**ASSIGNMENT 3**

**COMPUTATIONAL FLUID DYNAMICS**



**Kushagra Shrivastava**

**214103011**

**M.Tech Aerodynamics & Propulsion**

**Monsoon 2021**

**1D Linear Hyperbolic Wave Equation**

FTCS Scheme computation code (Page 2-12)

**# include <stdlib.h>**

**# include <stdio.h>**

**# include <math.h>**

**# include <time.h>**

**# include <string.h>**

**int main ( );**

**int i4\_modp ( int i, int j );**

**int i4\_wrap ( int ival, int ilo, int ihi );**

**double \*initial\_condition ( int nx, double x[] );**

**double \*r8vec\_linspace\_new ( int n, double a, double b );**

**void timestamp ( );**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**int main ( )**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**/\***

**Purpose:**

**FD1D\_ADVECTION\_FTCS solves the advection equation using the FTCS method.**

**Discussion:**

**The FTCS method is unstable for the advection problem.**

**Given a smooth initial condition, successive FTCS approximations will**

**exhibit erroneous oscillations of increasing magnitude.**

**\*/**

**{**

**double a;**

**double b;**

**double c;**

**char command\_filename[] = "advection\_commands.txt";**

**FILE \*command\_unit;**

**char data\_filename[] = "advection\_data.txt";**

**FILE \*data\_unit;**

**double dt;**

**double dx;**

**int i;**

**int j;**

**int jm1;**

**int jp1;**

**int nx;**

**int nt;**

**int nt\_step;**

**double t;**

**double \*u;**

**double \*unew;**

**double \*x;**

**timestamp ( );**

**printf ( "\n" );**

**printf ( "FD1D\_ADVECTION\_FTCS:\n" );**

**printf ( " C version\n" );**

**printf ( "\n" );**

**printf ( " Solve the constant-velocity advection equation in 1D,\n" );**

**printf ( " du/dt = - c du/dx\n" );**

**printf ( " over the interval:\n" );**

**printf ( " 0.0 <= x <= 1.0\n" );**

**printf ( " with periodic boundary conditions, and\n" );**

**printf ( " with a given initial condition\n" );**

**printf ( " u(0,x) = 0.0 for 0.5 <= x\n" );**

**printf ( " = 1.0 elsewhere.\n" );**

**printf ( "\n" );**

**printf ( " We use a method known as FTCS:\n" );**

**nx = 201;**

**dx = 1.0 / ( double ) ( nx - 1 );**

**a = 0.0;**

**b = 1.0;**

**x = r8vec\_linspace\_new ( nx, a, b );**

**nt = 50;**

**float lambda[] ={0.2,0.8,0.9,1.0,1.1};**

**float lmb;**

**dt = 1.0 / ( double ) ( nt );**

**c = 1.0;**

**u = initial\_condition ( nx, x );**

**/\***

**Open data file, and write solutions as they are computed.**

**\*/**

**data\_unit = fopen ( data\_filename, "wt" );**

**t = 0.0;**

**fprintf ( data\_unit, "%10.4f %10.4f %10.4f\n", x[0], t, u[0] );**

**for ( j = 0; j < nx; j++ )**

**{**

**fprintf ( data\_unit, "%10.4f %10.4f %10.4f\n", x[j], t, u[j] );**

**}**

**fprintf ( data\_unit, "\n" );**

**nt\_step = 10;**

**printf ( "\n" );**

**printf ( " Number of nodes NX = %d\n", nx );**

**printf ( " Number of time steps NT = %d\n", nt );**

**printf ( " Constant velocity C = %g\n", c );**

**unew = ( double \* ) malloc ( nx \* sizeof ( double ) );**

**for ( i = 0; i < nt; i++ )**

**{ lmb=lambda[i];**

**for ( j = 0; j < nx; j++ )**

**{**

**jm1 = i4\_wrap ( j - 1, 0, nx - 1 );**

**jp1 = i4\_wrap ( j + 1, 0, nx - 1 );**

**unew[j] = u[j] - lmb / 2.0 \* ( u[jp1] - u[jm1] );**

**}**

**for ( j = 0; j < nx; j++ )**

**{**

**u[j] = unew[j];**

**}**

**if ( i == nt\_step - 1 )**

**{**

**t = ( double ) ( i ) \* dt;**

**for ( j = 0; j < nx; j++ )**

**{**

**fprintf ( data\_unit, "%10.4f %10.4f %10.4f\n", x[j], t, u[j] );**

**}**

**fprintf ( data\_unit, "\n" );**

**nt\_step = nt\_step + 15;**

**}**

**}**

**/\***

**Close the data file once the computation is done.**

**\*/**

**fclose ( data\_unit );**

**printf ( "\n" );**

**printf ( " Plot data written to the file \"%s\"\n", data\_filename );**

**return 0;**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**int i4\_modp ( int i, int j )**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**/\***

**Purpose:**

**I4\_MODP returns the nonnegative remainder of I4 division.**

**Discussion:**

**If**

**NREM = I4\_MODP ( I, J )**

**NMULT = ( I - NREM ) / J**

**then**

**I = J \* NMULT + NREM**

**where NREM is always nonnegative.**

**The MOD function computes a result with the same sign as the**

**quantity being divided. Thus, suppose you had an angle A,**

**and you wanted to ensure that it was between 0 and 360.**

**Then mod(A,360) would do, if A was positive, but if A**

**was negative, your result would be between -360 and 0.**

**On the other hand, I4\_MODP(A,360) is between 0 and 360, always.**

**Example:**

**I J MOD I4\_MODP I4\_MODP Factorization**

**107 50 7 7 107 = 2 \* 50 + 7**

**107 -50 7 7 107 = -2 \* -50 + 7**

**-107 50 -7 43 -107 = -3 \* 50 + 43**

**-107 -50 -7 43 -107 = 3 \* -50 + 43**

**Parameters:**

**Input, int I, the number to be divided.**

**Input, int J, the number that divides I.**

**Output, int I4\_MODP, the nonnegative remainder when I is**

**divided by J.**

**\*/**

**{**

**int value;**

**if ( j == 0 )**

**{**

**fprintf ( stderr, "\n" );**

**fprintf ( stderr, "I4\_MODP - Fatal error!\n" );**

**fprintf ( stderr, " I4\_MODP ( I, J ) called with J = %d\n", j );**

**exit ( 1 );**

**}**

**value = i % j;**

**if ( value < 0 )**

**{**

**value = value + abs ( j );**

**}**

**return value;**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**int i4\_wrap ( int ival, int ilo, int ihi )**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**/\***

**Purpose:**

**I4\_WRAP forces an I4 to lie between given limits by wrapping.**

**Parameters:**

**Input, int IVAL, an integer value.**

**Input, int ILO, IHI, the desired bounds for the integer value.**

**Output, int I4\_WRAP, a "wrapped" version of IVAL.**

**\*/**

**{**

**int jhi;**

**int jlo;**

**int value;**

**int wide;**

**if ( ilo < ihi )**

**{**

**jlo = ilo;**

**jhi = ihi;**

**}**

**else**

**{**

**jlo = ihi;**

**jhi = ilo;**

**}**

**wide = jhi + 1 - jlo;**

**if ( wide == 1 )**

**{**

**value = jlo;**

**}**

**else**

**{**

**value = jlo + i4\_modp ( ival - jlo, wide );**

**}**

**return value;**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**double \*initial\_condition ( int nx, double x[] )**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**/\***

**Purpose:**

**INITIAL\_CONDITION sets the initial condition.**

**Parameters:**

**Input, int NX, the number of nodes.**

**Input, double X[NX], the coordinates of the nodes.**

**Output, double INITIAL\_CONDITION[NX], the value of the initial condition.**

**\*/**

**{**

**int i;**

**double \*u;**

**u = ( double \* ) malloc ( nx \* sizeof ( double ) );**

**for ( i = 0; i < nx; i++ )**

**{**

**if ( 0.5 <= x[i] )**

**{**

**u[i] = 0.0;**

**}**

**else**

**{**

**u[i] = 1.0;**

**}**

**}**

**return u;**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**double \*r8vec\_linspace\_new ( int n, double a, double b )**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**/\***

**Purpose:**

**R8VEC\_LINSPACE\_NEW creates a vector of linearly spaced values.**

**Discussion:**

**An R8VEC is a vector of R8's.**

**4 points evenly spaced between 0 and 12 will yield 0, 4, 8, 12.**

**In other words, the interval is divided into N-1 even subintervals,**

**and the endpoints of intervals are used as the points.**

**Parameters:**

**Input, int N, the number of entries in the vector.**

**Input, double A, B, the first and last entries.**

**Output, double R8VEC\_LINSPACE\_NEW[N], a vector of linearly spaced data.**

**\*/**

**{**

**int i;**

**double \*x;**

**x = ( double \* ) malloc ( n \* sizeof ( double ) );**

**if ( n == 1 )**

**{**

**x[0] = ( a + b ) / 2.0;**

**}**

**else**

**{**

**for ( i = 0; i < n; i++ )**

**{**

**x[i] = ( ( double ) ( n - 1 - i ) \* a**

**+ ( double ) ( i ) \* b )**

**/ ( double ) ( n - 1 );**

**}**

**}**

**return x;**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**Lax Wendroff Scheme (**Page 13 – 22)

# include <stdlib.h>

# include <stdio.h>

# include <math.h>

# include <time.h>

# include <string.h>

int main ( );

int i4\_modp ( int i, int j );

int i4\_wrap ( int ival, int ilo, int ihi );

double \*initial\_condition ( int nx, double x[] );

double \*r8vec\_linspace\_new ( int n, double a, double b );

void timestamp ( );

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main ( )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

FD1D\_ADVECTION\_LAX solves the advection equation using the Lax method.

Discussion:

The Lax method is stable for the advection problem, if the time step

satisifies the Courant-Friedrichs-Levy (CFL) condition:

dt <= dx / c

\*/

{

double a;

double b;

double c;

char command\_filename[] = "advection\_commands.txt";

FILE \*command\_unit;

char data\_filename[] = "advection\_data.txt";

FILE \*data\_unit;

double dt;

double dx;

int i;

int j;

int jm1;

int jp1;

int nx;

int nt;

int nt\_step;

double t;

double \*u;

double \*unew;

double \*x;

printf ( "\n" );

printf ( "FD1D\_ADVECTION\_LAX:\n" );

printf ( " C version\n" );

printf ( "\n" );

printf ( " Solve the constant-velocity advection equation in 1D,\n" );

printf ( " du/dt = - c du/dx\n" );

printf ( " over the interval:\n" );

printf ( " 0.0 <= x <= 1.0\n" );

printf ( " with periodic boundary conditions, and\n" );

printf ( " with a given initial condition\n" );

printf ( " u(0,x) = 0.0 for 0.5 <= x \n" );

printf ( " = 1.0 elsewhere.\n" );

printf ( "\n" );

printf ( " We modify the FTCS method using the Lax method:\n" );

nx = 201;

dx = 1.0 / ( double ) ( nx - 1 );

a = 0.0;

b = 1.0;

x = r8vec\_linspace\_new ( nx, a, b );

nt = 50;

float lambda[] ={0.2,0.8,0.9,1.0,1.1};

float lmb;

dt = 1.0 / ( double ) ( nt );

c = 1.0;

u = initial\_condition ( nx, x );

data\_unit = fopen ( data\_filename, "wt" );

t = 0.0;

fprintf ( data\_unit, "%10.4f %10.4f %10.4f\n", x[0], t, u[0] );

for ( j = 0; j < nx; j++ )

{

fprintf ( data\_unit, "%10.4f %10.4f %10.4f\n", x[j], t, u[j] );

}

fprintf ( data\_unit, "\n" );

nt\_step = 10;

printf ( "\n" );

printf ( " Number of nodes NX = %d\n", nx );

printf ( " Number of time steps NT = %d\n", nt );

printf ( " Constant velocity C = %g\n", c );

unew = ( double \* ) malloc ( nx \* sizeof ( double ) );

for ( i = 0; i < nt; i++ )

{ lmb=lambda[i];

for ( j = 0; j < nx; j++ )

{

jm1 = i4\_wrap ( j - 1, 0, nx - 1 );

jp1 = i4\_wrap ( j + 1, 0, nx - 1 );

unew[j] = 0.5 \* u[jm1] + 0.5 \* u[jp1]

- lmb \* ( u[jp1] - u[jm1] ) + (lmb\*lmb)/2.0\*(u[jp1] -2\*u[ceil((jpl+jml)/2.0)] +u[jm1]);

}

for ( j = 0; j < nx; j++ )

{

u[j] = unew[j];

}

if ( i == nt\_step - 1 )

{

t = ( double ) ( i ) \* dt;

for ( j = 0; j < nx; j++ )

{

fprintf ( data\_unit, "%10.4f %10.4f %10.4f\n", x[j], t, u[j] );

}

fprintf ( data\_unit, "\n" );

nt\_step = nt\_step + 15;

}

}

/\*

Close the data file once the computation is done.

\*/

fclose ( data\_unit );

printf ( "\n" );

printf ( " Plot data written to the file \"%s\"\n", data\_filename );

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int i4\_modp ( int i, int j )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

I4\_MODP returns the nonnegative remainder of I4 division.

Discussion:

If

NREM = I4\_MODP ( I, J )

NMULT = ( I - NREM ) / J

then

I = J \* NMULT + NREM

where NREM is always nonnegative.

The MOD function computes a result with the same sign as the

quantity being divided. Thus, suppose you had an angle A,

and you wanted to ensure that it was between 0 and 360.

Then mod(A,360) would do, if A was positive, but if A

was negative, your result would be between -360 and 0.

On the other hand, I4\_MODP(A,360) is between 0 and 360, always.

Example:

I J MOD I4\_MODP I4\_MODP Factorization

107 50 7 7 107 = 2 \* 50 + 7

107 -50 7 7 107 = -2 \* -50 + 7

-107 50 -7 43 -107 = -3 \* 50 + 43

-107 -50 -7 43 -107 = 3 \* -50 + 43

Parameters:

Input, int I, the number to be divided.

Input, int J, the number that divides I.

Output, int I4\_MODP, the nonnegative remainder when I is

divided by J.

\*/

{

int value;

if ( j == 0 )

{

fprintf ( stderr, "\n" );

fprintf ( stderr, "I4\_MODP - Fatal error!\n" );

fprintf ( stderr, " I4\_MODP ( I, J ) called with J = %d\n", j );

exit ( 1 );

}

value = i % j;

if ( value < 0 )

{

value = value + abs ( j );

}

return value;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int i4\_wrap ( int ival, int ilo, int ihi )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

I4\_WRAP forces an I4 to lie between given limits by wrapping.

Parameters:

Input, int IVAL, an integer value.

Input, int ILO, IHI, the desired bounds for the integer value.

Output, int I4\_WRAP, a "wrapped" version of IVAL.

\*/

{

int jhi;

int jlo;

int value;

int wide;

if ( ilo < ihi )

{

jlo = ilo;

jhi = ihi;

}

else

{

jlo = ihi;

jhi = ilo;

}

wide = jhi + 1 - jlo;

if ( wide == 1 )

{

value = jlo;

}

else

{

value = jlo + i4\_modp ( ival - jlo, wide );

}

return value;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

double \*initial\_condition ( int nx, double x[] )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

INITIAL\_CONDITION sets the initial condition.

Parameters:

Input, int NX, the number of nodes.

Input, double X[NX], the coordinates of the nodes.

Output, double INITIAL\_CONDITION[NX], the value of the initial condition.

\*/

{

int i;

double \*u;

u = ( double \* ) malloc ( nx \* sizeof ( double ) );

for ( i = 0; i < nx; i++ )

{

if ( 0.5 <= x[i] )

{

u[i] = 0.0;

}

else

{

u[i] = 1.0;

}

}

return u;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

double \*r8vec\_linspace\_new ( int n, double a, double b )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

{

int i;

double \*x;

x = ( double \* ) malloc ( n \* sizeof ( double ) );

if ( n == 1 )

{

x[0] = ( a + b ) / 2.0;

}

else

{

for ( i = 0; i < n; i++ )

{

x[i] = ( ( double ) ( n - 1 - i ) \* a

+ ( double ) ( i ) \* b )

/ ( double ) ( n - 1 );

}

}

return x;

}

**Plots**

**FTCS Scheme**





