```
from __future__ import print_function
from dolfin import *
from dolfin_adjoint import *
import numpy as np
import os, sys
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from matplotlib import colors

# Set log level
set_log_level(ERROR)

def make_artificial_obs(cd,xc,yc,R):
    '''
        Define the initial and target cellularity for testing purpose.
        Both are constant inside the tumor area.
        xc and yc are the coordinate of the center of the target tumor area
        R is the radius of the target tumor area.
        The initial tumor is defined by the MeshFunction cd.
    '''
    initial_p = Function(V0,annotation=False)
    target_p = Function(V0,annotation=False)
    initial_p.rename('initial','initial tumor fraction')
    target_p.rename('target','target tumor fraction')
    p_values_i = [0,0.4]
    p_values_f = [0,0.9]

    for cell in cells(mesh):
        cell_no = V0.dofmap().cell_dofs(cell.index())
        # center of the cell
        xc_cell = np.average(cell.get_coordinate_dofs()[0::2])
        yc_cell = np.average(cell.get_coordinate_dofs()[1::2])

        # cd is a mesh function defined by the physical region xml file
        initial_p.vector()[cell_no] = p_values_i[int(cd.array()[cell_no]-1)]
        if(pow(xc_cell-xc,2.)+pow(yc_cell-yc,2.) < R*R):
            target_p.vector()[cell_no] = p_values_f[1]
        else:
            target_p.vector()[cell_no] = p_values_f[0]

    return [initial_p,target_p]

def vis_obs(initial_p,target_p,label=0):
    '''
        Visualize the initial and target cellularity.
        When label >=0, the two distributions are saved to the global result file file_results
        with label=label.
    '''
    target_p = interpolate(target_p,V0)
    cm1 = colors.ListedColormap([[0,1,0,1],[0,0,0,1],[1,0,0,1]])
    cm2 = cm.get_cmap('jet')
    plt.figure()
    plt.subplot(1,2,1)
    plt.title('initial')
    plot(initial_p,cmap=cm2)
    plt.gca().invert_yaxis()

    diff = Function(V0,annotation=False)
    diff.rename('diff','diff tumor fraction')
    diff.vector()[:] = initial_p.vector()-target_p.vector()

    plt.subplot(1,2,2)
    plt.title('target')
    plot(target_p,cmap=cm2)
    plt.gca().invert_yaxis()
```

```
    plt.show()

    if(label >= 0):
        file_results.write(initial_p, label)
        file_results.write(target_p, label)
        file_results.write(diff, label)

def forward(initial_p,annotate=False,record=False,record_initial=False,record_target=False):
    """
        Here, we define the forward problem.
        See the attached pptx file for more information on the variables.
        Don't delete the necessary adjoint-dolfin argument: annotate.
    """
    def boundary(x, on_boundary):
        return on_boundary

    bc = DirichletBC(V, Constant(0.), boundary)

    # field of growth rate, treated as a local parameter here
    k.rename('k','k field')
    # current distribution of cell density (phi_T)
    p_n = Function(V, name = 'pn', annotate = annotate)
    p_n = interpolate(initial_p,V)
    p_n.rename('phi_T','tumor fraction')
    p = TrialFunction(V)
    q = TestFunction(V)
    # logistic source term
    S = p_n*(1-p_n/Theta)*k

    F = p*q*dx + D*dt*dot(grad(p), grad(q))*dx - (p_n + dt*S)*q*dx
    a, L = lhs(F), rhs(F)

    # Prepare the solution
    p = Function(V, name = 'p', annotate = annotate)

    t = 0.
    if record:
        # save the current solution, k field, initial and target solution
        file_results.write(p_n,t)
        file_results.write(k,t)
        if record_initial:
            file_results.write(initial_p,t)
        if record_target:
            file_results.write(target_p,t)

    for n in range(num_steps):
        # Update current time
        t += dt
        # Compute solution
        solve(a == L, p, annotate = annotate) # with trivial Neumann BC
        # Update previous solution
        p_n.assign(p, annotate = annotate)

        if record:
            file_results.write(p_n,t)
            file_results.write(k,t)
            if record_initial:
                file_results.write(initial_p,t)
            if record_target:
                file_results.write(target_p,t)
    return p

# Callback function for the optimizer
# Writes intermediate results to a logfile
def eval_cb(j, m):
```

```
    """ The callback function keeping a log """

    print("objective = %15.10e " % j)

def objective(p, target_p, r_coeff1, r_coeff2):
    return assemble(inner(p-target_p, p-target_p)*dx) + r_coeff1*assemble(k*k*dx) +
    r_coeff2*assemble(dot(grad(k),grad(k))*dx)

def optimize(dbg=False):
    """ The optimization routine """

    # Define the control
    m = [Control(k)]

    # Execute first time to annotate and record the tape
    p = forward(initial_p,True)

    J = objective(p, target_p, r_coeff1, r_coeff2)

    # Prepare the reduced functional
    rf = ReducedFunctional(J,m,eval_cb_post=eval_cb)
    # upper and lower bound for the parameter field
    k_lb = Function(V, annotate = False)
    k_ub = Function(V, annotate = False)
    k_lb.vector()[:] = 0
    k_ub.vector()[:] = 4

    # Run the optimization
    #m_opt = minimize(rf,method='L-BFGS-B', bounds = [k_lb,k_ub],
    tol=1.0e-6,options={"disp":True,"gtol":1.0e-6})

    m_opt = minimize(rf,method='L-BFGS-B', tol=1.0e-6,options={"disp":True,"gtol":1.0e-6})
    return m_opt

if __name__ == "__main__":
    # call the function with
    # python <this file> case r_coeff1 r_coeff2
    print("Syntax: python <this file> case r_coeff1 r_coeff2")
    if(len(sys.argv) <= 1):
        case = 0
    else:
        case = sys.argv[1]
    if(len(sys.argv) <= 2):
        r_coeff1 = 0.1
    else:
        r_coeff1 = float(sys.argv[2])
    if(len(sys.argv) <= 3):
        r_coeff2 = 0
    else:
        r_coeff2 = float(sys.argv[3])
    base_dir = './Output/validation'

    output_dir = os.path.join(base_dir,'run'+str(case))
    print('Regularization coefficient (order 0) = '+str(r_coeff1))
    print('Regularization coefficient (order 1) = '+str(r_coeff2))
    print('Output saved to '+output_dir)

    # Prepare a mesh
    mesh = Mesh("./gmsh_test.xml")
    cd = MeshFunction('size_t',mesh,"./gmsh_test_physical_region.xml")
    fd = MeshFunction('size_t',mesh,"./gmsh_test_facet_region.xml")
    V0 = FunctionSpace(mesh, 'DG', 0)
    V = FunctionSpace(mesh, 'CG', 1)

    # Model parameters
```

```
T = 10.0              # final time
num_steps = 100       # number of time steps
dt = T / num_steps    # time step size
D = Constant(10.)      # mobility or diffusion coefficient
k_true = Function(V)    # true growth rate
k_true.vector()[:] = 0.1 # initial distribution of the k field
k = Function(V)    # growth rate
Theta = Constant(1.0) # carrying capacity

# Prepare output file
file_results = XDMFFile(os.path.join(output_dir,'optimal.xdmf'))
file_results.parameters["flush_output"] = True
file_results.parameters["functions_share_mesh"] = True

# Optimization module
[initial_p, target_p] = make_artificial_obs(cd,135,135,50) # generate the initial and
target cellularity
k = k_true
target_p = forward(initial_p,False,False,False,False) # run the forward model given the
true k field
# ... note the previous target_p (generated by make_artificial_obs) is overwritten
target_p.rename('target','target tumor fraction')
vis_obs(initial_p,target_p,0) # visualize the initial and target cellularity

k.vector()[:] = 0.5 # initial guess of the k field
k = optimize() # optimize the k field using the adjoint method provided by adjoint_dolfin
model_p = forward(initial_p,False,True,True,True) # run the forward model using the
optimized k field
print('norm(k_opt) = '+str(norm(k)))
print('J_opt = '+str(objective(model_p, target_p, r_coeff1, r_coeff2)))
print('J_opt (without regularization) = '+str(objective(model_p, target_p, 0., 0.)))
```