Geology 575 Homework 4

Chris McKinney

This full document is available at https://tachibanatech.com/litdoc/hw4.full.pdf in PDF format and https://tachibanatech.com/litdoc/hw4.full.d/_book/ in HTML format.

A simplified document is available at https://tachibanatech.com/litdoc/hw4.simple.pdf in PDF format.

The source is available via git at https://ttech.click/geol575hw3.git (branch `hw4`).

The source is also available at https://ttech.click/geol575hw4.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

2	**Finite Difference Advection-Dispersion**
11	**Analytical Advection-Dispersion**
15	**Plotter**
18	**Image Generation Script**
19	**Info**
20	Plots
25	Performance
25	`a_advect_dispers.f90`
26	`fd_advect_dispers.f90`

Finite Difference Advection-Dispersion

Chris McKinney

Finite Difference Advection-Dispersion

1. Introduction

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

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 equation for forward finite difference is ($c_i := c_i(t)$)

$$\frac{c_i(t+\Delta t) - c_i}{\Delta t} = D\frac{c_{i+1} - 2c_i + c_{i-1}}{(\Delta x)^2} - v\frac{c_{i+1} - c_{i-1}}{2\Delta x}$$

Solving for $c_i(t + \Delta t)$,

$$c_i(t+\Delta t) = c_i + D\Delta t\frac{c_{i+1} - 2c_i + c_{i-1}}{(\Delta x)^2} - v\Delta t\frac{c_{i+1} - c_{i-1}}{2\Delta x}$$

3. Truncation Error

The order of the truncation error here is $O(\Delta t)$ in time and $O(\Delta x)$ in space. The foundation of the finite difference method is the Taylor series:

$$u(x+\Delta x) = u(x) + \Delta x\ \frac{\partial u}{\partial x}\bigg|_x + \frac{\Delta x^2}{2!}\ \frac{\partial^2 u}{\partial x^2}\bigg|_x + \frac{\Delta x^3}{3!}\ \frac{\partial^3 u}{\partial x^3}\bigg|_x + \dots$$

Rearranging the equation to solve for error,

$$\frac{u(x+\Delta x) - u(x)}{\Delta x} - \frac{\partial u}{\partial x}\bigg|_x = \frac{\Delta x}{2!}\ \frac{\partial^2 u}{\partial x^2}\bigg|_x + \frac{\Delta x^2}{3!}\ \frac{\partial^3 u}{\partial x^3}\bigg|_x + \dots$$

Assuming u and its derivatives can be bound by polynomials, the increasing order of the partial derivatives and the increasing factorial divisor should both conspire to ensure that $\frac{\Delta x}{2!}\ \frac{\partial^2 u}{\partial x^2}\big|_x$ is the largest element of the error. So the error is notated in reference to that term $O(\Delta x)$.

On the spatial dimensions, the first partial derivatives produce the most error (since the approximations for higher-order derivatives truncate at a higher order in the Taylor series), so we can characterize the error just using the approximation for the first partial derivative. The approximation is constructed as the mean of the "left" and "right" approximations on the spatial dimension:

$$\frac{u(x+\Delta x)-u(x)}{2\Delta x}+\frac{u(x)-u(x-\Delta x)}{2\Delta x}$$

This introduces the possibility of doubling the error but removes directional bias. This does not affect the *order* of the error.

4. About Coarrays

This program is written with Fortran 2008 coarrays. Coarray computation is a type of *Single Program, Multiple Data* (SPMD) computation. Under SPMD, each *image* (program instance/process) has the same code, but each has its own memory space. Variables declared as coarrays behave just the same as normal variables, except each image can access each other image's instance of that variable. All of the parameters for this program are declared as coarrays, since the first image reads the values and sends them to the other images using the `co_broadcast` intrinsic from TS 18508 (link below). The concentration array is also declared as a coarray. Each image only deals with its own assigned section, and the section boundaries are synced each time step.

TS 18508: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=62702

5. Environment Parameters

The parameters of the environment are:

{Variable Declarations 5}

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

Added to in sections 6, 7, 9, 10, 11 and 12 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.

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

{Input 5}

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

Added to in section 6 Used in section 16

6. Control Parameters

The program also takes control parameters:

{Variable Declarations 5} +=

```
real, codimension[*] :: dx, dt
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 7, 9, 10, 11 and 12 Used in section 16

where:

Variable	Units	Definition
`dx`	meters	$dx = \Delta x$, the spatial discretization.
`dt`	days	$dt = \Delta t$, the temporal discretization.
`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 two reals, the second is just `tout_length` (an integer), and the third is `tout_length` reals:

{Input 5} +=

```
if (this_image() == 1) then
    read *, dx, dt
    read *, tout_length
end if
{Allocate Output Times, 7}
if (this_image() == 1) then
    read *, tout_array
end if
```

Used in section 16

7. Output Times Allocation

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

{Allocate Output Times 7}

```
call co_broadcast(tout_length, source_image=1)
allocate (tout_array(tout_length) [*], stat=alloc_stat, errmsg=alloc_errmsg)
{Handle Allocation Error, 7}
```

Used in section 6

Handling allocation errors necessitates two more variables:

{Variable Declarations 5} +=

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

Added to in sections 6, 9, 10, 11 and 12 Used in section 16

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

{Handle Allocation Error 7}

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

```
        stop
    end if
```

Used in sections 9 and 9

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

{Use Statements 7}

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

Used in section 16

8. 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 8}

```
    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(dx, source_image=1)
    call co_broadcast(dt, source_image=1)
    call co_broadcast(tout_array, source_image=1)
```

Used in section 16

9. Splitting the Domain

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 5} +=

```
    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 6, 7, 10, 11 and 12 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 9}

```
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 9 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 9} +=

```
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 9 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 9} +=

```
allocate (concentrations(0:next_concentrations_length + 1) [*], &
            stat=alloc_stat, errmsg=alloc_errmsg)
{Handle Allocation Error, 7}
concentrations = ca
allocate (next_concentrations(1:next_concentrations_length), &
            stat=alloc_stat, errmsg=alloc_errmsg)
{Handle Allocation Error, 7}
```

Added to in section 9 Used in section 16

10. Table Header Output

{Variable Declarations 5} +=

```
integer :: xi
```

Added to in sections 6, 7, 9, 11 and 12 Used in section 16

This just outputs the header for the value table:

{Output Header 10}

```
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

11. 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 5} +=

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

Added to in sections 6, 7, 9, 10 and 12 Used in section 16

{Real Output Subroutine 11}

```
    subroutine writereal(r)
        real :: r
        write (unit=*, fmt=real_fmtstr, advance="no") r
    end subroutine writereal
```

Used in section 16

12. Main Loop

{Variable Declarations 5} +=

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

Added to in sections 6, 7, 9, 10 and 11 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 12}

```
    !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, 12}
        end if
    end do
```

Used in section 16

This outputs the concentration at each node:

{Print Record 12}

```
    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") ""
```

13. 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 13}

```
subroutine run_to(new_ti)
    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

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 equation for this step is ($c_i := c_i(t)$):

$$c_i(t + \Delta t) = c_i + D\Delta t \frac{c_{i+1} - 2c_i + c_{i-1}}{(\Delta x)^2} - v\Delta t \frac{c_{i+1} - c_{i-1}}{2\Delta x}$$

{Time Step Subroutine 14}

```
subroutine time_step
    integer :: i
    do i = 1, next_concentrations_length
        next_concentrations(i) = concentrations(i) + &
                    d*dt*(concentrations(i+1) - 2*concentrations(i) + &
                        concentrations(i-1))/(dx*dx) - &
                    v*dt*(concentrations(i+1) - concentrations(i-1))/(2*dx)
    end do
end subroutine time_step
```

Used in section 16

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
    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
if (this_image() == 1) then
    if (ti == 0) then
        concentrations(0) = cb
    end if
else
    concentrations(0) = concentrations( &
            next_concentrations_length[this_image() - 1])[this_image() - 1]
end if
if (this_image() == num_images()) then
    if (ti == 0) then
        concentrations(next_concentrations_length + 1) = cc
    end if
else
    concentrations(next_concentrations_length + 1) = &
                concentrations(1)[this_image() + 1]
end if
sync all
```

16. File Outline

{**fd_advect_dispers.f90** 16}

```
program fd_advect_dispers

    {Use Statements, 7}

    implicit none
    {Variable Declarations, 5}

    {Input, 5}
    {Broadcast Parameters, 8}
    sync all

    {Initialize Domain, 9}
    {Output Header, 10}

    {Main Loop, 12}
```

```
contains

    {Real Output Subroutine, 11}
    {Run To Subroutine, 13}
    {Time Step Subroutine, 14}
    {Writeback Subroutine, 15}

end program fd_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

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

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 / 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

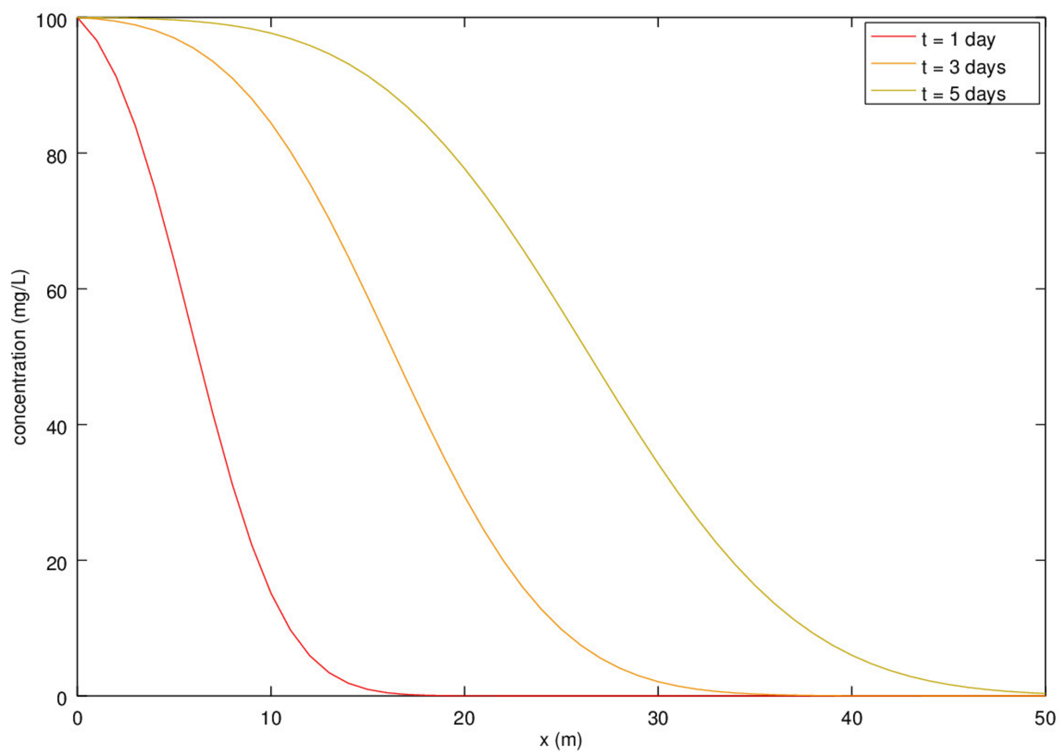
This is just the basic outline of the file.

{**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

This run has parameters $c_o = 100.0$ mg/L, $v = 5.0$ m/day, and $D = 8.0$ m^2/day:



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
```

Contents

.0.hw4.info **1**
Equations . . . 1
Truncation Error . . . 1
Plots . . . 2
Data . . . 3
Performance . . . 7

gensrc/a__advect__dispers.f90 **7**

gensrc/fd__advect__dispers.f90 **8**

.0.hw4.info

This simple document is available at https://tachibanatech.com/litdoc/hw4.simple.pdf in PDF format.

The full document is available at https://tachibanatech.com/litdoc/hw4.full.pdf in PDF format and https://tachibanatech.com/litdoc/hw4.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 equation for forward finite difference is ($c_i := c_i(t)$)

$$\frac{c_i(t+\Delta t) - c_i}{\Delta t} = D\frac{c_{i+1} - 2c_i + c_{i-1}}{(\Delta x)^2} - v\frac{c_{i+1} - c_{i-1}}{2\Delta x}$$

Solving for $c_i(t + \Delta t)$,

$$c_i(t+\Delta t) = c_i + D\Delta t\frac{c_{i+1} - 2c_i + c_{i-1}}{(\Delta x)^2} - v\Delta t\frac{c_{i+1} - c_{i-1}}{2\Delta x}$$

Truncation Error

The order of the truncation error here is $O(\Delta t)$ in time and $O(\Delta x)$ in space. The foundation of the finite difference method is the Taylor series:

$$u(x+\Delta x) = u(x) + \Delta x\ \frac{\partial u}{\partial x}\bigg|_x + \frac{\Delta x^2}{2!}\ \frac{\partial^2 u}{\partial x^2}\bigg|_x + \frac{\Delta x^3}{3!}\ \frac{\partial^3 u}{\partial x^3}\bigg|_x + \dots$$

Rearranging the equation to solve for error,

$$\frac{u(x+\Delta x) - u(x)}{\Delta x} - \frac{\partial u}{\partial x}\bigg|_x = \frac{\Delta x}{2!}\ \frac{\partial^2 u}{\partial x^2}\bigg|_x + \frac{\Delta x^2}{3!}\ \frac{\partial^3 u}{\partial x^3}\bigg|_x + \dots$$

Assuming u and its derivatives can be bound by polynomials, the increasing order of the partial derivatives and the increasing factorial divisor should both conspire to ensure that $\frac{\Delta x}{2!} \left. \frac{\partial^2 u}{\partial x^2} \right|_x$ is the largest element of the error. So the error is notated in reference to that term $O(\Delta x)$.

On the spatial dimensions, the first partial derivatives produce the most error (since the approximations for higher-order derivatives truncate at a higher order in the Taylor series), so we can characterize the error just using the approximation for the first partial derivative. The approximation is constructed as the mean of the "left" and "right" approximations on the spatial dimension:

$$\frac{u(x + \Delta x) - u(x)}{2\Delta x} + \frac{u(x) - u(x - \Delta x)}{2\Delta x}$$

This introduces the possibility of doubling the error but removes directional bias. This does not affect the *order* of the error.

Plots

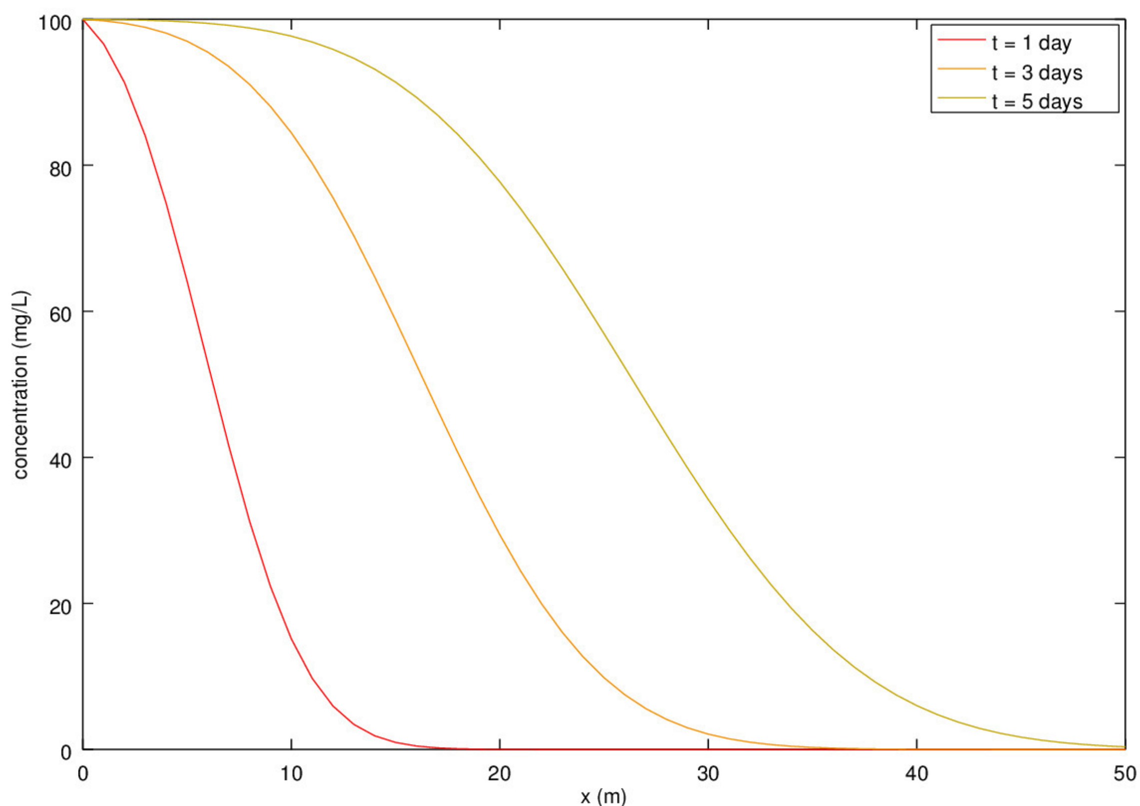


Figure 1: Analytic Solution

20 hw4.simple.markdown

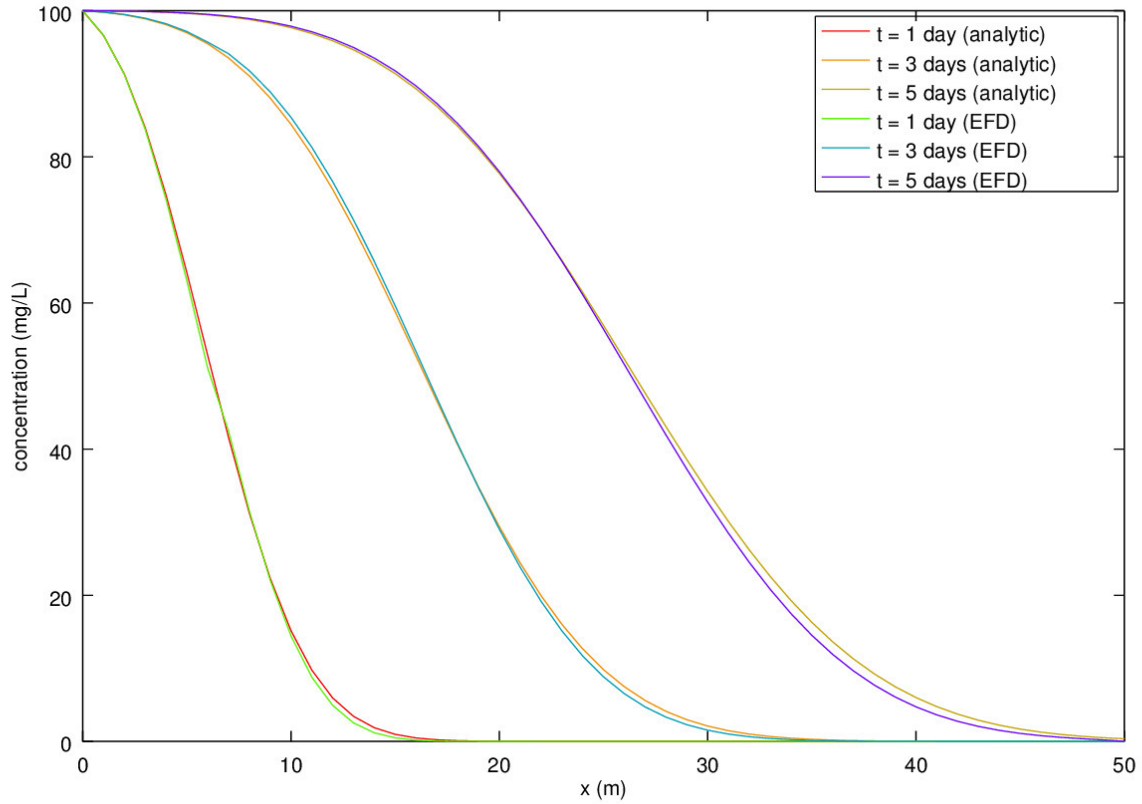


Figure 2: Explicit Finite Difference Model, $\Delta t = 0.05$ day

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 96.64 91.28 83.76 74.18 63.02 51.05 42.44 31.53 22.06 14.43 08.78 ...
(3)  03.00 100.0 99.81 99.48 98.98 98.22 97.15 95.70 94.11 91.79 88.90 85.41 81.31 ...
(4)  05.00 100.0 99.98 99.95 99.89 99.81 99.68 99.51 99.26 98.92 98.47 97.88 97.12 ...

(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)  04.93 02.52 01.17 00.48 00.17 00.05 00.01 00.00 00.00 00.00 00.00 00.00 00.00 ...
(3)  76.62 71.38 65.67 59.61 53.34 46.99 40.73 34.70 29.03 23.83 19.19 15.14 11.69 ...
(4)  96.16 94.97 93.51 91.77 89.70 87.30 84.55 81.44 77.98 74.18 70.08 65.72 61.14 ...

(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)  08.83 06.53 04.71 03.32 02.28 01.53 01.00 00.63 00.39 00.23 00.14 00.08 00.04 ...
(4)  56.41 51.58 46.73 41.93 37.25 32.75 28.48 24.49 20.82 17.49 14.52 11.91 09.64 ...

(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 00.00 00.00
(3)  00.02 00.01 00.01 00.00 00.00 00.00 00.00 00.00 00.00 00.00 00.00 00.00 00.00
(4)  07.70 06.07 04.73 03.63 02.74 02.05 01.50 01.09 00.77 00.54 00.36 00.20 00.00
```

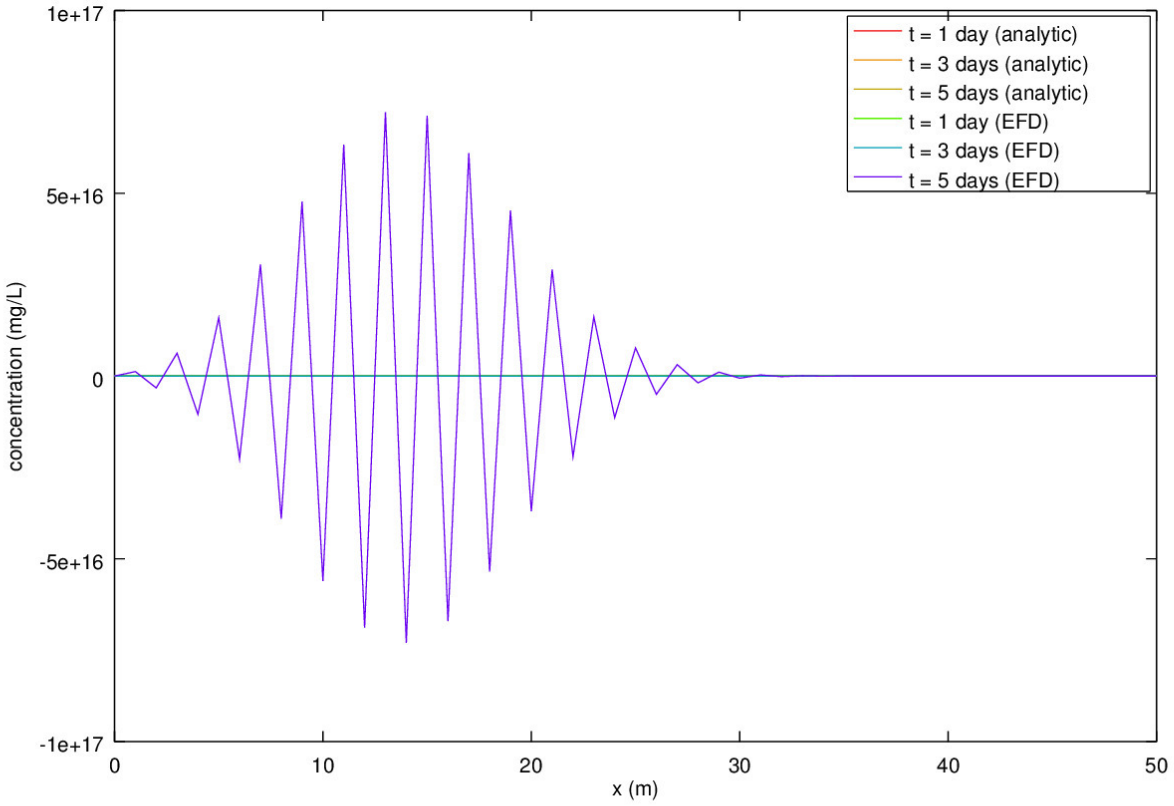


Figure 3: Explicit Finite Difference Model, $\Delta t = 0.1$ day

The stability criterion for EFD models of the advection-dispersion equation stipulates that $\frac{D\Delta t}{(\Delta x)^2} \leq \frac{1}{2}$ must be met. For this model, the criterion is not met: $\frac{(8.0 \text{ m}^2/\text{day})(0.1 \text{ day})}{(1.0 \text{ m})^2} = 0.8 \nleq \frac{1}{2}$. This is the reason for the explosion seen on the plot.

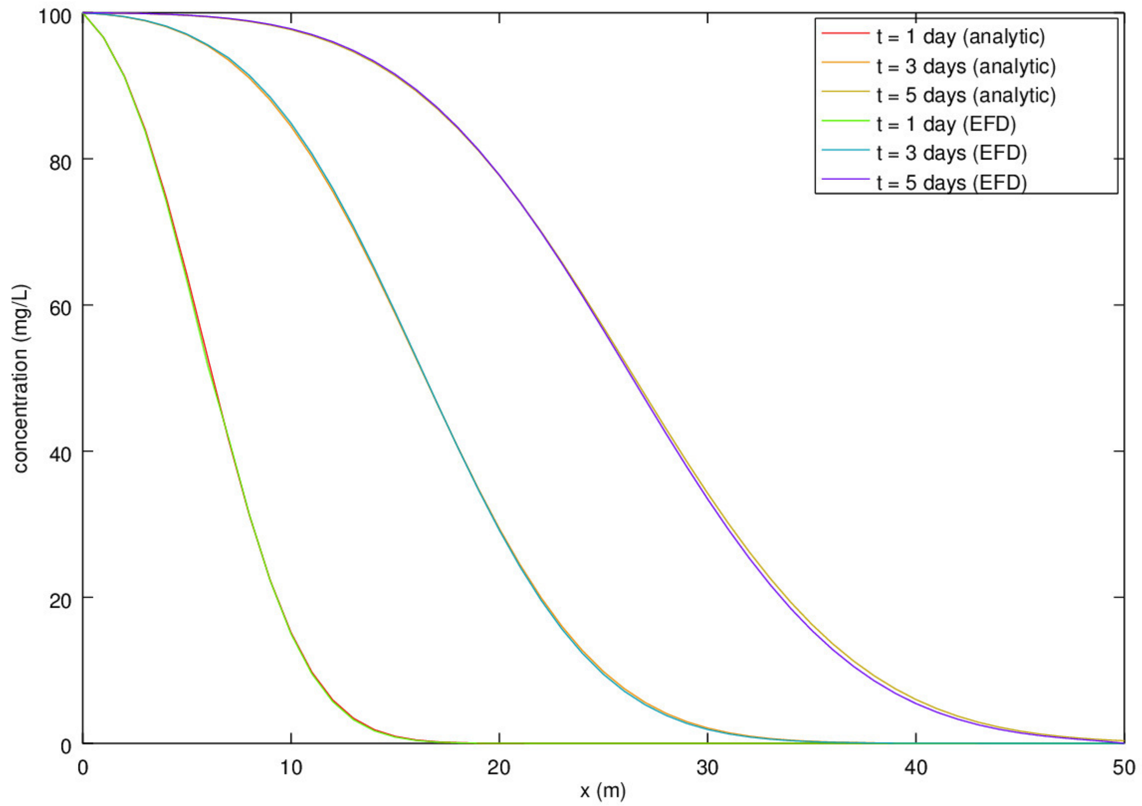
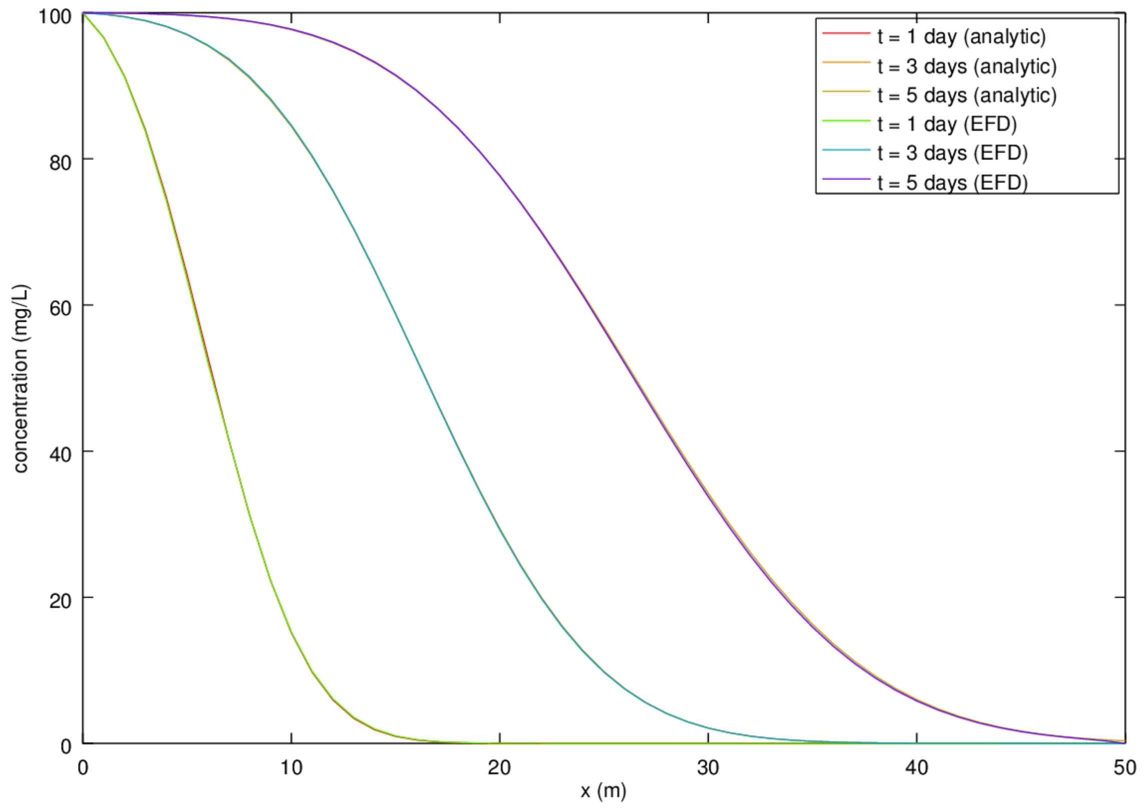
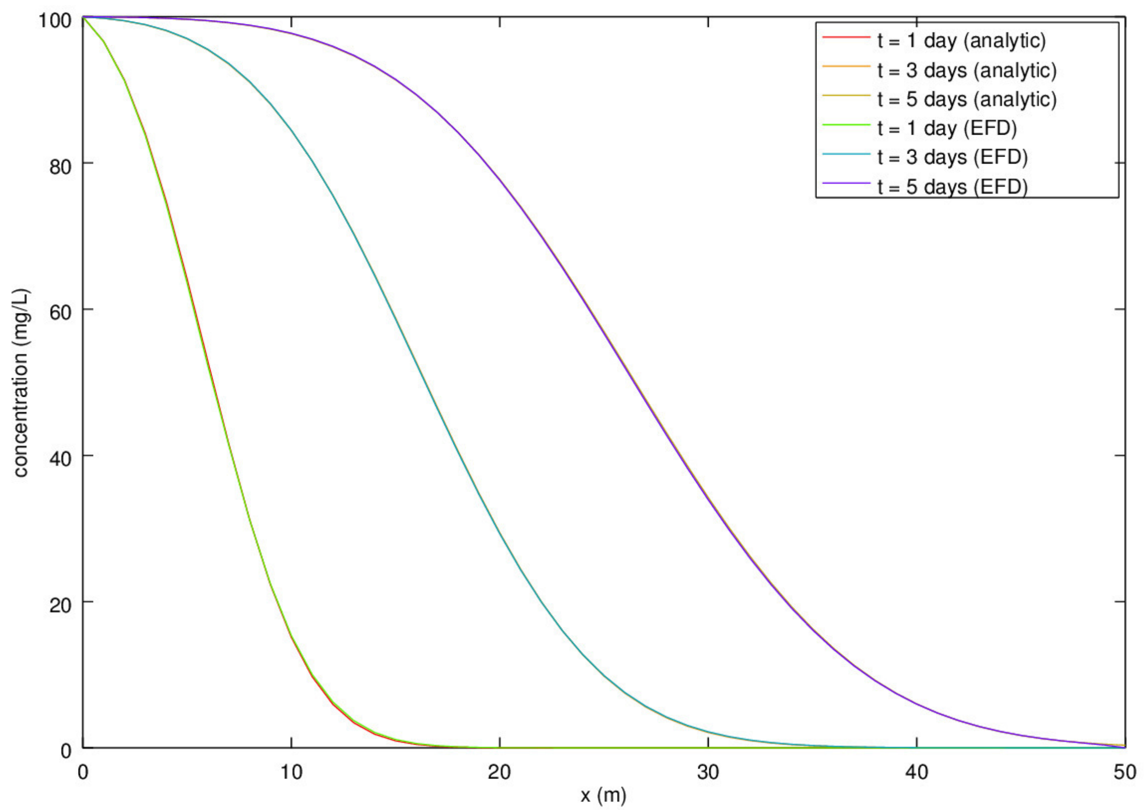


Figure 4: Explicit Finite Difference Model, $\Delta t = 0.025$ day

Reducing the time discretization reduces the coefficient of the truncation error, allowing for more accurate calculation from time step to time step.

On the next page are the plots for $\Delta t = 0.0125$ day and $\Delta t = 0.00625$ day (in that order). There are diminishing returns for decreasing the time discretization further, and so at the scale of these plots, I would use $\Delta t = 0.0125$ day for this problem.





24 hw4.simple.markdown

Performance

The finite difference program is designed to be parallel. I tested its performance on a Lenovo ideapad 700-15ISK with an i7-6700HQ CPU at 2.60GHz (4 cores, 8 hardware threads) and 16GiB of memory, running Ubuntu 16.10 with `gfortran` version 6.2.0 and `caf` version 1.8.4. Results:

Processes	Runtime Relative to Single Process
1	1.00
2	1.04
3	0.75
4	0.64
8	0.48
9	0.78
16	0.92

My conjecture is that the best performance is at the hardware thread count, unless the hardware thread count is 2, in which case the best performance is at a single process. The reason two processes is slower than one is that there is some basic overhead for multiprocessing, and the reason performance degrades over the hardware thread count is that there is quite a lot of overhead for OS process switching.

gensrc/a_advect_dispers.f90

```
1  ! a_advect_dispers.f90
2  program a_advect_dispers
3  
4      implicit none
5      ! Variable Declarations
6      real :: co, v, d
7      real :: xs, xint, xt
8      integer :: tlength
9      real, dimension(:), allocatable :: tarray
10     integer :: alloc_stat
11     character(len=80) :: alloc_errmsg
12     character(len=13), parameter :: real_fmtstr = "(sp,es16.7e3)"
13 
14     real :: x, t
15     integer :: xi, ti
16 
17     ! Input
18     read *, co, v, d
19     read *, xs, xint, xt
20     read *, tlength
21     allocate (tarray(tlength), stat=alloc_stat, errmsg=alloc_errmsg)
22     call errstop(alloc_stat, alloc_errmsg);
23     read *, tarray
24 
25     ! Output
26     write (unit=*, fmt=real_fmtstr, advance="no") 0.0
27     do xi = 0, floor(xt / xint)
28         x = xs + real(xi)*xint
29         write (unit=*, fmt=real_fmtstr, advance="no") x
30     end do
31     write (unit=*, fmt="(a)", advance="yes") ""
32     do ti = 1, size(tarray)
```

```
33          t = tarray(ti)
34          call at_time(t, xs, xint, xt)
35      end do
36
37
38  contains
39
40      ! Record Output Function
41      subroutine at_time(t, xs, xint, xt)
42          implicit none
43          real :: t, xs, xint, xt
44
45          write (unit=*, fmt=real_fmtstr, advance="no") t
46          do xi = 0, floor(xt / xint)
47              x = xs + xi*xint
48              write (unit=*, fmt=real_fmtstr, advance="no") c(x,t)
49          end do
50          write (unit=*, fmt="(a)", advance="yes") ""
51      end subroutine at_time
52
53      ! Concentration Function
54      function c(x,t) result(c_r)
55          implicit none
56          real, intent(in) :: x, t
57          real :: c_r
58
59          if (x == 0) then
60              c_r = co
61          else
62              c_r = co/2 * (erfc((x - v*t)/(2*sqrt(d*t))) &
63                      + exp(v*x/d)*erfc((x + v*t)/(2*sqrt(d*t))))
64          end if
65      end function c
66
67      ! Stop On Error
68      subroutine errstop(stat, errmsg)
69          integer :: stat
70          character(len=*) errmsg
71
72          if (stat > 0) then
73              print *, trim(errmsg)
74              stop
75          end if
76      end subroutine errstop
77
78
79  endprogram
```

gensrc/fd_advect_dispers.f90

```
1   ! fd_advect_dispers.f90
2   program fd_advect_dispers
3
4       ! Use Statements
5       use, intrinsic :: iso_fortran_env, only: error_unit
```

```
6
7
8     implicit none
9     ! Variable Declarations
10    real, codimension[*] :: ca, cb, cc
11    real, codimension[*] :: length, v, d
12    real, codimension[*] :: dx, dt
13    integer, codimension[*] :: tout_length
14    ! The dimensions of tout_array will be set later, at allocation.
15    real, dimension(:), codimension[:], allocatable :: tout_array
16    integer :: alloc_stat
17    character(len=80) :: alloc_errmsg
18    real, dimension(:), codimension[:], allocatable :: concentrations
19    real, dimension(:), allocatable :: next_concentrations
20    integer, codimension[*] :: next_concentrations_length
21    ! Intermediate values
22    real :: num_edges_real
23    integer :: num_edges
24    integer :: xi
25    character(len=13), parameter :: real_fmtstr = "(sp,es16.7e3)"
26    integer :: ti, tout_i, image_i
27
28
29    ! Input
30    if (this_image() == 1) then
31        read *, ca, cb, cc
32        read *, length, v, d
33    end if
34    if (this_image() == 1) then
35        read *, dx, dt
36        read *, tout_length
37    end if
38    ! Allocate Output Times
39    call co_broadcast(tout_length, source_image=1)
40    allocate (tout_array(tout_length) [*], stat=alloc_stat, errmsg=alloc_errmsg)
41    ! Handle Allocation Error
42    if (alloc_stat > 0) then
43        write (unit=error_unit, fmt="(a)") trim(alloc_errmsg)
44        stop
45    end if
46
47
48    if (this_image() == 1) then
49        read *, tout_array
50    end if
51
52    ! Broadcast Parameters
53    call co_broadcast(ca, source_image=1)
54    call co_broadcast(cb, source_image=1)
55    call co_broadcast(cc, source_image=1)
56    call co_broadcast(length, source_image=1)
57    call co_broadcast(v, source_image=1)
58    call co_broadcast(d, source_image=1)
59    call co_broadcast(dx, source_image=1)
60    call co_broadcast(dt, source_image=1)
61    call co_broadcast(tout_array, source_image=1)
62
```

```
63      sync all
64
65      ! Initialize Domain
66      num_edges_real = length / dx
67      num_edges = nint(num_edges_real)
68      if (abs(num_edges_real - num_edges) > 8*epsilon(num_edges_real) .or. &
69                  num_edges < 1) then
70          write (unit=error_unit, fmt="(a)") "ERROR: L/dx must be an integer >= 1."
71          stop
72      end if
73      if (this_image() == num_images()) then
74          next_concentrations_length = (num_edges - 1) - &
75                      (num_images()-1)*(num_edges / num_images())
76      else
77          next_concentrations_length = (num_edges / num_images())
78      end if
79      allocate (concentrations(0:next_concentrations_length + 1) [*], &
80                  stat=alloc_stat, errmsg=alloc_errmsg)
81      ! Handle Allocation Error
82      if (alloc_stat > 0) then
83          write (unit=error_unit, fmt="(a)") trim(alloc_errmsg)
84          stop
85      end if
86
87      concentrations = ca
88      allocate (next_concentrations(1:next_concentrations_length), &
89                  stat=alloc_stat, errmsg=alloc_errmsg)
90      ! Handle Allocation Error
91      if (alloc_stat > 0) then
92          write (unit=error_unit, fmt="(a)") trim(alloc_errmsg)
93          stop
94      end if
95
96
97      ! Output Header
98      if (this_image() == 1) then
99          call writereal(0.0)
100         do xi = 0, num_edges
101             call writereal(xi * dx)
102         end do
103         write (unit=*, fmt="(a)", advance="yes") ""
104     end if
105
106
107     ! Main Loop
108     !write (unit=error_unit, fmt=*) this_image(), next_concentrations_length
109     ti = 0
110     do tout_i = 1, size(tout_array)
111         call run_to(nint(tout_array(tout_i) / dt))
112         if (this_image() == 1) then
113             ! Print Record
114             call writereal(ti * dt)
115             call writereal(concentrations(0)[1])
116             do image_i = 1, num_images()
117                 do xi = 1, next_concentrations_length[image_i]
118                     call writereal(concentrations(xi)[image_i])
119                 end do
```

```
120             end do
121             xi = next_concentrations_length[num_images()] + 1
122             call writereal(concentrations(xi)[num_images()])
123             write (unit=*, fmt="(a)", advance="yes") ""
124
125         end if
126     end do
127
128
129 contains
130
131     ! Real Output Subroutine
132     subroutine writereal(r)
133         real :: r
134         write (unit=*, fmt=real_fmtstr, advance="no") r
135     end subroutine writereal
136
137     ! Run To Subroutine
138     subroutine run_to(new_ti)
139         integer, intent(in) :: new_ti
140
141         if (new_ti <= ti) then
142             return
143         end if
144         do ti = ti, new_ti - 1
145             call time_step
146             call writeback
147         end do
148         ti = new_ti
149     end subroutine run_to
150
151     ! Time Step Subroutine
152     subroutine time_step
153         integer :: i
154         do i = 1, next_concentrations_length
155             next_concentrations(i) = concentrations(i) + &
156                         d*dt*(concentrations(i+1) - 2*concentrations(i) + &
157                             concentrations(i-1))/(dx*dx) - &
158                         v*dt*(concentrations(i+1) - concentrations(i-1))/(2*dx)
159         end do
160     end subroutine time_step
161
162     ! Writeback Subroutine
163     subroutine writeback
164         integer :: i
165         do i = 1, next_concentrations_length
166             concentrations(i) = next_concentrations(i)
167         end do
168
169         ! Sync Boundaries
170         sync all
171         if (this_image() == 1) then
172             if (ti == 0) then
173                 concentrations(0) = cb
174             end if
175         else
176             concentrations(0) = concentrations( &
```

```
177                       next_concentrations_length[this_image() - 1])[this_image() - 1]
178         end if
179         if (this_image() == num_images()) then
180             if (ti == 0) then
181                 concentrations(next_concentrations_length + 1) = cc
182             end if
183         else
184             concentrations(next_concentrations_length + 1) = &
185                         concentrations(1)[this_image() + 1]
186         end if
187         sync all
188
189     end subroutine writeback
190
191
192 end program fd_advect_dispers
```