Supporting Information

Reactive Nanomessengers for Artificial Chemical Communication

*Luca Fichera, Giovanni Li-Destri, Roberta Ruffino, Grazia Maria Lucia Messina, Nunzio Tuccitto**

Jupyter notebook related to the typical code to simulate artificial molecular communication

Import libraries

`FiPy` is an object oriented, partial differential equation (PDE) solver, written in `Python`, based on a standard finite volume (FV) approach. The framework has been developed in the Materials Science and Engineering Division (MSED_) and Center for Theoretical and Computational Materials Science (CTCMS_), in the Material Measurement Laboratory (MML_) at the National Institute of Standards and Technology (NIST_).
`Matplotlib` is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.
`NumPy` is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

In [ ]:

```
from fipy import *
import matplotlib.pyplot as plt
import numpy as np
```

I define the simulation mesh in cm
“nx” is the number of steps in mesh along x
“ny” is the number of steps in mesh along y
“dx” is the step length in mesh along x
“dy” is the step length in mesh along y

In [ ]:

```
nx = 10
ny = 10
dx = 1.
dy = 0.1
```

Input physical parameters
"F" is applied steady pressure gradient dp/dx
“rho” is the fluid density
“nu” absolute viscosity
“ry” is the array representing the distance “r” from the center of the tube along y direction

In [ ]:

```
F = 1.
rho = 1 #g/cm3
nu = 8.9E-4 # Pa*s == g/cms
ry = np.linspace(-(ny*dy)/2,(ny*dy)/2,10)
```

the mesh is periodically mirrored 3 times

In [ ]:

```
mesh = Grid2D(nx=nx*3, dx=dx, ny=ny*3, dy=dy)
x = mesh.x
y = mesh.y

xl = np.arange(0, nx*dx*3, dx)
yl = np.arange(0, ny*dy*3, dy)
X, Y = np.meshgrid(xl, yl)
```

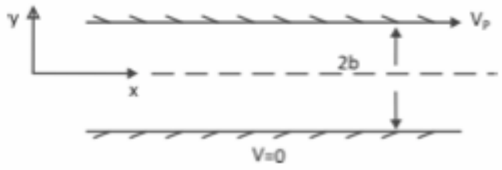
ininzialize the velocity field with 0 cm/s

In [ ]:

```
u = np.zeros((3*nx,3*ny)) #initialize u field
v = np.zeros((3*nx,3*ny)) #initialize v field
```

Calculate the profile of V(u,v) by solving the Navier-Stokes in a channel

$$\rho(\frac{\partial V_x}{\partial t} + V_x\frac{\partial V_x}{\partial x} + V_y\frac{\partial V_x}{\partial t}) = -\frac{\partial p}{\partial x} + \rho g_x + \mu(\frac{\partial^2 V_x}{\partial x^2} + \frac{\partial^2 V_y}{\partial y^2})$$



Boudary conditions:

$y = \pm b, V_x = 0$ (no splip condition)

$$\mu\frac{\partial^2 V_x}{\partial y^2} = \frac{\Delta P}{L}$$

$$V_{max} = (\frac{\Delta P}{L})(\frac{b^2}{2\mu})$$

$$V_x(u) = V_{max} * (1 - \frac{y^2}{b^2})$$

In [ ]:

```
def Poiseuille_flows(F,u):
    Vmax = (F*(ny*dy/2)**2)/(nx*dx*2*nu)
    Vy = Vmax*(1-(ry**2/(ny*dy/2)**2))

    for i in range(nx*3):
        u[ny:ny*2,i] = Vy
    return u
```

Now I initialize the concentration in the simulation meshes as 1E10-14

phi1_2D is the concentration of the messenger
phi2_2d is the concentration of the quencher
phiR_2D is the concentration of the quenched messenger

In [ ]:

```
phi1_2D = np.zeros_like(X)+(10**(-14))
phi2_2D = np.zeros_like(X)+(10**(-14))
phiR_2D = np.zeros_like(X)+(10**(-14))
xS = 15
yS = 3
xR = 15
yR = 5
u = Poiseuille_flows(F,u)
```

FiPy library works only for 1D variable so that I reshape phi1 phiR and phi2

In [ ]:

```
phi_value1 = phi1_2D.reshape(nx*3*ny*3)
phi1 = CellVariable(name="messenger", mesh=mesh, value=phi_value1)
phi_valueR = phiR_2D.reshape(nx*3*ny*3)
phiR = CellVariable(name="quenched messenger", mesh=mesh, value=phi_valueR)
phi_value2 = phi2_2D.reshape(nx*3*ny*3)
phi2 = CellVariable(name="quencher", mesh=mesh, value=phi_value2)
```

Differential equation to be solved

$$\partial C/\partial t = \nabla \cdot \left(D\nabla\phi - \vec{V}\phi \right) + R$$

FiPy solves field variables on the cell centers. Transient and source terms describe the change in the value of a field at the cell center, and so they take a CellVariable coefficient. Diffusion and convection terms involve fluxes between cell centers, and are calculated on the face between two cells, and so they take a FaceVariable coefficient.

In [ ]:

```
V = CellVariable(name= "speed", mesh=mesh, value = 0. ,rank = 1)
```

In [ ]:

```
D = np.array(((.1, 0.),(0., 0.1)))
```

In [ ]:

```
eqX = TransientTerm() == DiffusionTerm(coeff=D) - VanLeerConvectionTerm(coeff=V)
```

In [ ]:

```
#boundaries values
phi1_2D[:,1] = phi1_2D[:,-1]
phiR_2D[:,1] = phiR_2D[:,-1]
phi2_2D[:,1] = phi2_2D[:,-1]


# refleting walls
def reflect(phi2D):
    wall_bottom = phi2D[0:ny,:]
    phi2D[ny:ny*2,:] = phi2D[ny:ny*2,:] + np.flipud(wall_bottom)
    wall_top = phi2D[ny*2:ny*3,:]
    phi2D[ny:ny*2,:] = phi2D[ny:ny*2,:] + np.flipud(wall_top)
    phi2D[0:ny,:] = 1E-14
    phi2D[ny*2:ny*3,:] = 1E-14
    return phi2D
```

I define the time step duration of the numerical solution, the number of numerical time steps to solve and calculate time variable (named "tempo")

In [ ]:

```
timeStepDuration = 0.1
steps = 1200
tempo = np.arange(0,(steps*timeStepDuration+timeStepDuration),0.1)
```

I generate the array of coefficients that multiply the speed to make it variable

In [ ]:

```
F = np.ones(steps)
F[100:200] = 1.3
F[200:300] = 1.8
F[300:600] = 0.8

from scipy.signal import savgol_filter
F = savgol_filter(F, 701, 3) # window size 701, polynomial order 3
F =F/5
plt.plot(F)
plt.ylabel('F')
plt.xlabel('time steps')
plt.show()
```

I generate the carrier array

In [ ]:

```
Source = np.zeros(steps)
Source[0] = 1.
Source[200] = 1.
Source[400] = 1.
Source[600] = 1.
Source[800] = 1.
```

I generate the array of the reagent

In [ ]:

```
Reactant = np.zeros(steps)
Reactant[0:20] = 1.
Reactant[200:220] = 1.
Reactant[800:820] = 1.
```

Reaction

$$\phi 1 + R \xrightarrow{K_1} \phi 2$$

$$\partial[\phi 2]/\partial t = K_1 * [\phi 1] * [R]$$

In [ ]:

```
K1=1E-3
```

I initialize the array of the detected signal by placing the detector at -3 from the right edge of the simulation adding the values of all the rows of the column -3

In [ ]:

```
detector_phi1 = [np.sum(phi1_2D[:,-3])]
detector_phi2 = [np.sum(phi2_2D[:,-3])]
detector_phiR = [np.sum(phiR_2D[:,-3])]
```

Now I define the functions to solve the differential equations.

In [ ]:

```
def solve1():
    global phi1_2D
    phi1_value = phi1_2D.reshape(nx*3*ny*3)
    phi1.setValue(phi1_value)
    eqX.solve(var=phi1, dt=timeStepDuration, solver = DefaultAsymmetricSolver())
    phi1_2D = np.asarray(phi1).reshape((int(nx*3),int(ny*3)))
    phi1_2D = reflect(phi1_2D)
    phi1_2D[:,-2:] = 1E-14
    phi1_2D[:,1] = phi1_2D[:,-1]
```

In [ ]:

```
def solve2():
    global phi2_2D
    phi2_value = phi2_2D.reshape(nx*3*ny*3)
    phi2.setValue(phi2_value)
    eqX.solve(var=phi2, dt=timeStepDuration, solver = DefaultAsymmetricSolver())
    phi2_2D = np.asarray(phi2).reshape((int(nx*3),int(ny*3)))
    phi2_2D = reflect(phi2_2D)
    phi2_2D[:,-2:] = 1E-14
    phi2_2D[:,1] = phi2_2D[:,-1]
```

In [ ]:

```
def solveR():
    global phiR_2D
    phiR_value = phiR_2D.reshape(nx*3*ny*3)
    phiR.setValue(phiR_value)
    eqX.solve(var=phiR, dt=timeStepDuration, solver = DefaultAsymmetricSolver())
    phiR_2D = np.asarray(phiR).reshape((int(nx*3),int(ny*3)))
    phiR_2D = reflect(phiR_2D)
    phiR_2D[:,-2:] = 1E-14
    phiR_2D[:,1] = phiR_2D[:,-1]
```

numerical solution cycles

I use multithreading in order to speed up the simulations

In [ ]:

```
from threading import Thread
```

In [ ]:

```
for step in range(steps):

    u = Poiseuille_flows(F[step],u)
    u_1D = u.reshape(nx*3*ny*3)
    v_1D = v.reshape(nx*3*ny*3)
    velocity_value_new = np.vstack((u_1D,v_1D))
    V.setValue(velocity_value_new)
    #source
    phi1_2D[xS,yS] = phi1_2D[xS,yS] + Source[step]
    phiR_2D[xR,yR] = phiR_2D[xR,yR] + Reactant[step]


    #reaction
    phi1r_2D = phi1_2D * phiR_2D * K1
    phi1_2D = phi1_2D - phi1r_2D
    phiR_2D = phiR_2D - phi1r_2D
    phi2_2D = phi2_2D + phi1r_2D

    #solve
    t1 = Thread(target=solve1, args=())
    t2 = Thread(target=solveR, args=())
    t3 = Thread(target=solve2, args=())

    t1.start()
    t2.start()
    t3.start()

    t1.join()
    t2.join()
    t3.join()


    #detector
    detector_phi1 = np.append(detector_phi1,np.sum(phi1_2D[:,-4])) #signal phi1
    detector_phiR = np.append(detector_phiR,np.sum(phiR_2D[:,-4])) #signal phiR
    detector_phi2 = np.append(detector_phi2,np.sum(phi2_2D[:,-4])) #signal phi2
```

Now I save results

In [ ]:

```
detected_1 = np.vstack((tempo,detector_phi1))
detected_2 = np.vstack((tempo,detector_phi2))
detected_R = np.vstack((tempo,detector_phiR))
np.savetxt('Detector_1.txt', detected_1.T, delimiter=',')
np.savetxt('Detector_2.txt', detected_2.T, delimiter=',')
np.savetxt('Detector_R.txt', detected_R.T, delimiter=',')
```

Now I load UV-vis Spectrum of the molecular messenger

In [ ]:

```
UVvis = np.loadtxt('UVvis.csv', delimiter=',')
```

I calculate the total mocular messenger concentration (unreacted + quenched) and then I calculate the intensity od absorbance according to the UVvis spectrum

In [ ]:

```
conc_tot = (detector_phi1+detector_phi2)
ABS = conc_tot * UVvis[idx_max,1]
```

Reaction induces fluorescence quencing

The Stern-Volmer equation describes the kinetics of the phenomenon:

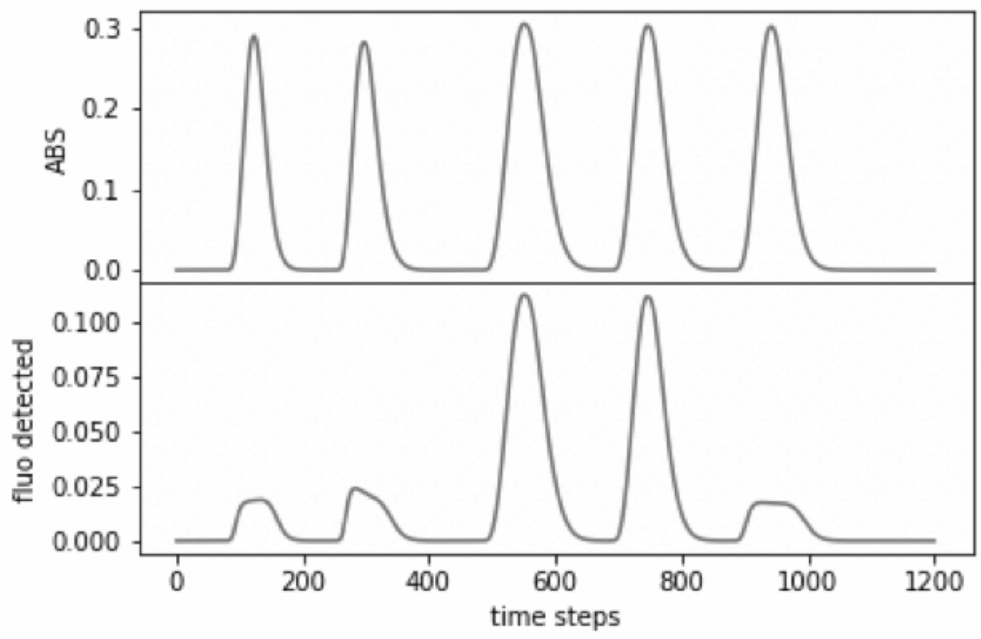
$$\frac{I_0}{I} = 1 + K_q\tau * [Q]$$

In [ ]:

```
R = 8.314472E-3 #L Pa K-1 mol-1
T = 300 #K
kq = 8*R*T/(3*nu) #L s-1 mol-1
tau = 0.01 #s

fluo = conc_tot*fluo_phi1[np.argmax(fluo_phi1[:,1]),1]/(1+kq*tau*detector_phi2)

f, (ax1, ax2) = plt.subplots(2, 1, sharex=True)
ax1.set_ylabel('ABS')
ax2.set_ylabel('fluo detected')
ax2.set_xlabel('time steps')
ax1.plot(ABS)
ax2.plot(fluo)
f.subplots_adjust(hspace=0)
plt.show()
```



In [ ]:

```

```