Contents

.0.hw5.info

This simple document is available at https://tachibanatech.com/litdoc/hw5.simple.pdf in PDF format.

The full document is available at https://tachibanatech.com/litdoc/hw5.full.pdf in PDF format and https://tachibanatech.com/litdoc/hw5.full.d/_book/ in HTML format.

Equations

The 1-D advection-dispersion equation is

$$\frac{\partial c}{\partial t} = D\frac{\partial^2 c}{\partial x^2} - v\frac{\partial c}{\partial x}$$

where D is the dispersion coefficient, and v is the average linear flow velocity.

The general equation for finite difference is

$$\begin{aligned}\frac{c_i(t+\Delta t) - c_i}{\Delta t} = \theta &\left[D\frac{c_{i+1}(t+\Delta t) - 2c_i(t+\Delta t) + c_{i-1}(t+\Delta t)}{(\Delta x)^2} - v\frac{c_{i+1}(t+\Delta t) - c_{i-1}(t+\Delta t)}{2\Delta x}\right] \\ &+ (1-\theta)\left[D\frac{c_{i+1}(t) - 2c_i(t) + c_{i-1}(t)}{(\Delta x)^2} - v\frac{c_{i+1}(t) - c_{i-1}(t)}{2\Delta x}\right]\end{aligned}$$

Rearranged,

$$-\theta A c_{i-1}(t+\Delta t) + B^* c_i(t+\Delta t) - \theta C c_{i+1}(t+\Delta t) = (1-\theta)A c_{i-1}(t) + B^{**} c_i(t) + (1-\theta)C c_{i+1}(t)$$

where

$$\begin{aligned}A &= \frac{D\Delta t}{(\Delta x)^2} + \frac{v\Delta t}{2\Delta x} \\ B^* &= 1 + 2\theta\frac{D\Delta t}{(\Delta x)^2} \\ B^{**} &= 1 + 2(\theta - 1)\frac{D\Delta t}{(\Delta x)^2} \\ C &= \frac{D\Delta t}{(\Delta x)^2} - \frac{v\Delta t}{2\Delta x}\end{aligned}$$

Converting to matrices,

$$\begin{bmatrix} 1 & & & & & & \\ -\theta A & B^* & -\theta C & & & & \\ & -\theta A & B^* & -\theta C & & & \\ & & . & . & . & & \\ & & & -\theta A & B^* & -\theta C & \\ & & & & -\theta A & B^* & -\theta C \\ & & & & & & 1 \end{bmatrix} \begin{bmatrix} c_0(t+\Delta t) \\ c_1(t+\Delta t) \\ . \\ . \\ . \\ c_\ell(t+\Delta t) \\ c_{\ell+1}(t+\Delta t) \end{bmatrix} = \begin{bmatrix} g_0(t) \\ g_1(t) \\ . \\ . \\ . \\ g_\ell(t) \\ g_{\ell+1}(t) \end{bmatrix}$$

where $g_i(t) = (1-\theta)Ac_{i-1}(t) + B^{**}c_i(t) + (1-\theta)Cc_{i+1}(t)$ for $1 \le g \le \ell$ and $g_0(t)$ and $g_{\ell+1}(t)$ are boundary conditions.

Removing the borders because they are known,

$$\begin{bmatrix} B^* & -\theta C & & & \\ -\theta A & B^* & -\theta C & & \\ & . & . & . & \\ & & -\theta A & B^* & -\theta C \\ & & & -\theta A & B^* \end{bmatrix} \begin{bmatrix} c_1(t+\Delta t) \\ c_2(t+\Delta t) \\ . \\ c_{\ell-1}(t+\Delta t) \\ c_\ell(t+\Delta t) \end{bmatrix} = \begin{bmatrix} g_1(t) + \theta A c_0(t+\Delta t) \\ g_2(t) \\ . \\ g_{\ell-1}(t) \\ g_\ell(t) + \theta C c_{\ell+1}(t+\Delta t) \end{bmatrix}$$

Suitability of the Thomas Algorithm

The LAPACK subroutine `dgtsv` uses Gaussian elimination since the Thomas algorithm is conditionally stable. Gaussian elimination, however, is $O(n^3)$, whereas the Thomas algorithm is $O(n)$.

The Thomas algorithm for this problem is

$$C'_i(t) = \begin{cases} \frac{-\theta C}{B^*} & ;\ i = 1 \\ \frac{-\theta C}{B^* + \theta A C'_{i-1}(t)} & ;\ i = 2, 3, \ldots, \ell - 1 \end{cases}$$

$$G'_i(t) = \begin{cases} \frac{g_1(t) + \theta A c_0(t+\Delta t)}{B^*} & ;\ i = 1 \\ \frac{g_i(t) + \theta A G'_{i-1}}{B^* + \theta A C'_{i-1}(t)} & ;\ i = 2, 3, \ldots, \ell - 1 \\ \frac{g_\ell(t) + \theta C c_{\ell+1}(t+\Delta t) + \theta A G'_{\ell-1}}{B^* + \theta A C'_{\ell-1}(t)} & ;\ i = \ell \end{cases}$$

$$c_\ell(t+\Delta t) = G'_\ell(t)$$

$$c_i(t+\Delta t) = G'_i(t) - C'_i(t) c_{i+1}(t+\Delta t) \quad ;\ i = \ell - 1, \ell - 2, \ldots, 1$$

The Thomas algorithm is stable when the matrix is diagonally dominant.

$$\begin{aligned} B^* &\ge \theta(A + C) \\ 1 + 2\theta \frac{D\Delta t}{(\Delta x)^2} &\ge 2\theta \frac{D\Delta t}{(\Delta x)^2} \\ 1 &\ge 0 \quad \checkmark \end{aligned}$$

Thus the Thomas algorithm is stable for this problem.

Plots

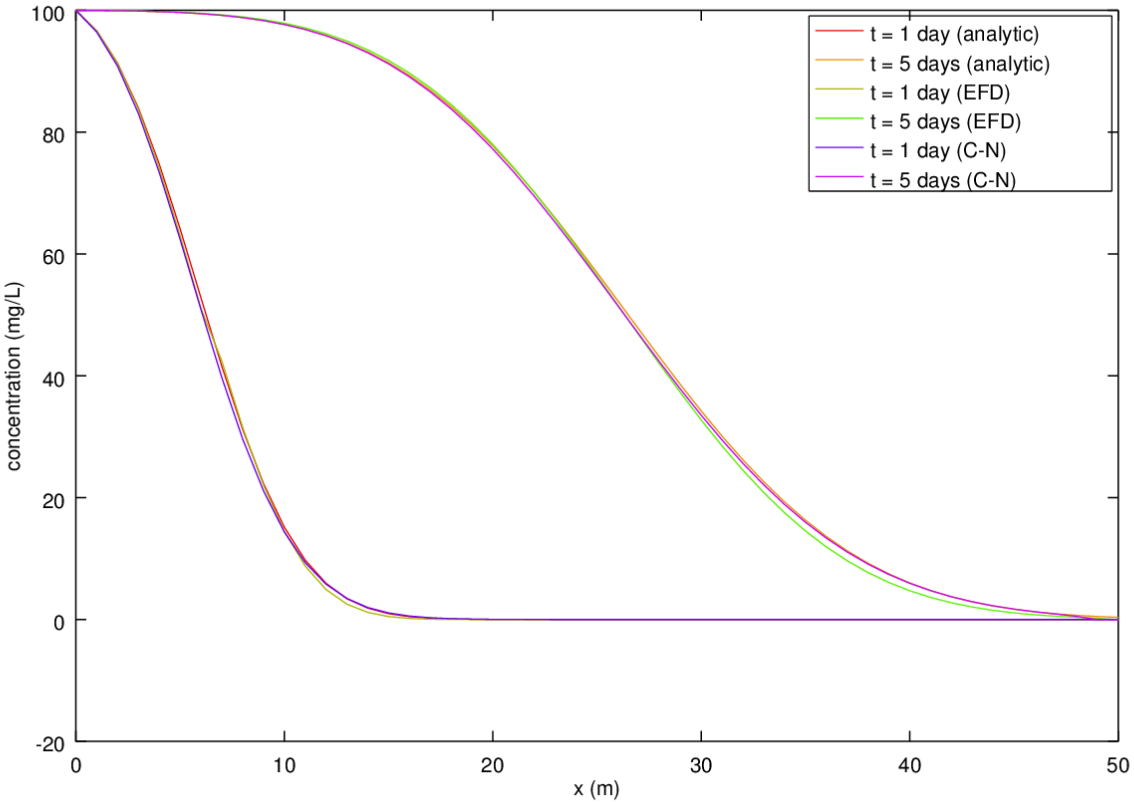

Figure 1: Analytic, EFD, and Crank-Nicolson with $D = 8.0\text{m}^2/\text{day}$

Crank-Nicolson Data

```
(1)  00.00 00.00 01.00 02.00 03.00 04.00 05.00 06.00 07.00 08.00 09.00 10.00 11.00 ...
(2)  01.00 100.0 96.44 90.82 83.06 73.39 62.40 50.89 39.72 29.63 21.11 14.36 09.34 ...
(3)  05.00 100.0 99.98 99.94 99.88 99.78 99.65 99.45 99.18 98.80 98.31 97.67 96.85 ...

(1)  12.00 13.00 14.00 15.00 16.00 17.00 18.00 19.00 20.00 21.00 22.00 23.00 24.00 ...
(2)  05.80 03.45 01.97 01.08 00.57 00.29 00.14 00.07 00.03 00.01 00.01 00.00 00.00 ...
(3)  95.82 94.56 93.03 91.21 89.08 86.61 83.81 80.68 77.22 73.45 69.42 65.16 60.71 ...

(1)  25.00 26.00 27.00 28.00 29.00 30.00 31.00 32.00 33.00 34.00 35.00 36.00 37.00 ...
(2)  00.00 00.00 00.00 00.00 00.00 00.00 00.00 00.00 00.00 00.00 00.00 00.00 00.00 ...
(3)  56.14 51.50 46.86 42.28 37.81 33.52 29.44 25.62 22.09 18.86 15.95 13.36 11.07 ...

(1)  38.00 39.00 40.00 41.00 42.00 43.00 44.00 45.00 46.00 47.00 48.00 49.00 50.00
(2)  00.00 00.00 00.00 00.00 00.00 00.00 00.00 00.00 00.00 00.00 00.00 -0.00 00.00
(3)  09.09 07.39 05.95 04.74 03.73 02.91 02.25 01.71 01.28 00.90 00.52 -0.02 00.00
```

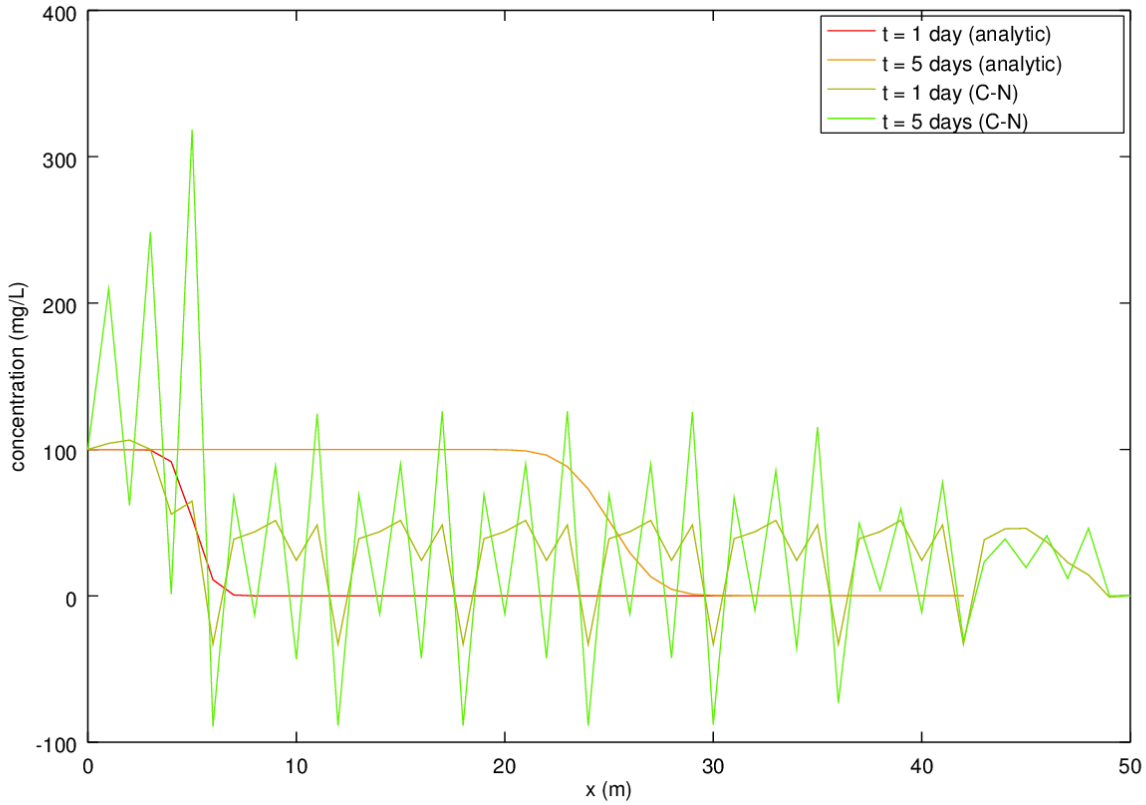

Figure 2: Analytic and Crank-Nicolson with $D = 0.3\text{m}^2/\text{day}$

This model has a Péclet number of

$$\text{Pe} = \frac{v\Delta x}{D} = \frac{(5\text{ m/day})(1\text{ m})}{0.3\text{ m}^2/\text{day}} = 16.67$$

This is far above the acceptable value, and as such, there are large oscillations.

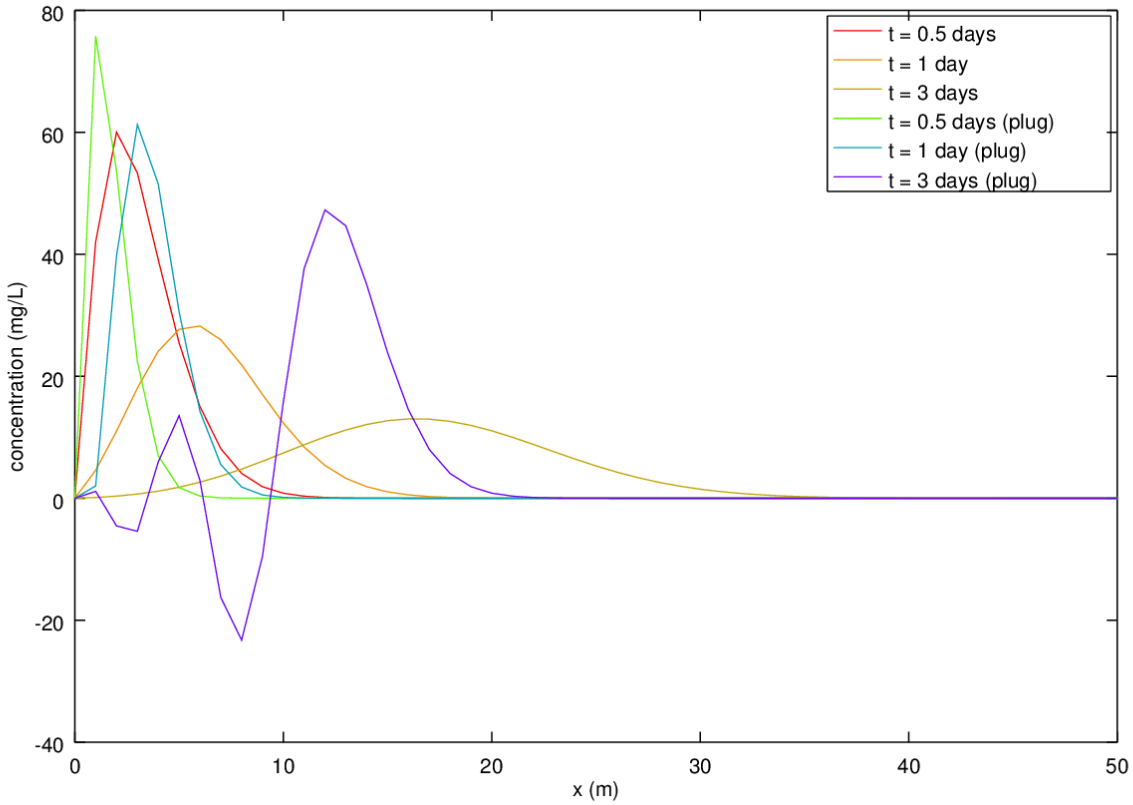

Figure 3: Crank-Nicolson Plume, $D = 8.0\text{m}^2/\text{day}$ & plug flow

gensrc/gfd_advect_dispers.f90

```
1  ! gfd_advect_dispers.f90
2  ! **WARNING: THIS CODE DOES NOT CURRENTLY WORK WITH > 1 IMAGE.**
3  ! Planning to fix this and do part 3 over the weekend.
4  program gfd_advect_dispers
5
6      ! Use Statements
7      use, intrinsic :: iso_fortran_env, only: error_unit
8
9
10     implicit none
11     ! Variable Declarations
12     real, codimension[*] :: ca, cb, cc
13     real, codimension[*] :: length, v, d
14     real, codimension[*] :: t_plume
15     real, codimension[*] :: dx, dt, theta
16     integer, codimension[*] :: tout_length
17     ! The dimensions of tout_array will be set later, at allocation.
18     real, dimension(:), codimension[:], allocatable :: tout_array
19     integer :: alloc_stat
20     character(len=80) :: alloc_errmsg
21     real, dimension(:), codimension[:], allocatable :: concentrations
22     real, dimension(:), allocatable :: next_concentrations
```

```
23      integer, codimension[*] :: next_concentrations_length
24      ! Intermediate values
25      real :: num_edges_real
26      integer :: num_edges
27      integer :: xi
28      character(len=13), parameter :: real_fmtstr = "(sp,es16.7e3)"
29      integer :: ti, tout_i, image_i
30      real :: coeff_p, coeff_r
31      real :: coeff_as, coeff_ass
32      real :: coeff_bs, coeff_bss
33      real :: coeff_cs, coeff_css
34
35
36      ! Input
37      if (this_image() == 1) then
38          read *, ca, cb, cc
39          read *, length, v, d
40          read *, t_plume
41      end if
42      if (this_image() == 1) then
43          read *, dx, dt, theta
44          read *, tout_length
45      end if
46      ! Allocate Output Times
47      call co_broadcast(tout_length, source_image=1)
48      allocate (tout_array(tout_length) [*], stat=alloc_stat, errmsg=alloc_errmsg)
49      ! Handle Allocation Error
50      if (alloc_stat > 0) then
51          write (unit=error_unit, fmt="(a)") trim(alloc_errmsg)
52          stop
53      end if
54
55
56      if (this_image() == 1) then
57          read *, tout_array
58      end if
59
60      ! Broadcast Parameters
61      call co_broadcast(ca, source_image=1)
62      call co_broadcast(cb, source_image=1)
63      call co_broadcast(cc, source_image=1)
64      call co_broadcast(length, source_image=1)
65      call co_broadcast(v, source_image=1)
66      call co_broadcast(d, source_image=1)
67      call co_broadcast(t_plume, source_image=1)
68      call co_broadcast(dx, source_image=1)
69      call co_broadcast(dt, source_image=1)
70      call co_broadcast(theta, source_image=1)
71      call co_broadcast(tout_array, source_image=1)
72
73      sync all
74
75      ! Initialize Domain
76      num_edges_real = length / dx
77      num_edges = nint(num_edges_real)
78      if (abs(num_edges_real - num_edges) > 8*epsilon(num_edges_real) .or. &
79                  num_edges < 1) then
```

```
80          write (unit=error_unit, fmt="(a)") "ERROR: L/dx must be an integer >= 1."
81          stop
82      end if
83      if (this_image() == num_images()) then
84          next_concentrations_length = (num_edges - 1) - &
85                      (num_images()-1)*(num_edges / num_images())
86      else
87          next_concentrations_length = (num_edges / num_images())
88      end if
89      allocate (concentrations(0:next_concentrations_length + 1) [*], &
90                  stat=alloc_stat, errmsg=alloc_errmsg)
91      ! Handle Allocation Error
92      if (alloc_stat > 0) then
93          write (unit=error_unit, fmt="(a)") trim(alloc_errmsg)
94          stop
95      end if
96
97      concentrations = ca
98      allocate (next_concentrations(1:next_concentrations_length), &
99                  stat=alloc_stat, errmsg=alloc_errmsg)
100     ! Handle Allocation Error
101     if (alloc_stat > 0) then
102         write (unit=error_unit, fmt="(a)") trim(alloc_errmsg)
103         stop
104     end if
105
106
107     ! Output Header
108     if (this_image() == 1) then
109         call writereal(0.0)
110         do xi = 0, num_edges
111             call writereal(xi * dx)
112         end do
113         write (unit=*, fmt="(a)", advance="yes") ""
114     end if
115
116
117     ! Calculate Coefficients
118     coeff_p = (d*dt) / (dx*dx)
119     coeff_r = (v*dt) / (2*dx)
120     coeff_as = -theta * (coeff_p + coeff_r)
121     coeff_ass = (1-theta) * (coeff_p + coeff_r)
122     coeff_bs = 1 + 2*theta*coeff_p
123     coeff_bss = 1 + 2*(theta-1)*coeff_p
124     coeff_cs = -theta * (coeff_p - coeff_r)
125     coeff_css = (1-theta) * (coeff_p - coeff_r)
126
127
128     ! Main Loop
129     !write (unit=error_unit, fmt=*) this_image(), next_concentrations_length
130     ti = 0
131     do tout_i = 1, size(tout_array)
132         call run_to(nint(tout_array(tout_i) / dt))
133         if (this_image() == 1) then
134             ! Print Record
135             call writereal(ti * dt)
136             call writereal(concentrations(0)[1])
```

```
137             do image_i = 1, num_images()
138                 do xi = 1, next_concentrations_length[image_i]
139                     call writereal(concentrations(xi)[image_i])
140                 end do
141             end do
142             xi = next_concentrations_length[num_images()] + 1
143             call writereal(concentrations(xi)[num_images()])
144             write (unit=*, fmt="(a)", advance="yes") ""
145 
146         end if
147     end do
148 
149 
150 contains
151 
152     ! Real Output Subroutine
153     subroutine writereal(r)
154         implicit none
155         real :: r
156         write (unit=*, fmt=real_fmtstr, advance="no") r
157     end subroutine writereal
158 
159     ! Run To Subroutine
160     subroutine run_to(new_ti)
161         implicit none
162         integer, intent(in) :: new_ti
163 
164         if (new_ti <= ti) then
165             return
166         end if
167         do ti = ti, new_ti - 1
168             call time_step
169             call writeback
170         end do
171         ti = new_ti
172     end subroutine run_to
173 
174     ! Time Step Subroutine
175     subroutine time_step
176         implicit none
177         integer :: i
178         real, dimension(1:next_concentrations_length - 1) :: c_prime, g_prime
179         real :: next_cb, next_cc
180 
181         ! Determine Boundary Updates
182         if (t_plume > 0 .and. (ti + 1) * dt > t_plume) then
183             next_cb = 0
184             next_cc = 0
185         else
186             next_cb = cb
187             next_cc = cc
188         end if
189 
190 
191         ! Populate C' and G'
192         c_prime(1) = coeff_cs / coeff_bs
193         g_prime(1) = (coeff_ass*concentrations(0) + coeff_bss*concentrations(1) + &
```

```
                    coeff_css*concentrations(2) - coeff_as*next_cb) / coeff_bs
        do i = 2, next_concentrations_length - 1
            c_prime(i) = coeff_cs / (coeff_bs - coeff_as*c_prime(i-1))
            g_prime(i) = (coeff_ass*concentrations(i-1) + &
                        coeff_bss*concentrations(i)+coeff_css*concentrations(i+1) &
                        - coeff_as*g_prime(i-1)) / (coeff_bs-coeff_as*c_prime(i-1))
        end do

        ! Back-substitute to solve
        i = next_concentrations_length
        next_concentrations(i) = (coeff_ass*concentrations(i-1) + &
                    coeff_bss*concentrations(i) + coeff_css*concentrations(i+1) - &
                    coeff_cs*next_cc + coeff_as*g_prime(i-1)) / (coeff_bs - &
                    coeff_as*c_prime(i-1))
        do i = (next_concentrations_length - 1), 1, -1
            next_concentrations(i) = g_prime(i)-c_prime(i)*next_concentrations(i+1)
        end do
    end subroutine time_step

    ! Writeback Subroutine
    subroutine writeback
        implicit none
        real :: next_cb, next_cc
        integer :: i
        do i = 1, next_concentrations_length
            concentrations(i) = next_concentrations(i)
        end do

        ! Sync Boundaries
        sync all
        ! Determine Boundary Updates
        if (t_plume > 0 .and. (ti + 1) * dt > t_plume) then
            next_cb = 0
            next_cc = 0
        else
            next_cb = cb
            next_cc = cc
        end if

        if (this_image() == 1) then
            concentrations(0) = next_cb
        else
            concentrations(0) = concentrations( &
                    next_concentrations_length[this_image() - 1])[this_image() - 1]
        end if
        if (this_image() == num_images()) then
            concentrations(next_concentrations_length + 1) = next_cc
        else
            concentrations(next_concentrations_length + 1) = &
                        concentrations(1)[this_image() + 1]
        end if
        sync all

    end subroutine writeback


end program gfd_advect_dispers
```