Using Python for electronic structure calculations, nonlinear solvers, FEM and symbolic manipulation

Ondřej Čertík

Institute of Physics, Academy of Sciences of the Czech Republic

August 5, 2007

Introduction

I'll talk about

- Density Functional Theory calculations
- SciPy nonlinear solvers
- Finite Element Method using python-petsc and libmesh
- SymPy – the symbolic manipulation package in Python

$$\hat{H}\,|\Psi\rangle = (\hat{T} + \hat{U} + \hat{V})\,|\Psi\rangle = E\,|\Psi\rangle$$

where

$$\hat{T} = \sum_{i}^{N} -\frac{1}{2}\nabla_i^2$$

$$\hat{U} = \sum_{i<j} U(\mathbf{r_i}, \mathbf{r_j}) = \frac{1}{2}\sum_{i,j} U(\mathbf{r_i}, \mathbf{r_j})$$

$$U(\mathbf{r_i}, \mathbf{r_j}) = U(\mathbf{r_j}, \mathbf{r_i}) = \frac{1}{|\mathbf{r_i} - \mathbf{r_j}|}$$

$$\hat{V} = \sum_{i}^{N} v(\mathbf{r_i})$$

$$v(\mathbf{r_i}) = \sum_{k} -\frac{Z_k}{|\mathbf{r_i} - \mathbf{R_k}|}$$

We solve the Kohn-Sham equations:

$$(-\tfrac{1}{2}\nabla^2 + V_H(\mathbf{r}) + V_{xc}(\mathbf{r}) + v(\mathbf{r}))\psi_i(\mathbf{r}) = \epsilon_i\psi(\mathbf{r})$$

that yield the orbitals ψ_i that reproduce the density $n(\mathbf{r})$ of the original interacting system

$$n(\mathbf{r}) = \sum_i^N |\psi_i(\mathbf{r})|^2$$

$$V_H(\mathbf{r}) = \frac{\delta E_H}{\delta n(\mathbf{r})} = \tfrac{1}{2}\int \frac{n(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|}\mathrm{d}^3 r'$$

$$E_{xc}[n] = (T+U)[n] - E_H[n] - T_S[n]$$

$$V_{xc}(\mathbf{r}) = \frac{\delta E_{xc}[n]}{\delta n(\mathbf{r})}$$

$$v(\mathbf{r}) = \sum_k -\frac{Z_k}{|\mathbf{r}-\mathbf{R_k}|}$$

Atomic wavefunctions

Spherically symmetric potential:

$$V(\mathbf{x}) = V(r)$$

$$\psi_{nlm}(\mathbf{x}) = R_{nl}(r)\, Y_{lm}\left(\frac{\mathbf{x}}{r}\right)$$

Radial Schrödinger equation:

$$R''_{nl} + \frac{2}{r} R'_{nl} + \frac{2M}{\hbar^2}(E - V)R_{nl} - \frac{l(l+1)}{r^2} R_{nl} = 0$$

Relativistic atomic wavefunctions

Dirac equation:

$$(ic\gamma^\mu D_\mu - mc^2)\psi = 0$$

$$D_\mu = \partial_\mu + ieA_\mu$$

Radial Dirac equation:

$$g_\kappa'' + \left(\frac{2}{r} + \frac{V'}{2Mc^2}\right) g_\kappa' + \left[(E - V) - \frac{\kappa(\kappa+1)}{2Mr^2} + \frac{\kappa+1}{4M^2c^2r}V'\right] 2Mg_\kappa = 0$$

$$f_\kappa = \frac{g_\kappa'}{2Mc} + \frac{\kappa+1}{r}\frac{g_\kappa}{2Mc}$$

$$R^2 = f^2 + g^2$$

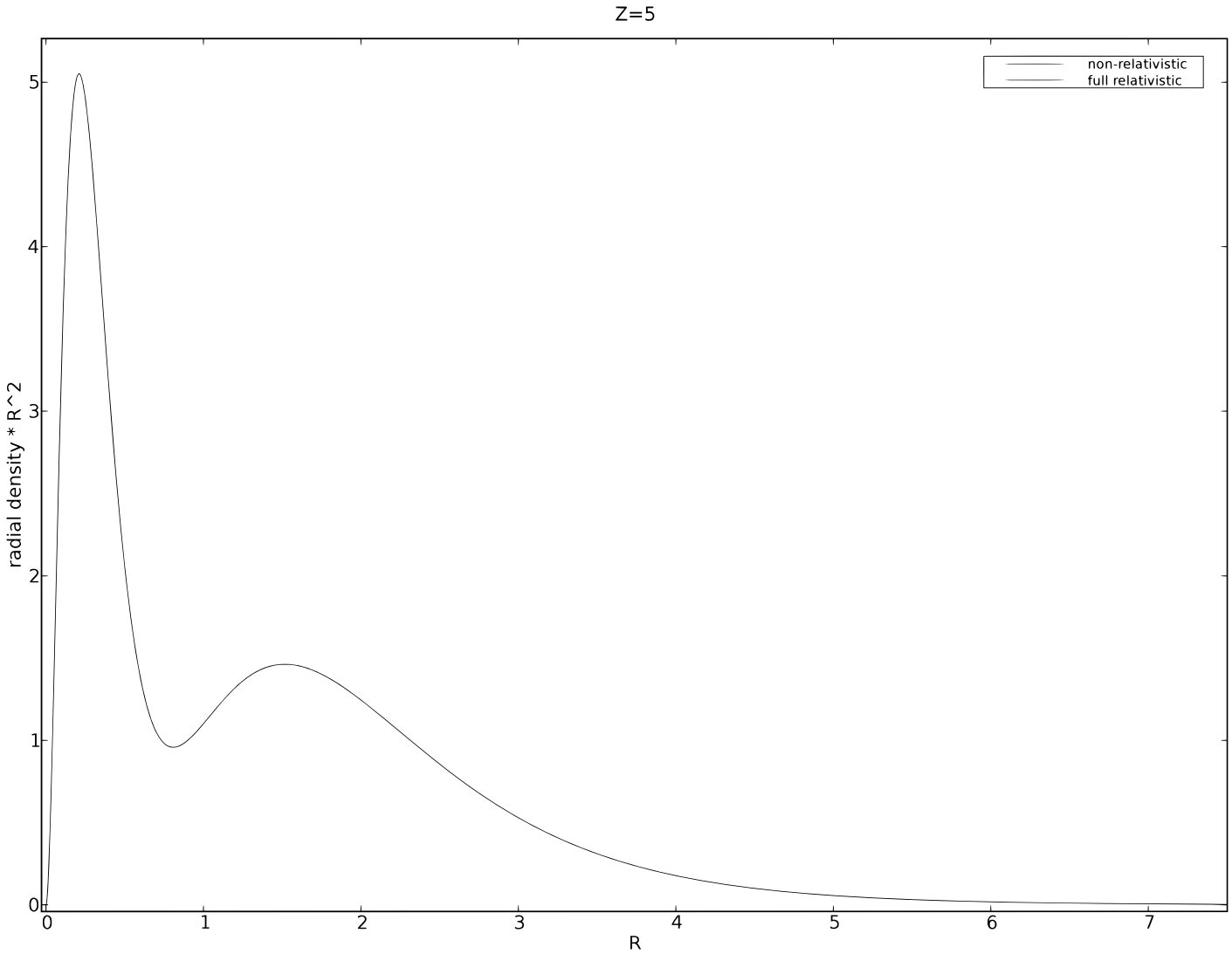
Radial Schrödinger equation:

$$R'' + \frac{2}{r}R' + \left[(E - V) - \frac{l(l+1)}{2Mr^2}\right] 2MR = 0$$

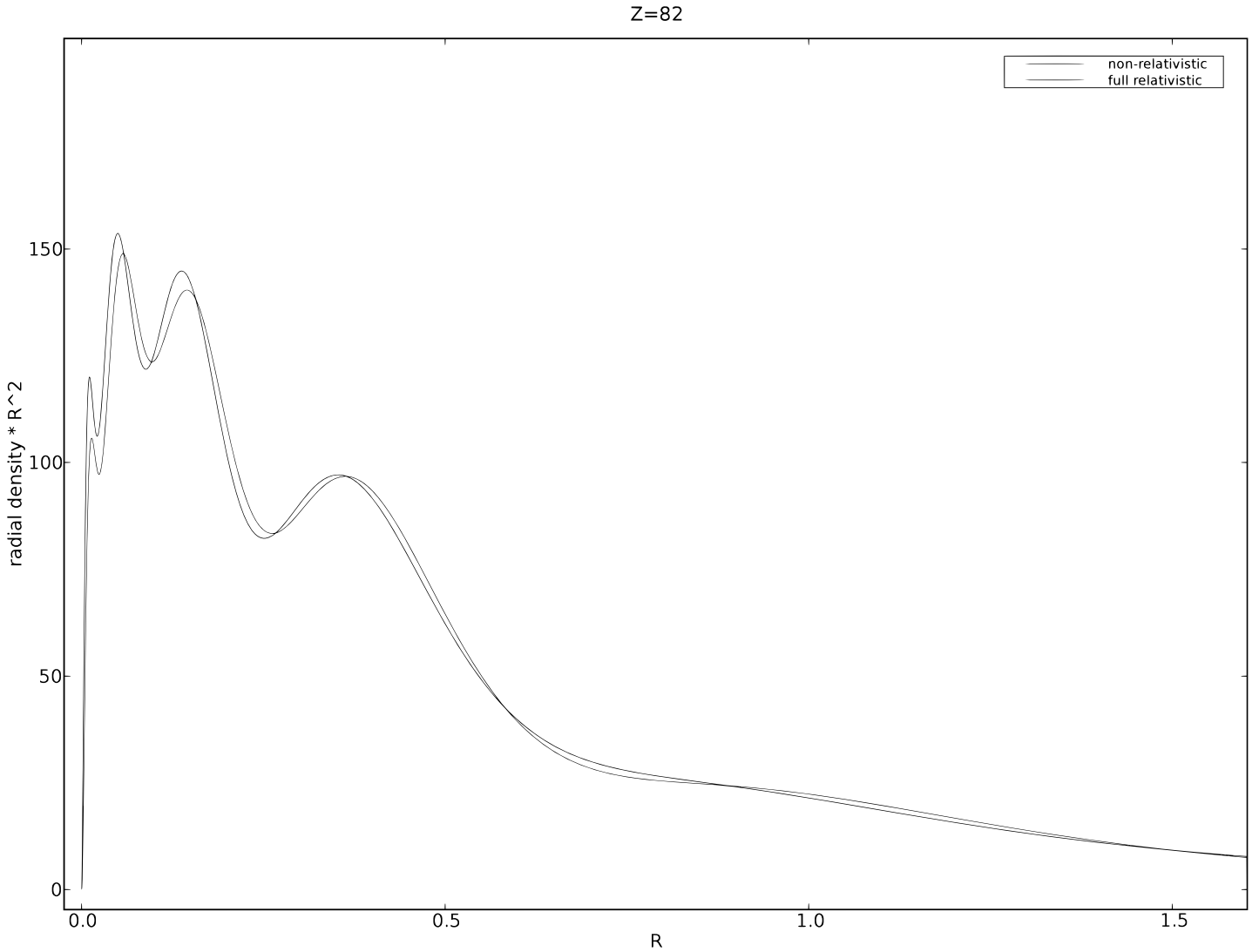
Code (300 lines in Python, 800 in Fortran)

```
from atom import atom, show
import radial
import utils
def do(Z):
    R = radial.create_log_grid(Z)
    s = atom(Z,alpha=0.3,iter=20,relat=0,grid=R)
    n5_lda = radial.KS_construct_density(s,R,Z)*R*R
    s = atom(Z,alpha=0.3,iter=20,relat=2,grid=R)
    n5_rlda = radial.KS_construct_density(s,R,Z)*R*R
    utils.makeplot(R,[
        (n5_lda,"b-","non-relativistic"),
        (n5_rlda,"g-","full relativistic"),
        ],title="Z=%d"%(Z),xleg="R",
        yleg="radial density * R^2")
do(5)
#do(82)
```

Boron


Z=5
non-relativistic
full relativistic
radial density * R^2
R
0
1
2
3
4
5
6
7

Lead


Z=82
non-relativistic
full relativistic
radial density * R^2
R
0
50
100
150
0.0
0.5
1.0
1.5

Lead - nonrelativistic calculation

Iterations: 20
$|F(x)| = 0.00003516$
Agrees with NIST:
http://physics.nist.gov/

```
1s( 2): -2901.078061
2s( 2): -488.8433352
2p( 6): -470.8777849
3s( 2): -116.526852
3p( 6): -107.950391
3d(10): -91.88992429
4s( 2): -25.75333021
4p( 6): -21.99056413
4d(10): -15.03002657
4f(14): -5.592531664
5s( 2): -4.206797624
5p( 6): -2.941656967
5d(10): -0.9023926829
6s( 2): -0.3571868295
6p( 2): -0.1418313263
```

Lead - relativistic calculation

Iterations: 20
$|F(x)| = 0.00000584$

```
1s( 2) j=l+1/2: -3209.51946
2s( 2) j=l+1/2: -574.1825655
2p( 6) j=l-1/2: -551.7234408
2p( 6) j=l+1/2: -472.3716103
3s( 2) j=l+1/2: -137.8642241
3p( 6) j=l-1/2: -127.6789451
3p( 6) j=l+1/2: -109.9540395
3d(10) j=l-1/2: -93.15817605
3d(10) j=l+1/2: -89.36399096
4s( 2) j=l+1/2: -31.15015728
4p( 6) j=l-1/2: -26.73281564
4p( 6) j=l+1/2: -22.38230707
4d(10) j=l-1/2: -15.1647618
4d(10) j=l+1/2: -14.3484973
5s( 2) j=l+1/2: -5.225938506
4f(14) j=l-1/2: -4.960490099
4f(14) j=l+1/2: -4.775660273
5p( 6) j=l-1/2: -3.710458943
5p( 6) j=l+1/2: -2.889127431
5d(10) j=l-1/2: -0.8020049565
5d(10) j=l+1/2: -0.7070299184
6s( 2) j=l+1/2: -0.4209603386
6p( 2) j=l-1/2: -0.1549640727
```

Iteration to self-consistency

The problem:

$$\mathbf{F}(\mathbf{x}) = \mathbf{x}$$

equivalently

$$\mathbf{R}(\mathbf{x}) = 0$$

for $\mathbf{R}(\mathbf{x}) = \mathbf{F}(\mathbf{x}) - \mathbf{x}$. We approximate

$$\mathbf{R}(\mathbf{x}_{M+1}) - \mathbf{R}(\mathbf{x}_M) \approx \mathbf{J} \cdot (\mathbf{x}_{M+1} - \mathbf{x}_M)$$

with the Jacobian

$$J_{ij} = \frac{\partial R_i}{\partial x_j}$$

We want $\mathbf{R}(\mathbf{x}_{M+1}) = 0$:

$$\mathbf{x}_{M+1} \approx \mathbf{x}_M - \mathbf{J}^{-1} \cdot \mathbf{R}(\mathbf{x}_M)$$

$\mathbf{J}$ is approximated by a sequence of $\mathbf{J}_0$, $\mathbf{J}_1$, $\mathbf{J}_2$, ...

Linear mixing

$$\mathbf{x}_{M+1} \approx \mathbf{x}_M - \mathbf{J}_M^{-1} \cdot \mathbf{R}(\mathbf{x}_M)$$

with

$$\mathbf{J}_M^{-1} = -\alpha \mathbb{1}$$

so

$$\mathbf{x}_{M+1} = \mathbf{x}_M + \alpha \mathbf{R}(\mathbf{x}_M) = \mathbf{x}_M + \alpha(\mathbf{F}(\mathbf{x}_M) - \mathbf{x}_M)$$

SciPy

```
from scipy.optimize.nonlin import linearmixing
```

"exciting" mixing

Used in the FP-LAPW DFT code (http://exciting.sourceforge.net/)

$$\mathbf{x}_{M+1} \approx \mathbf{x}_M - \mathbf{J}_M^{-1} \cdot \mathbf{R}(\mathbf{x}_M)$$

with

$$\mathbf{J}_M^{-1} = -\mathrm{diag}(\beta_1, \beta_2, \beta_3, \dots)$$

start with $\beta_1 = \beta_2 = \beta_3 = \cdots = \alpha$ and at every iteration adjust the parameters β_i according to this very simple algorithm: if $R_i(\mathbf{x}_{M-1})R_i(\mathbf{x}_M) > 0$ then increase β_i by α otherwise set $\beta_i = \alpha$ (if $\beta_i > \alpha_{max}$, set $\beta_i = \alpha_{max}$).

SciPy

```
from scipy.optimize.nonlin import excitingmixing
```

Broyden update

The *first Broyden method*:

$$\mathbf{J}_{M+1} = \mathbf{J}_M - \frac{(\Delta\mathbf{R}(\mathbf{x}_M) + \mathbf{J}_M \cdot \Delta\mathbf{x}_M)\Delta\mathbf{x}_M^T}{|\Delta\mathbf{x}_M|^2}$$

The *second Broyden method*:

$$\mathbf{J}_{M+1}^{-1} = \mathbf{J}_M^{-1} + \frac{(\Delta\mathbf{x}_M - \mathbf{J}_M^{-1} \cdot \Delta\mathbf{R}(\mathbf{x}_M))\Delta\mathbf{R}(\mathbf{x}_M)^T}{|\Delta\mathbf{R}(\mathbf{x}_M)|^2}$$

starting with the linear mixing:

$$\mathbf{J}_0^{-1} = -\alpha\mathbb{1}$$

SciPy

```
from scipy.optimize import broyden1, broyden2
```

low memory second Broyden update

The *second Broyden method*
$(\mathbf{J}_{M+1}^{-1} = \mathbf{J}_M^{-1} + \frac{(\Delta\mathbf{x}_M - \mathbf{J}_M^{-1}\cdot\Delta\mathbf{R}(\mathbf{x}_M))\Delta\mathbf{R}(\mathbf{x}_M)^T}{|\Delta\mathbf{R}(\mathbf{x}_M)|^2})$ can be written as

$$\mathbf{J}_{M+1}^{-1} = \mathbf{J}_M^{-1} + \mathbf{u}\mathbf{v}^T$$

with

$$\mathbf{u} = \Delta\mathbf{x}_M - \mathbf{J}_M^{-1}\cdot\Delta\mathbf{R}(\mathbf{x}_M)$$

$$\mathbf{v} = \frac{\Delta\mathbf{R}(\mathbf{x}_M)}{|\Delta\mathbf{R}(\mathbf{x}_M)|^2}$$

so the whole inverse Jacobian can be written as

$$\mathbf{J}_M^{-1} = -\alpha\mathbb{1} + \mathbf{u}_1\mathbf{v}_1^T + \mathbf{u}_2\mathbf{v}_2^T + \mathbf{u}_3\mathbf{v}_3^T + \cdots$$

$$\mathbf{J}_M^{-1}\cdot\mathbf{y} = -\alpha\mathbf{y} + \mathbf{u}_1(\mathbf{v}_1^T\mathbf{y}) + \mathbf{u}_2(\mathbf{v}_2^T\mathbf{y}) + \mathbf{u}_3(\mathbf{v}_3^T\mathbf{y}) + \cdots$$

SciPy

```
from scipy.optimize import broyden3
```

Other methods

The *generalized Broyden method* (modified Broyden method):

$$\sum_{p=M-k}^{M-1} (1+\omega_0^2\delta_{pn})\Delta\mathbf{R}(\mathbf{x}_n)^T\Delta\mathbf{R}(\mathbf{x}_p)\gamma_p = \Delta\mathbf{R}(\mathbf{x}_n)^T\mathbf{R}(\mathbf{x}_M)$$

$$\mathbf{x}_{M+1} = \mathbf{x}_M + \beta_M\mathbf{R}(\mathbf{x}_M) - \sum_{p=M-k}^{M-1} \gamma_p(\Delta\mathbf{x}_p + \beta_M\Delta\mathbf{R}(\mathbf{x}_p))$$

other methods: Anderson, extended Anderson

SciPy

```
from scipy.optimize import broyden_generalized,
    anderson, anderson2
```

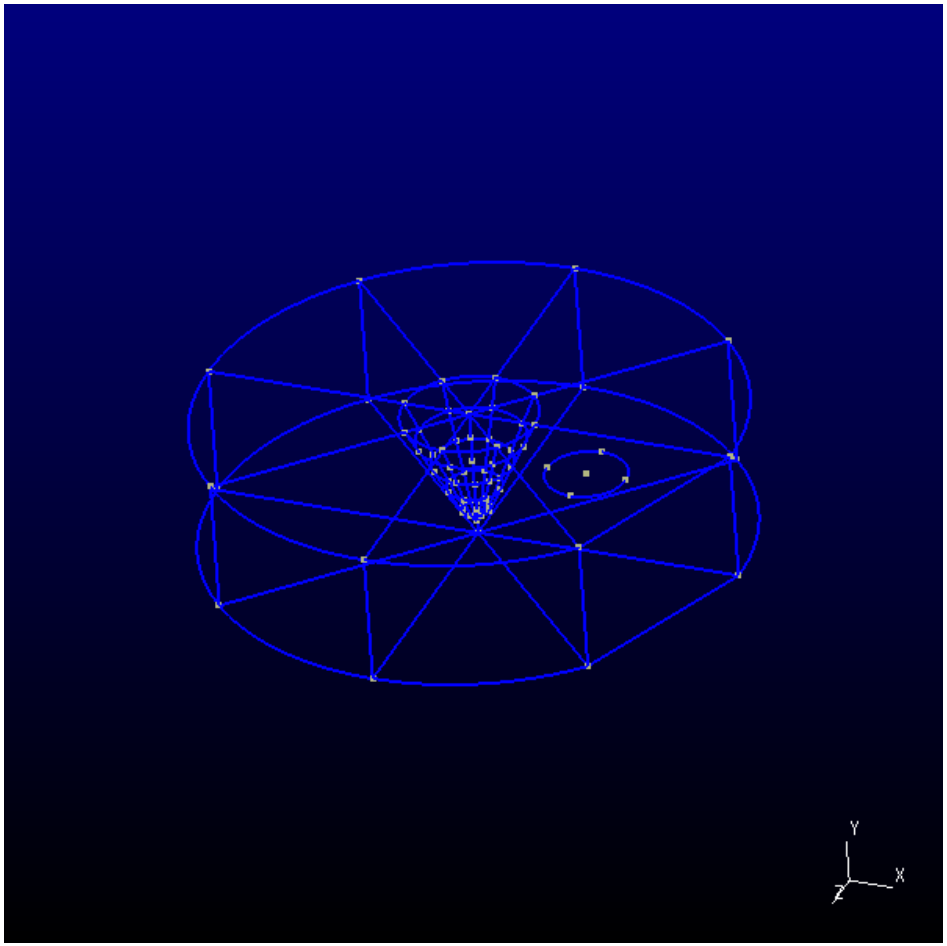
Finite element calculation

Tools I use (Debian package in parentheses):

- gmsh http://www.geuz.org/gmsh/ (gmsh)
- tetgen http://tetgen.berlios.de/ (non-free/tetgen)
- libmesh http://libmesh.sourceforge.net/ (libmesh0.6.0-pure-dev)
- petsc4py http://code.google.com/p/petsc4py/ (python-petsc)

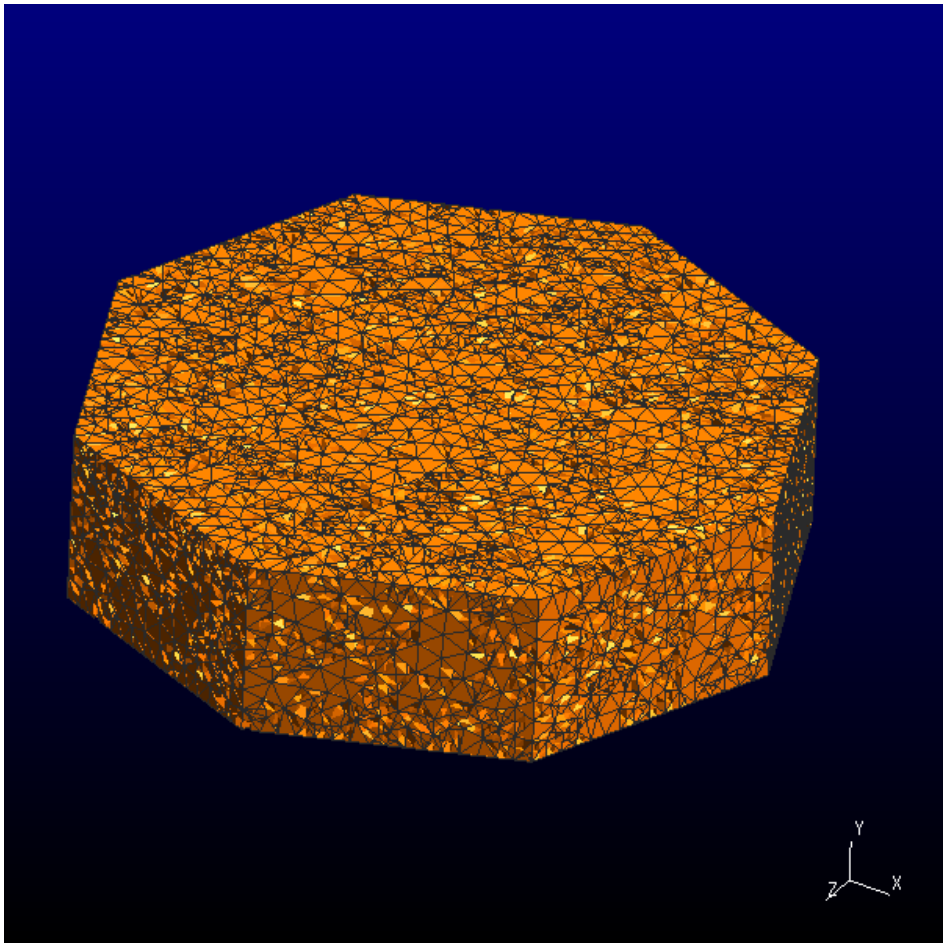
My code is at: http://code.google.com/p/grainmodel/

Geometry



Mesh

gmsh $\longrightarrow$ converter $\longrightarrow$ tetgen $\longrightarrow$ converter $\longrightarrow$ gmsh, libmesh



Formulation

Continuity equation:

$$\nabla \cdot \mathbf{j} = -\frac{\partial \rho}{\partial t}$$

and using the Ohm's law $\mathbf{j} = \sigma \mathbf{E} = -\sigma \nabla \varphi$ and setting $\frac{\partial \rho}{\partial t} = 0$ we get

$$\nabla \cdot \sigma \nabla \varphi = 0$$

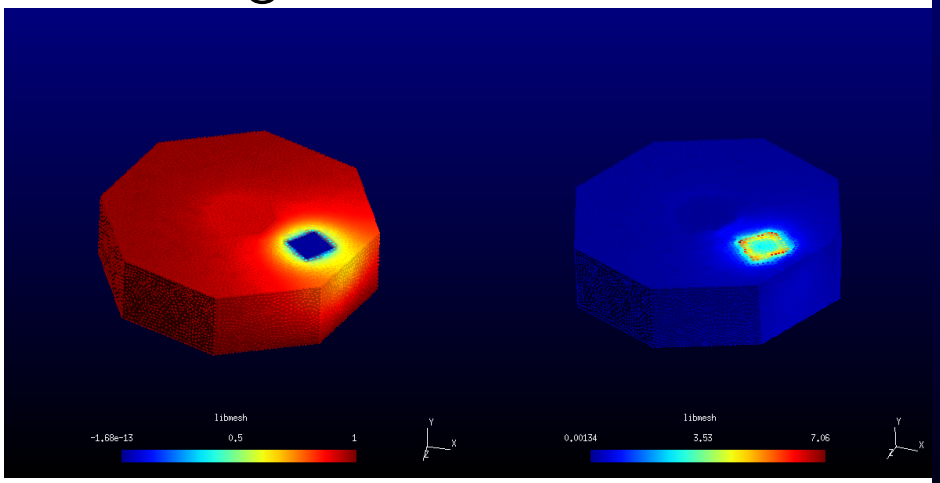
Standard FEM procedure: weak formulation, global matrix assembly, solving the large sparse problem $Ax = b$, etc.

Solver

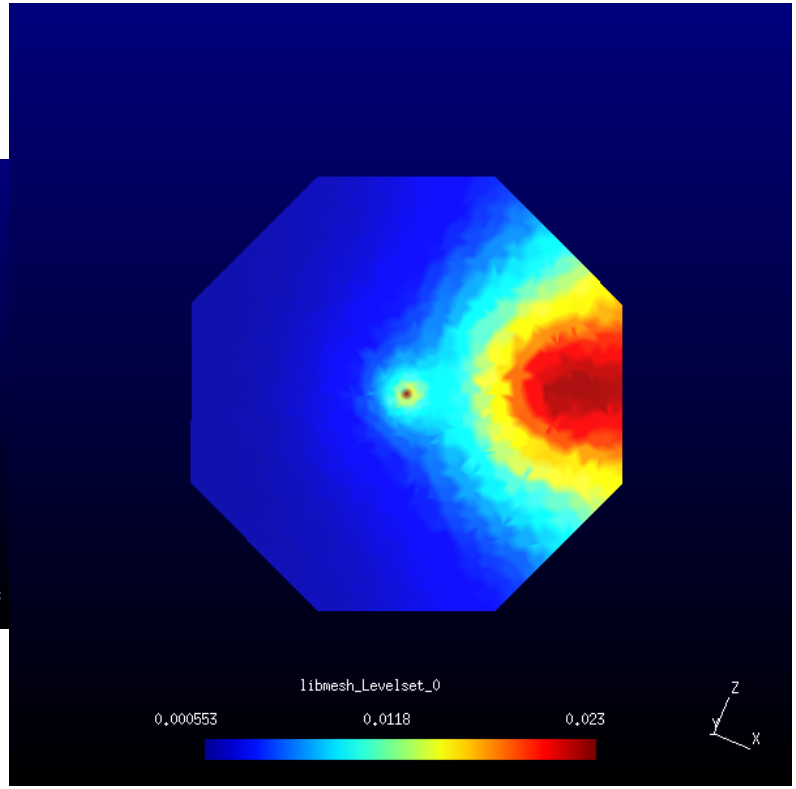
- main program in Python
- libmesh (C++), together with SWIG bindings, to assemble
- petsc (C), together with petsc4py (SWIG), to solve

Solution

solution+gradient



substrate plane



SymPy

- A Python library for symbolic mathematics
- http://code.google.com/p/sympy/

Why symbolic mathematics? The same reasons people use Maple/Mathematica, but we want to use it from Python.

```
>>> from sympy import Symbol, limit, sin, oo
>>> x=Symbol("x")
>>> limit(sin(x)/x, x, 0)
1
>>> limit((5**x+3**x)**(1/x), x, oo)
5
```

SymPy

What SymPy can do

- basics (expansion, complex numbers, differentiation, taylor (laurent) series, substitution, arbitrary precision integers, rationals and floats, pattern matching)
- noncommutative symbols
- limits and very simple integrals (so far)
- polynomials (division, gcd, square free decomposition, groebner bases, factorization)
- symbolic matrices (determinants, LU decomposition...)
- solvers (some algebraic and differential equations)
- 2D geometry module
- plotting (2D and 3D)

SymPy

In his introduction to the book A=B Donald Knuth wrote:

> Science is what we understand well enough to explain to a computer. Art is everything else we do. During the past several years an important part of mathematics has been transformed from an Art to a Science

- Being able to see the computer doing mathematics, I understand it better (symbolic limits, factorization, expansion, integration, ...)
- one example: epsilon delta gymnastics in limits

SymPy

Other symbolic manipulation software: GiNaC, Giac, Qalculate, Yacas, Eigenmath, Axiom, PARI, Maxima, SAGE, Singular, Mathomatic, Octave, ...
Problems:

- all use their own language (except GiNac, Giac and SAGE)
- GiNac and Giac still too complicated (C++), difficult to extend

What we want

- Python library and that's it (no environment, no new language, nothing)
- Rich funcionality
- Pure Python (non Python modules are only optional) – works on Linux, Windows, Mac out of the box

SAGE

- aims to glue together every useful open source mathematics software package and provide a transparent interface to all of them
- http://www.sagemath.org/

```
sage: import sympy
sage: sympy.__version__
'0.4.2'
sage: x=sympy.Symbol("x")
sage: y=sympy.Symbol("y")
sage: ((x+y)**sympy.Rational("5")).expand()
5*x**4*y+y**5+x**5+10*x**3*y**2+5*x*y**4+10*y**3*x**2
```

the Schwarzschild solution in the General Relativity

spherically symmetric metric ($diag(-e^{\nu(r)}, e^{\lambda(r)}, r^2, r^2 \sin^2\theta)$) $\rightarrow$ Christoffel symbols $\rightarrow$ Riemann tensor $\rightarrow$ Einstein equations $\rightarrow$ solver

```
ondra@pc232:~/sympy/examples$ time python relativity.py
...
[SKIP]
...
----------------------------------------
metric:
-C1 - C2/r 0 0 0
0 1/(C1 + C2/r) 0 0
0 0 r**2 0
0 0 0 r**2*sin(\theta)**2

real 0m1.092s
user 0m1.024s
sys 0m0.068s
```

How SymPy is developed

- I wrote first code 2 years ago, limits a year ago and then stopped working on it
- Fabian from Spain joined in February
- Google Summer of Code, SymPy is under the umbrella of Python Software Foundation, the Space Telescope Science Institute and Portland State University
- Now there are 8 regular contributors with svn access, other people send patches sometimes
- GSoC students wrote most of the modules, Pearu Peterson wrote a new core (10x to 100x faster than the old core), Fredrik wrote a fast floating point arithmetics in Python (faster than the Decimal module), Jurjen contributed pretty printing etc.