M3C 2018 Final Project Solution

Part 1: Parallel neural network (15 pts)

You will develop a parallel neural network simulation code based on *nnmodel* from HW2. *Full* gradient descent (*fgd*) will be used rather than *sgd* to train the model. Recall that the components of ∇C are calculated with sums over d training images, and the *sgd* approach truncated this sum so that one randomly chosen image was used instead of all d images. We now remove this truncation and use the full sum to compute the gradient of the cost function which is then used to update the vector of fitting parameters:

$$\mathbf{f}^{[q+1]} = \mathbf{f}^{[q]} - \alpha \nabla C^{[q]}$$

Complete Fortran routines for using *fgd* with *nnmodel* have been provided in *p1serial.f90*. The main program, *p1main.f90*, 1) reads in training and testing images, 2) reads in model parameters from *data.in* (which must be created), 3) trains the model with *fgd*, and 4) computes the test error. You should complete *p1mpi.f90* and *p1main_mpi.f90* so that the computation of ∇C is parallelized with MPI. The initial MPI setup and distribution of *dtrain* images to *numprocs* processes (each process works with *dlocal* training images) has been provided in the main program.

1. Complete *p1main_mpi.f90* so that a copy of the initial guess for the fitting parameters, *fvec0*, is available on each process prior to the call to *fgd*.

ANS: The initial guess should be broadcast from process 0 to the other processes:

```
    call mpi_bcast(fvec0,m*(n+2)+1,MPI_DOUBLE_PRECISION,0,MPI_COMM_WORLD,ierr)
```

2. Implement a parallelized computation of ∇C in *fgd_mpi* and *nnmodel_mpi*. After receiving output from *nnmodel_mpi*, code should be added to *fgd_mpi* to update *fvec* appropriately each iteration. The *mpi* and serial versions of the codes should produce the same output when using the same model parameters (*e.g.* *dtrain* and *m*).

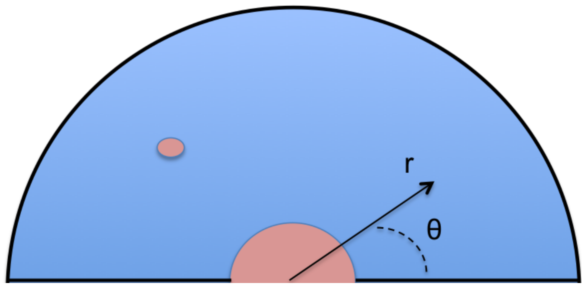
The completed *nnmodel_mpi* and *fgd_mpi* subroutines are below:

```
subroutine nnmodel_mpi(fvec,n,m,dlocal,d,c,cgrad)
  implicit none
  integer, intent(in) :: n,m,dlocal,d !training data and inner layer sizes
  real(kind=8), dimension(m*(n+2)+1), intent(in) :: fvec !fitting parameters
  real(kind=8), intent(out) :: c !cost
  real(kind=8), dimension(m*(n+2)+1), intent(out) :: cgrad !gradient of cost
  integer :: i1,j1,l1
  real(kind=8), dimension(m,n) :: w_inner
  real(kind=8), dimension(m) :: b_inner,w_outer
  real(kind=8) :: dinv,b_outer
  !Declare other variables as needed
  real(kind=8), dimension(m,dlocal) :: z_inner,a_inner,g_inner
  real(kind=8), dimension(dlocal) :: z_outer,a_outer,e_outer,g_outer,eg_outer
  real(kind=8) :: dcdb_outer
  real(kind=8),dimension(m) :: dcdw_outer
  real(kind=8), dimension(m) :: dcdb_inner
  real(kind=8), dimension(m,n) :: dcdw_inner

  dinv = 1.d0/dble(d)

  !unpack fitting parameters (use if needed)
  do i1=1,n
    j1 = (i1-1)*m+1
    w_inner(:,i1) = fvec(j1:j1+m-1) !inner layer weight matrix
  end do
  b_inner = fvec(n*m+1:n*m+m) !inner layer bias vector
  w_outer = fvec(n*m+m+1:n*m+2*m) !output layer weight vector
  b_outer  = fvec(n*m+2*m+1) !output layer bias

  !Add code to compute c and cgrad

  !Compute inner layer activation vector, a_inner
  z_inner = matmul(w_inner,nm_x)
```

```
    do i1=1,dlocal
      z_inner(:,i1) = z_inner(:,i1) + b_inner
    end do
    a_inner = 1.d0/(1.d0 + exp(-z_inner))

    !Compute outer layer activation (a_outer) and cost
    z_outer = matmul(w_outer,a_inner) + b_outer
    a_outer = 1.d0/(1.d0+exp(-z_outer))
    e_outer = a_outer-nm_y
    c = 0.5d0*dinv*sum((e_outer)**2)

    !Compute dc/db_outer and dc/dw_outer
    g_outer = a_outer*(1.d0-a_outer)
    eg_outer = e_outer*g_outer
    dcdb_outer = dinv*sum(eg_outer)
    dcdw_outer = dinv*matmul(a_inner,eg_outer)

    !Compute dc/db_inner and dc/dw_inner
    g_inner = a_inner*(1.d0-a_inner)
    dcdb_inner = dinv*w_outer*matmul(g_inner,eg_outer)
    do l1 = 1,n
      dcdw_inner(:,l1) = dinv*w_outer*matmul(g_inner,(nm_x(l1,:)*eg_outer))
    end do

    !Pack gradient into cgrad
    do i1=1,n
      j1 = (i1-1)*m+1
      cgrad(j1:j1+m-1) = dcdw_inner(:,i1)
    end do
    cgrad(n*m+1:n*m+m) = dcdb_inner
    cgrad(n*m+m+1:n*m+2*m) = dcdw_outer
    cgrad(n*m+2*m+1) = dcdb_outer

  end subroutine nnmodel_mpi

  !Use full gradient descent
  !to move towards optimal fitting parameters using
  ! nnmodel. Iterates for 400 "epochs" and final fitting
  !parameters are stored in fvec.
  !Input:
  !fvec_guess: initial vector of fitting parameters
  !n: number of pixels in each image (should be 784)
  !m: number of neurons in inner layer; snmodel is used if m=0
  !d: number of training images to be used; only the 1st d images and labels stored
  !in nm_x and nm_y are used in the optimization calculation
  !alpha: learning rate, it is fine to keep this as alpha=0.1 for this assignment
  !Output:
  !fvec: fitting parameters, see comments above for snmodel and nnmodel to see how
  !weights and biases are stored in the array.
  !Note: nm_x and nm_y must be allocated and set before calling this subroutine.

  subroutine fgd_mpi(comm,fvec_guess,n,m,dlocal,d,alpha,fvec)
    implicit none
    integer, intent(in) :: comm,n,m,dlocal,d
    real(kind=8), dimension(:), intent(in) :: fvec_guess
    real(kind=8), intent(in) :: alpha
    real(kind=8), dimension(size(fvec_guess)), intent(out) :: fvec
    integer :: i1, j1, i1max=1000, n_fvec
    real(kind=8) :: c,c_total
    real(kind=8), dimension(size(fvec_guess)) :: cgrad,cgrad_total
    integer :: myid,ierr

    call MPI_COMM_RANK(comm,myid,ierr)

    n_fvec = size(fvec_guess)
    fvec = fvec_guess
    do i1=1,i1max
        call nnmodel_mpi(fvec,n,m,dlocal,d,c,cgrad)
        call MPI_ALLREDUCE(cgrad,cgrad_total,n_fvec,MPI_DOUBLE_PRECISION,MPI_SUM,comm,ierr)
        fvec = fvec - alpha*cgrad_total !update fitting parameters using gradient descent step
        if ((myid==0) .and.(mod(i1,50)==0)) print *, 'completed epoch # ', i1, 'cost on process 0 =',c
    end do
```

```
end subroutine fgd_mpi
```

Part 2: Bacterial contamination



Consider the following model for the spread of bacteria on a liquid surface:

$$\frac{\partial C}{\partial t} = \nabla^2 C - gC + S$$

where $C(r, \theta, t)$ is the concentration of bacteria at location, (r, θ), and time, t, $S(r, \theta)$ is a source of contamination, and g is the death rate ($g \geq 0$). We consider the following configuration:

$$1 \leq r \leq \pi + 1$$
$$0 \leq \theta \leq \pi$$

and assume that antibiotics have been applied to surfaces of the container such that $C(r = \pi + 1) = C(\theta = 0) = C(\theta = \pi) = 0$. The surface, $r = 1$, however, is an additional source of contamination with $C(r = 1) = sin^2(k\theta)$ where k is a model parameter.

We will only consider the time-independent problem which, in polar coordinates, is:

$$\frac{\partial^2 C}{\partial r^2} + \frac{1}{r}\frac{\partial C}{\partial r} + \frac{1}{r^2}\frac{\partial^2 C}{\partial \theta^2} - gC = -S$$

We will use a Gaussian model for the source, $S = exp\left[-20\left((r - r_0)^2 + (\theta - \theta_0)^2\right)\right]$, where the parameters, r_0 and θ_0, set the source location.

In this part, you will complete Fortran and Python routines to 1) compute numerical solutions to this model using the over-step method described here: Iterative methods for contamination model (edited 6/12/18), 2) analyze the performance of your codes, and 3) analyze the contamination dynamics.

Note that you have been provided with both *Fortran* and *Python* codes for computing solutions using Jacobi iteration.

---

1) (10 pts) Complete the function *simulate* in *p2.py* and compute solutions to this model with the *OSI* method described in the accompanying notes. The function should return the final concentration distribution and an array (or list) containing the maximum change in concentration for each iteration ($max(|C^{k+1}_{i,j} - C^k_{i,j}|)$). Iterations should terminate when this maximum change falls below *tol* which is an input parameter.

ANS: The key here is to avoid calculations in loops as much as possible. Two approaches were acceptable. The first loops through the matrix going across rows and columns in the conventional order and vectorizes the calculation as much as possible:

The second (and better) acceptable solution was to iterate down the main diagonal of the matrix and update all elements along the "orthogonal" diagonal. For example, for index (2,2), the elements (1,2) and (2,1) are updated:

```
    Ctemp = C.copy()

    iarray = np.arange(1,n+1)

    for k in range(kmax):
        Cold= C.copy()
        #Compute Cnew
        Ctemp[1:-1,1:-1] = Sdel2[1:-1,1:-1]+w*(C[2:,1:-1]*facp[1:-1,1:-1]+C[1:-1,2:]*fac2[1:-1,1:-1])+

        #sweep through first set of diagonals
        for i in range(n):
            ind2 = iarray[:i+1]
            ind1 = ind2[-1::-1]
            C[ind1,ind2] = Ctemp[ind1,ind2] + w*(C[ind1-1,ind2]*facm[ind1,ind2] + C[ind1,ind2-1]*fac2[

        #sweep through second set of diagonals
        for i in range(n-1):
            ind2 = iarray[i+1:]
            ind1 = ind2[-1::-1]
            C[ind1,ind2] = Ctemp[ind1,ind2] + w*(C[ind1-1,ind2]*facm[ind1,ind2] + C[ind1,ind2-1]*fac2[

        #Compute delta_p
        deltac += [np.max(np.abs(C-Cold))]

        if k%1000==0: print("k,dcmax:",k,deltac[k])
        #check for convergence
        if deltac[k]<tol:
            print("Converged,k=%d,dc_max=%28.16f " %(k,deltac[k]))
            break
```

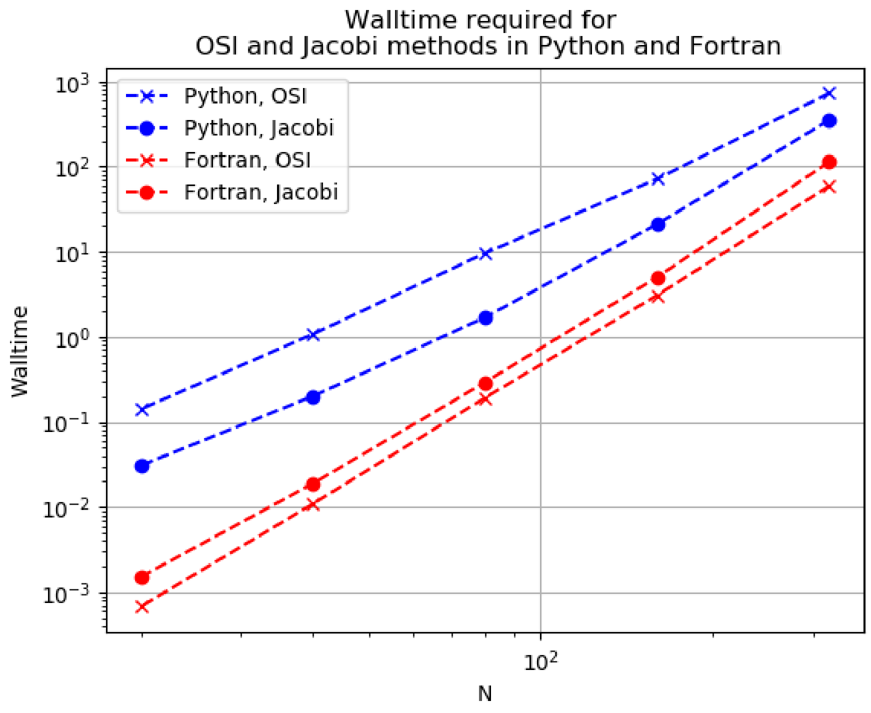
2) (10 pts) Complete the subroutine *simulate* in *p2.f90* and again compute solutions to this model with the *OSI* method. The routine should again return the final concentration matrix, but note now that several model and numerical parameters are module variables rather than input to the subroutine. Additionally the maximum change in *C* for each iteration should be stored in the module variable, *deltac*.

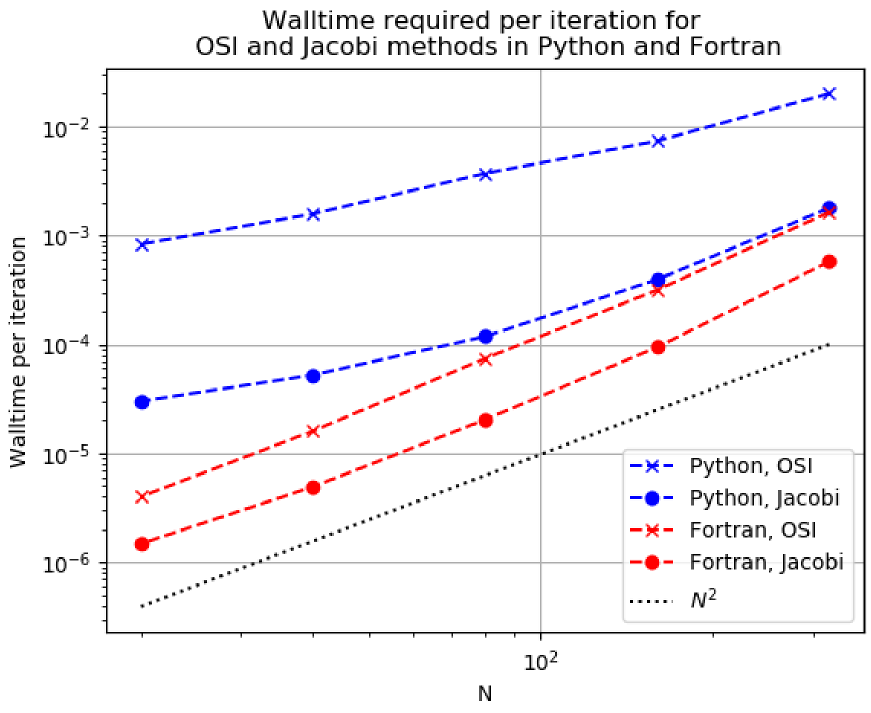
ANS: The Fortran version of the code does not need to be vectorized, and a straightforward loop-based implementation of the algorithm is sufficient. The code provided here is the Fortran version of the vectorized Python code above:

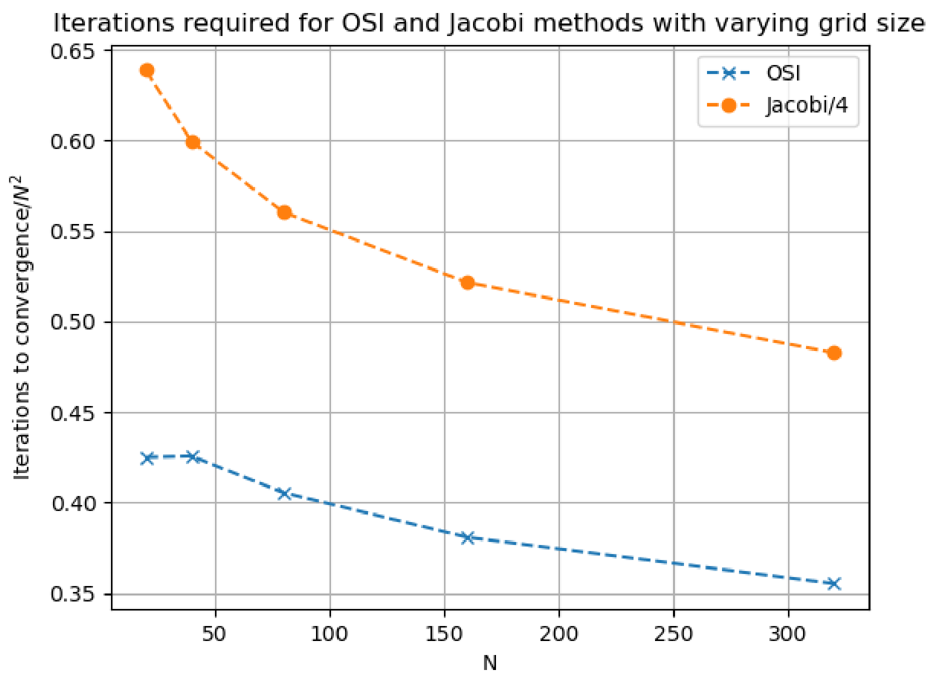
```
    do k1=1,bm_kmax
        Cold = C
        C(1:n,1:n) = Sdel2(1:n,1:n) + w*(Cold(2:n+1,1:n)*facp(1:n,1:n)+Cold(1:n,2:n+1)*fac2(1:n,1:n))+
        do i1=1,n
          C(i1,1:n) = C(i1,1:n)+w*C(i1-1,1:n)*facm(i1,1:n)
          do j1=1,n
            C(i1,j1) = C(i1,j1) + w*(C(i1,j1-1))*fac2(i1,j1)
          end do
        end do

        deltaC(k1) = maxval(abs(Cold(1:n,1:n)-C(1:n,1:n))) !compute relative error
        if (deltaC(k1)<bm_tol) exit !check convergence criterion
        !if (mod(k1,1000)==0) print *, k1,deltaC(k1)
    end do
```

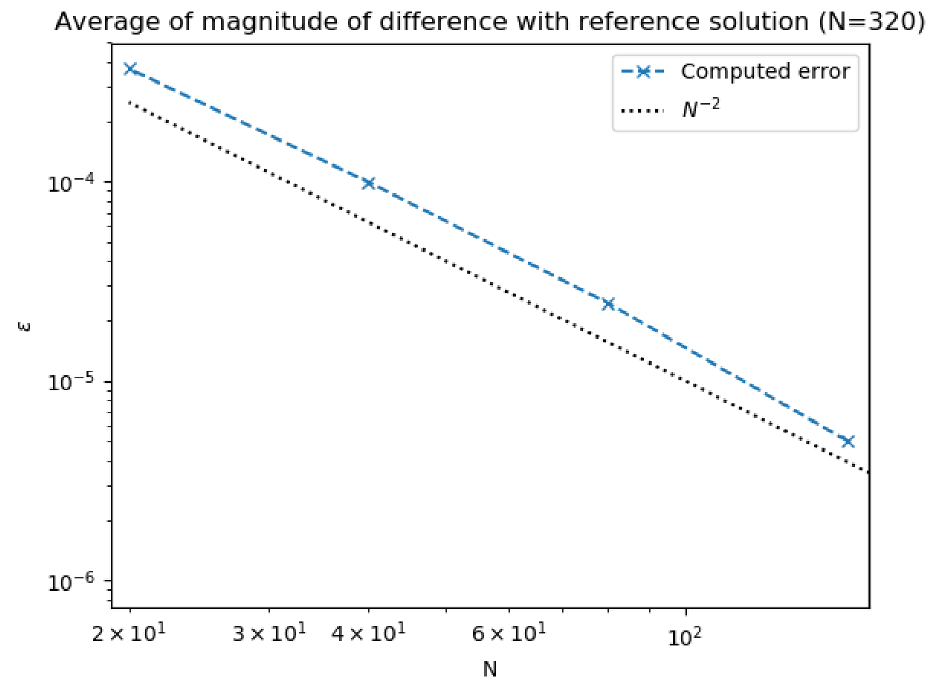
3) (15 pts) Analyze the performance of your Fortran and Python routines. Your analysis should consist of one or more figures and accompanying discussion. Place the code used to generate your figures in the function, *performance*, in *p2.py*, and add your discussion to the docstring of this function. You should save these figures and include them in your project repo.

ANS: We have two methods and two languages giving four routines to compare. The aim is to understand their performance as the problem size varies. The routines from parts 1 and 2 were modified to collect timing information and were then run for *N* between 20 and 320. Results for the walltime are shown in the first figure below. This figure is sufficient to see that Fortran+OSI is the best option (the tolerance criteria were adjusted to ensure that all computations converged) followed by Fortran+Jacobi, Python+Jacboi, and Python+OSI in that order. In order to understand these trends properly, we need to consider the walltime/iteration for each of the four routines as well as the number of iterations required for convergence for the two methods. Walltime/iteration is shown in the 2nd figure below, and we see that the cost of both Fortran routines scales approximately as N^2 which we expect since we are working with a *N+2* x *N+2* matrix. The behavior for the Python routines is more complicated. Jacobi+Python does seem to approach N^2 behavior at at larger values of *N* but otherwise the rate of increase of walltime is slower than N^2 for both methods. This suggests that accelaration due to vectorization is particularly significant at smaller values of *N*. While the walltime/iteration for OSI tends to be slower than Jacobi, the number of iterations required for convergence tends to be substantially smaller. This is is seen in the 3rd figure below which shows iterations/N^2. Note that the Jacobi results have been divided by 4 so that we can see the trends more clearly. The curves show that OSI requires approximately 5 times less iterations (for this range of *N*) and that rate of increase of the number of iterations is a little slower than N^2, so the total walltime for large *N* increase at a rate smaller than N^4. Finally, we should consider the accuracy of the solutions for this range of *N*. Taking the *N=320* result as 'exact', an error is computed as $\sum |C_{exact} - C_N|/N$ where C_N is the concentration field for a particular value of *N*, and the sum is over all elements in the concentration matrices. The results are shown in the fourth figure below. The error drops as N^{-2} as expected, and we now largely have all the information needed to assess the four routines. A full picture would require moving to larger values of *N* which have been omitted here due to the corresponding increase in computational time. The full *performance* function is near the bottom of this page.







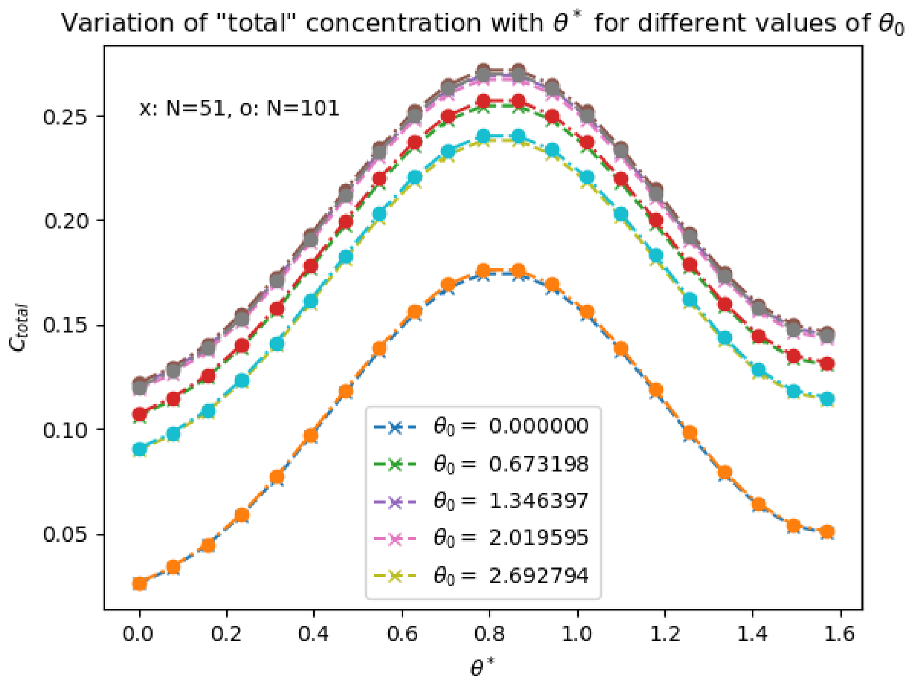


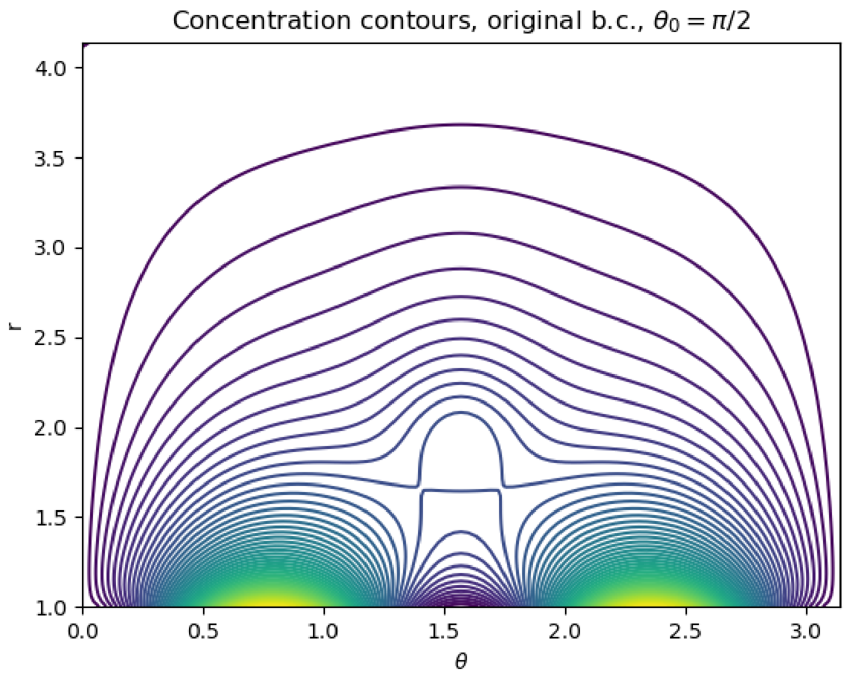
4) (20 pts) It is possible to fight against the contamination by applying antibacterial agents locally along the contaminated boundary. This modifies the boundary condition at *r=1* to, $C(r=1) = exp(-10(\theta-\theta^*)^2)sin^2(k\theta)$. What is the best choice of θ^*? It is sufficient to consider the case, $g=1, r_0 = 1+\pi/2$, *k=2*, and $S_0 = 2$, but you are encouraged to consider other parameter values as well. Place your answer, an explanation of how you arrived at it, and accompanying code in *analyze* in *p2.py*. You should also save and submit figures generated by your code.

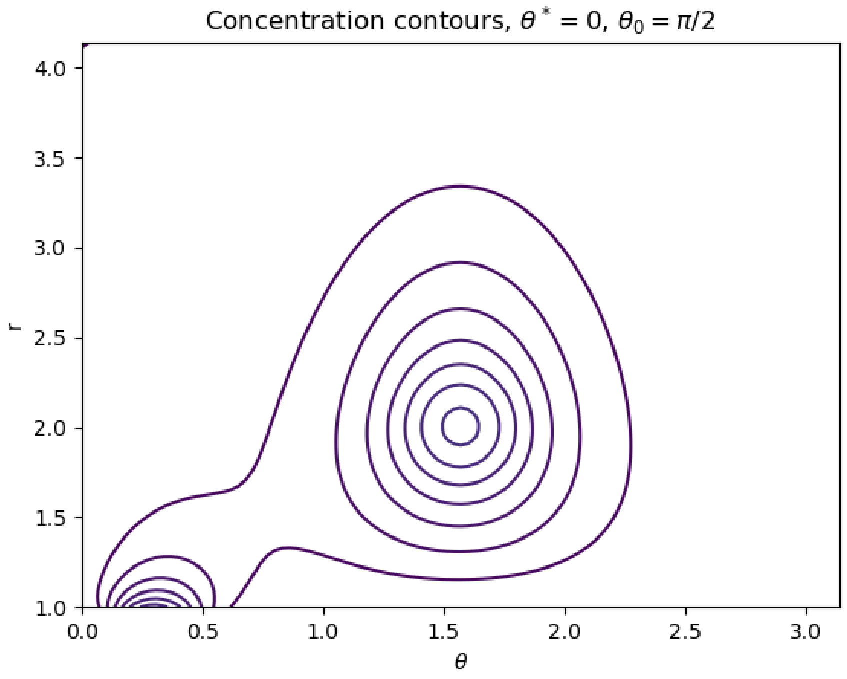
ANS: It is straightforward to modify the boundary condition used in any of the 4 routines and use the code here. There are two parameters to vary, θ_0 and θ^*. Additionally, we need to ensure that the solution is not dependent on *N*. As a measure of effectiveness, the area integral of the concentration, $C_{tot} = \int_1^{1+\pi}\int_0^{\pi} Crdrd\theta$, is computed using the function, *total_c*:

```
    def total_c(C):
        """ Compute integral(C r dr dtheta)
        """
        from scipy.integrate import simps
        n = C.shape[0]-2
        r = np.linspace(1,1+np.pi,n+2)
        t = np.linspace(0,np.pi,n+2)
        Csum_r = C.sum(axis=1)*r/(n+2)

        return simps(Csum_r,r)
```

C_{tot} is computed for a range of θ_0 and θ^* and two values of N. The resulting plot (shown below) shows that there are local minima at $\theta^* = 0, \pi/2, \pi$ for all computed values of θ_0 and that the results are insensitive to N. Note that results are only computed for $0 \leq \theta^* \pi/2$ due to the symmetry of the problem. The absolute minima occur at 0 and π and these are the best choices for θ^*. The 2nd and 3rd figures below show the dramatic effect of applying the antibacterial agent with $\theta^* = 0$







Part 3: MPI simulation of bacterial contamination

Here, you will develop a distributed-memory approach for the model introduced in part 2, and you will utilize the alternating over-step method (AOS) described here: Iterative methods for contamination model (edited 6/12/18). A radial domain-decomposition approach is used where process p is assigned a portion of the domain $r_{p1} \leq r \leq r_{p2}$. This decomposition is computed for you as described below.

The file, *p3.f90* contains a main program which initializes MPI, reads in input parameters from a text file (which you should create), calls the subroutine *simulate_mpi*, and writes the output from *simulate_mpi* to a file. You will have to complete *simulate_mpi*.

1. (25 pts) The domain decomposition takes place at the begining of *simulate_mpi*. The *n* radial points are distributed to *numprocs* processes using *MPE_DECOMP1D*. Points on the radial boundaries should be assigned to the first and last processes (there are *n+2* points total in each direction). Then, each process should solve the model from part 2 for r_i from *istart* to *iend* (and $0 \leq \theta \leq \pi$) using the AOS method. Since finite-difference approximations require information from *i+1* and *i-1*, each process must typically exchange information with one or more neighbors during each iteration.

Complete *simulate_mpi* so the model system of equations is distributed across the *numprocs* processes and iteratively solved in parallel using the AOS method.

ANS: Two sets of send/recvs are needed. The first provides updated white points from neighboring processes for the white update, while the second provides updated black points from neighboring processes for the black update. A reduction is used to determine the maximum delta_c. Since communication should be minimized, it is important to only send data corresponding to the points needed for the subsequent update. The solution code is below:

```fortran
!Simulate contamination model with AOS iteration
subroutine simulate_mpi(comm,numprocs,n,kmax,tol,g,S0,r0,t0,k_bc,C,deltaC)
    !input:
    !comm: MPI communicator
    !numprocs: total number of processes
    !n: total number of grid points in each direction (actually n+2 points in each direction)
    !tol: convergence criteria
    !kmax: total number of iterations
    !g: bacteria death rate
    !S0,r0,t0: source function parameters
    !k_bc: r=1 boundary condition parameter
    !output: C, final solution
    !deltaC: |max change in C| each iteration
    use mpi
```

```fortran
implicit none
integer, intent (in) :: comm,numprocs,n,kmax
real(kind=8), intent(in) ::tol,g,S0,r0,t0,k_bc
real(kind=8), dimension(0:n+1,0:n+1), intent(out) :: C
real(kind=8), intent(out) :: deltac(kmax)
integer :: i1,i2,j0,j1,k,istart,iend
integer :: myid,ierr,nlocal
real(kind=8) :: temp,t1d(0:n+1),del,pi,w=1.5
real(kind=8), allocatable, dimension(:) :: r1d
real(kind=8),allocatable, dimension(:,:) :: r,t,rinv2,Sdel2,Clocal,Cnew
real(kind=8), allocatable, dimension(:,:) :: fac,fac2,facp,facm
integer, dimension(numprocs) :: disps,Nper_proc
integer :: jsend,jrecv,Nbuf_send, Nbuf_recv, request
integer, dimension(MPI_STATUS_SIZE) :: status


call MPI_COMM_RANK(comm, myid, ierr)
print *, 'start simulate_mpi, myid=',myid

!Set up theta
pi = acos(-1.d0)
del = pi/dble(n+1)
do i1 = 0,n+1
  t1d(i1) = i1*del
end do



!generate decomposition and allocate sub-domain variables
!Note: this decomposition is for indices 1:n; it
!ignores the boundaries at i=0 and i=n+1 which
!should be assigned to the first and last processes
call mpe_decomp1d(n,numprocs,myid,istart,iend)
print *, 'istart,iend,threadID=',istart,iend,myid
nlocal = iend-istart+1
allocate(r1d(nlocal+2))

!generate local grid and allocate Clocal----
do i1=istart-1,iend+1
  r1d(i1-istart+2) = 1.d0 + i1*del
end do

allocate(r(0:nlocal+1,0:n+1),t(0:nlocal+1,0:n+1),rinv2(0:nlocal+1,0:n+1), &
    Sdel2(0:nlocal+1,0:n+1),Clocal(0:nlocal+1,0:n+1),Cnew(0:nlocal+1,0:n+1))
allocate(fac(0:nlocal+1,0:n+1),fac2(0:nlocal+1,0:n+1),facp(0:nlocal+1,0:n+1),facm(0:nlocal+1,

do j1=0,n+1
  r(:,j1) = r1d
end do
do i1=0,nlocal+1
  t(i1,:) = t1d
end do
!-----------------

!Coefficients for AOS
rinv2 = 1.d0/(r**2)
fac = 1.d0/(2.d0+2.d0*rinv2+del*del*g) !1/(del^2*k^2 + 2 + 2/r^2)
facp = (1.d0 + 0.5d0*del/r)*fac !(1 + del/(2r))*fac
facm = (1.d0 - 0.5d0*del/r)*fac !(1 - del/(2r))*fac
fac2 = fac*rinv2
!----------------

!set boundary conditions/initial condition
Clocal = 0.d0
Clocal =  (sin(k_bc*t)**2)*(pi+1.d0-r)/pi
Cnew = Clocal

!set source function, Sdel2 = 1.5*S*del^2*fac
Sdel2 = 1.5d0*S0*exp(-20.d0*((r-r0)**2+(t-t0)**2))*(del**2)*fac


!AOS iteration
do k = 1,kmax
```

```fortran
Cnew(1:nlocal,1:n) = Sdel2(1:nlocal,1:n) -0.5d0*Clocal(1:nlocal,1:n)

do i1=1,nlocal
  j0 = 2-mod(istart+i1-1,2)
  Cnew(i1,j0:n:2) =   Cnew(i1,j0:n:2) + &
          w*(Clocal(i1+1,j0:n:2)*facp(i1,j0:n:2) + Clocal(i1-1,j0:n:2)*facm(i1,j0:n:2) + &
                                 (Clocal(i1,j0-1:n-1:2) + Clocal(i1,j0+1:n+1:2))*fac2(i1
  if ((istart+i1-1)==11) then
  end if
end do

!Send either cnew(nlocal,1:2:n) or cnew(nlocal,2:2:n)
!to process myid+1 to be stored in clocal(0,1:2:n) or
!clocal(0,2:2:n)
if (myid<numprocs-1) then
  jsend = 2-mod(iend,2)
  Nbuf_send = size(Cnew(nlocal,jsend:n:2))
  call MPI_ISEND(Cnew(nlocal,jsend:n:2),Nbuf_send,MPI_DOUBLE_PRECISION,myid+1, &
    0,comm,request,ierr)
end if

if (myid>0) then
  jrecv = 2-mod(istart-1,2)
  Nbuf_recv = size(Cnew(0,jrecv:n:2))
  call MPI_RECV(Cnew(0,jrecv:n:2),Nbuf_recv,MPI_DOUBLE_PRECISION,myid-1,MPI_ANY_TAG,comm,s
end if

!Send either cnew(1,1:2:n) or cnew(1,2:2:n)
!to process myid+1 to be stored in clocal(nlocal+1)
if (myid>0) then
  jsend = 2-mod(istart,2)
  Nbuf_send = size(Cnew(1,jsend:n:2))
  call MPI_ISEND(Cnew(1,jsend:n:2),Nbuf_send,MPI_DOUBLE_PRECISION,myid-1, &
    0,comm,request,ierr)
end if

if (myid < numprocs-1) then
  jrecv = 2-mod(iend+1,2)
  Nbuf_recv = size(Cnew(nlocal+1,jrecv:n:2))
  call MPI_RECV(Cnew(nlocal+1,jrecv:n:2),Nbuf_recv,MPI_DOUBLE_PRECISION,myid+1,MPI_ANY_TAG
end if



do i1=1,nlocal
  j0 = 1+mod(istart+i1-1,2)
  Cnew(i1,j0:n:2) =   Cnew(i1,j0:n:2)+ &
        w*(Cnew(i1+1,j0:n:2)*facp(i1,j0:n:2) + Cnew(i1-1,j0:n:2)*facm(i1,j0:n:2) + &
                                 (Cnew(i1,j0-1:n-1:2) + Cnew(i1,j0+1:n+1:2))*fac2(i1,j0:
  if ((istart+i1-1)==10) then
  end if
end do

!Send either cnew(nlocal,1:2:n) or cnew(nlocal,2:2:n)
!to process myid+1 to be stored in clocal(0)
if (myid<numprocs-1) then
  jsend = 1+mod(iend,2)
  Nbuf_send = size(Cnew(nlocal,jsend:n:2))
  call MPI_ISEND(Cnew(nlocal,jsend:n:2),Nbuf_send,MPI_DOUBLE_PRECISION,myid+1, &
    0,comm,request,ierr)
end if
if (myid>0) then
  jrecv = 1+mod(istart-1,2)
  Nbuf_recv = size(Cnew(0,jrecv:n:2))
  call MPI_RECV(Cnew(0,jrecv:n:2),Nbuf_recv,MPI_DOUBLE_PRECISION,myid-1,MPI_ANY_TAG,comm,s
end if


!Send either cnew(1,1:2:n) or cnew(1,2:2:n)
!to process myid+1 to be stored in clocal(0)
if (myid>0) then
  jsend = 1+mod(istart,2)
```

```
          Nbuf_send = size(Cnew(1,jsend:n:2))
          call MPI_ISEND(Cnew(1,jsend:n:2),Nbuf_send,MPI_DOUBLE_PRECISION,myid-1, &
            0,comm,request,ierr)

        end if

        if (myid < numprocs-1) then
          jrecv = 1+mod(iend+1,2)
          Nbuf_recv = size(Cnew(nlocal+1,jrecv:n:2))
          call MPI_RECV(Cnew(nlocal+1,jrecv:n:2),Nbuf_recv,MPI_DOUBLE_PRECISION,myid+1,MPI_ANY_TAG

        end if

        temp = maxval(abs(Cnew(1:nlocal,:)-Clocal(1:nlocal,:))) !compute relative error
        Clocal = Cnew
        call MPI_ALLREDUCE(temp,deltaC(k),1,MPI_DOUBLE_PRECISION,MPI_MAX,comm,ierr)
        if (deltaC(k)<tol) exit !check convergence criterion
      !  if (mod(k,1000)==0) print *, k,deltaC(k),maxval(abs(Clocal)),myid,temp
    end do
  !  print *, 'k,error=',k,deltaC(min(k,kmax))


    !---------------------------------------------------------
    !Code below constructs C from the Clocal on each process
    !print *, 'before collection',myid, maxval(abs(Clocal))

    i1=1
    i2 = nlocal

    if (myid==0) then
      i1=0
      nlocal = nlocal+1
    elseif (myid==numprocs-1) then
      i2 = nlocal+1
      nlocal = nlocal + 1
    end if

    call MPI_GATHER(nlocal,1,MPI_INT,NPer_proc,1,MPI_INT,0,comm,ierr)
    !collect Clocal from each processor onto myid=0

    if (myid==0) then
        disps(1)=0
        do j1=2,numprocs
          disps(j1) = disps(j1-1) + Nper_proc(j1-1)*(n+2)
        end do

      !  print *, 'nper_proc=',NPer_proc
      !  print *, 'disps=',disps
    end if

  !collect Clocal from each processor onto myid=0

     call MPI_GATHERV(transpose(Clocal(i1:i2,:)),nlocal*(n+2),MPI_DOUBLE_PRECISION,C,Nper_proc*(n+
                 disps,MPI_DOUBLE_PRECISION,0,comm,ierr)

      C = transpose(C)
    if (myid==0) print *, 'finished',maxval(abs(C)),sum(C)

end subroutine simulate_mpi
```

2. (3 pts) It is possible to decompose the square grid into $numprocs = m^2$ squares (where m is an integer greater than 1) rather than $numprocs$ ‘rectangles’. Clearly and concisely describe one key advantage and one disadvantage of using ‘smaller-squares’ approach for this problem. Place your answer in a comment at the top of *p3.f90*

ANS: As discussed in lecture, the smaller-squares approach leads to a lower total number of points at the boundaries between processes which leads to reduced communication and better performance. However, this approach is relatively difficult to implement.

Python *performance* routine from part 2:

```python
def performance(Nmin=20,Ngrids=5,display=False):
    """Analyze performance of simulation codes
    Add input/output variables as needed.
    The code here will use kmax=1e6,tol=1e-12 and otherwise will
    use the default model parameters.
    """

    #Set up problem parameters for testing
    bm.bm_tol = 1e-12
    bm.bm_kmax=10**6
    input_num=(10**6,1e-12)

    Nlist=[Nmin]
    for i in range(Ngrids-1):
        Nlist.append(Nlist[i]*2)
    Nlist.reverse()

    dt_array_py = np.zeros((2,len(Nlist)))
    dt_array_f90 = np.zeros((2,len(Nlist)))
    iter_array = np.zeros((2,len(Nlist)))
    eps_array = np.zeros(len(Nlist)-1)
    Nlist = np.array(Nlist)

    #Run the four routines with varying N and collect timing information--------
    for i,N in enumerate(Nlist):
        N = N-1
        Cf90=np.zeros((N+2,N+2))
        Cpy = np.zeros((N+2,N+2))
        print("i,N=",i,N)
        #Run f90 routines
        Cf90 = bm.simulatev(N)
        dt_array_f90[0,i] = bm.bm_dt
        iter_array[0,i] = bm.niter
        Cf90 = bm.simulate_jacobi(N)
        dt_array_f90[1,i] = bm.bm_dt
        iter_array[1,i] = bm.niter

        #Run python routines
        Cpy,_,dt = simulate_t(N,input_num)
        dt_array_py[0,i] = dt
        Cpy,_,dt = simulate_jacobi(N,input_num)
        dt_array_py[1,i] = dt

        if i==0:
            C0 = Cf90
        else:
            eps_array[i-1] = np.mean(np.abs(Cf90-C0[::2**i,::2**i])) #Compute error
    #-----------------------------------------------------------------


    if display:

        plt.figure()
        plt.loglog(Nlist[1:],eps_array,'x--')
        plt.xlabel('N')
        plt.ylabel(r'$\epsilon$')
        plt.title('Average of magnitude of difference with reference solution (N=%d)'%Nlist[0])

        plt.figure()
        plt.plot(Nlist,iter_array[0,:]/Nlist**2,'x--')
        plt.plot(Nlist,iter_array[1,:]/Nlist**2/4,'o--')
        plt.xlabel('N')
        plt.ylabel('Iterations to convergence/$N^2$')
        plt.legend(('OSI','Jacobi/4'))
        plt.title('Iterations required for OSI and Jacobi methods with varying grid size')
        plt.grid()

        plt.figure()
        plt.loglog(Nlist,dt_array_py[0,:],'bx--')
        plt.loglog(Nlist,dt_array_py[1,:],'bo--')
        plt.loglog(Nlist,dt_array_f90[0,:],'rx--')
        plt.loglog(Nlist,dt_array_f90[1,:],'ro--')
        plt.legend(('Python, OSI','Python, Jacobi','Fortran, OSI','Fortran, Jacobi'))
        plt.xlabel('N')
        plt.ylabel('Walltime')
        plt.title('Walltime required for \n OSI and Jacobi methods in Python and Fortran')

        plt.figure()
        plt.loglog(Nlist,dt_array_py[0,:]/iter_array[0,:],'bx--')
        plt.loglog(Nlist,dt_array_py[1,:]/iter_array[1,:],'bo--')
        plt.loglog(Nlist,dt_array_f90[0,:]/iter_array[0,:],'rx--')
        plt.loglog(Nlist,dt_array_f90[1,:]/iter_array[1,:],'ro--')
        plt.plot(Nlist,(1e-2*Nlist/Nlist[0])**2,'k:')
        plt.legend(('Python, OSI','Python, Jacobi','Fortran, OSI','Fortran, Jacobi','$N^2$'))
        plt.xlabel('N')
        plt.ylabel('Walltime per iteration')
        plt.title('Walltime required per iteration for \n OSI and Jacobi methods in Python and For


    return Nlist,dt_array_py,dt_array_f90,iter_array,eps_array
```

Python *analyze* routine from part 2:

```
def analyze(Narray=[51,101],display=False):
    """Analyze influence of modified
    boundary condition on contamination dynamics
        Add input/output variables as needed.
    """

    #Increase bm_tol and bm_kmax for accuracy, set k_bc=2, S0 to 2, otherwise
    #use default model parameters
    bm.bm_tol = 1e-12
    bm.bm_kmax = 10**6
    bm.bm_s0 = 2
    bm.bm_kbc = 2

    #Set up theta_0 and theta*
    Nt0=15
    Nts=21

    t0_array = np.linspace(0,np.pi,Nt0)
    ts_array = np.linspace(0,np.pi/2,Nts)

    #Run modfied simulate routine for different theta_0, theta*
    Ctot = np.zeros((len(Narray),Nt0,Nts))
    for k,n in enumerate(Narray):
        for i,bm.bm_t0 in enumerate(t0_array):
            for j,bm.bm_tstar in enumerate(ts_array):
                print("***i,j=",i,j,'***')
                C = bm.simulate4(n)
                Ctot[k,i,j] = total_c(C)

    #Display results
    if display:
        plt.figure()
        for i in range(0,Nt0,3):
                plt.plot(ts_array,Ctot[0,i,:],'x--',label=r'$\theta_0=$ %f' %t0_array[i])
                plt.plot(ts_array,Ctot[1,i,:],'o-.')
        plt.legend()
        plt.xlabel(r'$\theta^*$')
        plt.ylabel(r'$C_{total}$')
        plt.title(r'Variation of "total" concentration with $\theta^*$ for different values of $\t


        r = np.linspace(1,1+np.pi,n+2)
        t = np.linspace(0,np.pi,n+2) #theta
        bm.bm_tstar = 0
        bm.bm_t0 = np.pi/2
        C1 = bm.simulate4(101)
        C2 = bm.simulate(101)
        plt.figure()
        plt.contour(t,r,C1,np.linspace(0,1,51))
        plt.xlabel(r'$\theta$')
        plt.ylabel('r')
        plt.title(r'Concentration contours, $\theta^*=0$, $\theta_0=\pi/2$')

        plt.figure()
        plt.contour(t,r,C2,np.linspace(0,1,51))
        plt.xlabel(r'$\theta$')
        plt.ylabel('r')
        plt.title(r'Concentration contours, original b.c., $\theta_0=\pi/2$')


    return t0_array,ts_array,Ctot
```