Geology 575 Homework 4

Chris McKinney

This 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.

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

The source is available via git at https://ttech.click/geol575hw3.git (branch `hw5`).

The source is also available at https://ttech.click/geol575hw5.tar.gz with the pre-compiled PDF (the PDF generation dependencies are many).

Compiling the target executables requires Literate (http://literate.zbyedidia.webfactional.com/), Python 3 (https://python.org/), and Bottle (https://bottlepy.org/).

Contents

General Finite Difference Advection-Dispersion

Chris McKinney

General Finite Difference Advection-Dispersion

1. Introduction

This Fortran program runs a general finite difference model of the advection-dispersion equation. It uses coarrays to allow for parallelization.

WARNING: THIS CODE DOES NOT CURRENTLY WORK WITH > 1 IMAGE.

Planning to fix this and do part 3 over the weekend.

2. 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,

$$\begin{bmatrix} B^* & -\theta C & & & \\ -\theta A & B^* & -\theta C & & \\ & . & . & . & \\ & & -\theta A & B^* & -\theta C \\ & & & -\theta A & B^* \end{bmatrix} \begin{bmatrix} c_1(t+\Delta t) \\ c_2(t+\Delta t) \\ . \\ c_{\ell-1}(t+\Delta t) \\ c_\ell(t+\Delta t) \end{bmatrix} = \begin{bmatrix} g_1(t) + \theta Ac_0(t+\Delta t) \\ g_2(t) \\ . \\ g_{\ell-1}(t) \\ g_\ell(t) + \theta Cc_{\ell+1}(t+\Delta t) \end{bmatrix}$$

3. 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

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

$$G'_i(t) = \begin{cases} \frac{g_1(t) + \theta Ac_0(t+\Delta t)}{B^*} & ;\ i = 1 \\ \frac{g_i(t) + \theta AG'_{i-1}}{B^* + \theta AC'_{i-1}(t)} & ;\ i = 2, 3, \ldots, \ell - 1 \\ \frac{g_\ell(t) + \theta Cc_{\ell+1}(t+\Delta t) + \theta AG'_{\ell-1}}{B^* + \theta AC'_{\ell-1}(t)} & ;\ i = \ell \end{cases}$$

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

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

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

$$B^* \ge \theta(A + C)$$

$$1 + 2\theta\frac{D\Delta t}{(\Delta x)^2} \ge 2\theta\frac{D\Delta t}{(\Delta x)^2}$$

$$1 \ge 0 \quad \checkmark$$

Thus the Thomas algorithm is stable for this problem.

4. Environment Parameters

The parameters of the environment are (note the addition of t_{plume}):

{Variable Declarations 4}

```
real, codimension[*] :: ca, cb, cc
real, codimension[*] :: length, v, d
real, codimension[*] :: t_plume
```

Added to in sections 5, 6, 8, 9, 10, 11 and 13 Used in section 16

where ($c(x,t) = c_{x/dx}(t),\ x/dx \in \mathbb{Z}$. dx is declared in the next section.):

Variable	Units	Definition
`ca`	mg/L	$c_a = c(x,0),\ 0 \le x \le L$
`cb`	mg/L	$c_b = c(0,t),\ t > 0$
`cc`	mg/L	$c_c = c(L,t),\ t > 0$
`length`	meters	L is the length to model.
`v`	m/day	v is the average linear flow velocity.
`d`	m^2/day	D is the dispersion coefficient.
`t_plume`	days	t_{plume} is the length of the plume; forever if < 0.

These parameters are read in as two records of three values each:

{Input 4}

```
if (this_image() == 1) then
    read *, ca, cb, cc
    read *, length, v, d
    read *, t_plume
end if
```

Added to in section 5 Used in section 16

5. Control Parameters

The program also takes control parameters (note the addition of θ):

{Variable Declarations 4} +=

```
real, codimension[*] :: dx, dt, theta
integer, codimension[*] :: tout_length
! The dimensions of tout_array will be set later, at allocation.
real, dimension(:), codimension[:], allocatable :: tout_array
```

Added to in sections 6, 8, 9, 10, 11 and 13 Used in section 16

where:

Variable	Units	Definition
`dx`	meters	$dx = \Delta x$, the spatial discretization.
`dt`	days	$dt = \Delta t$, the temporal discretization.
`theta`		θ is the model's implicitness.
`tout_length`		Length of `tout_array`
`tout_array`	days	Times at which to output

$t_{\max}$ is the last element of `tout_array`.

These parameters are read in as three records. The first is three reals, the second is just `tout_length` (an integer), and the third is `tout_length` reals:

{Input 4} +=

```
if (this_image() == 1) then
    read *, dx, dt, theta
    read *, tout_length
end if
{Allocate Output Times, 6}
if (this_image() == 1) then
    read *, tout_array
end if
```

Used in section 16

6. Output Times Allocation

All images have to allocate the array at the same time:

{Allocate Output Times 6}

```
call co_broadcast(tout_length, source_image=1)
allocate (tout_array(tout_length) [*], stat=alloc_stat, errmsg=alloc_errmsg)
{Handle Allocation Error, 6}
```

Used in section 5

Handling allocation errors necessitates two more variables:

{Variable Declarations 4} +=

```
integer :: alloc_stat
character(len=80) :: alloc_errmsg
```

Added to in sections 5, 8, 9, 10, 11 and 13 Used in section 16

If an error does occur, output the error message to standard error and exit:

{Handle Allocation Error 6}

```
if (alloc_stat > 0) then
    write (unit=error_unit, fmt="(a)") trim(alloc_errmsg)
    stop
end if
```

Used in sections 8 and 8

To output to standard error, the `error_unit` number is needed:

{Use Statements 6}

```
use, intrinsic :: iso_fortran_env, only: error_unit
```

Used in section 16

7. Broadcasting Parameters

This broadcasts all the parameters (except `tout_length`, which has already been broadcast) from the first image (which reads the values) to all the others:

{Broadcast Parameters 7}

```
call co_broadcast(ca, source_image=1)
call co_broadcast(cb, source_image=1)
call co_broadcast(cc, source_image=1)
call co_broadcast(length, source_image=1)
call co_broadcast(v, source_image=1)
call co_broadcast(d, source_image=1)
call co_broadcast(t_plume, source_image=1)
call co_broadcast(dx, source_image=1)
call co_broadcast(dt, source_image=1)
call co_broadcast(theta, source_image=1)
call co_broadcast(tout_array, source_image=1)
```

Used in section 16

8. Splitting the Domain

This section and the following four (through "Simulating to a Particular Time Step" are unchanged from forward FD.

For one image to handle the full length L with discretization Δx, it would need an array of size $L/\Delta x + 1$ to represent the concentrations (including the boundaries). We will assume L and Δx are such that their quotient is an integer. To describe the splitting method among n images, I will use an example ($L = 14,\ \Delta x = 1,\ n = 3$):

```
|BB   image 1   BB|     |BB    image 3     BB|
 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14
            |BB   image 2   BB|
```

`BB` indicates that the corresponding element is treated as a boundary condition for that image. We essentially have here three separate models. The only communication between the models is that at each time step, the boundary values of each model are updated to the corresponding "active" values in the image's neighbors. Each image 1 through $n-1$ needs an array of size $\lfloor L/(n\Delta x) \rfloor + 2$, and image n needs an array of size $(L/\Delta x + 1) - (n-1)\lfloor L/(n\Delta x) \rfloor$. In light of this:

{Variable Declarations 4} +=

```
real, dimension(:), codimension[:], allocatable :: concentrations
real, dimension(:), allocatable :: next_concentrations
integer, codimension[*] :: next_concentrations_length
! Intermediate values
real :: num_edges_real
integer :: num_edges
```

Added to in sections 5, 6, 9, 10, 11 and 13 Used in section 16

Before initializing the domain arrays, the program checks that $L/\Delta x$ is an integer greater than 0. If it is not, it stops:

{Initialize Domain 8}

```
num_edges_real = length / dx
num_edges = nint(num_edges_real)
if (abs(num_edges_real - num_edges) > 8*epsilon(num_edges_real) .or. &
            num_edges < 1) then
    write (unit=error_unit, fmt="(a)") "ERROR: L/dx must be an integer >= 1."
    stop
end if
```

Added to in section 8 Used in section 16

Then the size of the `next_concentrations` array is calculated (2 less than `concentrations`, since it excludes the boundaries. Note that the division of positive integers is equivalent to floor of division:

{Initialize Domain 8} +=

```
if (this_image() == num_images()) then
    next_concentrations_length = (num_edges - 1) - &
                (num_images()-1)*(num_edges / num_images())
else
    next_concentrations_length = (num_edges / num_images())
end if
```

Added to in section 8 Used in section 16

Finally, the arrays are allocated and initialized. Note the difference in starting index. This is so that the indexes directly correspond:

{Initialize Domain 8} +=

```
allocate (concentrations(0:next_concentrations_length + 1) [*], &
            stat=alloc_stat, errmsg=alloc_errmsg)
{Handle Allocation Error, 6}
concentrations = ca
allocate (next_concentrations(1:next_concentrations_length), &
            stat=alloc_stat, errmsg=alloc_errmsg)
{Handle Allocation Error, 6}
```

Added to in section 8 Used in section 16

9. Table Header Output

{Variable Declarations 4} +=

```
integer :: xi
```

Added to in sections 5, 6, 8, 10, 11 and 13 Used in section 16

This just outputs the header for the value table:

{Output Header 9}

```
if (this_image() == 1) then
    call writereal(0.0)
    do xi = 0, num_edges
        call writereal(xi * dx)
    end do
    write (unit=*, fmt="(a)", advance="yes") ""
end if
```

Used in section 16

10. Outputting Reals

This just makes writing reals a bit less wordy. For information on the output format, see https://tachibanatech.com/litdoc/hw3.full.d/_book/a_heat_conduct.html#1:5

{Variable Declarations 4} +=

```
character(len=13), parameter :: real_fmtstr = "(sp,es16.7e3)"
```

Added to in sections 5, 6, 8, 9, 11 and 13 Used in section 16

{Real Output Subroutine 10}

```
subroutine writereal(r)
    implicit none
    real :: r
    write (unit=*, fmt=real_fmtstr, advance="no") r
end subroutine writereal
```

Used in section 16

11. Main Loop

{Variable Declarations 4} +=

```
integer :: ti, tout_i, image_i
```

Added to in sections 5, 6, 8, 9, 10 and 13 Used in section 16

The main loop basically just "fast-forwards" to each requested output time, then prints a record of the concentrations:

{Main Loop 11}

```
!write (unit=error_unit, fmt=*) this_image(), next_concentrations_length
ti = 0
do tout_i = 1, size(tout_array)
    call run_to(nint(tout_array(tout_i) / dt))
    if (this_image() == 1) then
        {Print Record, 11}
    end if
end do
```

Used in section 16

This outputs the concentration at each node:

{Print Record 11}

```
call writereal(ti * dt)
call writereal(concentrations(0)[1])
do image_i = 1, num_images()
    do xi = 1, next_concentrations_length[image_i]
        call writereal(concentrations(xi)[image_i])
    end do
end do
xi = next_concentrations_length[num_images()] + 1
call writereal(concentrations(xi)[num_images()])
write (unit=*, fmt="(a)", advance="yes") ""
```

12. Simulating to a Particular Time Step

This subroutine just calls `time_step` and `writeback` until it has reached a particular time. Note that on entry to `time_step`, `ti` should be the index of the time *already calculated*. `time_step` calculates the values for time `ti + 1`.

{Run To Subroutine 12}

```
subroutine run_to(new_ti)
    implicit none
    integer, intent(in) :: new_ti

    if (new_ti <= ti) then
        return
    end if
    do ti = ti, new_ti - 1
        call time_step
        call writeback
    end do
    ti = new_ti
end subroutine run_to
```

Used in section 16

13. Calculating Coefficients

Note that this is executed before the main loop starts.

{Variable Declarations 4} +=

```
real :: coeff_p, coeff_r
real :: coeff_as, coeff_ass
real :: coeff_bs, coeff_bss
real :: coeff_cs, coeff_css
```

Added to in sections 5, 6, 8, 9, 10 and 11 Used in section 16

These are the coefficients used for calculating time steps ($A^*, A^{**}, B^*, B^{**}, C^*, C^{**}$)

{Calculate Coefficients 13}

```
coeff_p = (d*dt) / (dx*dx)
coeff_r = (v*dt) / (2*dx)
coeff_as = -theta * (coeff_p + coeff_r)
coeff_ass = (1-theta) * (coeff_p + coeff_r)
coeff_bs = 1 + 2*theta*coeff_p
coeff_bss = 1 + 2*(theta-1)*coeff_p
coeff_cs = -theta * (coeff_p - coeff_r)
coeff_css = (1-theta) * (coeff_p - coeff_r)
```

Used in section 16

14. Stepping through Time

This subroutine calculates concentrations at time `ti + 1`. Note that it does *not* update `ti`: this is the responsibility of the caller. Neither does it copy the values it puts in `next_concentrations` back into `concentrations`. For this copying, use `writeback`.

The equations for this step are (see “Equations” and “Suitability of the Thomas Algorithm”)

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

{Time Step Subroutine 14}

```
subroutine time_step
    implicit none
    integer :: i
    real, dimension(1:next_concentrations_length - 1) :: c_prime, g_prime
    real :: next_cb, next_cc

    {Determine Boundary Updates, 14}

    ! Populate C' and G'
    c_prime(1) = coeff_cs / coeff_bs
    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
```

Used in section 16

After the plume is over,

{Determine Boundary Updates 14}

```
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
```

Used in section 15

15. Write Back Concentrations

This subroutine copies `next_concentrations` back into `concentrations` and also syncs the boundaries for each image.

{Writeback Subroutine 15}

```
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, 15}
end subroutine writeback
```

Used in section 16

To sync boundaries, the "left" boundary of each image is set to the last "active" concentration in the previous image (except for on the first image, on which the "left" boundary is a real boundary condition), and the "right" boundary is set to the first "active" concentration in the next image (except for on the lat image, on which the "right" boundary is a real boundary condition).

{Sync Boundaries 15}

```
sync all
{Determine Boundary Updates, 14}
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
```

16. File Outline

{**gfd_advect_dispers.f90** 16}

```
! **WARNING: THIS CODE DOES NOT CURRENTLY WORK WITH > 1 IMAGE.**
! Planning to fix this and do part 3 over the weekend.
program gfd_advect_dispers

    {Use Statements, 6}

    implicit none
    {Variable Declarations, 4}

    {Input, 4}
    {Broadcast Parameters, 7}
    sync all

    {Initialize Domain, 8}
    {Output Header, 9}

    {Calculate Coefficients, 13}

    {Main Loop, 11}

contains

    {Real Output Subroutine, 10}
    {Run To Subroutine, 12}
    {Time Step Subroutine, 14}
    {Writeback Subroutine, 15}

end program gfd_advect_dispers
```

Analytical Advection-Dispersion

Chris McKinney

Analytical Advection-Dispersion

1. Introduction

This is the analytical advection-dispersion model from last week. I've modified it slightly to allow it to read parameters and also to make the code slightly less terrible (though the main focus this week is on the *numerical* model).

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.

For the initial and boundary conditions

$c(x,0) = 0$	$0 \le x \le \infty$	IC
$c(0,t) = c_o$	$t \ge 0$	BC
$c(\infty,t) = 0$	$t \ge 0$	BC

the analytical solution is

$$c = \frac{c_o}{2}\left(\operatorname{erfc}\left(\frac{x - vt}{2\sqrt{Dt}}\right) + e^{vx/D}\operatorname{erfc}\left(\frac{x + vt}{2\sqrt{Dt}}\right)\right)$$

2. Parameters

The variables that are not arguments to c are stored as module variables:

{Variable Declarations 2}

```
real :: co, v, d
```

Added to in section 4 Used in section 6

`co` is a (unitless) concentration. `v` is in meters per day. `d` is in square meters per day.

These are read in as a record from standard input in standard Fortran format:

{Input 2}

```
read *, co, v, d
```

Used in section 6

The program also takes control parameters: start, interval, and total for x (meters) and list of t values (days):

{Variable Declarations 2} +=

```
real :: xs, xint, xt
integer :: tlength
real, dimension(:), allocatable :: tarray
```

```
integer :: alloc_stat
character(len=80) :: alloc_errmsg
```

Added to in section 4 Used in section 6

These are read as three records. The first reads the control parameters for x. The second is the number of t values that will be provided, and the third is the t values:

{Input 2} +=

```
read *, xs, xint, xt
read *, tlength
allocate (tarray(tlength), stat=alloc_stat, errmsg=alloc_errmsg)
call errstop(alloc_stat, alloc_errmsg);
read *, tarray
```

Used in section 6

3. Concentration Function

The implementation is fairly straightforward, basically just transcribing the initial condition and concentration formula.

{Concentration Function 3}

```
function c(x,t) result(c_r)
    implicit none
    real, intent(in) :: x, t
    real :: c_r

    if (x == 0) then
        c_r = co
    else
        c_r = co/2 * (erfc((x - v*t)/(2*sqrt(d*t))) &
                + exp(v*x/d)*erfc((x + v*t)/(2*sqrt(d*t))))
    end if
end function c
```

Used in section 6

4. Output

The program first outputs a record with a zero and then reference x values. It then outputs records with one t value and then c values. For details on the output format, see
https://tachibanatech.com/litdoc/hw3.full.d/_book/a_heat_conduct.html#1:5

{Variable Declarations 2} +=

```
character(len=13), parameter :: real_fmtstr = "(sp,es16.7e3)"

real :: x, t
integer :: xi, ti
```

Added to in section 2 Used in section 6

This loop outputs the first record of reference x values:

{Output 4}

```
write (unit=*, fmt=real_fmtstr, advance="no") 0.0
do xi = 0, floor((xt + epsilon(xt)) / (xint - epsilon(xint)))
    x = xs + real(xi)*xint
    write (unit=*, fmt=real_fmtstr, advance="no") x
end do
write (unit=*, fmt="(a)", advance="yes") ""
```

Used in section 6

This loop outputs all the subsequent records:

{Output 4} +=

```
do ti = 1, size(tarray)
    t = tarray(ti)
    call at_time(t, xs, xint, xt)
end do
```

Used in section 6

This is the subroutine that outputs a single record for a single point in time:

{Record Output Function 4}

```
subroutine at_time(t, xs, xint, xt)
    implicit none
    real :: t, xs, xint, xt

    write (unit=*, fmt=real_fmtstr, advance="no") t
    do xi = 0, floor(xt / xint)
        x = xs + xi*xint
        write (unit=*, fmt=real_fmtstr, advance="no") c(x,t)
    end do
    write (unit=*, fmt="(a)", advance="yes") ""
end subroutine at_time
```

Used in section 6

5. Error Handling

This subroutine prints an error message and stops the program if an error has occurred.

{Stop On Error 5}

```
subroutine errstop(stat, errmsg)
    integer :: stat
    character(len=*) errmsg

    if (stat > 0) then
        print *, trim(errmsg)
        stop
    end if
end subroutine errstop
```

Used in section 6

6. Code

This is just the basic outline of the file.

{**a_advect_dispers.f90** 6}

```
program a_advect_dispers

    implicit none
    {Variable Declarations, 2}
    {Input, 2}
    {Output, 4}

contains

    {Record Output Function, 4}
    {Concentration Function, 3}
    {Stop On Error, 5}

endprogram
```

7. Plots

Plotter

Chris McKinney

Plotter

1. Introduction

This Octave script (Yes, Octave specifically. From my googling, it seems MATLAB doesn't have a reasonable way to handle command line arguments. Octave has `argv`.) plots output from the Fortran programs.

2. Command-Line Argument Handling

The script takes $4p + c$ command line arguments, where c is the total number of curves to plot, and p is the number of separate plots on which to plot them. Usage:

```
plotter.m [ ( XPLOT | YPLOT ) <filename> <xlabel> <ylabel> [ <legend> ]... ]...
```

- `XPLOT` and `YPLOT` starts a new plot with the specified dependent axis.
- `<filename>` is the image file to output to.
- `<xlabel>` and `<ylabel>` specify the axis labels.
- `<legend>` is a legend entry. `NOP` will skip records.

{Usage 2}

```
function usage_and_exit
    fprintf(stderr(), '%s [ ( XPLOT | YPLOT ) <filename> ', program_name());
    fprintf(stderr(), '<xlabel> <ylabel> [ <legend> ]... ]...\n\n');
    fdisp(stderr(), '`XPLOT` and `YPLOT` starts a new plot with the specified dependent axis.');
    fdisp(stderr(), '<filename> is the image file to output to.');
    fdisp(stderr(), '<xlabel> and <ylabel> specify the axis labels.');
    fdisp(stderr(), '<legend> is a legend entry. `NOP` will skip records.');
    exit(1);
endfunction
```

Used in section 4

Here the script loops through the arguments and builds up the figures as the relevant arguments are read. If there is a fatal error later in the argument list, plots earlier in the list should be unaffected, since they have already been output. `read_fortran_line` is implemented in the next section.

{Argument Handling 2}

```
args = argv(); % Cell array of arguments
independent = read_fortran_line(); % Read independent values
i_in_plot = 1; % Keeps track of which field we're on
dependent_axis = ''; % 'X' or 'Y'
filename = '';
x_label = '';
y_label = '';
legend_array = {}; % Cell array of legend items
for i = 1:(nargin+1)
    if i == nargin+1 || strcmp(args{i}, 'XPLOT') || strcmp(args{i}, 'YPLOT')

        {Prepare New Figure, 2}
```

```
        elseif i == 1

            fdisp(stderr(), 'ERROR: The first argument must be either XPLOT or YPLOT.');
            usage_and_exit();

        elseif i_in_plot == 2

            filename = args{i};

        elseif i_in_plot == 3

            x_label = args{i};

        elseif i_in_plot == 4

            y_label = args{i};

        else

            {Plot Curve, 2}

        endif

        i_in_plot = i_in_plot + 1;
    end
```

Used in section 4

When a new figure is created, the previous one is saved if it exists, and the figure-specific variables are reset.

{Prepare New Figure 2}

```
    if i > 1
        if i_in_plot < 4
            fdisp(stderr(), 'ERROR: Early termination of plot specification.');
        endif
        % Apply saved figure adornments
        xlabel(x_label);
        ylabel(y_label);
        legend(legend_array);
        printf('SAVED %s\n', filename);
        saveas(gcf(), filename); % Output current figure to file
    endif
    if i == nargin+1
        break % End of arguments
    endif
    % Create new invisible plot, set current
    figure('visible', 'off', 'paperposition', [0.25, 2.5, 9.0, 6.0]);
    hold on;
    % Reset for new plot
    i_in_plot = 1;
    dependent_axis = args{i}(1); % First character of the arg ('X' or 'Y')
    filename = '';
    legend_array = {};
```

Plotting a curve is as simple as reading the dependent values and calling `plot`.

{Plot Curve 2}

```
if strcmp(args{i}, 'NOP')
    read_fortran_line(); % Skip
else
    legend_array{length(legend_array) + 1} = args{i}; % Append to legend
    dependent = read_fortran_line(); % Read dependent values
    %color = prism(i_in_plot - 4)((i_in_plot - 4),1:3);
    color = sqrt(rainbow(8)(mod(i_in_plot-5, 8) + 1, 1:3) * 0.9);
    yellowness = color(1) + color(2) - color(3) - 0.5;
    if yellowness > 1
        color /= yellowness;
    endif
    if dependent_axis == 'X'
        plot(dependent, independent, 'color', color);
    else
        plot(independent, dependent, 'color', color);
    endif
endif
```

3. Standard Input Handling

The script takes in $c+1$ lines of space-separated reals in scientific notation. The first is for the independent variable array, and the subsequent lines are for the dependent variable arrays to plot. **Important:** the first value of each line is ignored.

{Input Handler 3}

```
function values = read_fortran_line
    line = fgetl(stdin()); % Retrieve the next line of standard input
    values = sscanf(line, '%e'); % Find all reals in line
    values = values(2:length(values)); % Remove first value
endfunction
```

Used in section 4

4. File Outline

{**plotter.m** 4}

{Usage, 2}

{Input Handler, 3}

{Argument Handling, 2}

Image Generation Script

Chris McKinney

Image Generation Script

1. Code

This is just a little script to generate the plots.

{**launcher.sh** 1}

```
cd "$(dirname "$0")/.."
## Analytic
#printf '1 0.05 0.5\n 0 1 200\n 3\n 365 1280 1825\n' \
#    | bin/a_advect_dispers | script/plotter.m \
#        YPLOT lit/images/hw3-2.png 'x (m)' 'concentration' \
#            't = 1 yr' 't = 3.5 yr' 't = 5 yr'
#printf '100.0 5.0 8.0\n 0 1 50\n 3\n 1 3 5\n' \
#    | bin/a_advect_dispers | tee lit/images/a_ad.data | script/plotter.m \
#        YPLOT lit/images/hw4a.png 'x (m)' 'concentration (mg/L)' \
#            't = 1 day' 't = 3 days' 't = 5 days'
## Finite Difference
#for dt in 0.05 0.1 0.025 0.0125 0.00625; do
#    ( cat lit/images/a_ad.data \
#            && ( printf "0 100.0 0\n 50 5.0 8.0\n 1.0 $dt\n 3\n 1 3 5\n" \
#                | cafrun -np 8 bin/fd_advect_dispers \
#                | tee "lit/images/fd_ad_$dt.data" ) ) \
#        | script/plotter.m \
#            YPLOT "lit/images/fd_ad_$dt.png" 'x (m)' 'concentration (mg/L)' \
#                't = 1 day (analytic)' 't = 3 days (analytic)' \
#                    't = 5 days (analytic)' NOP \
#                't = 1 day (EFD)' 't = 3 days (EFD)' 't = 5 days (EFD)'
#done

## Homework 5 ##
## D = 8.0 m^2/day
# Analytic
printf '100.0 5.0 8.0\n 0.0 1.0 50.0\n 2\n 1 5\n' \
    | bin/a_advect_dispers > lit/images/a_ad.data
# Explicit Finite Difference
printf "0.0 100.0 0.0\n 50.0 5.0 8.0\n -1\n 1.0 0.05 0.0\n 2\n 1 5\n" \
    | cafrun -np 8 bin/gfd_advect_dispers > lit/images/efd_ad.data
# Crank-Nicolson Finite Difference
printf "0.0 100.0 0.0\n 50.0 5.0 8.0\n -1\n 1.0 0.05 0.5\n 2\n 1 5\n" \
    | cafrun -np 1 bin/gfd_advect_dispers > lit/images/cn_ad.data
# Plot
cat lit/images/a_ad.data lit/images/efd_ad.data lit/images/cn_ad.data \
    | script/plotter.m \
        YPLOT 'lit/images/d_8.0.png' 'x (m)' 'concentration (mg/L)' \
            't = 1 day (analytic)' 't = 5 days (analytic)' \
            NOP 't = 1 day (EFD)' 't = 5 days (EFD)' \
            NOP 't = 1 day (C-N)' 't = 5 days (C-N)'
```

```
## D = 0.3 m^2/day
# Analytic
printf '100.0 5.0 0.3\n 0.0 1.0 50.0\n 2\n 1 5\n' \
    | bin/a_advect_dispers > lit/images/a_ad_lowd.data
# Crank-Nicolson Finite Difference
printf "0.0 100.0 0.0\n 50 5.0 0.3\n -1\n 1.0 0.05 0.5\n 2\n 1 5\n" \
    | cafrun -np 8 bin/gfd_advect_dispers > lit/images/cn_ad_lowd.data
# Plot
cat lit/images/a_ad_lowd.data lit/images/cn_ad_lowd.data \
    | script/plotter.m \
        YPLOT 'lit/images/d_0.3.png' 'x (m)' 'concentration (mg/L)' \
            't = 1 day (analytic)' 't = 5 days (analytic)' \
            NOP 't = 1 day (C-N)' 't = 5 days (C-N)'
## Plume
# Crank-Nicolson Finite Difference
printf "0.0 100.0 0.0\n 50.0 5.0 8.0\n 0.4\n 1.0 0.05 0.5\n 3\n 0.5 1 3\n" \
    | cafrun -np 1 bin/gfd_advect_dispers > lit/images/cn_ad_plume.data
printf "0.0 100.0 0.0\n 50.0 5.0 0.0\n 0.4\n 1.0 0.05 0.5\n 3\n 0.5 1 3\n" \
    | cafrun -np 1 bin/gfd_advect_dispers > lit/images/cn_ad_plug.data
# Plot
cat lit/images/cn_ad_plume.data lit/images/cn_ad_plug.data \
    | script/plotter.m \
        YPLOT 'lit/images/d_plume.png' 'x (m)' 'concentration (mg/L)' \
            't = 0.5 days' 't = 1 day' 't = 3 days' NOP \
            't = 0.5 days (plug)' 't = 1 day (plug)' 't = 3 days (plug)'
```

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.

23 hw5.simple.markdown

Plots

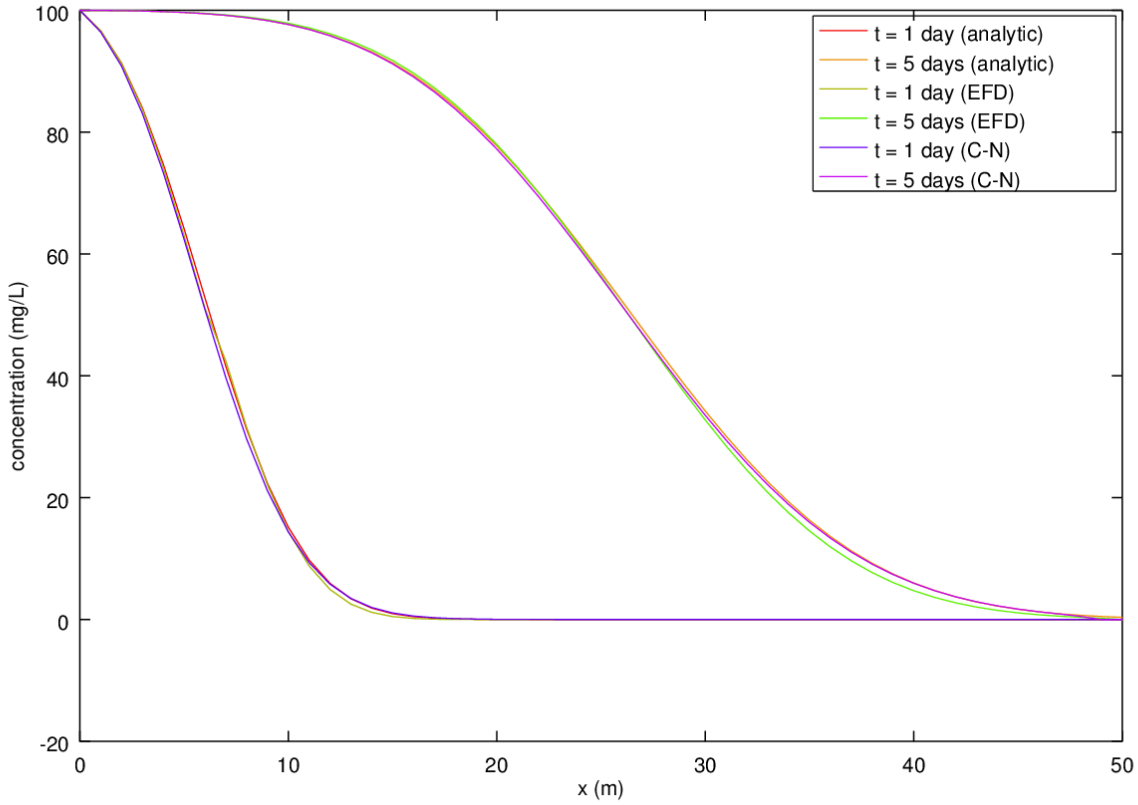

Figure 1: Analytic, EFD, and Crank-Nicolson with $D = 8.0\text{m}^2/\text{day}$

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
```

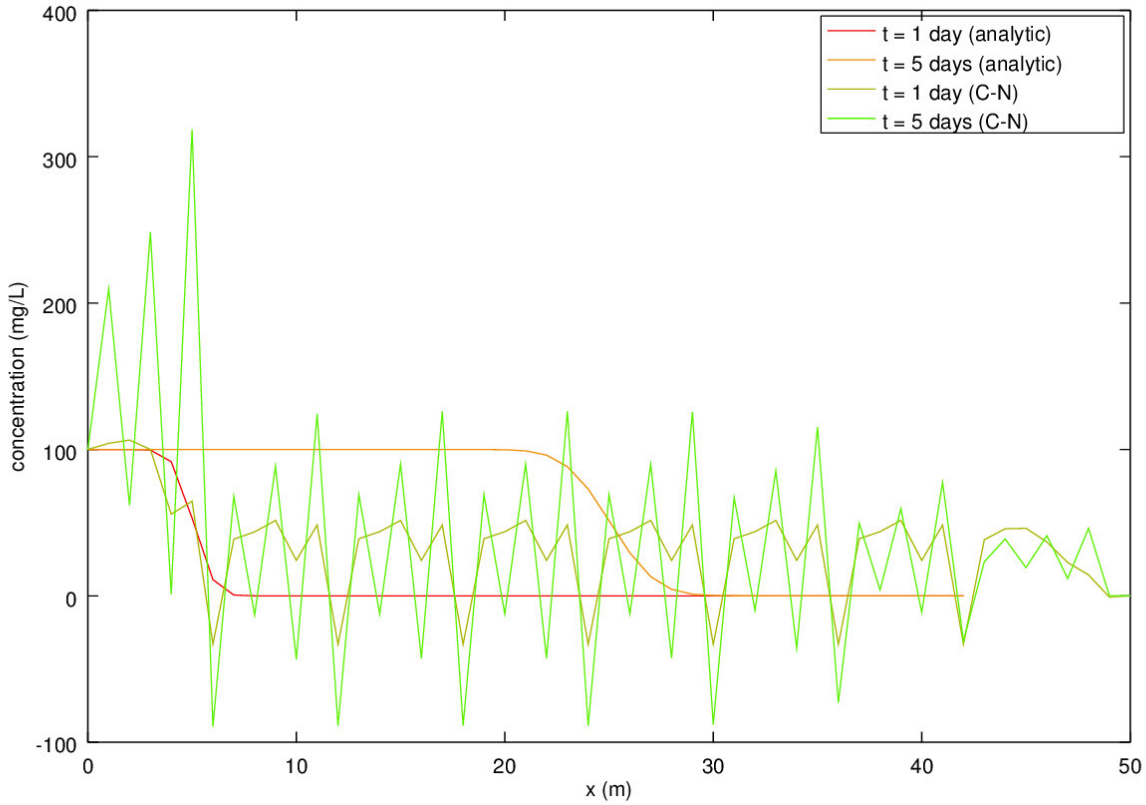

Figure 2: Analytic and Crank-Nicolson with $D = 0.3\text{m}^2/\text{day}$

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.

25 hw5.simple.markdown

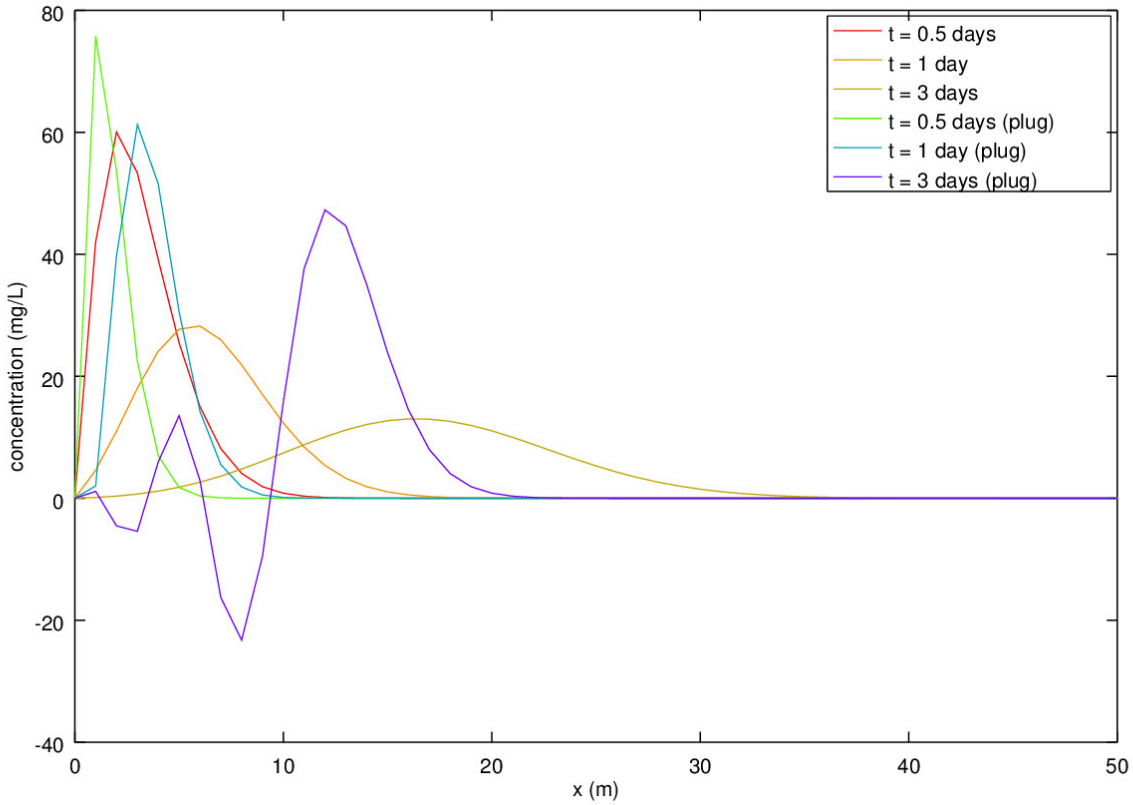

Figure 3: Crank-Nicolson Plume, $D = 8.0\text{m}^2/\text{day}$ & plug flow

gensrc/gfd_advect_dispers.f90

```
! gfd_advect_dispers.f90
! **WARNING: THIS CODE DOES NOT CURRENTLY WORK WITH > 1 IMAGE.**
! Planning to fix this and do part 3 over the weekend.
program gfd_advect_dispers

    ! Use Statements
    use, intrinsic :: iso_fortran_env, only: error_unit


    implicit none
    ! Variable Declarations
    real, codimension[*] :: ca, cb, cc
    real, codimension[*] :: length, v, d
    real, codimension[*] :: t_plume
    real, codimension[*] :: dx, dt, theta
    integer, codimension[*] :: tout_length
    ! The dimensions of tout_array will be set later, at allocation.
    real, dimension(:), codimension[:], allocatable :: tout_array
    integer :: alloc_stat
    character(len=80) :: alloc_errmsg
    real, dimension(:), codimension[:], allocatable :: concentrations
    real, dimension(:), allocatable :: next_concentrations
```

26 hw5.simple.markdown

```
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
```

```
        write (unit=error_unit, fmt="(a)") "ERROR: L/dx must be an integer >= 1."
        stop
    end if
    if (this_image() == num_images()) then
        next_concentrations_length = (num_edges - 1) - &
                    (num_images()-1)*(num_edges / num_images())
    else
        next_concentrations_length = (num_edges / num_images())
    end if
    allocate (concentrations(0:next_concentrations_length + 1) [*], &
                stat=alloc_stat, errmsg=alloc_errmsg)
    ! Handle Allocation Error
    if (alloc_stat > 0) then
        write (unit=error_unit, fmt="(a)") trim(alloc_errmsg)
        stop
    end if

    concentrations = ca
    allocate (next_concentrations(1:next_concentrations_length), &
                stat=alloc_stat, errmsg=alloc_errmsg)
    ! Handle Allocation Error
    if (alloc_stat > 0) then
        write (unit=error_unit, fmt="(a)") trim(alloc_errmsg)
        stop
    end if


    ! Output Header
    if (this_image() == 1) then
        call writereal(0.0)
        do xi = 0, num_edges
            call writereal(xi * dx)
        end do
        write (unit=*, fmt="(a)", advance="yes") ""
    end if


    ! Calculate Coefficients
    coeff_p = (d*dt) / (dx*dx)
    coeff_r = (v*dt) / (2*dx)
    coeff_as = -theta * (coeff_p + coeff_r)
    coeff_ass = (1-theta) * (coeff_p + coeff_r)
    coeff_bs = 1 + 2*theta*coeff_p
    coeff_bss = 1 + 2*(theta-1)*coeff_p
    coeff_cs = -theta * (coeff_p - coeff_r)
    coeff_css = (1-theta) * (coeff_p - coeff_r)


    ! Main Loop
    !write (unit=error_unit, fmt=*) this_image(), next_concentrations_length
    ti = 0
    do tout_i = 1, size(tout_array)
        call run_to(nint(tout_array(tout_i) / dt))
        if (this_image() == 1) then
            ! Print Record
            call writereal(ti * dt)
            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) + &
```

29 hw5.simple.markdown

```
                    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
```

30 hw5.simple.markdown