第一次大作业

迁移方程离散格式求解



姓名：李春蕾

学号：SY1804410

指导教师：宁方飞

## 一、要求

空间计算域：[0.00-1.00]，用至少100个点离散。

方程如下：

初始条件如下：

在[0.1,0.2]处有一个方形波。

边界条件：

左端点：u=10。表示不断从左端输入速度通量

右端点：零法向梯度。即u[N+1]=u[N];

方波高度取为100，沿时间推进求解至时刻t=0.7/a，(a>0时)

## 二、程序说明

本迁移方程离散求解题目采用C++编写，共计算了11种离散格式，分别是欧拉向后显式，欧拉向前显式，中心差分显式，蛙跳显式，Lax-Wendroff显式，欧拉向后隐式，欧拉向前隐式，中心差分隐式，三点欧拉中心显式, Crank-Nicolson。除中心差分隐式外，每一种格式采用一个函数。中心差分隐式由于需要求解三对角矩阵，单独编写到一个cpp文件。

为了提高代码复用率，将相近的差分格式编写到同一个函数中，用”//”注释掉不用的格式。因此除中心差分隐式外，源文件包含4个函数：Explicit()用于实现欧拉向前、向后显式，中心差分显式，三点欧拉中心显式和三点欧拉向后显式； Implicit()用于欧拉向前、向后隐式；FogJump()用于实现蛙跳格式；LaxWendroff()用于实现Lax-Wendroff格式；以及MacCormark（）和Crank-Nicolson（）。

为了处理向后格式的前两步，即u[i-1]和u[i-2]，的值，需要先进行两步欧拉向前显式求解u[0]和u[1]。为此编写first2Steps（）函数，并设置flag值，flag=0表示不需要处理，flag=1表示需要处理第一步，flag=2表示需要处理第一和第二步。空间上的循环变量i从flag开始，一直循环到maxI-1（本程序中为100）结束。

在左端点边界上，给定第一类边界条件，即固定值10。在右端点边界上，给定第二类边界条件，即固定法向梯度0。在无u[i+1]项的格式中，只需要把循环变量i设置到maxI结束；在包含u[i+1]的格式中，i循环到maxI-1, 只需要令u[maxI]=u[maxI-1]即可得到最右边界点的值。

**离散参数**

取a=0.7，，求解至t=1.0时刻，自x[0]至x[N+1]共101个空间点，。

## 三、计算结果

图 1 差分格式比较

稳定的计算结果如图 1 所示。除图中所示的结果以外其余结果均为不稳定的。稳定的格式有6种，分别为：欧拉向后显式，LaxWendroff, Crank-Nicolson，MacCormark，欧拉向后隐式和中心差分隐式。

可以看到，经过1s后，波峰大概传递到8.5~9.0的位置。各个格式均有耗散。耗散最小的是LaxWendroff, 其次是欧拉向后显式，中心差分隐式，Crank-Nicolson, 最后是欧拉向后隐式。

还可以发现，MacCormark与Lax-Wendroff格式基本重合，这是因为MacCormark是Lax-Wendroff的一种变体。

## 四、离散格式精度分析

### 1. 欧拉向后显式



时间离散误差分析，将在点Taylor展开

然后代入到方程左端。

同理，对空间离散进行误差分析。将在点Taylor展开

然后代入到方程右端。

用上面第二个式子减去第四个式子，可以得到

显然其时间离散误差为，空间离散误差为，时间与空间皆为一阶精度。

### 2、Euler中心差显式格式



同上面一样，进行误差分析，最后得到的方程为：

由此可知为时间一阶精度，空间二阶精度。

### 3、Euler中心差隐式格式



在这种格式中，对空间项不仅要在空间点上Taylor展开,还要进行时间点的泰勒展开。

首先仿照Euler中心差分显式，仅仅将空间项中上标的n改成n+1, 写出，

然后再对空间项进行时间点上的Taylor展开， 为了方便起见，将记为，记为

带回原式中，得到

误差分析结果为:

整理一下，方程右边可以写成

由此可知为时间一阶精度，空间二阶精度。

对于该格式由于是隐式格式，采用追赶法求解隐式格式形成三对角方程组



### 4、Lax-Wendroff 格式

与前面不同的是，前面的格式大都通过将Taylor展开式代入到控制方程中得到，而Lax-Wendroff格式则通过将控制方程代入Taylor展开得到。

综合前面的推导，我们发现最终想要得到的是下一个时间步在当前网格点的值，即。这也叫做时间推进法。于是将在时间点上进行Taylor展开。

然后用控制方程代替和，将时间项转换为空间项

再对空间项进行中分差分离散

最后带回Taylor展开式。

可以看出它拥有二阶空间，三阶时间精度，并且是显式的。

### C:\Users\Administrator\Documents\Tencent Files\895227780\FileRecv\MobileFile\Image\G}K@LE~{DZLGWRR33RCV@17.png5、Crank-Nicolson格式

Crank-Nicolson是一种隐式格式，它具有二阶精度。如右图所示，Crank-Nicolson的空间离散点选取为两个时间步的中点。

即

其余推导与欧拉向后差分隐式相似。

### 6、MacCormark格式

MacCormark是LaxWendroff格式的一种变体，它也是二阶时间、空间精度的，也是利用空间离散时间。MacCormark是一种预报-校正格式。

首先用向前差分计算出预报步：

然后用向后差分计算出校正步：

最后两者平均，得到修正后的下一个时间点的u

注意，预报步要求出与两个值，然后再代入校正步。因此，可以先进行预报步的在所有空间步上的循环，然后再进行校正步的循环。

## 五、源程序

### 1. 除中心差分隐式以外

#include<iostream>

#define I 101

using namespace std;

double x=0.0;

double const dx=0.01, dt=0.01, alpha=0.7, C=alpha\*dt/dx ;

int i=0,n=0;

int const maxI=1.0/dx;

double const maxN=0.7/alpha/dt;

double u[I]={0.0}, u1[I]={0.0}, u0[I]={0.0};

void update();

void first2Steps(int flag);

void Explicit();

void Implicit();

void FogJump();

void CrankNicolson();

void MacCormark();

void LaxWendroff();

int main()

{

//u1: next time step u; u0 former timestep

// I.C. @n=0, x=[0.1,0.2] : u=100; @anywhere else, u=0;

for(i=10, x=0.1; x<=0.2; i++, x+=dx)

{

u[i]=100;

}

cout<<"maxI="<<maxI<<endl;

cout<<"maxN="<<maxN<<endl;

//B.C. @x=0 : u1=0

u1[0]=10;

// u[459]=0;

//Explicit

//Explicit();

//LaxWendroff();

//FogJump();

MacCormark();

//Implicit

//Implicit();

//CrankNicolson();

//print

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

cout<<u[i]<<"\n";

return 0;

}

void update()

{

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

{

u[i]=u1[i];

}

}

inline void first2Steps(int flag)

{

for(i=1;i<=flag;i++)

u1[i]=u[i] - C \* ( u[i+1] - u[i]);

}

// Euler Explicit & Central Diff Explicit

void Explicit()

{

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

{

//change the flag value to accomodate different schemes

int flag=1; //flag==0 means no need to solve the first 2 space step with Euler Explicit

if(flag>=1) //flag==1 means need to solve the first step, flag==2 means need to solve the first and second step

first2Steps(flag);

for(i=flag; i<=maxI-1; i++)

{

//u1[i]=u[i] - C \* ( u[i+1] - u[i]);//Euler Forewards Explicit //instable!!

u1[i]=u[i] - C \* ( u[i] - u[i-1]);//Euler Backwards Explicit //flag=1

//u1[i]=u[i] - C/2 \* ( u[i+1] - u[i-1]);//Central Differencing Explicit// instable!! //falg=1

//u1[i] = u[i] - C/2 \* ( 3\*u[i+1] -4\*u[i] + u[i-1] );//Triple Point Euler Central Explicit //instable!! //flag=1

//u1[i] = u[i] - C/2 \* ( 3\*u[i] -4\*u[i-1] + u[i-2] );//Triple Point Euler Backwards Explicit //instable!!// flag=2

}

u1[maxI]=u1[maxI-1]; //right BC: zero gradient

update();

}

}

//FogJump Explicit

void FogJump()

{

n=1;

for(i=1;i<=maxI;i++) //must solved by Euler Explict before Fogjump, to get value when n=1 (t=dt).

u1[i]=u[i] - C \* ( u[i+1] - u[i]);

for(n=2; n<=maxN; n++)

{

int flag=1;

first2Steps(flag);

for(i=flag; i<=maxI-1; i++)

{

u1[i] = u0[i] - C\*( u[i+1] - u[i-1] ); //FogJump instable!!

}

u1[maxI]=u1[maxI-1];

update();

}

}

//Lax-Wendroff Explicit

void LaxWendroff()

{

int flag=1;

first2Steps(flag);

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

{

for(i=flag; i<=maxI-1; i++)

{

u1[i] = u[i] - C/2 \* ( u[i+1] - u[i-1] ) + C\*C/2 \* (u[i+1] - 2\*u[i] + u[i-1]); //LaxWendroff

}

u1[maxI]=u1[maxI-1];

update();

}

}

//Euler Implicit

void Implicit()

{

int flag=1;

if(flag>=1) first2Steps(flag);

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

{

for(i=flag; i<=maxI-1; i++)

{

//u1[i]=( u[i] - C \* u1[i+1] ) / (1-C); //Euler Forewards Implicit //instable!!//flag=0

u1[i]=( u[i] + C \* u1[i-1] ) / (1+C); //Euler Backwards Implicit //flag=1

}

u1[maxI]=u1[maxI-1];

update();

}

}

//Crank-Nicolson

void CrankNicolson()

{

int flag=1;

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

{

first2Steps(flag);

for(i=flag;i<=maxI-1;i++)

{

u1[i] = ( C/2 \* u1[i-1] + ( 1 - C/2 ) \* u[i] + C/2 \* u[i-1] ) / ( 1 + C/2 );

}

u1[maxI]=u1[maxI-1];

update();

}

}

//MacCormark

void MacCormark()

{

int flag=1;

double dudt[I], dudt1\_, dudtav, u1\_[I];

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

{

first2Steps(flag);

//predict

for(i=flag;i<=maxI-1;i++)

{

dudt[i] = -alpha \* (u[i+1] - u[i]) /dx;

u1\_[i] = u[i]+ dudt[i] \* dt;

}

//revise

for(i=flag;i<=maxI-1;i++)

{

dudt1\_ = -alpha \* (u1\_[i] - u1\_[i-1] ) /dx;

dudtav = 0.5\* (dudt1\_ + dudt[i] );

u1[i] = u[i] + dudtav \* dt;

}

u1[maxI]=u1[maxI-1];

update();

}

}

### 2. 中分差分隐式

/\*---------------------------------------------------------------------------\*\

Description

Implementing the Central Differencing Implicit Discretization for transport equation, i.e. :

ddt(u) + a \* ddx(u) == 0;

after discretizing, it becomes :

-C/2 \* u1[i-1] + u1[i] + C/2 \* u1[i+1] == u[i];

where C = a\*dt/dx is the Courant number;

u1 denotes velocity in next time step, and u denotes velocity in current time step.

Essentially, it is a tridiagonal matrix equation, i.e. :

A x == f

To solve this tridiagonal equation, we use Thomas Algorithm,

which is the main object of this cpp file.

\\*---------------------------------------------------------------------------\*/

#include<iostream>

#define N 100

using namespace std;

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

int main()

{

//define variables and matrix. maxT denotes maximum running time, a.k.a stop time

const double dx=0.01, dt=0.01, alpha=0.7, C=alpha\*dt/dx, maxT=0.7/alpha;

//x[] denotes solution vector, in this case standing for u[] in next time step, which is to solve.

//f[] denotes constant vector, in this case standing for u[] in current time step, which is already known.

//beta[] is the auxilary vector, storing the corresponding coefficients after LU.

double t=0, x[N+2]={0}, f[N+2]={0}, beta[N], y[N];

int i=0;

//I.C.

for(i=10; i<=20; i++)

{

f[i]=100;

}

//B.C.

x[0]=10;

double a=-C/2, b=1.0, c=C/2;

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

//Thomas Algorithm Begin

//LU, once is enough

beta[1]=c/b;

for(i=2;i<=N-1;i++)

{

beta[i]=c / ( b - a \* beta[i-1] );

}

//In every time step, solve Ly=f, Ux=y

for(t=0; t<=maxT; t+=dt)

{

//B.C. in the right side: zero gradient

x[N+1]=f[N];

//treatment of maxtix, change it to tridiagonal

f[1] = f[1] - a \* x[0];

f[N] = f[N] - c \* x[N+1];

//chasing, to solve Ly=f

y[1] = f[1] / b;

for(i=2;i<=N;i++)

{

y[i]=( f[i] - a \* y[i-1] ) / (b - a \* beta[i-1] );

}

//passing, to solve Ux=y

x[N]=y[N] ;

for(i=N-1;i>=1;i--)

{

x[i] = y[i] - beta[i] \* x[i+1];

}

//before moving to next time step, swap x and f

for(i=0;i<=N+1;i++)

{

f[i]=x[i];

}

// change back f[]

f[1] = f[1] - a \* x[0];

f[N] = f[N] - c \* x[N+1];

}

//print

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

{

cout<<x[i]<<"\n";

}

return 0;

}

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