PDE Project Course 05/06

The Linearized Euler Equations and the Incompressible Navier Stokes equations

Maarten Hornikx (Euler Equations)
Email: maarten.hornikx@chalmers.se

Andreas Fischer (Navier Stokes Equations)
Email: afish@mdstud.chalmers.se
Contents

1 Introduction 4

2 Differential equation 4
   2.1 Incompressible Navier Stokes Equations . . . . . . . . . . . . 4
   2.2 Linearized Euler Equations . . . . . . . . . . . . . . . . . . . 6

3 Method 7
   3.1 Incompressible Navier Stokes equation . . . . . . . . . . . . 7
   3.2 Linearized Euler Equations . . . . . . . . . . . . . . . . . . . 8

4 Implementation 10
   4.1 FEniCS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
      4.1.1 DOLFIN/pyDOLFIN . . . . . . . . . . . . . . . . . . . . 11
      4.1.2 FFC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
      4.1.3 FIAT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
      4.1.4 PETSc . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
   4.2 Incompressible Navier-Stokes equation . . . . . . . . . . . . . 12
   4.3 Linearized Euler Equations . . . . . . . . . . . . . . . . . . . . 12

5 Results 14
   5.1 Incompressible Navier Stokes equation . . . . . . . . . . . . 14
   5.2 Linearized Euler Equations . . . . . . . . . . . . . . . . . . . 15
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.1 The box problem</td>
<td>15</td>
</tr>
<tr>
<td>5.2.2 The applied problem</td>
<td>19</td>
</tr>
<tr>
<td>5.3 Linearized Euler Equations + Incompressible Navier Stokes equations</td>
<td>19</td>
</tr>
<tr>
<td>6 Conclusions</td>
<td>21</td>
</tr>
<tr>
<td>6.1 Incompressible Navier-Stokes equation</td>
<td>21</td>
</tr>
<tr>
<td>6.2 Linearized Euler Equations</td>
<td>22</td>
</tr>
<tr>
<td>6.3 Linearized Euler Equations + Incompressible Navier Stokes equations</td>
<td>22</td>
</tr>
<tr>
<td>A Code</td>
<td>25</td>
</tr>
<tr>
<td>A.1 Incompressible Navier Stokes Equations</td>
<td>25</td>
</tr>
<tr>
<td>A.1.1 NSSolver.py</td>
<td>25</td>
</tr>
<tr>
<td>A.1.2 NSEMomentum2D.form</td>
<td>30</td>
</tr>
<tr>
<td>A.1.3 NSEContinuity2D.form</td>
<td>31</td>
</tr>
<tr>
<td>A.2 Linearized Euler Equations</td>
<td>32</td>
</tr>
<tr>
<td>A.2.1 Eulermomentum.py</td>
<td>32</td>
</tr>
<tr>
<td>A.2.2 Eulercontinuity.py</td>
<td>33</td>
</tr>
<tr>
<td>A.2.3 LEE.py</td>
<td>34</td>
</tr>
</tbody>
</table>
1 Introduction

The problem we have solved concerns an urban sound propagation problem. Sound propagation in an urban environment is a complex process. The sound waves due to, for example, road traffic are reflected by the ground and objects like buildings and screens. The finiteness of the objects causes diffraction at the ends of the them. Wind velocities change the local speed of sound, influencing the direction of sound waves. In an urban environment, we aim at modeling the sound waves that reach a shadow zone behind an urban traffic flow, situated between two building blocks. This situation, especially when taking into account the influence of a wind gradient, is hard to model by analytical models. Recently, time domain models were used to model this problem (e.g. [13]). In these methods, the governing equations were solved by the finite difference method in time and space. The problem will here be solved with finite differences in time and the finite element method in a 2-dimensional space.

2 Differential equation

The equations governing sound propagation in a moving medium may be divided up in two sets of equations, that first will be solved separately: the acoustical equations (Linearized Euler Equations, see below) and the equations solving for the atmospheric background flow (Incompressible Navier Stokes equation). We are allowed to do so, since the acoustical variables do not influence the atmospheric wind velocities (in good approximation). After having solved both equation systems, the steady state atmospheric background flow will be inserted in the acoustical equations.

2.1 Incompressible Navier Stokes Equations

\[
\begin{align*}
\frac{\partial \vec{u}_{av}}{\partial t} + (\vec{u}_{av} \cdot \nabla) \vec{u}_{av} - \varepsilon \Delta \vec{u}_{av} + \nabla p_{av} &= f, \quad \text{in } \Omega \\
\nabla \cdot \vec{u}_{av} &= 0, \quad \text{in } \Omega
\end{align*}
\]
\( \vec{u}_{av} = \begin{bmatrix} 2x-1 \\ 0 \end{bmatrix}, \quad \text{on } \partial \Omega_1 \) \hfill (2)

\( \vec{u}_{av} = 0, \quad \text{on } \partial \Omega_i, \quad i = 1, \ldots, 8 \) \hfill (3)

\( \frac{\partial \vec{v}_{av}}{\partial x} \cdot n = 0, \quad \text{on } \partial \Omega_{10} \) \hfill (4)

\( \vec{v}_{av} \big|_y = 0, \quad \text{on } \partial \Omega_{10} \) \hfill (5)

\( p_{av} = 0, \quad \text{on } \partial \Omega_1 \) \hfill (6)

\( \frac{\partial p_{av}}{\partial n} = 0, \quad \text{on } \partial \Omega_i, \quad i = 1, 2, \ldots, 8, 10. \) \hfill (7)

where \( \vec{v}_{av} \) is the atmospheric velocity, \( p_{av} \) is the atmospheric pressure, \( f \) is the source term, \( \varepsilon \) is the dynamic viscosity. The boundary’s, \( \partial \Omega_i, \quad i = 1, \ldots, 10, \) is according to Figure 1. A simplification of (1) is to consider the source term \( f \), which consists of gravity and other forces acting on the air mass, to be negligible to the forces from the pressure and incoming velocity, i.e. \( f = 0 \). In (1), we define \( \vec{v}_{av} \) as the solution of Navier-Stokes equation.
2.2 Linearized Euler Equations

The Linearized Euler Equations for sound propagation read:

\[
\begin{align*}
\dot{\mathbf{u}} + (\mathbf{u}_{av} \cdot \nabla)\mathbf{u} + (\mathbf{u} \cdot \nabla)\mathbf{u}_{av} + 1/\rho_{av} \nabla p &= 0, \\
\dot{p} + \mathbf{u}_{av} \cdot \nabla p + c_0^2 \rho_{av} \nabla \cdot \mathbf{u} &= 0.
\end{align*}
\]

(8) \hspace{2cm} (9)

With \(c_0\) the adiabatic speed of sound, \(p\) the pressure, \(\rho\) the density and \(\mathbf{u}\) the velocity. The variables were decomposed in atmospheric (denoted by subscript \(\text{av}\)) and acoustic components, which can be regarded as small compared to the atmospheric components:

\[
\begin{align*}
\mathbf{u}_a &= \mathbf{u}_{av} + \mathbf{u}, \\
p_a &= p_{av} + p, \\
\rho_a &= \rho_{av} + \rho.
\end{align*}
\]

(10) \hspace{2cm} (11)

The Euler Equations were linearized with respect to the acoustic variables. Also, the following assumptions, common within the sound propagation society (see e.g. [8]) were made:

\[
\dot{\rho}_{av} = \nabla \rho_{av} = \nabla p_{av} = \nabla \rho_{av} = \nabla \cdot \mathbf{u}_{av} = 0.
\]

(12)

Also, entropy variations are neglected and the process is adiabatic (no heat transfer). Two different domains were used to solve the problem, see the results section. In both cases, sound propagation is initialized by a wavelet pressure distribution with center frequency \(2f_c\) and a zero acoustical velocity. The pressure envelope contains a band passed frequency content, see figure 6. Acoustically hard boundaries and reflection free boundaries close the computational domain. For the non reflecting boundaries (see figure 4), an artificial absorption layer is introduced to ensure no reflections (note that imposing reflection free conditions at the boundary \(\Gamma\) is not straightforward, since the normalized boundary impedance has to be 1 for all incident angles of sound waves). The absorption layer is modelled by an artificial viscosity term in the momentum equation, which was chosen analogue to the molecular viscosity term:

\[
\dot{\mathbf{u}} + (\mathbf{u}_{av} \cdot \nabla)\mathbf{u} + (\mathbf{u} \cdot \nabla)\mathbf{u}_{av} + 1/\rho_{av} \nabla p + \mu \Delta \mathbf{u} = 0.
\]

(13)
The thickness of the layer has to be several wavelengths for best results and $\mu$ is set gradually increasing with the thickness of the layer to have a smooth change of the medium impedance:

$$
\mu(x) = \alpha \left( \frac{x - d}{d} \right)^4.
$$

(14)

Where $x$ is the position in the absorption layer, $d$ the thickness of the absorption layer and $\alpha$ and empirical constant. The other boundary and initial conditions read:

$$
p(\cdot, 0) = \cos \left( \frac{2\pi f_c}{c_0} \sqrt{(x - x_0)^2 + (y - y_0)^2} \right) e^{-\sigma((x-x_0)^2+(y-y_0)^2)},
$$
on $\Omega_{\text{hard}},$ 

$$
\bar{u}_n(\cdot, 0) = 0, \quad \text{on } \Omega_{\text{hard}},
$$

$$
\nabla_n p = 0, \quad \text{on } \Gamma_{\text{hard}},
$$

$$
\dot{u}_n(\cdot, 0) = 0, \quad \text{on } \Gamma_{\text{hard}}.
$$

Where $\sigma$ determines the bandwidth of the signal.

3 Method

3.1 Incompressible Navier Stokes equation

One way to get a steady-state solution to (1) is to solve for a long enough time span for the transients to disappear from the solution. It might be that there doesn’t exist a steady-state solution because of turbulence or vortex formation, but the low inflow velocity gives a hint that this isn’t the case for this particular problem. The method to solve (1) will be the continuous Galerkin method of degree 1 in both space and time, $cG(1)cG(1)$. It reads: Find $(U^n, P^n) \in [W_0]^2 \times W$ such that:

$$
\left( \frac{U^n - U^{n-1}}{k}, v \right) + ((U^n \cdot \nabla) U^n, v) + (\varepsilon \nabla U^n, \nabla v) +
$$

(16)
(\nabla P^n, v) + (\nabla \cdot \bar{U}^n, q) = 0 \\
\forall (v, q) \in [W_0^n]^2 \times W^n$

where $W$ is the piecewise linear functions on the triangulation $\mathcal{T}$, $W_0$ is the piecewise linear functions on the triangulation $\mathcal{T}$ with compact support, $k$ is the time step $k = t_n - t_{n-1}$, $U^n = U(t_n)$, $\bar{U}^n = \frac{U^n + U^{n-1}}{2}$ and $(a, b) = \int_{\Omega} a \cdot b \, d\bar{x}$. In order to stabilize the solution a least squares stability factor:

$$
(\delta_1 (\bar{U}^n \cdot \nabla U^n + \nabla P^n), \bar{U}^n \cdot \nabla v + \nabla q) + (\delta_2 \nabla \cdot U^n, \bar{U}^n \cdot \nabla v + q)
$$

(17)
is added to (16), see [7], to make the resulting stiffness matrix positive definite.

### 3.2 Linearized Euler Equations

A staggered in time procedure and the finite element method are used to solve the Linearized Euler Equations. We write the equations using the Crank-Nicolson method in time. This means that we use a second order accurate centered finite difference scheme in time and evaluate the other variables at time step $\frac{1}{2} \Delta t$:

$$
\bar{u}_{\Delta t} + \frac{1}{2} \Delta t (\bar{u}_{av} \cdot \nabla) \bar{u}_{\Delta t} + \frac{1}{2} \Delta t (\bar{u}_{\Delta t} \cdot \nabla) \bar{u}_{av} + \frac{1}{2} \Delta t \mu \Delta \bar{u}_{\Delta t} =
\bar{u}_0 - \frac{1}{2} \Delta t (\bar{u}_{av} \cdot \nabla) \bar{u}_0 - \frac{1}{2} \Delta t (\bar{u}_0 \cdot \nabla) \bar{u}_{av} - \frac{\Delta t}{\rho_{av}} \nabla p_{\Delta t - \frac{1}{2}} - \frac{1}{2} \Delta t \mu \Delta \bar{u}_0.
$$

(18)

$$
p_{\Delta t + \frac{1}{2}} + \frac{1}{2} \Delta t (\bar{u}_{av} \cdot \nabla) p_{\Delta t + \frac{1}{2}} =
p_{\Delta t - \frac{1}{2}} - \Delta t c_0^2 \rho_{av} \nabla \cdot \bar{u}_{\Delta t} - \frac{1}{2} \Delta t (\bar{u}_{av} \cdot \nabla p_{\Delta t - \frac{1}{2}}).
$$

(19)

The first equation is expanded around $t = \frac{1}{2} \Delta t$, whereas the second equation is expanded around $t = \Delta t$. The momentum equation 18 is solved first for time step $t = \Delta t$ using initial values for the pressures at $t = \frac{1}{2} \Delta t$ and
velocity values at \( t = 0 \). Then, the continuity equation is solved for the pressures at \( t = \frac{3}{2} \Delta t \) using the initial values for the pressures and the just calculated velocities at \( t = \Delta t \). The momentum equation is then used for the calculating \( \vec{u} \) at \( t = 2 \Delta t \) using the former calculated velocity at \( t = \Delta t \) and the just calculated pressure at \( t = \frac{3}{2} \Delta t \). This procedure is called a staggered in time procedure, e.g. the velocity and the pressure are calculated at a displaced mesh in time.

The Linearized Euler equations 18 and 19 are multiplied by test functions \( v \) and \( \vec{v} \) and integrated over the 2D spatial domain to obtain the variational form:

\[
\int_{\Omega} \vec{u}_t \Delta t v \, dx + \int_{\Omega} \frac{1}{2} \Delta t (\vec{u}_{av} \cdot \nabla) \vec{u}_t v \, dx + \int_{\Omega} \frac{1}{2} \Delta t (\vec{u}_{av} \cdot \nabla) \vec{u}_{av} v \, dx + \int_{\Omega} \frac{1}{2} \Delta t \mu (\nabla \cdot \vec{u}_t) (\nabla \cdot v) \, dx = \int_{\Omega} \vec{u}_0 v \, dx - \int_{\Omega} \frac{1}{2} \Delta t (\vec{u}_{av} \cdot \nabla) \vec{u}_0 v \, dx - \int_{\Omega} \frac{1}{2} \Delta t (\nabla \cdot \vec{u}_0) (\nabla \cdot v) \, dx - \int_{\Omega} \frac{\Delta t}{\rho_{av}} \nabla p_{\Delta t - \frac{1}{2}} v \, dx - \int_{\Omega} \frac{1}{2} \Delta t \mu (\nabla \cdot \vec{u}_0) (\nabla \cdot v) \, dx. \tag{20}
\]

\[
\int_{\Omega} p_{\Delta t + \frac{1}{2}} \vec{v} \, dx + \int_{\Omega} \frac{1}{2} \Delta t (\vec{u}_{av} \cdot \nabla) p_{\Delta t + \frac{1}{2}} \vec{v} \, dx = \int_{\Omega} p_{\Delta t - \frac{1}{2}} \vec{v} \, dx - \int_{\Omega} \Delta t c^2 \rho_{av} \nabla \cdot (\vec{u}_t) \vec{v} \, dx - \int_{\Omega} \frac{1}{2} \Delta t (\vec{u}_{av} \cdot \nabla p_{\Delta t - \frac{1}{2}}) \vec{v} \, dx. \tag{21}
\]

We can write equations 20 and 21 in a bilinear and linear form:
\[ a(\vec{u}, \vec{v}) = L(\vec{v}), \quad (22) \]
\[ a(p, v) = L(v). \quad (23) \]

The left and right hand sides of equations 20 and 21 correspond to the left and right hand sides of equations 22 and 23. The program FFC [4] is used to compile the variational forms for a reference element. Triangles are used as elements in the 2 dimensional space together with linear shape functions. Section 4.1 describes the FFC program. The program Dolfin [2] is used to solve the linear problem on a specified mesh using the output from FFC. Results have been analysed in Matlab.

4 Implementation

4.1 FEniCS

The FEniCS (Finite Element in Computer Science) project is aiming at providing a Automation of Computational Mathematical Modeling (ACMM), that is: For any given PDE and tolerance to automatically compute a solution. The FEniCS project is a GPL project so anyone interested can (and are encouraged to) participate, see [5].

The project consists of:

- DOLFIN/pyDOLFIN, Dynamic Object oriented Library for FINite Element computation: The C++/Python interface for solving PDE’s
- FFC, The FEniCS Form Compiler: Takes the variational form of the PDE and a finite element to compute the reference tensor and multiplies it with the geometry tensor, see below.
- FIAT, The Finite element Automatic Tabulator: Constructs a large range of different finite elements of various degree.
- PETSc, The Portable, Extensible Toolkit for Scientific computation: A library to, amongst other things, solve large, sparse linear equations.
DOLFIN/pyDOLFIN, FFC, FIAT can be found at [3] and PETSc at [11]

4.1.1 DOLFIN/pyDOLFIN

The DOLFIN/pyDOLFIN (henceforth DOLFIN) is responsible for assembling the mass, stiffness and convection matrix, applying the boundary conditions and solving the discrete system. There is three levels of DOLFIN

- **User level.** In this level it’s, in theory, just to specify a variational form and a mesh of the problem and solve to get a solution. This is the level which an average user should be able to solve a problem.

- **Modular level.** In this level one can solve a problem which is not in the user level. Requires a deeper knowledge of the FEM and is therefore more advanced.

- **Kernel level.** In this level one can specify more basic things such as adding a different linear solver or change how to assemble the matrices. This level is mostly for developers of DOLFIN.

Work done in the modular or the kernel level should be incorporated into the user level so that more users can use this level.

4.1.2 FFC

When computing the integrals over a triangle in the FEM there is a possibility to compute the integral over a reference triangle and then map the result to the actual triangle. The integral over the reference triangle can be computed, either exact or using quadrature, but since it does not depend on the geometry it can be precomputed. FFC take a variational form and a finite element basis to precompute the integrals over the reference triangle and multiplies it with the mapping to the actual triangle in a effective way.
4.1.3 FIAT

Since simple finite elements may not be suited to be used on all PDE’s this program supplies more advanced finite elements, e.g. $H(div)$ for electromagnetic PDE’s. It does this by using a prime basis to build the finite element, see [9].

4.1.4 PETSc

PETSc is written in C, but can be used with C++ and Fortran, and uses MPI to communicate, so it can be used in a parallel environment. It is mainly designed to be used in PDE calculations, but it can be used by any application that needs a sparse solver. It has a wide range of linear system, nonlinear system and ODE solvers.

4.2 Incompressible Navier-Stokes equation

An existing Navier-Stokes solver written in C++ was translated to Python with minor changes to speed the code up and to reflect the different boundary conditions for the problem. The code uses a fixed point iteration to solve the nonlinear equation see page 25. A mesh was done with the help of PDE Toolbox in MATLAB [10]. The mesh consisted of 34560 triangles in order to be able to catch the interesting frequencies in the Euler equation. It should be rather straightforward to change the boundary conditions, the parameters to the equation and add a source term, but to change, for example, the stabilization takes a bit more understanding of the language used in the form file.

4.3 Linearized Euler Equations

In the appendix, the Dolfin code for solving equations 8 and 9 are included. The code uses Python [12] as a language. In the script, the initial values and boundary conditions for the variables $p$ and $\vec{u}$ are prescribed as well as values
for the constants $c_0$, $\mu$ and $\Delta t$ and a mesh. For the calculation of the geometry with the buildings, the triangular mesh was obtained from the PDEtool of MatLab [10]. The 2 dimensional space is discretized in triangles. In a second step, the variational form files are compiled with FFC (or precalculated FFC outputs are used). Three form files were created for the projection of space dependent variables on the mesh ($p$, $\vec{u}$ and $\mu$). The main part of the code is the loop over the time steps up to the predefined $t_{\text{max}}$. Using the input data, an equation system for all nodes is formed using predefined functions in *Dolf in*. The unknown velocities are then calculated by an iterative solver (Krylov solver). For the calculation loop, the old variables are overwritten by the new calculated ones. In a next time step, the pressures are calculated likewise.

The code is flexible regarding the geometry and boundary conditions, which only have to be altered in the input section of the code. In the equation system, we haven’t included source terms, yet a pressure distribution was used to initialize the calculation. Source terms could be included by writing them at the right hand sides of the equations 8 (a volume source) and 9 (a force source). The code is prepared to include atmospheric wind velocity terms. When these values are set to zero, a situation without wind is obtained. For underwater problems (for example), the ambient density is not constant over space. This has to be taken into account in equations 8 and 9. The same solution method can then be used. Temperature difference (which could occur in the atmosphere for example) can be modelled by setting the adiabatic sound speed $c_0$, which is dependent on $T$, spatially dependent. For a full finite difference scheme, stability is determined by the ratio $\frac{\Delta c_0}{\Delta x}$, which has to be smaller than 1 (see e.g. [13]). Here, this ratio was found to be smaller 0.5 for stability. For accuracy (see results section) $\Delta t$ and $\Delta x$ have to be much smaller than a corresponding wavelength of the wavelet signal. This limits the computations regarding computation time, especially for the higher frequencies. Since the pressures and velocities are solved for all nodes at every time step, the code is not very effective (an adaptive grid would be a possibility when sound waves are only present in a certain region). Also, an iterative solver that is more efficient than the one used here might be an option.
5 Results

5.1 Incompressible Navier Stokes equation

The resulting code was run on a 4 CPU Opteron 270 at 2 GHz with approximately 4000 MB of RAM. The program used several hours to complete, but this can be attributed to the fine mesh and to the long time span used. The first timestep was more timeconsuming since the initial condition was the zero solution so the stabilization parameters was set to accomplish convergence of this timestep.

The solution can be seen in Figure 2. When adding streamlines to the solution Figure 3 one sees that a vortex is formed between the “buildings” and after the last “building”. This is to be expected since a part of the geometry (between the “buildings”) specify a open box problem. No steady state solution exists. To solve the Navier-Stokes problem on our geometry has been tried with the help of the *comsol* program [1], but the boundary conditions was not easily implemented in the program so this idea was abandoned.

![Figure 2: Solution of the Navier-Stokes equation](image)
5.2 Linearized Euler Equations

Two geometries were used to solve the LEE, a square box and a geometry with building blocks (see figure 4). This geometry can be regarded as a 1:20 representation of a geometry in reality. The box is used to display accuracy of the method, whereas the second geometry represents a more applied problem.

5.2.1 The box problem

Figures 5 show three snapshots of a calculation in the box at various time steps. We clearly see the wave front expanding, reflecting at the left boundary and being absorbed at the other boundaries. In this calculation, $\Delta t = 1e^{-5} \text{s}$, $\Delta x \approx 0.01 \text{m}$, $f_c = 2000 \text{Hz}$, $\sigma = 20$, $\alpha = 1e^3$ and $d = 0.3 \text{m}$. To calculate accuracy, we do the following. The first wave (we gate out the reflected wave from the time signal) reaching the point $(0.4, 0.5)$ is taken. A fast Fourier transform is applied to obtain the complex pressure amplitude in the frequency domain. The analytic solution to a problem with a point source $\delta(x - 0.5)\delta(y - 0.5)$ reads:
Figure 4: Left) The box problem (for the sound propagation problem only. Right) The urban geometry.

\[
p(f, r) = -A(f)\frac{j}{4}H_0^2\left(\frac{2\pi f}{c_0}r\right),
\]

with:
\[
r = \sqrt{(x - 0.5)^2 + (y - 0.5)^2}
\]
\[
H_0^{(2)} = \text{Hankel function of order zero and second kind},
\]
\[
A(f) = \text{Frequency dependent source amplitude}.
\]

The source amplitude is due to the initial pressure distribution and can be retrieved for the position (0.4, 0.5) by;

\[
A(f) = \frac{p_n(f, 0.1)}{-\frac{1}{4}H_0^2\left(\frac{2\pi f}{c_0}0.1\right)},
\]

with:
\[
p_n(f, 0.1) = \text{the calculated complex pressure at 0.1 m from the source}.
\]

Now, we can retrieve the point source solution from our numerical results.
by dividing the Fourier transformed signal at an arbitrary point in the box $p_n (f, (x - 0.5, y - 0.5))$ by the source amplitude $A(f)$. The error of the method can than be displayed by comparing this result with an analytic result. Figures 7 show amplitude results from the numerical and analytic model for different positions. For the region around $2f_c$ the agreement is fine. Another error of interest is the phase error (does the calculated results travel faster or slower than the analytic one?). Therefore, the phase of the calculated result is plotted together with the phase of the analytic expression. Also here, the agreement is fine.

Figure 5: Snapshots of the absolute pressure in the box at 4 time instances: $t = 0.5$ ms, $t = 1.5$ ms, $t = 2.5$ ms and $t = 3.5$ ms
Figure 6: The box problem. First 1 ms of signal arriving at (0.4, 0.5) and its frequency content $A(f)$, see text.

Figure 7: The box problem. Comparison between the calculated amplitude and phase at three different positions with analytical results. Dashed) calculated, Solid) analytical. Upper) receiver position at (0.3, 0.5), Middle) receiver position at (0, 0.5) Lower) receiver position at (0.4, 0.5) (a direct and reflected wave)
5.2.2 The applied problem

We now turn to the problem of sound propagation in an urban environment. Figures 8 show six snapshots of the sound propagation. It displays the various effects of reflection, diffraction and the influence of the absorption layer. In this calculation, $\Delta t = 2e^{-5}s$, $\Delta x \approx 0.02m$, $f_c = 500Hz$, $\sigma = 200$, $\alpha = 1e^3$ and $d = 0.5m$. Clearly, sound waves reaching the backside of the building blocks are strongly reduced. Figure 9 shows the amplitude relative to the amplitude in free field (i.e. without the buildings) at a receiver point at $(3.25, 0.5)$. The reference results has been calculated using a diffraction model for a point source [6]. Although this diffraction model is for a point source, the level relative to the free field level for these geometries is considered being equal for point and line sources (the calculated solution is a 2D solution, a line source solution) [14]. Also, this model is known to be slightly inaccurate when the wavelength is large compared to the distances in the problem. The results show that the computed amplitude values are inaccurate, especially at the higher frequencies. At the lower frequencies, the error is smaller, yet a certain error exists everywhere. The error at the lower frequencies could also be due to the inaccuracy of the diffraction model at these frequencies. With increasing the number of triangles, the results get better. It is known that the ratio $\frac{\lambda}{\Delta x}$, where $\lambda$ the wavelength, has to be around ten to obtain a reasonable accuracy. At 1000 Hz however, we already have a ratio of 17. It is remarkable that the results in the box problem are more accurate than the ones here. The phase of the calculated result are quite close to the phase of the diffraction model result. There is a small error everywhere, denoting a small phase difference for all frequencies.

5.3 Linearized Euler Equations + Incompressible Navier Stokes equations

The Incompressible Navier Stokes equations are used to calculate the background velocities at the same mesh as used for the Euler Equations variables of section 5.2.2. In the Dolfin code LEE.py, see appendix, these results are loaded and the values for $\vec{u}_{av}$ are used in the calculation. The calculation results showed a stable calculation. Snapshots of the new calculation did visibly
Figure 8: Snapshots of the positive pressure in the urban geometry at 6 time instances: \( t = 0.5 \) ms, \( t = 1 \) ms, \( t = 1.5 \) ms, \( t = 2 \) ms, \( t = 2.5 \) ms and \( t = 3.0 \) ms
6 Conclusions

6.1 Incompressible Navier-Stokes equation

This project shows that it is timeconsuming to get a steady-state solution of the Navier-Stokes equation using the time dependent equations (1). A way to dynamically change the stabilization parameters when solving the problem should shorten the solving time.
6.2 Linearized Euler Equations

The Linearized Euler Equations were solved using the Crank-Nicholson method in time and the finite element method in space. Calculations in a model problem (a box) with hard and absorbing boundary conditions show a good agreement with analytical results for both amplitude and phase. A calculation in an urban geometry (building blocks are present) show however clear inaccuracies in amplitude (and to a less extent) phase. The meshsize has an influence on the results, yet the source of the inaccuracies has not been fully understood. For all situations, the calculations were stable.

6.3 Linearized Euler Equations + Incompressible Navier Stokes equations

The combination of the Linearized Euler Equations and the Incompressible Navier Stokes equations yielded a stable result. The results for the case without wind however already showed inaccuracies. When comparing the results of the case with wind to the case without wind showed a clear difference for the amplitude of the signal. Phase differences were however not obvious, which could be attributed to the low wind velocities.
References


A Code

A.1 Incompressible Navier Stokes Equations

A.1.1 NSSolver.py

```python
from dolfin import *
from math import *

def GetMinCellSize(Mesh):
    ret=1e6
    ci=CellIterator(mesh)
    while not ci.end):
        if (ci.diameter() < ret):
            ret=ci.diameter()
        ci.increment()
    return ret

def ComputeStabilization(mesh,w,ny,k,d1vec,d2vec):
    C1=4.0
    C2=2.0

    ci=CellIterator(mesh)
    n=VertexIterator(ci)
    normw=0.0
    while not ci.end() :
        while not n.end() :
            normw+= (1.0/(k**2)+(normw/ci.diameter())**2)
            n.increment()
        normw/=ci.numVertices()
    if( ((ci.diameter()/ny) > 1.0 ) or (ny < 1.0e-10)):
        d1vec[ci.id()]=C1*(0.5/sqrt(1.0/(k**2)+(normw/ci.diameter())**2))
        d2vec[ci.id()]=C2*ci.diameter()
    else:
        d1vec[ci.id()]=C1*(ci.diameter())**2
        d2vec[ci.id()]=C2*(ci.diameter())**2
    ci.increment()

class Source (Function):
    def eval(self,point,i):
```

return 0.0

class BC_momentum(BoundaryCondition):
    def eval(self,value,point,i):
        if( (point.y == 1.0 ) and ((point.x >= 0.0) and (point.x <= 1.0))):
            ## Boundary no 2
            value.set(0.0)
        elif((point.x -1.0)<1.0e-3 and ((point.y >= 0.0) and (point.y <= 1.0))):
            ## Boundary no 3
            value.set(0.0)
        elif((point.y == 0.0) and ((point.x >= 1.0) and (point.x <= 2.0))):
            ## Boundary no 4
            value.set(0.0)
        elif( ( (point.x- 2.0)<1.0e-3) and ((point.y >= 0.0) and (point.y <= 1.0))):
            ## Boundary no 5
            value.set(0.0)
        elif( (point.y < (1.0+1e-6) ) and (( point.x >= 2.0) and (point.x <= 3.0))):
            ## Boundary no 6
            value.set(0.0)
        elif( ( ((point.x- 3.0 )<1.0e-3) and ((point.y >= 0.0) and (point.y <= 1.0)) )):
            ## Boundary no 7
            value.set(0.0)
        elif( (point.y == 0.0 ) and(point.x >= 3.0)):
            ## Boundary no 8
            value.set(0.0)
        if(i==0):
            if((point.x < 1.0e-6) and (point.y >= 1.0)):
                ## Boundary no 1 (x-vel)
                value.set(2*(point.y-1.0)/(point.y+1e-12))
        if(i==1):
            if(point.x == 0 and point.y >=1.0 ):
                ## Boundary no 1 (y-vel)
                value.set(0.0)
            if(point.y == 2.0):
                ## Boundary no 10 (y-vel)
                value.set(0.0)

class BC_cont(BoundaryCondition):
    def eval(self,value,point,i):
        if(point.x == 4):
            value.set(0.0)
            ## Boundary no 9 pressure
        return value
f=Source()
bcmom=BC_momentum()
bc_con=BC_cont()
mesh=Mesh("bygnaderfinal2.xml.gz")

Am=Matrix()
bm=Vector()

Ac=Matrix()
bc=Vector()

filu=File("byggvel.pvd")
filp=File("byggpre.pvd")

T0=0.0
t=0.0
T=10

ny=17.4-6/1.6

hmin=GetMinCellSize(mesh)

hmin=0.01/sqrt(2.0)
if( (T-T0)/100 < 0.1*hmin ):
   k=(T-T0)/100
else:
k=.1*hmin
dolfin_log(False)

nsd=mesh.numSpaceDim()

x0vel=Vector(nsd*mesh.numVertices())
xcvel=Vector(nsd*mesh.numVertices())
vel=Vector(nsd*mesh.numVertices())
xpre=Vector(mesh.numVertices())
x0vel.copy(0.0)
xvel.copy(10.0)
xvel.copy(0.0)
xpre.copy(0.0)

#SetInitialVelocity(xvel)
resm=Vector(nsd*mesh.numVertices())
resc=Vector(mesh.numVertices())
resm.copy(1.0e3)
resc.copy(1.0e3)

#linsolv=LU()
linsolv=KrylovSolver()
linsolv.setRtol(1.0e-6)

u0=Function(x0vel,mesh)
uc=Function(xcvel,mesh)
p=Function(xpre,mesh)

d1vec=Vector(mesh.numCells())
d2vec=Vector(mesh.numCells())

delta1=Function(d1vec)
delta2=Function(d2vec)

mom_forms=import_formfile("NSEMomentum2D.form")
con_forms=import_formfile("NSEContinuity2D.form")

am=mom_forms.NSEMomentum2DBilinearForm(uc,delta1,delta2,k,ny)
Lm=mom_forms.NSEMomentum2DLinearForm(uc,u0,f,p,delta1,delta2,k,ny)
ac=con_forms.NSEContinuity2DBilinearForm(delta1)
Lc=con_forms.NSEContinuity2DLinearForm(uc,f,delta1)

trialelement=mom_forms.NSEMomentum2DBilinearFormTrialElement()
u=Function(xvel,mesh,trialelement)

u.sync(t)
p.sync(t)
bc_con.sync(t)
bcmom.sync(t)

ComputeStabilization(mesh,u0,ny,k,d1vec,d2vec)

FEM_assemble(ac,Ac,mesh)
# print "assemble Acont 1"
FEM_assemble(am,Am,mesh)
# print "assemble Amom 1"

time_step=0
```python
sample=0
no_samples=30
rtol=1.0e-3
max_iter=100

while(t<T):
    time_step+=1
    x0vel.copy(xvel)

    ComputeStabilization(mesh,u0,ny,k,d1vec,d2vec)
    residual=2*rtol
    iteration=0

    FEM_assemble(Lc,bc,mesh)
    #print "assemble bcont 1"
    FEM_applyBC(Ac,bc,mesh,ac.trial(),bc_con)
    #print "apply bc_cont 1"

    while(residual > rtol and iteration<max_iter):
        linsolv.solve(Ac,xpre,bc)
        FEM_assemble(Lm,bm,mesh)
        FEM_applyBC(Am,bm,mesh,am.trial(),bc_mom)
        linsolv.solve(Am,xvel,bm)
        xcvel.copy(xvel)

        FEM_assemble(am,Lm,Am,bm,mesh,bc_mom)
        FEM_applyBC(Am,bm,mesh,am.trial(),bc_mom)

        Am.mult(xvel,resm)
        resm-=bm

        FEM_assemble(Lc,bc,mesh)
        FEM_applyBC(Ac,bc,mesh,ac.trial(),bc_con)

        Ac.mult(xpre,resc)
        resc-=bc

        residual=sqrt(resm.norm()**2+resc.norm()**2)
        iteration+=1

        if(residual > rtol):
```

```
dolfin_warning("Fixpoint iteration did not converge")

if( (time_step ==1 ) or (t > (T-T0)*sample/no_samples)):
    dolfin_log(True)

    print t
    filu << u
    filp << p
    sample+=1
    dolfin_log(False)

    t+=k
    prog=t/T

filu << u
filp << p

xmlvel=File("velocity.xml",File.xml)
xmpre=File("pressure.xml",File.xml)

xmlvel << xvel
xmlpre << xpre

A.1.2 NSEMomentum2D.form

# Copyright (c) 2005 Johan Hoffman (hoffman@cims.nyu.edu)
# Licensed under the GNU GPL Version 2
#
# The momentum equation for the incompressible
# Navier-Stokes equations using cG(1)cG(1)
#
# Compile this form with FFC: ffc NSEMomentum.form.
# Changed 2006 by Andreas Fischer

name = "NSEMomentum2D"
scalar = FiniteElement("Lagrange", "triangle", 1)
vector = FiniteElement("Vector Lagrange", "triangle", 1)
constant_scalar = FiniteElement("Discontinuous Lagrange", "triangle", 0)

v = BasisFunction(vector) # test basis function
u = BasisFunction(vector) # trial basis function
uc = Function(vector) # linearized velocity
\[ u_0 = \text{Function(vector)} \# \text{velocity from previous time step} \]

\[ f = \text{Function(vector)} \# \text{force term} \]

\[ p = \text{Function(scalar)} \# \text{pressure} \]

\[ \text{delta1} = \text{Function(constant\_scalar)} \# \text{stabilization parameter} \]

\[ \text{delta2} = \text{Function(constant\_scalar)} \# \text{stabilization parameter} \]

\[ k = \text{Constant()} \# \text{time step} \]

\[ \text{nu} = \text{Constant()} \# \text{viscosity} \]

\[ \text{um} = \text{mean(uc)}; \# \text{cell mean value of linearized velocity} \]

\[ i_0 = \text{Index()} \# \text{index for tensor notation} \]

\[ i_1 = \text{Index()} \# \text{index for tensor notation} \]

\[ i_2 = \text{Index()} \# \text{index for tensor notation} \]

# Galerkin discretization of bilinear form
\[
G_a = v[i0]*u[i0]*dx + k*nu*0.5*v[i0].dx(i1)*u[i0].dx(i1)*dx + 0.5*k*v[i0]*uc[i1]*u[i0].dx(i1)*dx
\]

# Least squares stabilization of bilinear form
\[
SD_a = \text{delta1}*k*0.5*um[i1]*v[i0].dx(i1)*um[i2]*u[i0].dx(i2)*dx + \text{delta2}*k*0.5*v[i0].dx(i0)*u[i1].dx(i1)*dx
\]

# Galerkin discretization of linear form
\[
G_L = v[i0]*u0[i0]*dx + k*v[i0]*f[i0]*dx + k*v[i0].dx(i0)*p*dx - k*nu*0.5*v[i0].dx(i1)*u0[i0].dx(i1)*dx - 0.5*k*v[i0]*uc[i1]*u0[i0].dx(i1)*dx
\]

# Least squares stabilization of linear form
\[
SD_L = \text{delta1}*k*um[i1]*v[i0].dx(i1)*f[i0]*dx - \text{delta1}*k*0.5*um[i1]*v[i0].dx(i1)*um[i2]*u0[i0].dx(i2)*dx - \text{delta1}*k*um[i1]*v[i0].dx(i1)*p.dx(i0)*dx - \text{delta2}*k*0.5*v[i0].dx(i0)*u0[i1].dx(i1)*dx
\]

# Bilinear and linear forms
\[ a = G_a + SD_a \]
\[ L = G_L + SD_L \]

A.1.3 NSEContinuity2D.form
# The continuity equation for the incompressible 
# Navier-Stokes equations using cG(1)cG(1) 
# Compile this form with FFC: ffc NSEContinuity.form.

name = "NSEContinuity2D"
scalar = FiniteElement("Lagrange", "triangle", 1)
vector = FiniteElement("Vector Lagrange", "triangle", 1)
constant_scalar = FiniteElement("Discontinuous Lagrange", "triangle", 0)

q = BasisFunction(scalar)  # test basis function
p = BasisFunction(scalar)  # trial basis function
uc = Function(vector)  # linearized velocity
f = Function(vector)  # force term

delta1 = Function(constant_scalar)  # stabilization parameter

um = mean(uc);  # cell mean value of linearized velocity

i0 = Index()  # index for tensor notation
i1 = Index()  # index for tensor notation

# Bilinear and linear forms
a = delta1*dot(grad(q), grad(p))*dx;
L = delta1*dot(grad(q), f)*dx - q*uc[i0].dx(i0)*dx -
delta1*q.dx(i0)*um[i1]*uc[i0].dx(i1)*dx

A.2 Linearized Euler Equations

A.2.1 Eulermomentum.py

# Eulermomentum.form
#
# input code for the FFC program
#
# Momentum equation from the Euler equations.

scalar = FiniteElement("Lagrange", "triangle", 1)
vector = FiniteElement("Vector Lagrange", "triangle", 1)
v = BasisFunction(vector)  # test function
unew = BasisFunction(vector)  # pressure at time step 1
uold = Function(vector)  # pressure at previous time step
pold = Function(scalar)  # acoustic velocity at half time step
ua = Function(vector)  # atmospheric velocity at half time step
artvis = Function(scalar)  # artificial viscosity
dt = Constant()  # time step
rhoa = Constant()  # atmospheric density

artvis2 = 0.5*dt*artvis*dot(grad(uold),grad(v))*dx

a = unew[i]*v[i]*dx + 0.5*dt*ua[j]*D(unew[i],j)*v[i]*dx +
0.5*dt*unew[j]*D(ua[i],j)*v[i]*dx + 0.5*dt*artvis*dot(grad(unew),grad(v))*dx

L = uold[i]*v[i]*dx - dt/rhoa*D(pold,i)*v[i]*dx -
0.5*dt*ua[j]*D(uold[i],j)*v[i]*dx - 0.5*dt*uold[j]*D(ua[i],j)*v[i]*dx - artvis2

A.2.2 Eulercontinuity.py

# Eulercontinuity.form
#
# input code for the FFC program

scalar = FiniteElement("Lagrange", "triangle", 1)
vector = FiniteElement("Vector Lagrange", "triangle", 1)

v = BasisFunction(scalar)  # test function
pnew = BasisFunction(scalar)  # pressure at time step 1
pold = Function(scalar)  # pressure at previous time step
uold = Function(vector)  # acoustic velocity at half time step
ua = Function(vector)  # atmospheric velocity
dt = Constant()  # time step
c = Constant()  # adiabatic speed of sound
rhoa = Constant()  # atmospheric density

a = pnew*v*dx + 0.5*dt*ua[j]*D(pnew,j)*v*dx +
0.5*dt*alfap*pnew*v*dx

L = pold*v*dx - dt*c*c*rhoa*div(uold)*v*dx -
0.5*dt*ua[j]*D(pold,j)*v*dx - 0.5*dt*alfap*pold*v*dx
A.2.3 LEE.py

#___________________________________________________________
#LEE solver
#(linearized equation equation solver)
#
from dolfin import *
from math import *

#____________________________________________________________
# initial conditions and BC's

class SimpleBCu(BoundaryCondition):
    def eval(self, value, point, i):
        if (i == 0 and point.x == 0.0 and point.y >= 1):
            value.set(0.0)
        if (i == 1 and point.y == 1.0 and point.x >= 0.0 and point.x <= 1.0):
            value.set(0.0)
        if (i == 0 and point.x == 1.0 and point.y >= 0.0 and point.y <= 1.0):
            value.set(0.0)
        if (i == 1 and point.y == 0.0 and point.x >= 1.0 and point.x <= 2.0):
            value.set(0.0)
        if (i == 0 and point.x == 2.0 and point.y >= 0.0 and point.y <= 1.0):
            value.set(0.0)
        if (i == 1 and point.y == 0.0 and point.x >= 2.0 and point.x <= 3.0):
            value.set(0.0)
        if (i == 0 and point.x == 3.0 and point.y >= 0.0 and point.y <= 1.0):
            value.set(0.0)
        if (i == 1 and point.y == 2.0):
            value.set(0.0)


class InitialPressure(Function):
    def eval(self, point, i):
        if point.x >= 0.0 and point.y >= 0.0:
            return cos(pow((pow((point.x-1.5),2)+pow((point.y -0.5),2)),0.5)*2*pi/1.376)*
            exp(-200*pow((point.x-1.5),2)-200*pow((point.y-0.5),2))


class InitialVelocity(Function):
    def eval(self, point, i):

return 0.0

class AtmosphericVelocity(Function):
    def eval(self, point, i):
        if i == 0:
            return 10.0
        else:
            return 0.0

class Artificialviscosity(Function):
    def eval(self, point, i):
        if point.x < 0.5 and point.y >= 1.0:
            return 1000*pow(((0.5-point.x)/0.5),4)
        if point.x > 3.5 and point.y >= 0.0:
            return 1000*pow(((point.x-3.5)/0.5),4)
        if point.y > 1.5:
            return 1000*pow(((point.y-1.5)/0.5),4)
        else:
            return 0.0

artvisold = Artificialviscosity()

mesh=Mesh("byggnaderfinal2.xml.gz")
bcu = SimpleBCu()
u0 = InitialVelocity()  # u at t = 1/2 dt
p0 = InitialPressure()  # p at t = 0 dt

#-----------------------------------------------
# importing backgroundflow values

# tmpvariables in order to make the trialelement
nsd=mesh.numSpaceDim()
x=Vector(nsd*mesh.numVertices())
d=Vector(mesh.numCells())
uc=Function(x,mesh)
delta1=Function(d)
delta2=Function(d)
k=1
ny=1

# Import form
mom_forms=import_formfile("NSEMomentum2D.form")

# Get the biliearform
am=mom_forms.NSEMomentum2DBilinearForm(uc,delta1,delta2,k,ny)
# Get the trialelement
trialelement=am.trial()

# Import the dof
ufile = File("velocity.xml",File.xml)
ufile >> x

# Make the velocity function
ua = Function(x,mesh,trial)

savefile = File("backgroundflow.m")
savefile << ua

# startvalues

dth = 0.00002/2 # time step in Leap-Frog algorithm (= dt/2)
t = 0 # initial time
dt = 0.00002 # time step
c = 344 # speed of sound
tmax = 0.008 # maximum time
rhoa = 1.2 # atmospheric density
counter = 0 # step counter

# calculate Bilinear forms

eulercontinuity_forms = import_formfile("Eulercontinuity.form")
eulermomentum_forms = import_formfile("Eulermomentum.form")
ap = eulercontinuity_forms.EulercontinuityBilinearForm(ua*10,alfap,dt)
au = eulermomentum_forms.EulermomentumBilinearForm(ua*10,artvisold,dt)

# calculate forms for the projection of the initial conditions on the mesh

eulerinp_forms = import_formfile("Eulerinp.form")
eulerinartvis_forms = import_formfile("Eulerinartvis.form")
eulerinu_forms = import_formfile("Eulerinu.form")

# projection of known values on the mesh

Linp = eulerinp_forms.EulerinLinearForm(p0)
Linartvis = eulerinartvis_forms.EulerinartvisLinearForm(artvisold)
Linu = eulerinu_forms.EulerinuLinearForm(u0)
ainp = eulerinp_forms.EulerinpBilinearForm()
ainartvis = eulerinartvis_forms.EulerinartvisBilinearForm()
ainu = eulerinu_forms.EulerinuBilinearForm()

trialelementp = eulercontinuity_forms.EulercontinuityBilinearFormTrialElement()
trialelementartvis = eulercontinuity_forms.EulercontinuityBilinearFormTrialElement()
trialelementu = eulermomentum_forms.EulermomentumBilinearFormTrialElement()

# initialize matrices and vectors

Ap = Matrix()  # mass matrix p
bp = Vector()  # load vector p
Au = Matrix()  # mass matrix u
bu = Vector()  # load vector u
pold = Vector()  
mp = Vector()  
uold = Vector()  
martvis = Vector()  
mu = Vector()  

FEM_assemble(ainp, Mp, mesh)
FEM_assemble(ainartvis, Martvis, mesh)
FEM_assemble(ainu, Mu, mesh)
FEM_lump(Mp, mp)
FEM_lump(Martvis, martvis)
FEM_lump(Mu, mu)
FEM_assemble(Linp, pold, mesh)
FEM_assemble(Linartvis, artvistemp, mesh)
FEM_assemble(Linu, uold, mesh)
pnew.copy(pold)
artvistemp2.copy(artvistemp)
artvis = Function(artvistemp2, mesh, trialelementartvis)
unew.copy(uold)
# set output files
matlabp = File("pressure.m")
matlabu = File("velocity.m")

# loop over time steps

ptest = pnew[1]
pold = Function(pnew, mesh, trialelementp)
uold = Function(unew, mesh, trialelementu)
matlabp << pold
matlabu << uold

while(t < tmax):
    print "t: ", t
    # velocity values updating
    # calculate linear form of continuity equation
    Lu = eulermomentum_forms.EulermomentumLinearForm(uold,pold,ua,dt,rhoa)
    FEM_assemble(au, Au, mesh)
    FEM_assemble(Lu, bu, mesh)
    FEM_applyBC(Au, bu, mesh, trialelementu, bcu)
    linesolver = KrylovSolver()
    linesolver.solve(Au, unew, bu)

    # update t and counter
    t += dth
    counter += 1
    uold = Function(unew,mesh,trialelementu)

    # pressure values updating
    # calculate linear form of momentum equation
    Lp = eulercontinuity_forms.EulercontinuityLinearForm(pold,uold,ua*10,alfap,dt,c,rhoa)
FEM_assemble(ap, Ap, mesh)
FEM_assemble(Lp, bp, mesh)

linearsolver = KrylovSolver()
linearsolver.solve(Ap, pnew, bp)

# update t and counter
  t += dth
  counter += 1

  p = Function(pnew, mesh, trialelementp)
  matlabp << p

  pold = Function(pnew, mesh, trialelementp)

  print "t: ", t