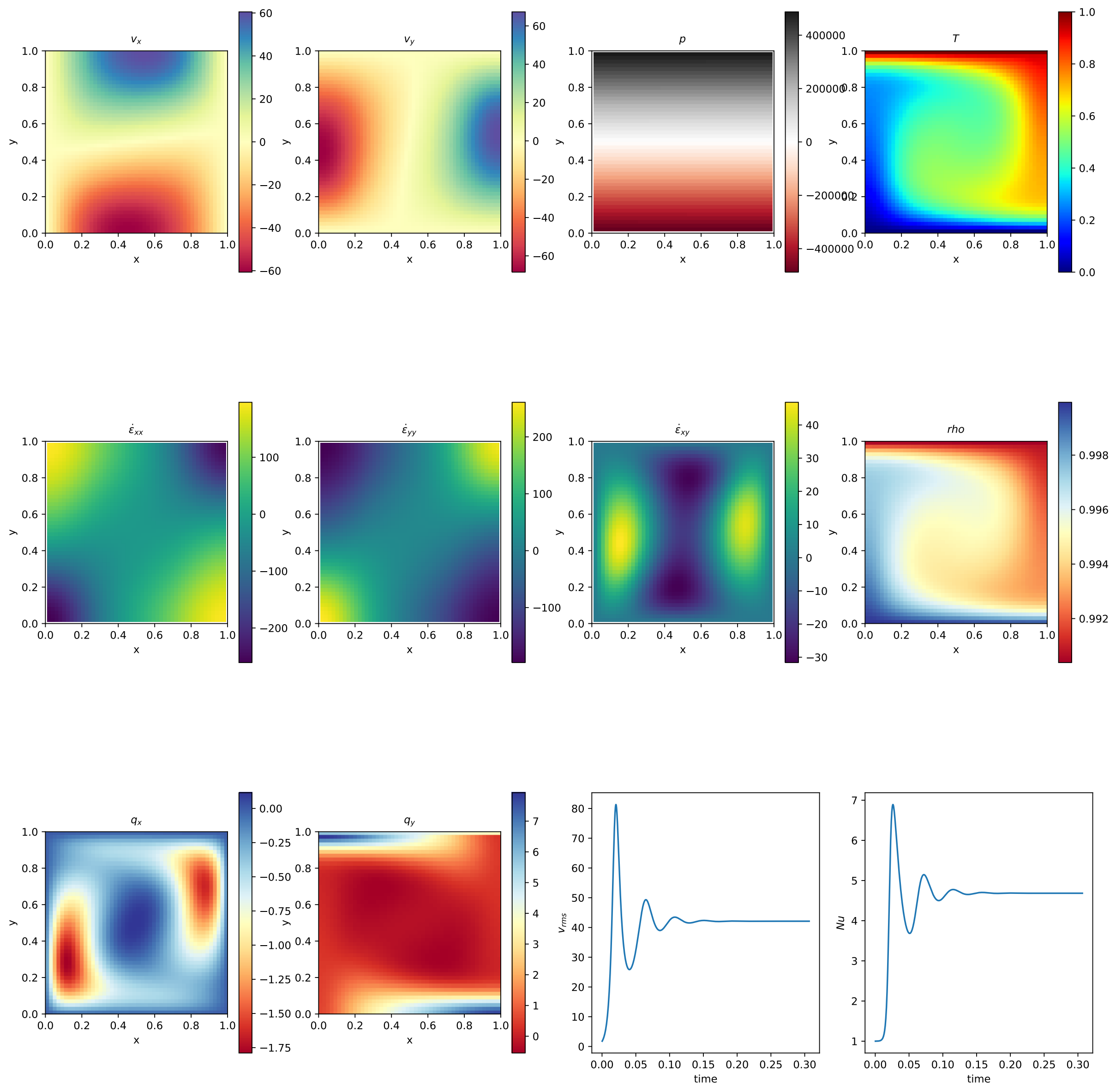
$$T(x, y) = (1 - y) - 0.01 \cos(\pi x) \sin(\pi z) \tag{10}$$

The perturbation in the initial temperature fields leads to a perturbation of the density field and sets the fluid in motion.

Depending on the initial Rayleigh number, the system ultimately reaches a steady state after some time. The steady-state temperature fields of case 1a,b,c are shown hereunder:

features

- $Q_1 \times P_0$ element
- incompressible flow
- penalty formulation
- Dirichlet boundary conditions (free-slip)
- Boussinesq approximation
- direct solver
- non-isothermal
- buoyancy-driven flow
- isoviscous
- CFL-condition



ToDo:
implement steady state criterion
reach steady state
do Ra=1e4, 1e5, 1e6
plot against blankenbach paper and aspect
look at critical Ra number

8 `fieldstone`: solcx benchmark

The SolCx benchmark is intended to test the accuracy of the solution to a problem that has a large jump in the viscosity along a line through the domain. Such situations are common in geophysics: for example, the viscosity in a cold, subducting slab is much larger than in the surrounding, relatively hot mantle material.

The SolCx benchmark computes the Stokes flow field of a fluid driven by spatial density variations, subject to a spatially variable viscosity. Specifically, the domain is $\Omega = [0,1]^2$, gravity is $\boldsymbol{g} = (0,-1)^T$ and the density is given by

$$\rho(x,y) = \sin(\pi y)\cos(\pi x) \tag{11}$$

Boundary conditions are free slip on all of the sides of the domain and the temperature plays no role in this benchmark. The viscosity is prescribed as follows:

$$\mu(x,y) = \left\{ \begin{array}{lll} 1 & for & x < 0.5 \\ 10^6 & for & x > 0.5 \end{array} \right. \tag{12}$$

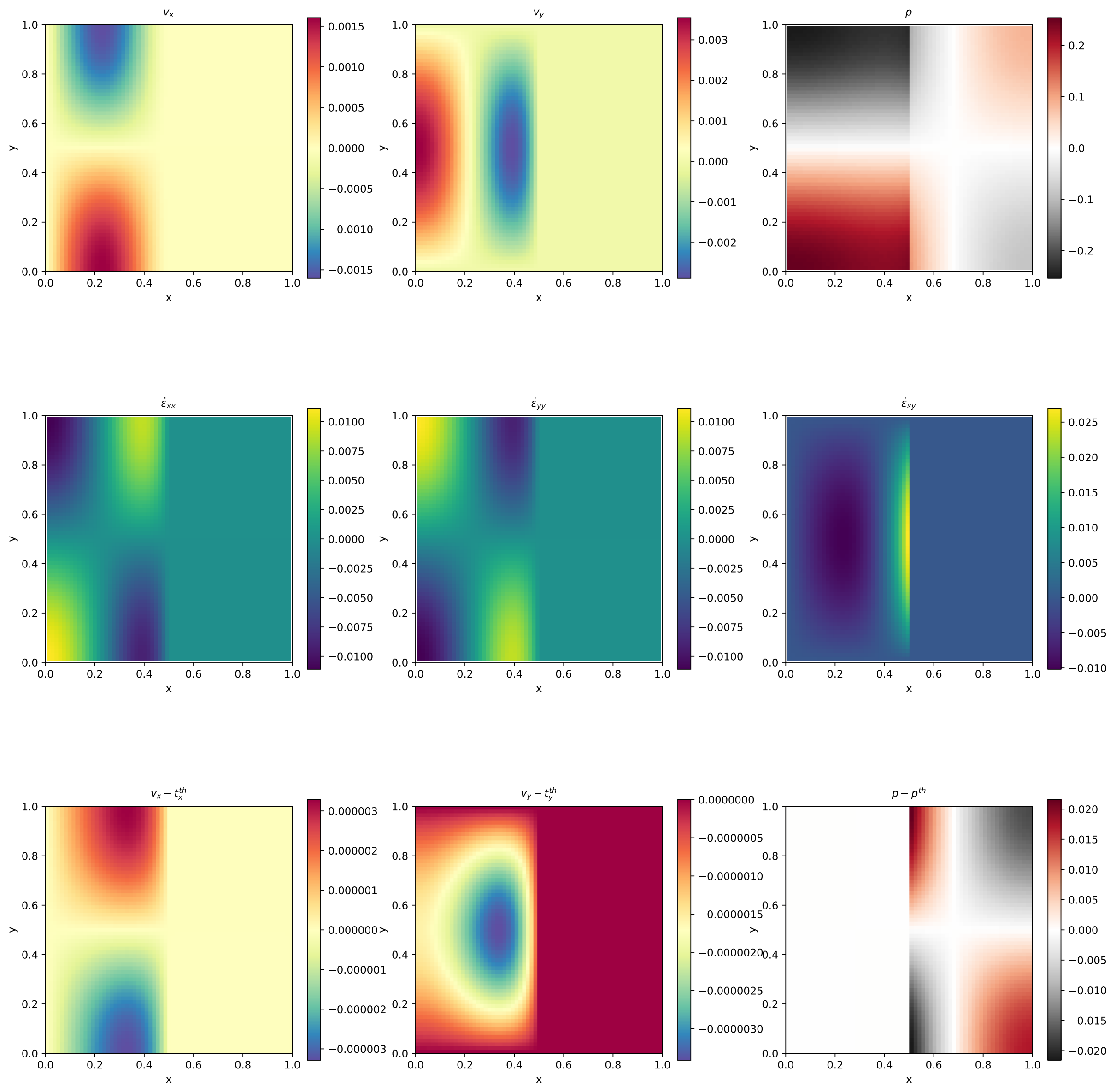
Note the strongly discontinuous viscosity field yields a stagnant flow in the right half of the domain and thereby yields a pressure discontinuity along the interface.

The SolCx benchmark was previously used in [12] (references to earlier uses of the benchmark are available there) and its analytic solution is given in [36]. It has been carried out in [22] and [14]. Note that the source code which evaluates the velocity and pressure fields for both SolCx and SolKz is distributed as part of the open source package Underworld ([25], http://underworldproject.org).

In this particular example, the viscosity is computed analytically at the quadrature points (i.e. tracers are not used to attribute a viscosity to the element). If the number of elements is even in any direction, all elements (and their associated quadrature points) have a constant viscosity(1 or 10^6). If it is odd, then the elements situated at the viscosity jump have half their integration points with $\mu = 1$ and half with $\mu = 10^6$ (which is a pathological case since the used quadrature rule inside elements cannot represent accurately such a jump).

features

- $Q_1 \times P_0$ element
- incompressible flow
- penalty formulation
- Dirichlet boundary conditions (free-slip)
- direct solver
- isothermal
- non-isoviscous
- analytical solution



What we learn from this

9 `fieldstone`: solkz benchmark

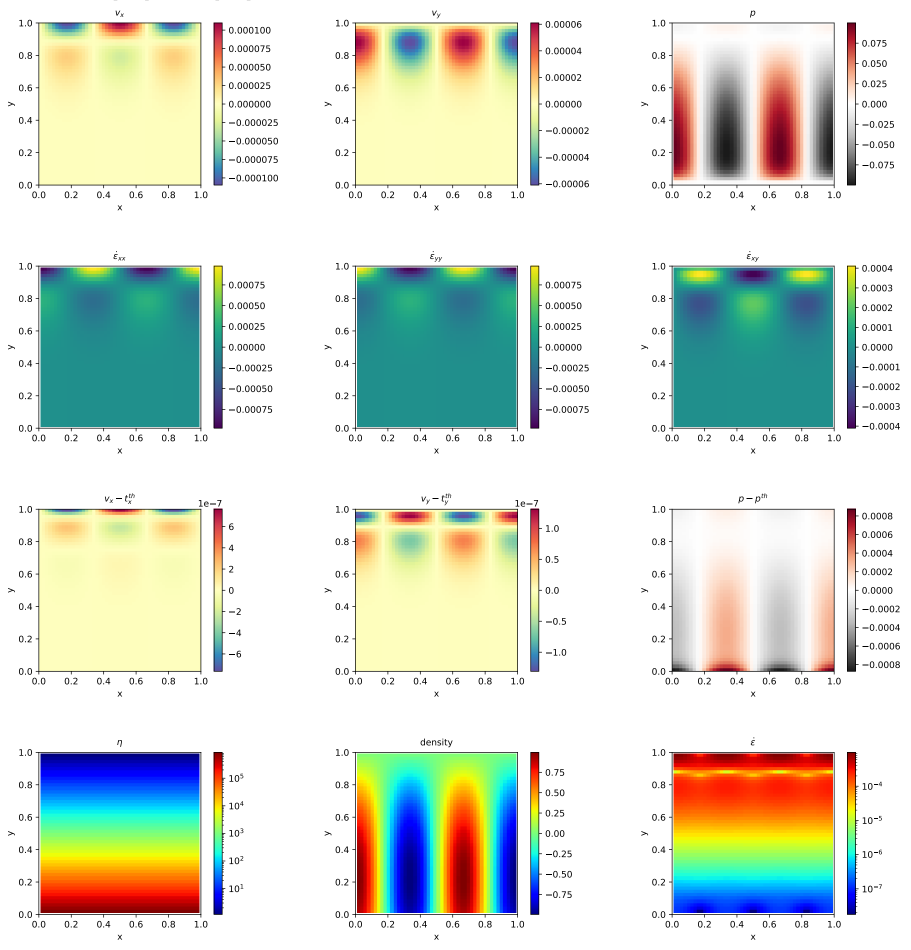
The SolKz benchmark [29] is similar to the SolCx benchmark. but the viscosity is now a function of the space coordinates:

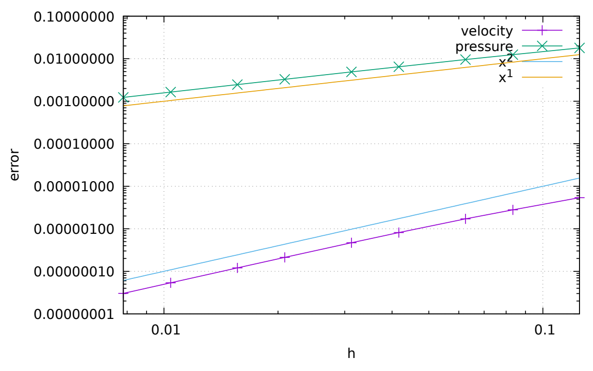
$$\mu(y) = \exp(By) \quad \text{with} \quad B = 13.8155 \tag{13}$$

It is however not a discontinuous function but grows exponentially with the vertical coordinate so that its overall variation is again 10^6. The forcing is again chosen by imposing a spatially variable density variation as follows:

$$\rho(x,y) = \sin(2y)\cos(3\pi x) \tag{14}$$

Free slip boundary conditions are imposed on all sides of the domain. This benchmark is presented in [36] as well and is studied in [12] and [14].





10 `fieldstone`: solvi benchmark

Following SolCx and SolKz, the SolVi inclusion benchmark solves a problem with a discontinuous viscosity field, but in this case the viscosity field is chosen in such a way that the discontinuity is along a circle. Given the regular nature of the grid used by a majority of codes and the present one, this ensures that the discontinuity in the viscosity never aligns to cell boundaries. This in turns leads to almost discontinuous pressures along the interface which are difficult to represent accurately. [30] derived a simple analytic solution for the pressure and velocity fields for a circular inclusion under simple shear and it was used in [9], [32], [12], [22] and [14].

Because of the symmetry of the problem, we only have to solve over the top right quarter of the domain (see Fig. **??**a).

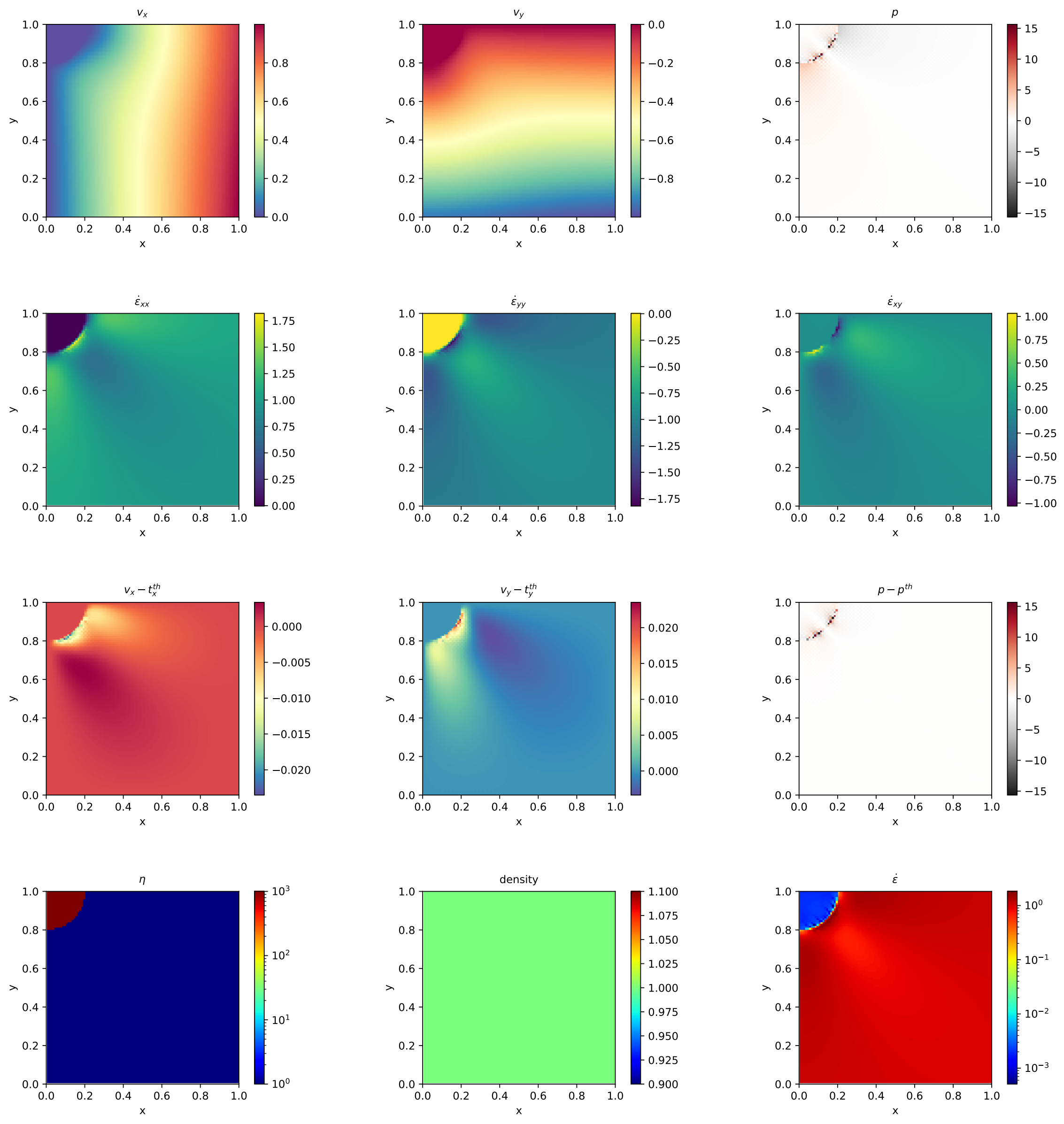
The analytical solution requires a strain rate boundary condition (e.g., pure shear) to be applied far away from the inclusion. In order to avoid using very large domains and/or dealing with this type of boundary condition altogether, the analytical solution is evaluated and imposed on the boundaries of the domain. By doing so, the truncation error introduced while discretizing the strain rate boundary condition is removed.

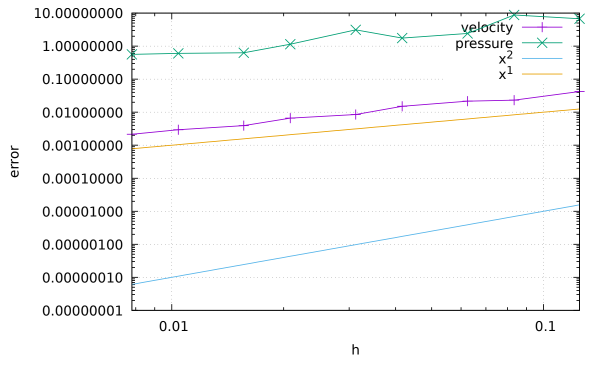
A characteristic of the analytic solution is that the pressure is zero inside the inclusion, while outside it follows the relation

$$p_m = 4\dot{\epsilon}\frac{\mu_m(\mu_i-\mu_m)}{\mu_i+\mu_m}\frac{r_i^2}{r^2}\cos(2\theta) \tag{15}$$

where $\mu_i = 10^3$ is the viscosity of the inclusion and $\mu_m = 1$ is the viscosity of the background media, $\theta = \tan^{-1}(y/x)$, and $\dot{\epsilon} = 1$ is the applied strain rate.

[9] thoroughly investigated this problem with various numerical methods (FEM, FDM), with and without tracers, and conclusively showed how various averagings lead to different results. [12] obtained a first order convergence for both pressure and velocity, while [22] and [14] showed that the use of adaptive mesh refinement in respectively the FEM and FDM yields convergence rates which depend on refinement strategies.


v_x
v_y
p
$\dot{\varepsilon}_{xx}$
$\dot{\varepsilon}_{yy}$
$\dot{\varepsilon}_{xy}$
$v_x - t_x^{th}$
$v_y - t_y^{th}$
$p - p^{th}$
η
density
$\dot{\varepsilon}$
x
y


velocity
pressure
x^2
x^1
error
h

11 `fieldstone`: the indentor benchmark

The punch benchmark is one of the few boundary value problems involving plastic solids for which there exists an exact solution. Such solutions are usually either for highly simplified geometries (spherical or axial symmetry, for instance) or simplified material models (such as rigid plastic solids) [21].

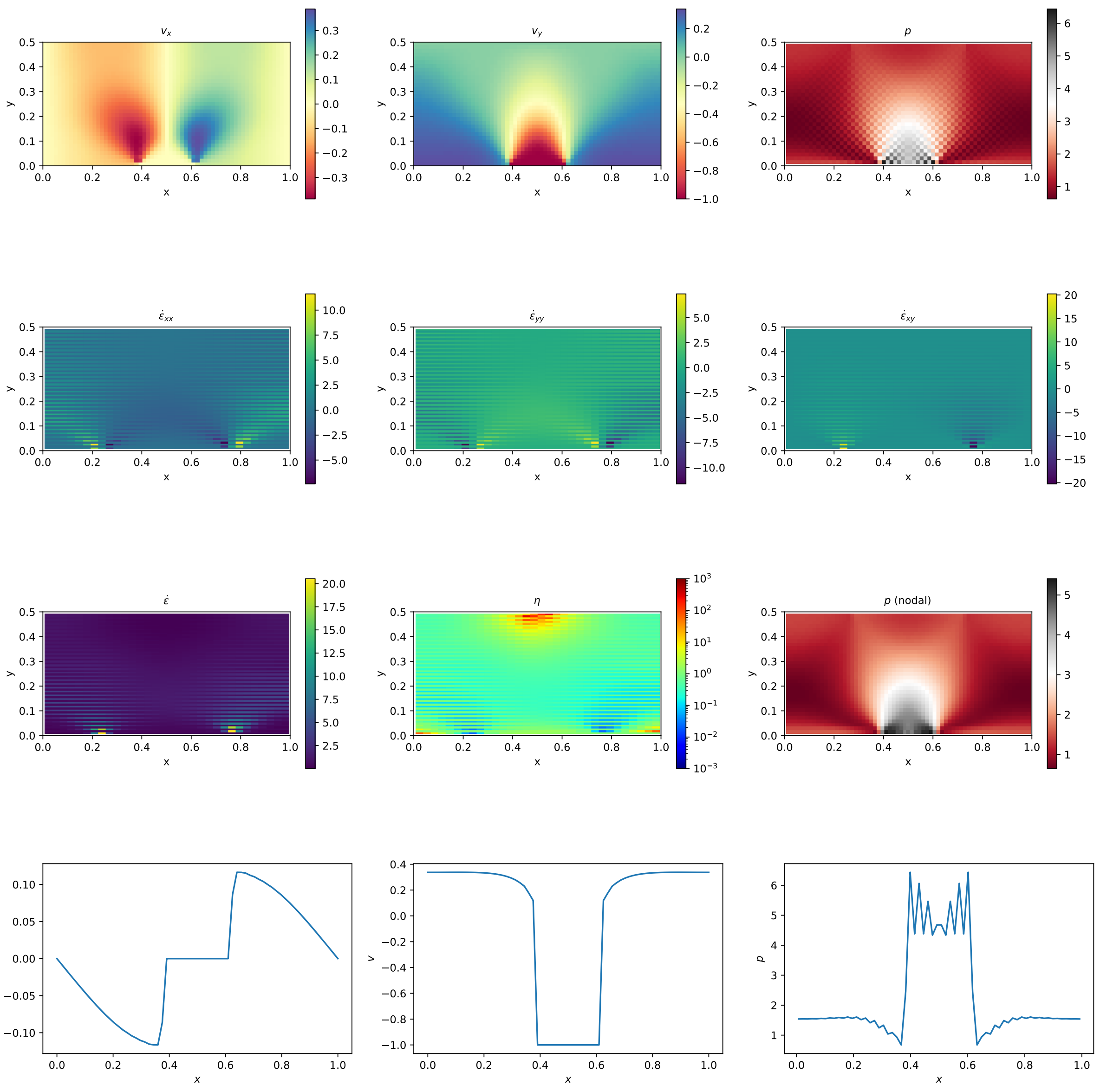
In this experiment, a rigid punch indents a rigid plastic half space; the slip line field theory gives exact solutions as shown in Fig. **??**a. The plane strain formulation of the equations and the detailed solution to the problem were derived in the Appendix of [34] and are also presented in [13].

The two dimensional punch problem has been extensively studied numerically for the past 40 years [39, 38, 6, 5, 19, 35, 4, 27] and has been used to draw a parallel with the tectonics of eastern China in the context of the India-Eurasia collision [33, 24]. It is also worth noting that it has been carried out in one form or another in series of analogue modelling articles concerning the same region, with a rigid indenter colliding with a rheologically stratified lithosphere [26, 8, 20].

Numerically, the one-time step punch experiment is performed on a two-dimensional domain of purely plastic von Mises material. Given that the von Mises rheology yield criterion does not depend on pressure, the density of the material and/or the gravity vector is set to zero. Sides are set to free slip boundary conditions, the bottom to no slip, while a vertical velocity $(0, -v_p)$ is prescribed at the top boundary for nodes whose x coordinate is within $[L_x/2 - \delta/2, L_x/2 + \delta/2]$.

The following parameters are used: $L_x = 1$, $L_y = 0.5$, $\mu_{min} = 10^{-3}$, $\mu_{max} = 10^3$, $v_p = 1$, $\delta = 0.123456789$ and the yield value of the material is set to $k = 1$.

The analytical solution predicts that the angle of the shear bands stemming from the sides of the punch is $\pi/4$, that the pressure right under the punch is $1 + \pi$, and that the velocity of the rigid blocks on each side of the punch is $v_p/\sqrt{2}$ (this is simply explained by invoking conservation of mass).



ToDo: smooth punch

12 `fieldstone`: the annulus benchmark

This benchmark is based on Thieulot & Puckett [Subm.] in which an analytical solution to the isoviscous incompressible Stokes equations is derived in an annulus geometry. The velocity and pressure fields are as follows:

$$
\begin{aligned}
v_r(r,\theta) &= g(r) k \sin(k\theta), && (16)\\
v_\theta(r,\theta) &= f(r)\cos(k\theta), && (17)\\
p(r,\theta) &= k h(r) \sin(k\theta), && (18)\\
\rho(r,\theta) &= \aleph(r) k \sin(k\theta), && (19)
\end{aligned}
$$

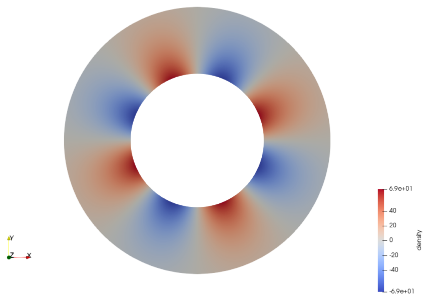
with

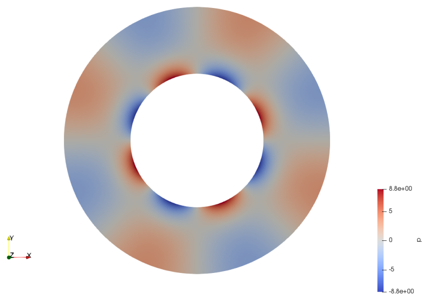
$$
\begin{aligned}
f(r) &= Ar + B/r, && (20)\\
g(r) &= \frac{A}{2} r + \frac{B}{r}\ln r + \frac{C}{r}, && (21)\\
h(r) &= \frac{2g(r) - f(r)}{r}, && (22)\\
\aleph(r) &= g'' - \frac{g'}{r} - \frac{g}{r^2}(k^2-1) + \frac{f}{r^2} + \frac{f'}{r}, && (23)\\
A &= -C\frac{2(\ln R_1 - \ln R_2)}{R_2^2 \ln R_1 - R_1^2 \ln R_2}, && (24)\\
B &= -C\frac{R_2^2 - R_1^2}{R_2^2 \ln R_1 - R_1^2 \ln R_2}. && (25)
\end{aligned}
$$

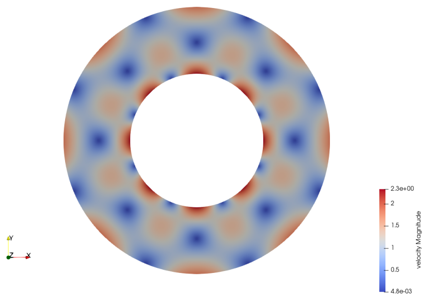
The parameters A and B are chosen so that $v_r(R_1) = v_r(R_2) = 0$, i.e. the velocity is tangential to both inner and outer surfaces. The gravity vector is radial and of unit length. In the present case, we set $R_1 = 1$, $R_2 = 2$ and $C = -1$.

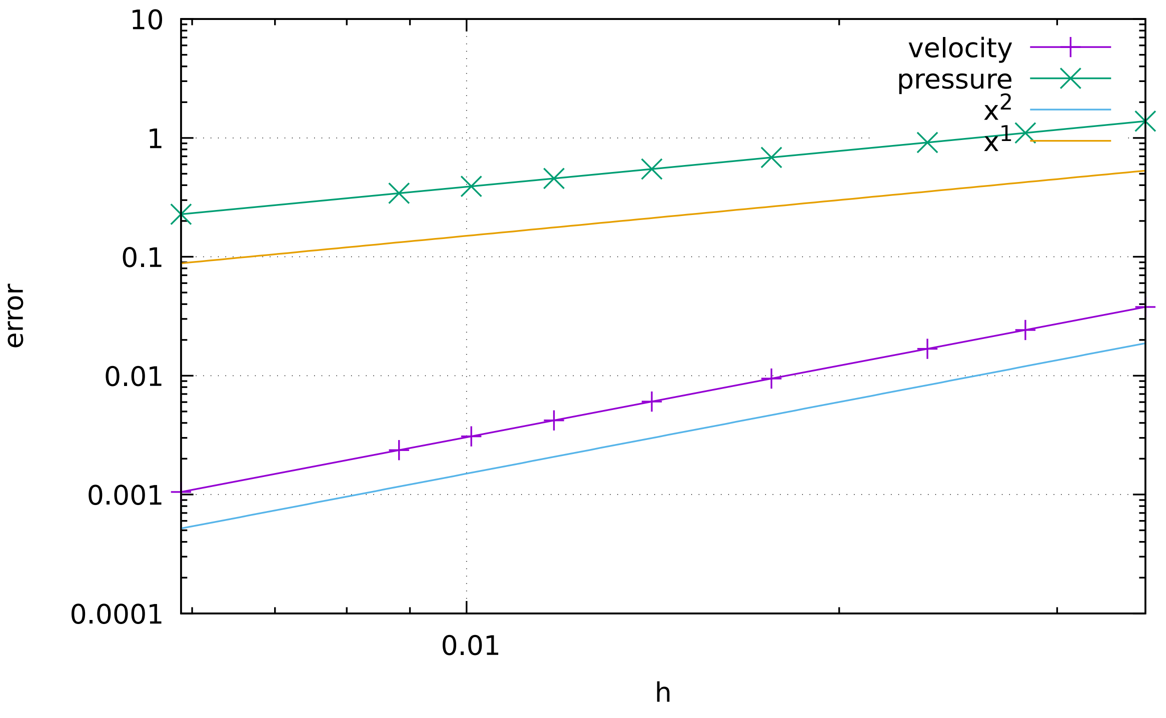
features

- $Q_1 \times P_0$ element
- incompressible flow
- penalty formulation
- Dirichlet boundary conditions
- direct solver
- isothermal
- isoviscous
- analytical solution
- annulus geometry
- elemental boundary conditions





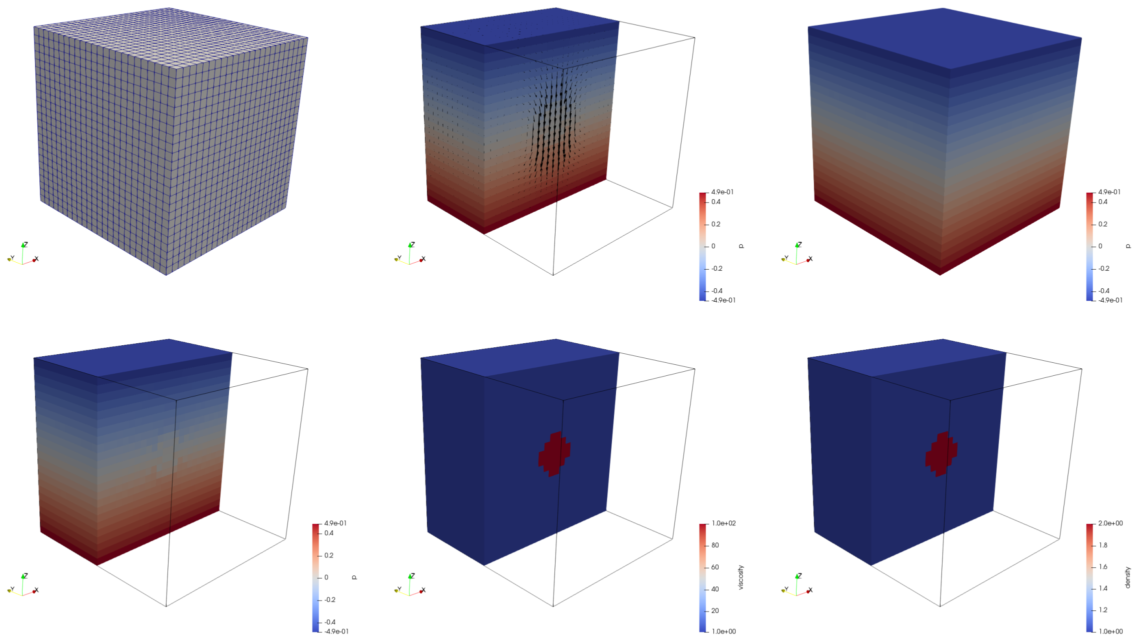



10
1
0.1
0.01
0.001
0.0001
error
0.01
h
velocity
pressure
x^2
x^1

13 `fieldstone`: stokes sphere (3D)

features

- $Q_1 \times P_0$ element
- incompressible flow
- penalty formulation
- Dirichlet boundary conditions (free-slip)
- direct solver
- isothermal
- non-isoviscous
- 3D
- elemental b.c.
- buoyancy driven

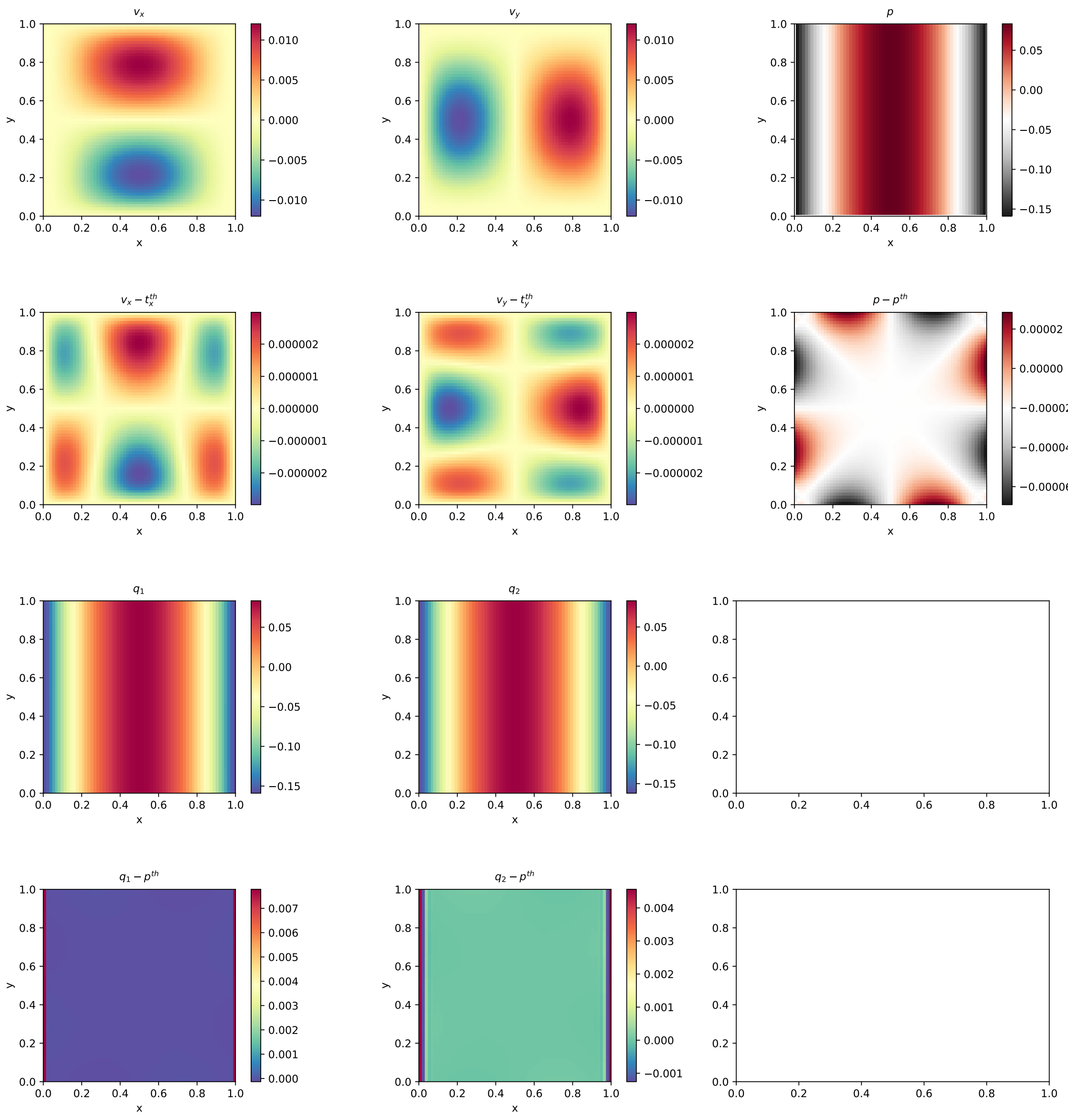


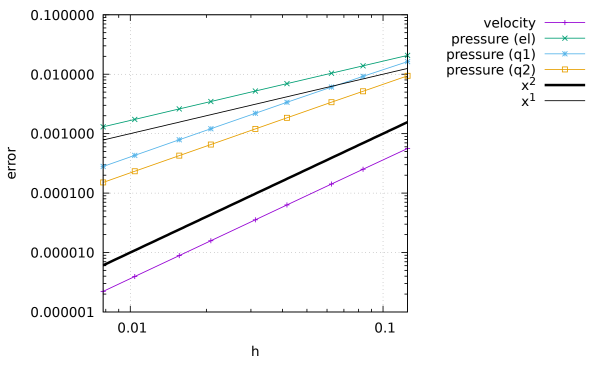
resolution is 24x24x24

14 `fieldstone`: consistent pressure recovery

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

q_1 is smoothed pressure obtained with the center-to-node approach.
q_2 is recovered pressure obtained with [37].
All three fulfill the zero average condition: $\int p d\Omega = 0$.





In terms of pressure error, q_2 is better than q_1 which is better than elemental.
QUESTION: why are the averages exactly zero ?!
TODO:

- add randomness to internal node positions.
- look at elefant algorithms

15 `fieldstone`: the Particle in Cell technique (1) - the effect of averaging

A The main codes in computational geodynamics

In what follows I make a quick inventory of the main codes of computational geodynamics, for crust, lithosphere and/or mantle modelling.

A.1 ADELI

A.2 ASPECT

A.3 CITCOMS and CITCOMCU

A.4 DOUAR

A.5 GAIA

A.6 GALE

A.7 GTECTON

A.8 ELVIS

A.9 ELEFANT

A.10 ELLIPSIS

A.11 FANTOM

A.12 FLUIDITY

A.13 LAMEM

A.14 MILAMIN

A.15 PARAVOZ/FLAMAR

A.16 PTATIN

A.17 RHEA

A.18 SEPRAN

A.19 SOPALE

A.20 STAGYY

A.21 SULEC

SULEC is a finite element code that solves the incompressible Navier-Stokes equations for slow creeping flows. The code is developed by Susan Ellis (GNS Sciences, NZ) and Susanne Buiter (NGU).

A.22 TERRA

A.23 UNDERWORLD 1&2

B fieldstone.py

C fieldstone_stokes_sphere.py

D fieldstone_convection_box.py

E fieldstone_solcx.py

```
import numpy as np
import math as math
import sys as sys
import scipy
import scipy.sparse as sps
from scipy.sparse.linalg.dsolve import linsolve
import time as time
import matplotlib.pyplot as plt
import tkinter
import solcx as solcx

#------------------------------------------------------------------------------
def viscosity(x,y):
    if x<0.5:
       val=1.
    else:
       val=1.e6
    return val

def density(x,y):
    val=math.sin(math.pi*y)*math.cos(math.pi*x)
    return val
#------------------------------------------------------------------------------

print("-----------------------------")
print("----------fieldstone----------")
print("-----------------------------")

# declare variables
print("variable declaration")

m=4      # number of nodes making up an element
ndof=2  # number of degrees of freedom per node

Lx=1.  # horizontal extent of the domain
Ly=1.  # vertical extent of the domain

gx=0
gy=1

assert (Lx>0.), "Lx should be positive"
assert (Ly>0.), "Ly should be positive"

# allowing for argument parsing through command line
if int(len(sys.argv) == 4):
   nelx = int(sys.argv[1])
   nely = int(sys.argv[2])
   visu = int(sys.argv[3])
else:
   nelx = 64
   nely = 64
   visu = 1

assert (nelx>0.), "nnx should be positive"
assert (nely>0.), "nny should be positive"

nnx=nelx+1  # number of elements, x direction
nny=nely+1  # number of elements, y direction

nnp=nnx*nny  # number of nodes

nel=nelx*nely  # number of elements, total

penalty=1.e7  # penalty coefficient value

Nfem=nnp*ndof  # Total number of degrees of freedom

eps=1.e-10

sqrt3=np.sqrt(3.)

# declare arrays
print("declaring arrays")
```

```
#################################################################
# grid point setup
#################################################################

print("grid_point_setup")

x = np.empty(nnp, dtype=np.float64)  # x coordinates
y = np.empty(nnp, dtype=np.float64)  # y coordinates
counter = 0
for j in range(0, nny):
    for i in range(0, nnx):
        x[counter]=i*Lx/float(nelx)
        y[counter]=j*Ly/float(nely)
        counter += 1

#################################################################
# connectivity
#################################################################

print("connectivity")

icon =np.zeros((m, nel),dtype=np.int16)
counter = 0
for j in range(0, nely):
    for i in range(0, nelx):
        icon[0, counter] = i + j * (nelx + 1)
        icon[1, counter] = i + 1 + j * (nelx + 1)
        icon[2, counter] = i + 1 + (j + 1) * (nelx + 1)
        icon[3, counter] = i + (j + 1) * (nelx + 1)
        counter += 1

# for iel in range (0,nel):
#     print ("iel=",iel)
#     print ("node 1",icon[0][iel],"at pos.",x[icon[0][iel]], y[icon[0][iel]])
#     print ("node 2",icon[1][iel],"at pos.",x[icon[1][iel]], y[icon[1][iel]])
#     print ("node 3",icon[2][iel],"at pos.",x[icon[2][iel]], y[icon[2][iel]])
#     print ("node 4",icon[3][iel],"at pos.",x[icon[3][iel]], y[icon[3][iel]])

#################################################################
# define boundary conditions
#################################################################

print("defining_boundary_conditions")

bc_fix = np.zeros(Nfem, dtype=np.bool)  # boundary condition, yes/no
bc_val = np.zeros(Nfem, dtype=np.float64)  # boundary condition, value
for i in range(0, nnp):
    if x[i]<eps:
       bc_fix[i*ndof]   = True ; bc_val[i*ndof]   = 0.
    if x[i]>(Lx-eps):
       bc_fix[i*ndof]   = True ; bc_val[i*ndof]   = 0.
    if y[i]<eps:
       bc_fix[i*ndof+1] = True ; bc_val[i*ndof+1] = 0.
    if y[i]>(Ly-eps):
       bc_fix[i*ndof+1] = True ; bc_val[i*ndof+1] = 0.

#################################################################
# build FE matrix
#################################################################

print("building_FE_matrix")

a_mat = np.zeros((Nfem,Nfem),dtype=np.float64)  # matrix of Ax=b
b_mat = np.zeros((3,ndof*m),dtype=np.float64)   # gradient matrix B
rhs   = np.zeros(Nfem,dtype=np.float64)         # right hand side of Ax=b
N     = np.zeros(m,dtype=np.float64)            # shape functions
dNdx  = np.zeros(m,dtype=np.float64)            # shape functions derivatives
dNdy  = np.zeros(m,dtype=np.float64)            # shape functions derivatives
dNdr  = np.zeros(m,dtype=np.float64)            # shape functions derivatives
dNds  = np.zeros(m,dtype=np.float64)            # shape functions derivatives
u     = np.zeros(nnp,dtype=np.float64)          # x-component velocity
v     = np.zeros(nnp,dtype=np.float64)          # y-component velocity
k_mat = np.array([[1,1,0],[1,1,0],[0,0,0]],dtype=np.float64)
c_mat = np.array([[2,0,0],[0,2,0],[0,0,1]],dtype=np.float64)

for iel in range(0, nel):
```

```
# set 2 arrays to 0 every loop
b_el = np.zeros(m * ndof)
a_el = np.zeros((m * ndof, m * ndof), dtype=float)

# integrate viscous term at 4 quadrature points
for iq in [-1, 1]:
    for jq in [-1, 1]:

        # position & weight of quad. point
        rq=iq/sqrt3
        sq=jq/sqrt3
        wq=1.*1.

        # calculate shape functions
        N[0]=0.25*(1.-rq)*(1.-sq)
        N[1]=0.25*(1.+rq)*(1.-sq)
        N[2]=0.25*(1.+rq)*(1.+sq)
        N[3]=0.25*(1.-rq)*(1.+sq)

        # calculate shape function derivatives
        dNdr[0]=-0.25*(1.-sq) ; dNds[0]=-0.25*(1.-rq)
        dNdr[1]=+0.25*(1.-sq) ; dNds[1]=-0.25*(1.+rq)
        dNdr[2]=+0.25*(1.+sq) ; dNds[2]=+0.25*(1.+rq)
        dNdr[3]=-0.25*(1.+sq) ; dNds[3]=+0.25*(1.-rq)

        # calculate jacobian matrix
        jcb = np.zeros((2, 2),dtype=float)
        for k in range(0,m):
            jcb[0, 0] += dNdr[k]*x[icon[k,iel]]
            jcb[0, 1] += dNdr[k]*y[icon[k,iel]]
            jcb[1, 0] += dNds[k]*x[icon[k,iel]]
            jcb[1, 1] += dNds[k]*y[icon[k,iel]]

        # calculate the determinant of the jacobian
        jcob = np.linalg.det(jcb)

        # calculate inverse of the jacobian matrix
        jcbi = np.linalg.inv(jcb)

        # compute dNdx & dNdy
        xq=0.0
        yq=0.0
        for k in range(0, m):
            xq+=N[k]*x[icon[k,iel]]
            yq+=N[k]*y[icon[k,iel]]
            dNdx[k]=jcbi[0,0]*dNdr[k]+jcbi[0,1]*dNds[k]
            dNdy[k]=jcbi[1,0]*dNdr[k]+jcbi[1,1]*dNds[k]

        # construct 3x8 b_mat matrix
        for i in range(0, m):
            b_mat[0:3, 2*i:2*i+2] = [[dNdx[i],0.      ],
                                     [0.      ,dNdy[i]],
                                     [dNdy[i],dNdx[i]]]

        # compute elemental a_mat matrix
        a_el += b_mat.T.dot(c_mat.dot(b_mat))*viscosity(xq,yq)*wq*jcob

        # compute elemental rhs vector
        for i in range(0, m):
            b_el[2*i  ]+=N[i]*jcob*wq*density(xq,yq)*gx
            b_el[2*i+1]+=N[i]*jcob*wq*density(xq,yq)*gy

# integrate penalty term at 1 point
rq=0.
sq=0.
wq=2.*2.

N[0]=0.25*(1.-rq)*(1.-sq)
N[1]=0.25*(1.+rq)*(1.-sq)
N[2]=0.25*(1.+rq)*(1.+sq)
N[3]=0.25*(1.-rq)*(1.+sq)

dNdr[0]=-0.25*(1.-sq) ; dNds[0]=-0.25*(1.-rq)
dNdr[1]=+0.25*(1.-sq) ; dNds[1]=-0.25*(1.+rq)
dNdr[2]=+0.25*(1.+sq) ; dNds[2]=+0.25*(1.+rq)
```

```
    dNdr[3]=-0.25*(1.+sq) ; dNds[3]=+0.25*(1.-rq)

    # compute the jacobian
    jcb=np.zeros((2,2),dtype=float)
    for k in range(0, m):
        jcb[0,0]+=dNdr[k]*x[icon[k,iel]]
        jcb[0,1]+=dNdr[k]*y[icon[k,iel]]
        jcb[1,0]+=dNds[k]*x[icon[k,iel]]
        jcb[1,1]+=dNds[k]*y[icon[k,iel]]

    # calculate determinant of the jacobian
    jcob = np.linalg.det(jcb)

    # calculate the inverse of the jacobian
    jcbi = np.linalg.inv(jcb)

    # compute dNdx and dNdy
    for k in range(0,m):
        dNdx[k]=jcbi[0,0]*dNdr[k]+jcbi[0,1]*dNds[k]
        dNdy[k]=jcbi[1,0]*dNdr[k]+jcbi[1,1]*dNds[k]

    # compute gradient matrix
    for i in range(0,m):
        b_mat[0:3,2*i:2*i+2]=[[dNdx[i],0.     ],
                              [0.     ,dNdy[i]],
                              [dNdy[i],dNdx[i]]]

    # compute elemental matrix
    a_el += b_mat.T.dot(k_mat.dot(b_mat))*penalty*wq*jcob

    # assemble matrix a_mat and right hand side rhs
    for k1 in range(0,m):
        for i1 in range(0,ndof):
            ikk=ndof*k1          +i1
            m1 =ndof*icon[k1,iel]+i1
            for k2 in range(0,m):
                for i2 in range(0,ndof):
                    jkk=ndof*k2          +i2
                    m2 =ndof*icon[k2,iel]+i2
                    a_mat[m1,m2]+=a_el[ikk,jkk]
            rhs[m1]+=b_el[ikk]

#################################################################
# impose boundary conditions
#################################################################

print("imposing boundary conditions")

for i in range(0, Nfem):
    if bc_fix[i]:
       a_matref = a_mat[i,i]
       for j in range(0,Nfem):
           rhs[j]-= a_mat[i, j] * bc_val[i]
           a_mat[i,j]=0.
           a_mat[j,i]=0.
           a_mat[i,i] = a_matref
       rhs[i]=a_matref*bc_val[i]

#print("a_mat (m,M) = %.4f %.4f" %(np.min(a_mat),np.max(a_mat)))
#print("rhs   (m,M) = %.6f %.6f" %(np.min(rhs),np.max(rhs)))

#################################################################
# solve system
#################################################################

start = time.time()
sol = sps.linalg.spsolve(sps.csr_matrix(a_mat),rhs)
print("solve time: %.3f s" % (time.time() - start))
print("----------------------------------------")

#####################################################################
# put solution into separate x,y velocity arrays
#####################################################################

u,v=np.reshape(sol,(nnp,2)).T
```

```
print("u (m,M) %.4f %.4f " %(np.min(u),np.max(u)))
print("v (m,M) %.4f %.4f " %(np.min(v),np.max(v)))

np.savetxt('velocity.ascii',np.array([x,y,u,v]).T,header='# x,y,u,v')

#####################################################################
# retrieve pressure
#####################################################################

xc = np.zeros(nel,dtype=np.float64)
yc = np.zeros(nel,dtype=np.float64)
p  = np.zeros(nel,dtype=np.float64)
exx = np.zeros(nel,dtype=np.float64)
eyy = np.zeros(nel,dtype=np.float64)
exy = np.zeros(nel,dtype=np.float64)

for iel in range(0,nel):

    rq = 0.0
    sq = 0.0
    wq = 2.0 * 2.0

    N[0]=0.25*(1.-rq)*(1.-sq)
    N[1]=0.25*(1.+rq)*(1.-sq)
    N[2]=0.25*(1.+rq)*(1.+sq)
    N[3]=0.25*(1.-rq)*(1.+sq)

    dNdr[0]=-0.25*(1.-sq) ; dNds[0]=-0.25*(1.-rq)
    dNdr[1]=+0.25*(1.-sq) ; dNds[1]=-0.25*(1.+rq)
    dNdr[2]=+0.25*(1.+sq) ; dNds[2]=+0.25*(1.+rq)
    dNdr[3]=-0.25*(1.+sq) ; dNds[3]=+0.25*(1.-rq)

    jcb=np.zeros((2,2),dtype=float)
    for k in range(0, m):
        jcb[0,0]+=dNdr[k]*x[icon[k,iel]]
        jcb[0,1]+=dNdr[k]*y[icon[k,iel]]
        jcb[1,0]+=dNds[k]*x[icon[k,iel]]
        jcb[1,1]+=dNds[k]*y[icon[k,iel]]

    # calculate determinant of the jacobian
    jcob=np.linalg.det(jcb)

    # calculate the inverse of the jacobian
    jcbi=np.linalg.inv(jcb)

    for k in range(0, m):
        dNdx[k]=jcbi[0,0]*dNdr[k]+jcbi[0,1]*dNds[k]
        dNdy[k]=jcbi[1,0]*dNdr[k]+jcbi[1,1]*dNds[k]

    for k in range(0, m):
        xc[iel] += N[k]*x[icon[k,iel]]
        yc[iel] += N[k]*y[icon[k,iel]]
        exx[iel] += dNdx[k]*u[icon[k,iel]]
        eyy[iel] += dNdy[k]*v[icon[k,iel]]
        exy[iel] += 0.5*dNdy[k]*u[icon[k,iel]]+ 0.5*dNdx[k]*v[icon[k,iel]]

    p[iel]=-penalty*(exx[iel]+eyy[iel])

print("p (m,M) %.4f %.4f " %(np.min(p),np.max(p)))
print("exx (m,M) %.4f %.4f " %(np.min(exx),np.max(exx)))
print("eyy (m,M) %.4f %.4f " %(np.min(eyy),np.max(eyy)))
print("exy (m,M) %.4f %.4f " %(np.min(exy),np.max(exy)))

np.savetxt('pressure.ascii',np.array([xc,yc,p]).T,header='# xc,yc,p')

np.savetxt('strainrate.ascii',np.array([xc,yc,exx,eyy,exy]).T,header='# xc,yc,exx,eyy,exy')

#################################################################
# compute error
#################################################################

error_u = np.empty(nnp,dtype=np.float64)
error_v = np.empty(nnp,dtype=np.float64)
error_p = np.empty(nel,dtype=np.float64)

for i in range(0,nnp):
    ui,vi,pi=solcx.SolCxSolution(x[i],y[i])
    error_u[i]=u[i]-ui
    error_v[i]=v[i]-vi

for i in range(0,nel):
    ui,vi,pi=solcx.SolCxSolution(xc[i],yc[i])
    error_p[i]=p[i]-pi

errv=0.
errp=0.
for iel in range (0,nel):
    for iq in [-1,1]:
        for jq in [-1,1]:
            rq=iq/sqrt3
            sq=jq/sqrt3
            wq=1.*1.
            N[0]=0.25*(1.-rq)*(1.-sq)
            N[1]=0.25*(1.+rq)*(1.-sq)
            N[2]=0.25*(1.+rq)*(1.+sq)
            N[3]=0.25*(1.-rq)*(1.+sq)
            dNdr[0]=-0.25*(1.-sq)  ; dNds[0]=-0.25*(1.-rq)
            dNdr[1]=+0.25*(1.-sq)  ; dNds[1]=-0.25*(1.+rq)
            dNdr[2]=+0.25*(1.+sq)  ; dNds[2]=+0.25*(1.+rq)
            dNdr[3]=-0.25*(1.+sq)  ; dNds[3]=+0.25*(1.-rq)
            jcb=np.zeros((2,2),dtype=np.float64)
            for k in range(0,m):
                jcb[0,0]+=dNdr[k]*x[icon[k,iel]]
                jcb[0,1]+=dNdr[k]*y[icon[k,iel]]
                jcb[1,0]+=dNds[k]*x[icon[k,iel]]
                jcb[1,1]+=dNds[k]*y[icon[k,iel]]
            jcob=np.linalg.det(jcb)
            xq=0.0
            yq=0.0
            uq=0.0
            vq=0.0
            for k in range(0,m):
                xq+=N[k]*x[icon[k,iel]]
                yq+=N[k]*y[icon[k,iel]]
                uq+=N[k]*u[icon[k,iel]]
                vq+=N[k]*v[icon[k,iel]]
            ui,vi,pi=solcx.SolCxSolution(xq,yq)
            errv+=((uq-ui)**2+(vq-vi)**2)*wq*jcob
            errp+=(p[iel]-pi)**2*wq*jcob

errv=np.sqrt(errv)
errp=np.sqrt(errp)

print("nel= %6d ; errv= %.8f ; errp= %.8f" %(nel,errv,errp))

#####################################################################
# plot of solution
#####################################################################

u_temp=np.reshape(u,(nnx,nny))
v_temp=np.reshape(v,(nnx,nny))
p_temp=np.reshape(p,(nelx,nely))
exx_temp=np.reshape(exx,(nelx,nely))
eyy_temp=np.reshape(eyy,(nelx,nely))
exy_temp=np.reshape(exy,(nelx,nely))
error_u_temp=np.reshape(error_u,(nnx,nny))
error_v_temp=np.reshape(error_v,(nnx,nny))
error_p_temp=np.reshape(error_p,(nelx,nely))

fig,axes = plt.subplots(nrows=3,ncols=3,figsize=(18,18))

uextent=(np.amin(x),np.amax(x),np.amin(y),np.amax(y))
pextent=(np.amin(xc),np.amax(xc),np.amin(yc),np.amax(yc))

im = axes[0][0].imshow(u_temp,extent=uextent,cmap='Spectral_r',interpolation='nearest')
axes[0][0].set_title('$v_x$', fontsize=10, y=1.01)
axes[0][0].set_xlabel('x')
axes[0][0].set_ylabel('y')
fig.colorbar(im,ax=axes[0][0])

im = axes[0][1].imshow(v_temp,extent=uextent,cmap='Spectral_r',interpolation='nearest')
axes[0][1].set_title('$v_y$', fontsize=10, y=1.01)
```

```
axes[0][1].set_xlabel('x')
axes[0][1].set_ylabel('y')
fig.colorbar(im,ax=axes[0][1])

im = axes[0][2].imshow(p_temp,extent=pextent,cmap='RdGy_r',interpolation='nearest')
axes[0][2].set_title('$p$', fontsize=10, y=1.01)
axes[0][2].set_xlim(0,Lx)
axes[0][2].set_ylim(0,Ly)
axes[0][2].set_xlabel('x')
axes[0][2].set_ylabel('y')
fig.colorbar(im,ax=axes[0][2])

im = axes[1][0].imshow(exx_temp,extent=pextent, cmap='viridis',interpolation='nearest')
axes[1][0].set_title('$\dot{\epsilon}_{xx}$',fontsize=10, y=1.01)
axes[1][0].set_xlim(0,Lx)
axes[1][0].set_ylim(0,Ly)
axes[1][0].set_xlabel('x')
axes[1][0].set_ylabel('y')
fig.colorbar(im,ax=axes[1][0])

im = axes[1][1].imshow(eyy_temp,extent=pextent,cmap='viridis',interpolation='nearest')
axes[1][1].set_title('$\dot{\epsilon}_{yy}$',fontsize=10,y=1.01)
axes[1][1].set_xlim(0,Lx)
axes[1][1].set_ylim(0,Ly)
axes[1][1].set_xlabel('x')
axes[1][1].set_ylabel('y')
fig.colorbar(im,ax=axes[1][1])

im = axes[1][2].imshow(exy_temp,extent=pextent,cmap='viridis',interpolation='nearest')
axes[1][2].set_title('$\dot{\epsilon}_{xy}$',fontsize=10,y=1.01)
axes[1][2].set_xlim(0,Lx)
axes[1][2].set_ylim(0,Ly)
axes[1][2].set_xlabel('x')
axes[1][2].set_ylabel('y')
fig.colorbar(im,ax=axes[1][2])

im = axes[2][0].imshow(error_u_temp,extent=uextent,cmap='Spectral_r',interpolation='nearest')
axes[2][0].set_title('$v_x-t^{th}_x$',fontsize=10,y=1.01)
axes[2][0].set_xlabel('x')
axes[2][0].set_ylabel('y')
fig.colorbar(im,ax=axes[2][0])

im = axes[2][1].imshow(error_v_temp,extent=uextent,cmap='Spectral_r',interpolation='nearest')
axes[2][1].set_title('$v_y-t^{th}_y$',fontsize=10,y=1.01)
axes[2][1].set_xlabel('x')
axes[2][1].set_ylabel('y')
fig.colorbar(im,ax=axes[2][1])

im = axes[2][2].imshow(error_p_temp, extent=uextent, cmap='RdGy_r',interpolation='nearest')
axes[2][2].set_title('$p-p^{th}$',fontsize=10,y=1.01)
axes[2][2].set_xlabel('x')
axes[2][2].set_ylabel('y')
fig.colorbar(im,ax=axes[2][2])

plt.subplots_adjust(hspace=0.5)

if visu==1:
   plt.savefig('solution.pdf', bbox_inches='tight')
   plt.show()

print("-----------------------------")
```

F fieldstone_indentor.py

```
import numpy as np
import math as math
import sys as sys
import scipy
import scipy.sparse as sps
from scipy.sparse.linalg.dsolve import linsolve
import time as time
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm
import tkinter

#------------------------------------------------------------------------------

def viscosity(exx,eyy,exy):
    e2=np.sqrt(0.5*(exx*exx+eyy*eyy)+exy*exy)
    e2=max(1e-8,e2)
    sigmay=1.
    val=sigmay/2./e2
    val=min(1.e3,val)
    val=max(1.e-3,val)
    #val=1.
    return val

#------------------------------------------------------------------------------

print("-----------------------------")
print("---------- fieldstone ----------")
print("-----------------------------")

# declare variables
print("variable declaration")

m=4     # number of nodes making up an element
ndof=2  # number of degrees of freedom per node

Lx=1.  # horizontal extent of the domain
Ly=0.5  # vertical extent of the domain

assert (Lx>0.), "Lx should be positive"
assert (Ly>0.), "Ly should be positive"

# allowing for argument parsing through command line
if int(len(sys.argv) == 4):
   nelx = int(sys.argv[1])
   nely = int(sys.argv[2])
   visu = int(sys.argv[3])
else:
   nelx = 64
   nely = 32
   visu = 1

assert (nelx>0.), "nnx should be positive"
assert (nely>0.), "nny should be positive"

nnx=nelx+1  # number of elements, x direction
nny=nely+1  # number of elements, y direction

nnp=nnx*nny  # number of nodes

nel=nelx*nely  # number of elements, total

penalty=1.e7  # penalty coefficient value

Nfem=nnp*ndof  # Total number of degrees of freedom

eps=1.e-10

sqrt3=np.sqrt(3.)

width=0.123456789
niter=3

gx=0.
gy=0.
```

```
density=1.

# declare arrays
print("declaring arrays")

#######################################################################
# grid point setup
#######################################################################

print("grid point setup")

x = np.empty(nnp, dtype=np.float64)  # x coordinates
y = np.empty(nnp, dtype=np.float64)  # y coordinates

counter = 0
for j in range(0, nny):
    for i in range(0, nnx):
        x[counter]=i*Lx/float(nelx)
        y[counter]=j*Ly/float(nely)
        counter+=1

#######################################################################
# connectivity
#######################################################################

print("connectivity")

icon =np.zeros((m, nel),dtype=np.int16)

counter = 0
for j in range(0, nely):
    for i in range(0, nelx):
        icon[0, counter] = i + j * (nelx + 1)
        icon[1, counter] = i + 1 + j * (nelx + 1)
        icon[2, counter] = i + 1 + (j + 1) * (nelx + 1)
        icon[3, counter] = i + (j + 1) * (nelx + 1)
        counter += 1

# for iel in range (0,nel):
#     print ("iel=",iel)
#     print ("node 1",icon[0][iel],"at pos.",x[icon[0][iel]], y[icon[0][iel]])
#     print ("node 2",icon[1][iel],"at pos.",x[icon[1][iel]], y[icon[1][iel]])
#     print ("node 3",icon[2][iel],"at pos.",x[icon[2][iel]], y[icon[2][iel]])
#     print ("node 4",icon[3][iel],"at pos.",x[icon[3][iel]], y[icon[3][iel]])

#######################################################################
# define boundary conditions
#######################################################################

print("defining boundary conditions")

bc_fix = np.zeros(Nfem, dtype=np.bool)  # boundary condition, yes/no
bc_val = np.zeros(Nfem, dtype=np.float64)  # boundary condition, value

for i in range(0, nnp):
    if x[i]<eps:
       bc_fix[i*ndof]   = True ; bc_val[i*ndof]   = 0.
    if x[i]>(Lx-eps):
       bc_fix[i*ndof]   = True ; bc_val[i*ndof]   = 0.
    if y[i]<eps:
       bc_fix[i*ndof+1] = True ; bc_val[i*ndof+1] = 0.
    if y[i]>(Ly-eps) and abs(x[i]-Lx/2.)<width:
       bc_fix[i*ndof]   = True ; bc_val[i*ndof]   = 0.
       bc_fix[i*ndof+1] = True ; bc_val[i*ndof+1] = -1.

#######################################################################

N     = np.zeros(m,dtype=np.float64)            # shape functions
dNdx  = np.zeros(m,dtype=np.float64)            # shape functions derivatives
dNdy  = np.zeros(m,dtype=np.float64)            # shape functions derivatives
dNdr  = np.zeros(m,dtype=np.float64)            # shape functions derivatives
dNds  = np.zeros(m,dtype=np.float64)            # shape functions derivatives
u     = np.zeros(nnp,dtype=np.float64)          # x-component velocity
v     = np.zeros(nnp,dtype=np.float64)          # y-component velocity
k_mat = np.array([[1,1,0],[1,1,0],[0,0,0]],dtype=np.float64)
c_mat = np.array([[2,0,0],[0,2,0],[0,0,1]],dtype=np.float64)
```

```
Res    = np.zeros(Nfem,dtype=np.float64)          # non-linear residual
sol    = np.zeros(Nfem,dtype=np.float64)          # solution vector

#------------------------------------------------------------------------------
# non-linear iterations
#------------------------------------------------------------------------------

for iter in range(0,niter):

    print("--------------------------------")
    print("iter=", iter)
    print("--------------------------------")

    #################################################################
    # build FE matrix
    #################################################################

    print("building FE matrix")

    a_mat = np.zeros((Nfem,Nfem),dtype=np.float64)  # matrix of Ax=b
    b_mat = np.zeros((3,ndof*m),dtype=np.float64)   # gradient matrix B
    rhs   = np.zeros(Nfem,dtype=np.float64)         # right hand side of Ax=b

    for iel in range(0, nel):

        # set 2 arrays to 0 every loop
        b_el = np.zeros(m * ndof)
        a_el = np.zeros((m * ndof, m * ndof), dtype=float)

        # integrate viscous term at 4 quadrature points
        for iq in [-1, 1]:
            for jq in [-1, 1]:

                # position & weight of quad. point
                rq=iq/sqrt3
                sq=jq/sqrt3
                wq=1.*1.

                # calculate shape functions
                N[0]=0.25*(1.-rq)*(1.-sq)
                N[1]=0.25*(1.+rq)*(1.-sq)
                N[2]=0.25*(1.+rq)*(1.+sq)
                N[3]=0.25*(1.-rq)*(1.+sq)

                # calculate shape function derivatives
                dNdr[0]=-0.25*(1.-sq)  ; dNds[0]=-0.25*(1.-rq)
                dNdr[1]=+0.25*(1.-sq)  ; dNds[1]=-0.25*(1.+rq)
                dNdr[2]=+0.25*(1.+sq)  ; dNds[2]=+0.25*(1.+rq)
                dNdr[3]=-0.25*(1.+sq)  ; dNds[3]=+0.25*(1.-rq)

                # calculate jacobian matrix
                jcb = np.zeros((2, 2),dtype=float)
                for k in range(0,m):
                    jcb[0, 0] += dNdr[k]*x[icon[k,iel]]
                    jcb[0, 1] += dNdr[k]*y[icon[k,iel]]
                    jcb[1, 0] += dNds[k]*x[icon[k,iel]]
                    jcb[1, 1] += dNds[k]*y[icon[k,iel]]

                # calculate the determinant of the jacobian
                jcob = np.linalg.det(jcb)

                # calculate inverse of the jacobian matrix
                jcbi = np.linalg.inv(jcb)

                # compute dNdx & dNdy
                xq=0.0
                yq=0.0
                exxq=0.0
                eyyq=0.0
                exyq=0.0
                for k in range(0, m):
                    xq+=N[k]*x[icon[k,iel]]
                    yq+=N[k]*y[icon[k,iel]]
                    dNdx[k]=jcbi[0,0]*dNdr[k]+jcbi[0,1]*dNds[k]
                    dNdy[k]=jcbi[1,0]*dNdr[k]+jcbi[1,1]*dNds[k]
                    exxq+=dNdx[k]*u[icon[k,iel]]
                    eyyq+=dNdy[k]*v[icon[k,iel]]
                    exyq+=0.5*dNdy[k]*u[icon[k,iel]]+ 0.5*dNdx[k]*v[icon[k,iel]]

                # construct 3x8 b_mat matrix
                for i in range(0, m):
                    b_mat[0:3, 2*i:2*i+2] = [[dNdx[i],0.      ],
                                             [0.      ,dNdy[i]],
                                             [dNdy[i],dNdx[i]]]

                # compute elemental a_mat matrix
                a_el += b_mat.T.dot(c_mat.dot(b_mat))*viscosity(exxq,eyyq,exyq)*wq*jcob

                # compute elemental rhs vector
                for i in range(0,m):
                    b_el[2*i  ]+=N[i]*jcob*wq*density*gx
                    b_el[2*i+1]+=N[i]*jcob*wq*density*gy


        # integrate penalty term at 1 point
        rq=0.
        sq=0.
        wq=2.*2.

        N[0]=0.25*(1.-rq)*(1.-sq)
        N[1]=0.25*(1.+rq)*(1.-sq)
        N[2]=0.25*(1.+rq)*(1.+sq)
        N[3]=0.25*(1.-rq)*(1.+sq)

        dNdr[0]=-0.25*(1.-sq) ; dNds[0]=-0.25*(1.-rq)
        dNdr[1]=+0.25*(1.-sq) ; dNds[1]=-0.25*(1.+rq)
        dNdr[2]=+0.25*(1.+sq) ; dNds[2]=+0.25*(1.+rq)
        dNdr[3]=-0.25*(1.+sq) ; dNds[3]=+0.25*(1.-rq)

        # compute the jacobian
        jcb=np.zeros((2,2),dtype=float)
        for k in range(0, m):
            jcb[0,0]+=dNdr[k]*x[icon[k,iel]]
            jcb[0,1]+=dNdr[k]*y[icon[k,iel]]
            jcb[1,0]+=dNds[k]*x[icon[k,iel]]
            jcb[1,1]+=dNds[k]*y[icon[k,iel]]

        # calculate determinant of the jacobian
        jcob = np.linalg.det(jcb)

        # calculate the inverse of the jacobian
        jcbi = np.linalg.inv(jcb)

        # compute dNdx and dNdy
        for k in range(0,m):
            dNdx[k]=jcbi[0,0]*dNdr[k]+jcbi[0,1]*dNds[k]
            dNdy[k]=jcbi[1,0]*dNdr[k]+jcbi[1,1]*dNds[k]

        # compute gradient matrix
        for i in range(0,m):
            b_mat[0:3,2*i:2*i+2]=[[dNdx[i],0.      ],
                                  [0.      ,dNdy[i]],
                                  [dNdy[i],dNdx[i]]]

        # compute elemental matrix
        a_el+=b_mat.T.dot(k_mat.dot(b_mat))*penalty*wq*jcob

        # assemble matrix a_mat and right hand side rhs
        for k1 in range(0,m):
            for i1 in range(0,ndof):
                ikk=ndof*k1          +i1
                m1 =ndof*icon[k1,iel]+i1
                for k2 in range(0,m):
                    for i2 in range(0,ndof):
                        jkk=ndof*k2          +i2
                        m2 =ndof*icon[k2,iel]+i2
                        a_mat[m1,m2]+=a_el[ikk,jkk]
                rhs[m1]+=b_el[ikk]


    #################################################################
    # impose boundary conditions
```

```
    ################################################################

    print("imposing boundary conditions")

    for i in range(0, Nfem):
        if bc_fix[i]:
           a_matref=a_mat[i,i]
           for j in range(0,Nfem):
               rhs[j]-=a_mat[i,j]*bc_val[i]
               a_mat[i,j]=0.
               a_mat[j,i]=0.
               a_mat[i,i]=a_matref
           rhs[i]=a_matref*bc_val[i]
    ################################################################
    # compute non-linear residual
    ################################################################

    Res=a_mat.dot(sol)-rhs

    if iter==0:
       Res0=np.max(abs(Res))

    print("Nonlinear residual (inf. norm) %.7e" % (np.max(abs(Res))/Res0))

    ################################################################
    # solve system
    ################################################################

    start = time.time()
    sol = sps.linalg.spsolve(sps.csr_matrix(a_mat),rhs)
    print("solve time: %.3f s" % (time.time() - start))

    ####################################################################
    # put solution into separate x,y velocity arrays
    ####################################################################

    u,v=np.reshape(sol,(nnp,2)).T

    print("u (m,M) %.4f %.4f " %(np.min(u),np.max(u)))
    print("v (m,M) %.4f %.4f " %(np.min(v),np.max(v)))


#------------------------------------------------------------------------------
# end of non-linear iterations
#------------------------------------------------------------------------------

################################################################
# retrieve pressure and elemental strain rate components
################################################################

xc=np.zeros(nel,dtype=np.float64)
yc=np.zeros(nel,dtype=np.float64)
p=np.zeros(nel,dtype=np.float64)
exx=np.zeros(nel,dtype=np.float64)
eyy=np.zeros(nel,dtype=np.float64)
exy=np.zeros(nel,dtype=np.float64)
eta=np.zeros(nel,dtype=np.float64)
e=np.zeros(nel,dtype=np.float64)

for iel in range(0,nel):

    rq = 0.0
    sq = 0.0
    wq = 2.0 * 2.0

    N[0]=0.25*(1.-rq)*(1.-sq)
    N[1]=0.25*(1.+rq)*(1.-sq)
    N[2]=0.25*(1.+rq)*(1.+sq)
    N[3]=0.25*(1.-rq)*(1.+sq)

    dNdr[0]=-0.25*(1.-sq) ; dNds[0]=-0.25*(1.-rq)
    dNdr[1]=+0.25*(1.-sq) ; dNds[1]=-0.25*(1.+rq)
    dNdr[2]=+0.25*(1.+sq) ; dNds[2]=+0.25*(1.+rq)
    dNdr[3]=-0.25*(1.+sq) ; dNds[3]=+0.25*(1.-rq)

    jcb=np.zeros((2,2),dtype=float)
    for k in range(0, m):
        jcb[0,0]+=dNdr[k]*x[icon[k,iel]]
        jcb[0,1]+=dNdr[k]*y[icon[k,iel]]
        jcb[1,0]+=dNds[k]*x[icon[k,iel]]
        jcb[1,1]+=dNds[k]*y[icon[k,iel]]

    # calculate determinant of the jacobian
    jcob=np.linalg.det(jcb)

    # calculate the inverse of the jacobian
    jcbi=np.linalg.inv(jcb)

    for k in range(0,m):
        dNdx[k]=jcbi[0,0]*dNdr[k]+jcbi[0,1]*dNds[k]
        dNdy[k]=jcbi[1,0]*dNdr[k]+jcbi[1,1]*dNds[k]

    for k in range(0,m):
        xc[iel]+=N[k]*x[icon[k,iel]]
        yc[iel]+=N[k]*y[icon[k,iel]]
        exx[iel]+=dNdx[k]*u[icon[k,iel]]
        eyy[iel]+=dNdy[k]*v[icon[k,iel]]
        exy[iel]+=0.5*dNdy[k]*u[icon[k,iel]]+ 0.5*dNdx[k]*v[icon[k,iel]]

    e[iel]=np.sqrt(0.5*(exx[iel]*exx[iel]+eyy[iel]*eyy[iel])+exy[iel]*exy[iel])
    eta[iel]=viscosity(exx[iel],eyy[iel],exy[iel])
    p[iel]=-penalty*(exx[iel]+eyy[iel])

print("p (m,M) %.4f %.4f " %(np.min(p),np.max(p)))
print("exx (m,M) %.4f %.4f " %(np.min(exx),np.max(exx)))
print("eyy (m,M) %.4f %.4f " %(np.min(eyy),np.max(eyy)))
print("exy (m,M) %.4f %.4f " %(np.min(exy),np.max(exy)))
print("eta (m,M) %.4f %.4f " %(np.min(eta),np.max(eta)))

np.savetxt('velocity.ascii',np.array([x,y,u,v]).T,header='# x,y,u,v')
np.savetxt('pressure.ascii',np.array([xc,yc,p]).T,header='# xc,yc,p')
np.savetxt('strainrate.ascii',np.array([xc,yc,exx,eyy,exy]).T,header='# xc,yc,exx,eyy,exy')

#################################################################
# smoothing pressure
#################################################################

q=np.zeros(nnp,dtype=np.float64)
count=np.zeros(nnp,dtype=np.float64)

for iel in range(0,nel):
    q[icon[0,iel]]+=p[iel]
    q[icon[1,iel]]+=p[iel]
    q[icon[2,iel]]+=p[iel]
    q[icon[3,iel]]+=p[iel]
    count[icon[0,iel]]+=1
    count[icon[1,iel]]+=1
    count[icon[2,iel]]+=1
    count[icon[3,iel]]+=1

q=q/count

#################################################################
# extract velocity field at domain top
#################################################################

xtop=np.zeros(nnx,dtype=np.float64)
utop=np.zeros(nnx,dtype=np.float64)
vtop=np.zeros(nnx,dtype=np.float64)
ptop=np.zeros(nelx,dtype=np.float64)
xctop=np.zeros(nelx,dtype=np.float64)

counter=0
for i in range(0,nnp):
    if y[i]>Ly-eps:
       xtop[counter]=x[i]
       utop[counter]=u[i]
       vtop[counter]=v[i]
       counter+=1

counter=0
for iel in range(0,nel):
    if y[icon[3,iel]]>Ly-eps:
       ptop[counter]=p[iel]
       xctop[counter]=xc[iel]
       counter+=1

#####################################################################
# plot of solution
#####################################################################

u_temp=np.reshape(u,(nny,nnx))
np.flipud(u_temp)

v_temp=np.reshape(v,(nny,nnx))
p_temp=np.reshape(p,(nely,nelx))
q_temp=np.reshape(q,(nny,nnx))
exx_temp=np.reshape(exx,(nelx,nely))
eyy_temp=np.reshape(eyy,(nelx,nely))
exy_temp=np.reshape(exy,(nelx,nely))
eta_temp=np.reshape(eta,(nelx,nely))
e_temp=np.reshape(e,(nelx,nely))

fig,axes = plt.subplots(nrows=4,ncols=3,figsize=(18,18))

uextent=(np.amin(x),np.amax(x),np.amin(y),np.amax(y))
pextent=(np.amin(xc),np.amax(xc),np.amin(yc),np.amax(yc))

im = axes[0][0].imshow(u_temp,extent=uextent,cmap='Spectral',interpolation='nearest')
axes[0][0].set_title('$v_x$', fontsize=10, y=1.01)
axes[0][0].set_xlabel('x')
axes[0][0].set_ylabel('y')
fig.colorbar(im,ax=axes[0][0])

im = axes[0][1].imshow(v_temp,extent=uextent,cmap='Spectral',interpolation='nearest')
axes[0][1].set_title('$v_y$', fontsize=10, y=1.01)
axes[0][1].set_xlabel('x')
axes[0][1].set_ylabel('y')
fig.colorbar(im,ax=axes[0][1])

im = axes[0][2].imshow(p_temp,extent=pextent,cmap='RdGy',interpolation='nearest')
axes[0][2].set_title('$p$', fontsize=10, y=1.01)
axes[0][2].set_xlim(0,Lx)
axes[0][2].set_ylim(0,Ly)
axes[0][2].set_xlabel('x')
axes[0][2].set_ylabel('y')
fig.colorbar(im,ax=axes[0][2])

im = axes[1][0].imshow(exx_temp,extent=pextent,cmap='viridis',interpolation='nearest')
axes[1][0].set_title('$\dot{\epsilon}_{xx}$',fontsize=10, y=1.01)
axes[1][0].set_xlim(0,Lx)
axes[1][0].set_ylim(0,Ly)
axes[1][0].set_xlabel('x')
axes[1][0].set_ylabel('y')
fig.colorbar(im,ax=axes[1][0])

im = axes[1][1].imshow(eyy_temp,extent=pextent,cmap='viridis',interpolation='nearest')
axes[1][1].set_title('$\dot{\epsilon}_{yy}$',fontsize=10,y=1.01)
axes[1][1].set_xlim(0,Lx)
axes[1][1].set_ylim(0,Ly)
axes[1][1].set_xlabel('x')
axes[1][1].set_ylabel('y')
fig.colorbar(im,ax=axes[1][1])

im = axes[1][2].imshow(exy_temp,extent=pextent,cmap='viridis',interpolation='nearest')
axes[1][2].set_title('$\dot{\epsilon}_{xy}$',fontsize=10,y=1.01)
axes[1][2].set_xlim(0,Lx)
axes[1][2].set_ylim(0,Ly)
axes[1][2].set_xlabel('x')
axes[1][2].set_ylabel('y')
fig.colorbar(im,ax=axes[1][2])

im = axes[2][0].imshow(e_temp,extent=pextent,cmap='viridis',interpolation='nearest')
axes[2][0].set_title('$\dot{\epsilon}$',fontsize=10,y=1.01)
axes[2][0].set_xlim(0,Lx)
axes[2][0].set_ylim(0,Ly)
axes[2][0].set_xlabel('x')
axes[2][0].set_ylabel('y')
```

```
fig.colorbar(im,ax=axes[2][0])

im = axes[2][1].imshow(eta_temp,extent=pextent,cmap='jet',interpolation='nearest',norm=LogNorm(vmin
    =1e-3,vmax=1e3))
axes[2][1].set_title('$\eta$',fontsize=10, y=1.01)
axes[2][1].set_xlim(0,Lx)
axes[2][1].set_ylim(0,Ly)
axes[2][1].set_xlabel('x')
axes[2][1].set_ylabel('y')
fig.colorbar(im,ax=axes[2][1])

im = axes[2][2].imshow(q_temp,extent=pextent,cmap='RdGy',interpolation='nearest')
axes[2][2].set_title('$p$ (nodal)',fontsize=10, y=1.01)
axes[2][2].set_xlim(0,Lx)
axes[2][2].set_ylim(0,Ly)
axes[2][2].set_xlabel('x')
axes[2][2].set_ylabel('y')
fig.colorbar(im,ax=axes[2][2])

im = axes[3][0].plot(xtop,utop)
axes[3][0].set_xlabel('$x$')
axes[3][0].set_ylabel('$u$')

im = axes[3][1].plot(xtop,vtop)
axes[3][1].set_xlabel('$x$')
axes[3][1].set_ylabel('$v$')

im = axes[3][2].plot(xctop,ptop)
axes[3][2].set_xlabel('$x$')
axes[3][2].set_ylabel('$p$')



plt.subplots_adjust(hspace=0.5)

if visu==1:
   plt.savefig('solution.pdf', bbox_inches='tight')
   plt.show()

print("-----------------------------")
print("---------the end-----------")
print("-----------------------------")
```

References

[1] K.-J. Bathe. *Finite Element Procedures in Engineering Analysis.* Prentice-Hall, 1982.

[2] M. Benzi, G.H. Golub, and J. Liesen. Numerical solution of saddle point problems. *Acta Numerica*, 14:1–137, 2005.

[3] B. Blankenbach, F. Busse, U. Christensen, L. Cserepes, D. Gunkel, U. Hansen, H. Harder, G. Jarvis, M. Koch, G. Marquart, D. Moore, P. Olson, H. Schmeling, and T. Schnaubelt. A benchmark comparison for mantle convection codes. *Geophys. J. Int.*, 98:23–38, 1989.

[4] H.H. Bui, R. Fukugawa, K. Sako, and S. Ohno. Lagrangian meshfree particles method (SPH) for large deformation and failure flows of geomaterial using elasticplastic soil constitutive model. *Int. J. Numer. Anal. Geomech.*, 32(12):1537–1570, 2008.

[5] Edmund Christiansen and Knud D. Andersen. Computation of collapse states with von mises type yield condition. *International Journal for Numerical Methods in Engineering*, 46:1185–1202, 1999.

[6] Edmund Christiansen and Ole S. Pedersen. Automatic mesh refinement in limit analysis. *International Journal for Numerical Methods in Engineering*, 50:1331–1346, 2001.

[7] C. Cuvelier, A. Segal, and A.A. van Steenhoven. *Finite Element Methods and Navier-Stokes Equations.* D. Reidel Publishing Company, 1986.

[8] P. Davy and P. Cobbold. Indentation tectonics in nature and experiment. 1. experiments scaled for gravity. *Bulletin of the Geological Institutions of Uppsala*, 14:129–141, 1988.

[9] Y. Deubelbeiss and B.J.P. Kaus. Comparison of Eulerian and Lagrangian numerical techniques for the Stokes equations in the presence of strongly varying viscosity. *Phys. Earth Planet. Interiors*, 171:92–111, 2008.

[10] J. Donea and A. Huerta. *Finite Element Methods for Flow Problems.* 2003.

[11] Jean Donea and Antonio Huerta. *Finite Element Methods for Flow Problems.* John Wiley & Sons, 2003.

[12] T. Duretz, D.A. May, T.V. Gerya, and P.J. Tackley. Discretization errors and free surface stabilisation in the finite difference and marker-in-cell method for applied geodynamics: A numerical study. *Geochem. Geophys. Geosyst.*, 12(Q07004), 2011.

[13] M. Gerbault, A.N.B. Poliakov, and M. Daignieres. Prediction of faulting from the theories of elasticity and plasticity: what are the limits? *Journal of Structural Geology*, 20:301–320, 1998.

[14] T.V. Gerya, D.A. May, and T. Duretz. An adaptive staggered grid finite difference method for modeling geodynamic Stokes flows with strongly variable viscosity. *Geochem. Geophys. Geosyst.*, 14(4), 2013.

[15] G.H. Golub and C.F. van Loan. *Matrix Computations, 4th edition.* John Hopkins University Press, 2013.

[16] M. Gunzburger. *Finite Element Methods for Viscous Incompressible Flows: A Guide to Theory, Practice and Algorithms.* Academic, Boston, 1989.

[17] T.J.R. Hughes. *The Finite Element Method. Linear Static and Dynamic Finite Element Analysis.* Dover Publications, Inc., 2000.

[18] T.J.R. Hughes, W.K. Liu, and A. Brooks. Finite element analysis of Incompressible viscous flows by the penalty function formulation. *J. Comp. Phys.*, 30:1–60, 1979.

[19] Hoon Huh, Choong Ho Lee, and Wei H. Yang. A general algorithm for plastic flow simulation by finite element limit analysis. *International Journal of Solids and Structures*, 36:1193–1207, 1999.

[20] L. Jolivet, P. Davy, and P. Cobbold. Right-lateral shear along the Northwest Pacific margin and the India-Eurasia collision. *Tectonics*, 9(6):1409–1419, 1990.

[21] L.M. Kachanov. *Fundamentals of the Theory of Plasticity.* Dover Publications, Inc., 2004.

[22] M. Kronbichler, T. Heister, and W. Bangerth. High accuracy mantle convection simulation through modern numerical methods . *Geophy. J. Int.*, 191:12–29, 2012.

[23] D.S. Malkus and T.J.R. Hughes. Mixed finite element methods - reduced and selective integration techniques: a unification of concepts. *Comput. Meth. Appl. Mech. Eng.*, 15:63–81, 1978.

[24] P. Molnar and P. Tapponnier. Relation of the tectonics of eastern China to the India-Eurasia collision: Application of the slip-line field theory to large-scale continental tectonics. *Geology*, 5:212–216, 1977.

[25] L. Moresi, S. Quenette, V. Lemiale, C. Mériaux, B. Appelbe, and H.-B. Mühlhaus. Computational approaches to studying non-linear dynamics of the crust and mantle. *Phys. Earth. Planet. Inter.*, 163:69–82, 2007.

[26] G. Peltzer and P. Tapponnier. Formation and evolution of strike-slip faults, rifts, and basins during the india-asia collision: an experimental approach. *J. Geophys. Res.*, 93(B12):15085–15177, 1988.

[27] T. Rabczuk, P.M.A. Areias, and T. Belytschko. A simplified mesh-free method for shear bands with cohesive surfaces . *Int. J. Num. Meth. Eng.*, 69:993–1021, 2007.

[28] J.N. Reddy. On penalty function methods in the finite element analysis of flow problems. *Int. J. Num. Meth. Fluids*, 2:151–171, 1982.

[29] J. Revenaugh and B. Parsons. Dynamic topography and gravity anomalies for fluid layers whose viscosity varies exponentially with depth. *Geophysical Journal of the Royal Astronomical Society*, 90(2):349–368, 1987.

[30] D.W. Schmid and Y.Y. Podlachikov. Analytical solutions for deformable elliptical inclusions in general shear. *Geophy. J. Int.*, 155:269–288, 2003.

[31] G. Schubert, D.L. Turcotte, and P. Olson. *Mantle Convection in the Earth and Planets.* Cambridge University Press, 2001.

[32] J. Suckale, J.-C. Nave, and B.H. Hager. It takes three to tango: 1. Simulating buoyancy-driven flow in the presence of large viscosity contrasts. *J. Geophys. Res.*, 115(B07409), 2010.

[33] Paul Tapponnier and Peter Molnar. Slip-line field theory and large-scale continental tectonics. *Nature*, 264:319–324, November 1976.

[34] C. Thieulot, P. Fullsack, and J. Braun. Adaptive octree-based finite element analysis of two- and three-dimensional indentation problems. *J. Geophys. Res.*, 113:B12207, 2008.

[35] X. Yu and F. Tin-Loi. A simple mixed finite element for static limit analysis. *Computers and Structures*, 84:1906–1917, 2006.

[36] S. Zhong. Analytic solutions for Stokes flow with lateral variations in viscosity. *Geophys. J. Int.*, 124:18–28, 1996.

[37] O. Zienkiewicz and S. Nakazawa. The penalty function method and its application to the numerical solution of boundary value problems. *The American Society of Mechanical Engineers*, 51, 1982.

[38] O.C. Zienkiewicz, M. Huang, and M. Pastor. Localization problems in plasticity using finite elements with adaptive remeshing. *International Journal for Numerical and Analytical Methods in Geomechanics*, 19:127–148, 1995.

[39] O.C. Zienkiewicz, C. Humpheson, and R.W. Lewis. Associated and non-associated visco-plasticity and plasticity in soil mechanics . *Géotechnique*, 25(4):671–689, 1975.