- Code development 26/09/21
- Incompressible, cylindrical, axisymmetric jet, resolvent analysis

- Conclusion:

-resolvent analysis and modal analysis results qualitatively in good agreement with Garnaud’s reference results.

Base flow (obtained by Newton)

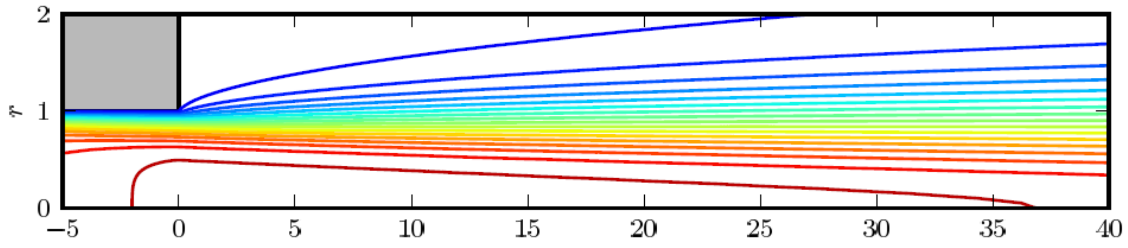
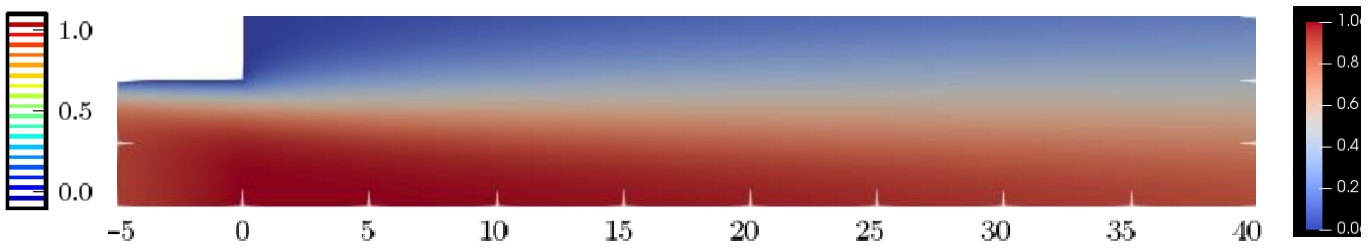
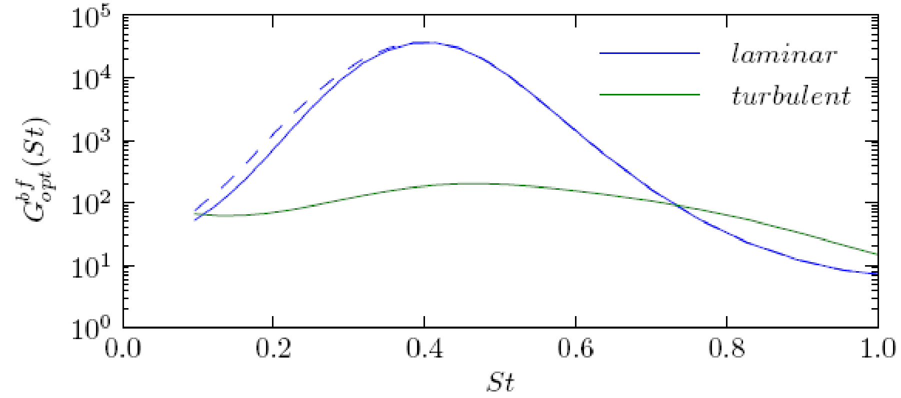


Figure 4.2 in Garnaud Ph. D thesis



new code result

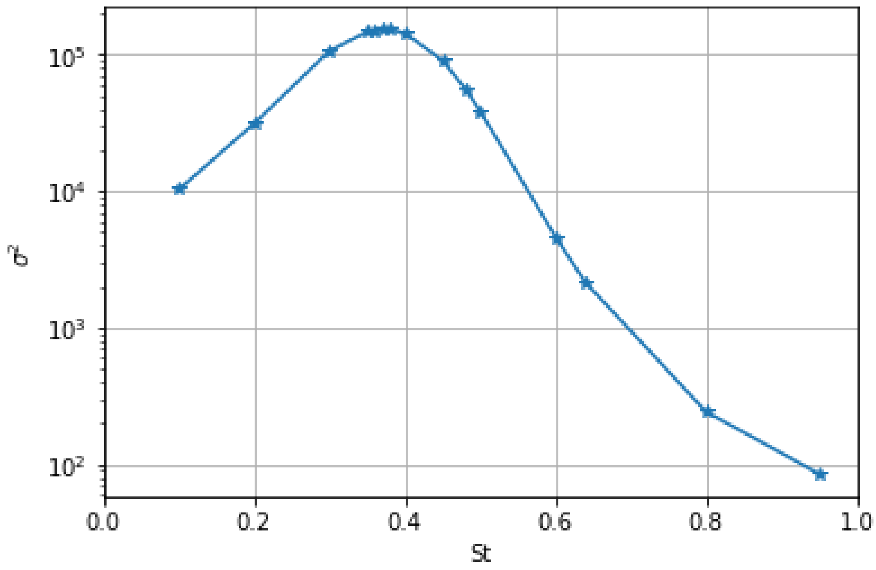
Resolvent gain curve



We work with laminar x_infini=40

Figure 6.6: Optimal amplification of periodic body forcing for the laminar and turbulent mean flows. The dashed lime displays the results obtained for domain of length $x_\infty = 60$ ($x_\infty = 40$ is used otherwise).

Figure 6.6 in Garnaud Ph. D thesis



Note: no restriction of forcing domain inside the nozzle, as Garnaud did in his calculation. This can lead to quantitative differences.

Resolvent modes

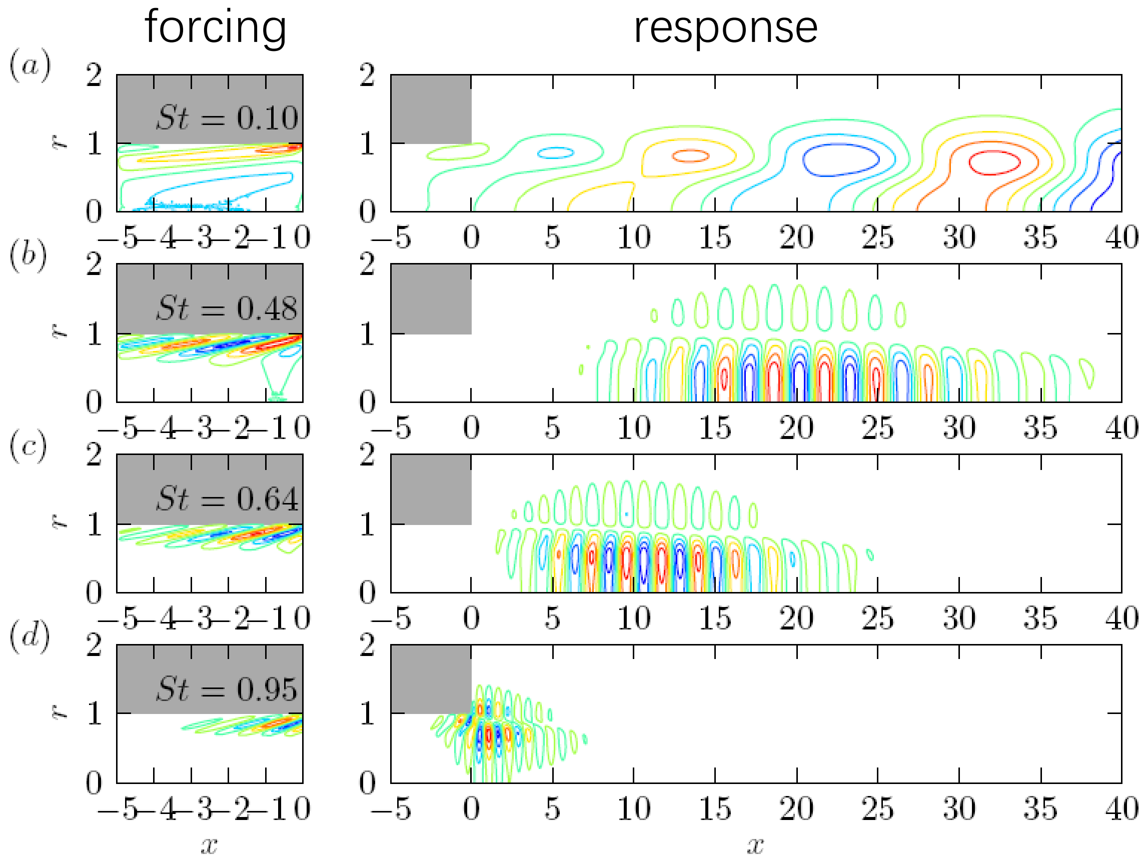
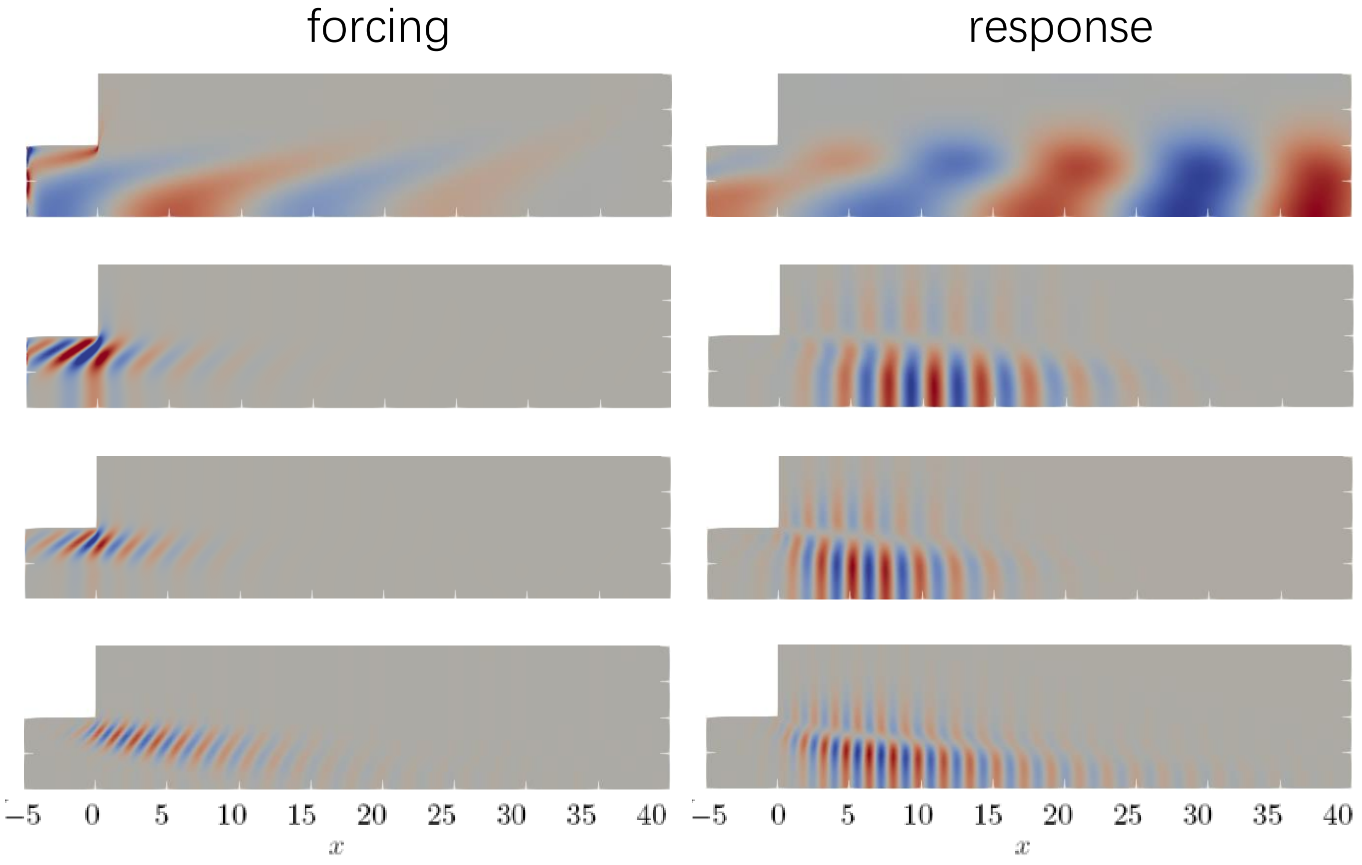


Figure 6.7 in Garnaud Ph. D. thesis



new code result

Note: no restriction of forcing domain inside the nozzle, as Garnaud did in his calculation. This can lead to quantitative differences.

Modal analysis --- Eigenvalues

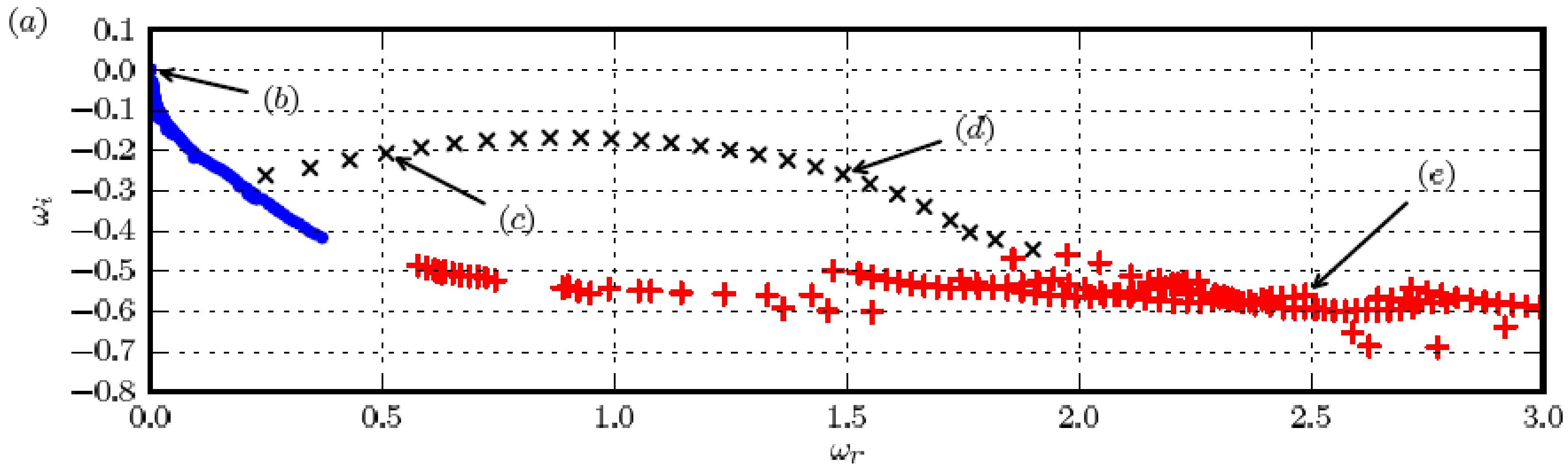
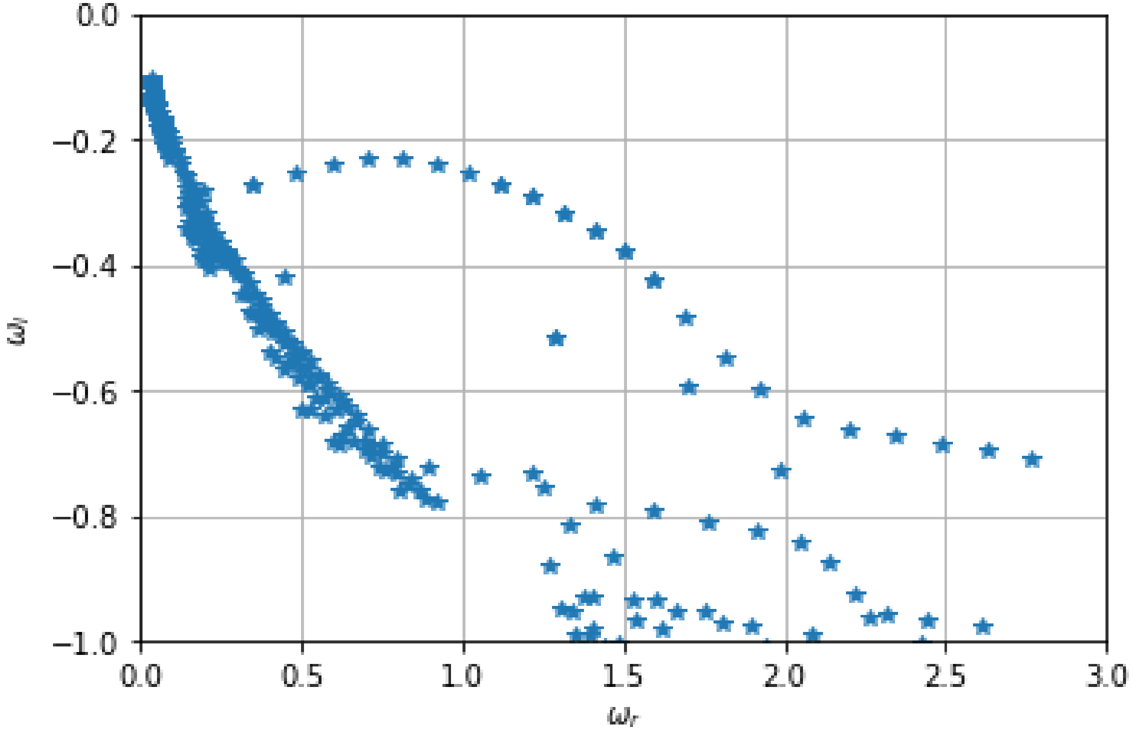


Figure 7 in Garnaud POF 2013



New code results

Code review (brief)

Anything with label “lowMach” or “lowMach_reacting” has not been implemented. Only “incompressible” is implemented.

Code structure

-Mesh

-Matlab: calculate eigenvalues from Matlab for modal analysis if required

-jet_incompressible: result path folder

-command_yaj.py: command file

-yaj.py: main script

command_yaj.py

- Newton
- Resolvent
- Eigenvalues (modal anaysis)

```
import_flag=1 #1: import base flow from file #0: not import
flow_mode='incompressible' #currently only incompressible is implemented.
yo=yaj(MeshPath,flow_mode,datapath,import_flag)
```

We need to import base flow not only when doing resolvent analysis and modal analysis, but also when doing Newton iteration from a previous calculated flow field.
If put it to 0, Newton iteration starts from the flow field given by self.InitialConditions().

self.OperatorNonlinear()

- rhs of nonlinear governing equations
- A good reference for nonlinear governing equations is https://demichie.github.io/NS_cylindrical/
- Can's code defines the differential operators in cylinder coordinates for axisymmetric case. The second-order stress tensor in grad_cyl is reduced from (19) in ref.

```
############################################################
# Can's code for cylindrical coordinates
# Define operator for axi-symmetric polar coordinate system
# in x,r,th it comes [Sxx  Sxr  Sxth ]
#                    [Srx  Srr  Srth ]
#                    [Sthx Sthr Sthth]

# Gradient with x[0] is x and x[1] is r
def grad_cyl(v):
    return as_tensor([[v[0].dx(0), v[0].dx(1), 0],
                      [v[1].dx(0), v[1].dx(1), 0],
                      [0, 0, v[1]/self.r]])

def div_cyl(v):
    c=(1./self.r)*(self.r*v[1]).dx(1) + v[0].dx(0)
    return c
```

$$\nabla \mathbf{u} = \begin{bmatrix} \frac{\partial u_r}{\partial r} & \frac{1}{r}\frac{\partial u_r}{\partial \theta} - \frac{u_\theta}{r} & \frac{\partial u_r}{\partial z} \\ \frac{\partial u_\theta}{\partial r} & \frac{1}{r}\frac{\partial u_\theta}{\partial \theta} + \frac{u_r}{r} & \frac{\partial u_\theta}{\partial z} \\ \frac{\partial u_z}{\partial r} & \frac{1}{r}\frac{\partial u_z}{\partial \theta} & \frac{\partial u_z}{\partial z} \end{bmatrix} \tag{19}$$

self.OperatorNonlinear()

- The axisymmetric equations read in (24)-(26) in ref. (\rho and g are not considered.)
- Integration by parts is used for diffusion and pressure term, with neglecting boundary parts.

```
#mass
F = div_cyl(u)*self.Test[1]*self.r*dx
#momentum
F -= inner(grad(u)*u, self.Test[0])*self.r*dx
F -= self.mu*inner(grad_cyl(u), grad_cyl(self.Test[0]))*self.r*dx
F -= -inner(p, div_cyl(self.Test[0]))*self.r*dx
return F
```

$$\frac{\partial u_r}{\partial t}+u_r\frac{\partial u_r}{\partial r}+u_z\frac{\partial u_r}{\partial z}=-\frac{1}{\rho}\frac{\partial p}{\partial r}+\frac{\mu}{\rho}\left\{-\frac{u_r}{r^2}+\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u_r}{\partial r}\right)+\frac{\partial^2 u_r}{\partial z^2}\right\}+g_r, \tag{24}$$

$$\frac{\partial u_z}{\partial t}+u_r\frac{\partial u_z}{\partial r}+u_z\frac{\partial u_z}{\partial z}=-\frac{1}{\rho}\frac{\partial p}{\partial z}+\frac{\mu}{\rho}\left\{\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u_z}{\partial r}\right)+\frac{\partial^2 u_z}{\partial z^2}\right\}+g_z, \tag{25}$$

while the continuity equation is

$$\frac{1}{r}\frac{\partial}{\partial r}(ru_r)+\frac{\partial u_z}{\partial z}=0. \tag{26}$$

- When doing non-axisymmetric, one may simply consider using the full second-order tensor in (19).

self.Newton()

```
self.mu=self.mu*self.play #update viscosity
```

```
self.play-=self.minus_play #increment of viscosity
```

```
if self.label=='incompressible':
    #write results in private_path for a given mu
    u_r,p_r = self.ru.split()
    ll="mu"+str(np.round(self.play, decimals=2))
    File(self.dnspath+self.private_path+"u"+f"{ifile:03d}"+l
    File(self.dnspath+self.private_path+"baseflow"+f"{ifile:
    print(self.dnspath+self.private_path+"baseflow"+f"{ifile
elif self.label=='lowMach':
    pass

#write result of current mu
File(self.dnspath+"u"+".pvd") << u_r
File(self.dnspath+"baseflow000"+".xml") << self.ru.vector()
print(self.dnspath+"baseflow000"+".xml written!")
```

- When doing Newton iterations, we sometimes need to start with a larger viscosity to get an initial field, and than gradually decrease the viscosity. Thus we add a prefactor self.play multiplied in front of self.mu. For example, when doing lowMach for cooled cylinder, we may need to set self.play=10 at the beginning, and then gradually decrease self.play to 1.
- For this reason, the results issued from each iteration are saved twice: a folder self.private_path='doing/' records all the history files; the current results are saved in self.datapath, and they are covered each time when a new iteration is done.
- However, in the present case, we can directly set self.play=1, and Newton can converge.

Boundary conditions

Get all free indices without Dirichlet condition

```
def get_indices(self):
```

- Newton

```
def BoundaryConditions(self):
```

```
        # define boundary conditions for Newton/timestpper
        if self.label=='incompressible':
            ux_tanh=Expression('tanh(5*(1-x[1]))', degree=2)
            bcs_inflow_x = DirichletBC(self.Space.sub(0).sub(0), ux_tanh, inlet)
            bcs_inflow_r = DirichletBC(self.Space.sub(0).sub(1), 0, inlet)
            bcs_wall=DirichletBC(self.Space.sub(0), (0,0), wall)
            bcs_symmetry=DirichletBC(self.Space.sub(0).sub(1), 0, symmetry)
            return [bcs_inflow_x,bcs_inflow_r,bcs_wall,bcs_symmetry]
```

- Resolvent (open ux at inlet)

```
def BoundaryConditionsPerturbations(self):
```

```
        if self.label=='incompressible':
            bcs_inflow = DirichletBC(self.Space.sub(0).sub(1), 0, inlet)
            bcs_wall=DirichletBC(self.Space.sub(0), (0,0), wall)
            bcs_symmetry=DirichletBC(self.Space.sub(0).sub(1), 0, symmetry)
            return [bcs_inflow,bcs_wall,bcs_symmetry]
```

- Modal (closed ux at inlet)

```
def BoundaryConditionsPerturbations(self):
```

```
        if self.label=='incompressible':
            bcs_inflow = DirichletBC(self.Space.sub(0), (0,0), inlet)
            bcs_wall=DirichletBC(self.Space.sub(0), (0,0), wall)
            bcs_symmetry=DirichletBC(self.Space.sub(0).sub(1), 0, symmetry)
            return [bcs_inflow,bcs_wall,bcs_symmetry]
```

self.Resolvent()

- Implementation of $P^H Q^H R^H M_r RQPf = \sigma^2 I^H M_f If$
- The matrix P, Q are real, so the Hermitian is the same as the Transpose. Only R is complex. LU decomposition is used for R.
- Detailed comments are given in the code.
- Some details with different notations can be found in document by Lutz: resolvent_doc.pdf.
- The quadrature matrix is required to compensate the quadrature in resolvent operator R, because R=(A-i*omegaB)^(-1), and there is quadrature in A and B.
- The matrix I reshapes forcing matrix Mf (m*m) to I^T*Mf*I (n*n). Normally the matrix I is the same as P. But this is not the case if we use values not equal to 0 or 1 in P to damp (but not entirely eliminate) the forcing in some regions.
- The output functionalities only work for setting nb of resolvent modes as 1, presently.

self.Eigenvalues()

- We find in some cases, shift-invert method works better in Matlab then Python especially for reacting flow, and we have not exactly understood why.
- For this reason, three operational mode by setting flag_mode

0: save matrix as .mat with file name "savematt"; 1: load result matrix from .mat with file name "loadmatt"; 2: calculate eigenvalues in python

- One could use mode 0 first and then use get_eigs.m in folder Matlab/. Then use mode 1 to load and output the eigenmodes.