Differenciálegyenletek numerikus megoldása
Jegyzet-féleség...

Csanád Máté

2008. december 12.

Tartalomjegyzék

1. Közönséges differenciálegyenletek

1.1. Diffegyenletek diszkretizációja, diszkretizációs lépés

Egy f függvény deriváltját az

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{1}$$

formulával definiáljuk. Ezt a deriváltat közelítőleg számolhatjuk numerikusan, a probléma által megkövetelt pontosságnak megfelelően kicsi Δx-et választva, a határérték elhagyásával:

$$f'(x) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x}. \tag{2}$$

Ezt nevezzük a differenciálás diszkretizálásának. A fenti formula elsőrendben pontos, azaz a hiba arányos a lépésköz, azaz Δx négyzetével. Vannak más, pontosabb formulák, amelyekben az f függvény x-en és $x+\Delta x$-en kívül más pontokban is vett értékeinek segítségével számítjuk ki az f' deriváltat az x pontban, azaz $f'(x)$-et. Általánosságban így néz ki egy diszkretizáció:

$$f'(x) \approx D(f(x_1), f(x_2), ..., f(x_n); \Delta x), \tag{3}$$

ahol x_i-k x egy környezetében lévő értékek, ezeknek egy kombinációját jelöli D.

Amennyiben egy elsőrendű differenciál-egyenletet akarunk megoldani, többnyire az alábbi egyenletünk van:

$$h(f'(x), f(x), x) = 0, \tag{4}$$

például

$$f'(x) - \lambda f(x) = 0 \tag{5}$$

vagy

$$f'(x) - \frac{2x}{x^2 - 1} = 0. \tag{6}$$

Az (4) egyenletet a diszkretizáció segítségével a

$$h(D(f(x_1), f(x_2), ..., f(x_n); \Delta x), f(x), x) = 0 \tag{7}$$

egyenletté alakíthatjuk. Ekkor vesszük az $\{x, x_1, ..., x_n\}$ halmazból a legnagyobb értéket, legyen ez x_k, és erre megoldjuk az egyenletet. Ekkor a következőre jutunk:

$$f(x_k) = H(x_1, .., x_{k-1}, x_{k+1}, .., x_n, x). \tag{8}$$

A fenti (8) egyenlet segítségével *léphetünk* egyet előre a megoldásban, ugyanis kisebb x_i értékek ismeretében kiszámítottuk egy következő x_k helyen a függvény értékét. Azaz, ha ismerjük a függvény értékét megfelelő mennyiségű kezdeti pontban (ezek a kezdeti feltételek), tetszőleges további helyeken kiszámíthatjuk a rá vonatkozó diffegyenlet segítségével!

Lássunk most a fenti sémának megfelelő módszereket!

1.2. Euler-módszer

Az Euler módszer a fenti (2) diszkretizációs formulán alapul. Ha van egy

$$f'(x) = H(f(x), x) \tag{9}$$

egyenletünk, akkor $f'(x)$-et meghatározzuk (2)-ből, és behelyettesítjük (9)-be. Ekkor a következőre jutunk:

$$\frac{f(x + \Delta x) - f(x)}{\Delta x} = H(f(x), x). \tag{10}$$

Innen pedig

$$f(x+\Delta x) = f(x) + \Delta x H(f(x), x)! \tag{11}$$

Így $f(x_0) = f_0$ ismeretében kiszámíthatjuk a $f(x+n\Delta x)$ mennyiségek közelítő értékét. Ezt az

$$f'(x) = -f(x) \tag{12}$$

diffegyenlet esetére a következőképpen programozhatjuk be:

```
#include <iostream>
#include <cmath>
#include <fstream>

using namespace std;

int main()
{
  double f0, f1; //a függvényérték az első és második pontban
  double dx, x; //a futó változó és a lépésköz
  int Nx; //a lépesek száma

  Nx = 1000;
  dx = 0.01; //így 1000 × 0.01 = 10-ig jutunk majd el

  f0 = 1; //a kezdeti érték megadása
  x = 0; //x-ben nulláról indulunk
  ofstream outfile;
  outfile.open("result.out",ios::out); //a kiemeneti fájl
                                       //megnyitása írásra
  for(int i=1;i<Nx;i++)
    {
      outfile << x << " " << f0 << endl; //írás a kimeneti fájlba
      x += dx; //a futó változó inkrementálása
      f1 = f0+dx*(-f0); //a diszkretizációs lépés
      f0 = f1; //az új függvényérték visszarása az előző helyére
    }

  return 0;
}
```

A program lefordítása és lefuttatása után elindíthatjuk a GNUPLOT nevű programot, és ábrázolhatjuk az eredményt. Ezt a következőképpen tehetjük meg:

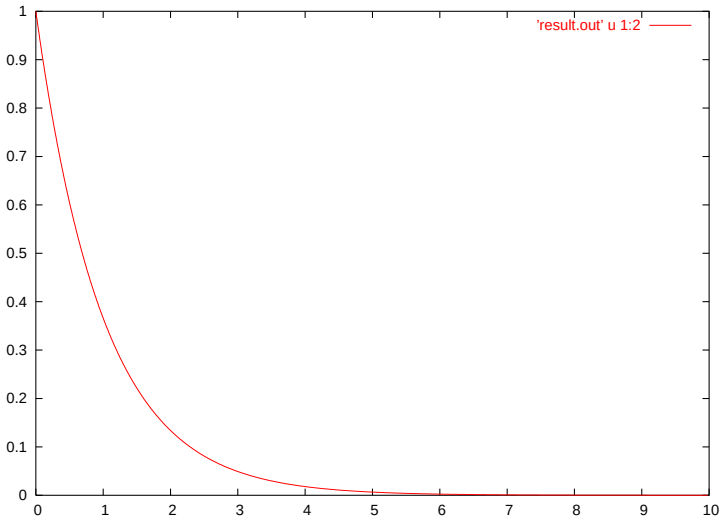
```
csanad@itl84:~/temp> g++ program.cc -o program.exe //Fordítás
csanad@itl84:~/temp> ./program.exe //Futtatás
csanad@itl84:~/temp> gnuplot //A GNUPLOT elindítása
gnuplot> plot 'result.out' u 1:2 w l //Ábrázolás vonalakkal
```

Az eredmény az 1. ábrán látható.

1.3. Runge-Kutta módszer

A fenti Euler-módszer hibája a lépésköz mégyzetével arányos, azaz

$$\Delta f \sim \Delta x^2. \tag{13}$$



1. ábra. Az Euler-módszer használatának az eredménye

A Runge-Kutta módszer alkalmazásával ennél kisebb hiba érhető el, avagy megfordítva, ugyanakkora hiba nagyobb lépésköz alkalmazásával kapható, azaz ugyanolyan távolságba kevesebb lépéssel juthatunk el, értékes processzor-időt takarítva meg ezzel. A módszer azon alapul, hogy a függvény deriváltját másik pontban tekintjük, mint ahonnan indulunk.

Továbbra is az (4) diffegyenletet akarjuk megoldani. Ehhez bevezetjük a

$$\Delta f_1 = \Delta x h(x_n, f_n) \text{ és} \tag{14}$$

$$\Delta f_2 = \Delta x h(x_n + \frac{\Delta x}{2}, f_n + \frac{\Delta f_1}{2}) \tag{15}$$

mennyiségeket. Ezekkel pedig a

$$f_{n+1} = f_n + \Delta f_2 \tag{16}$$

lépést tesszük meg. Ha utánaszámolunk a keresett f függvény Taylor-sorának segítségével, látjuk, hogy ennek a módszernek a hibája a lépésköz köbével arányos, azaz harmadrendben pontos. Egyes tanulmányokban, programokban ezt a módszert továbbfejlesztett sokszögek módszerének hívják, és bevezetnek egy másikfajta (a Heun féle) másodrendű Runge-Kutta módszert, ahol

$$\Delta f_1 = \Delta x h(x_n, f_n) \text{ és} \tag{17}$$

$$\Delta f_2 = \Delta x h(x_n + \Delta x, f_n + \Delta f_1) \tag{18}$$

mennyiségeket. Ezekkel pedig a

$$f_{n+1} = f_n + \frac{\Delta f_1 + \Delta f_2}{2} \tag{19}$$

lépést teszik meg. Ez ugyanahoz a pontossághoz vezet.

A leggyakrabban a negyedrendű Runge-Kutta módszer használják, ez pedig a következőképpen működik. Itt a következő négy segéd-mennyiséget vezetjük be:

$$\Delta f_1 = \Delta x h(x_n, f_n), \tag{20}$$

$$\Delta f_2 = \Delta x h(x_n + \frac{\Delta x}{2}, f_n + \frac{\Delta f_1}{2}), \tag{21}$$

$$\Delta f_3 = \Delta x h(x_n + \frac{\Delta x}{2}, f_n + \frac{\Delta f_2}{2}) \text{ és} \tag{22}$$

$$\Delta f_3 = \Delta x h(x_n + \Delta x, f_n + \Delta f_3). \tag{23}$$

Ezekkel pedig:

$$f_{n+1} = f_n + \frac{\Delta f_1 + 2\Delta f_2 + 2\Delta f_3 + \Delta f_4}{6}. \tag{24}$$

1.3.1. A Runge-Kutta módszer a gyakorlatban

Ha másodrendű egyenletünk van, azaz például az

$$\ddot{I} + a\dot{I} + bI = c \tag{25}$$

egyenlet, akkor ebből elsőrendű egyenletrendszert csinálnunk, azaz

$$\dot{I} = V \tag{26}$$

$$\dot{V} = c - aV - bI. \tag{27}$$

Itt a Runge-Kutta lépéseket párhuzamosan kell elvégeznünk. A diszkrecitációs lépés programozva a következőképpen nézne ki:

```
I0=0; //kezdeti
V0=1; //értékek

kI1=dt*V0; //első segéd-mennyiség    I-re
kV1=dt*(c-a*V0-b*I0); //első segéd-mennyiség    V-re

kI2=dt*(V0+kV1/2); //második segéd-mennyiség    I-re
kV2=dt*(c-a*(V0+kV1/2-b*(I0+kI1)/2);//második segéd-mennyiség    V-re

kI3=dt*(V0+kV2/2);
kV3=dt*(c-a*(V0+kV2/2-b*(I0+kV2/2));

kI4=dt*(V0+kV3);
kV4=dt*(c-a*(V0+kV3/2-b*(I0+kI3));

V1=V0+kV1/6+kV2/3+kV3/3+kV4/6;
I1=I0+kI1/6+kI2/3+kI3/3+kI4/6;

V0=V1; //az új függvényérték visszamásolása az előzőbe
I0=I1;
```

Azaz a fenti sorokat lehetne beírni a 1.2-ben szereplő diszkretizációs lépés helyére.
Ennek a módszernek a hibája arányos a lépésköz ötödik hatványával.

1.4. Adaptív lépéshossz-változtatás

Ha egy olyan diffegyenletünk van, amelynek a megoldása egyes intervallumokon gyorsan oszcillál, míg máshol kellemesen lapos, akkor nehéz megválasztani a lépésközt, mivel ha nagyot választunk, az a gyorsan oszcilláló tartományon rossz eredményt ad, ha viszont kis lépésközt választunk, lassan haladunk.

Erre a problémára ad egy lehetséges megoldást az adaptív lépéshossz-változtatás, melynek működésének lényegét az alábbiakban láthatjuk. Tegyük fel, hogy egy adott f_n-ből indulunk, Δx lépésközzel! Azt szeretnénk, hogy a hiba ne lépjen túl egy előre megadott $\widetilde{\Delta f}$ értéket. Ekkor a következőket tesszük:

1. Megteszünk egy lépést Δx lépésközzel, így y_n-ből y_{n+1}-be jutunk.
2. Megteszünk egy lépést $\frac{\Delta x}{2}$ lépésközzel, így $y_{n+1}^{(1/2)}$-be jutunk.
3. $y_{n+1}^{(1/2)}$-ből még egy lépést megteszünk $\frac{\Delta x}{2}$ lépésközzel, így $y_{n+2}^{(1/2)}$-be jutunk.
4. Összehasonlítjuk y_{n+1}-t és $y_{n+2}^{(1/2)}$-t. Ez a két érték közelítőleg egyenlő kellene, hogy legyen.
5. A két fenti érték alapján kapunk egy becslést a hibára:

$$\Delta f = \left| y_{n+1} - y_{n+2}^{(1/2)} \right|. \tag{28}$$

6. Mivel tudjuk, hogy $\Delta f \sim \Delta x^5$ negyedrendű Runge-Kutta módszer esetén, ebből a kívánt hibához szükséges lépésköz:

$$\widetilde{\Delta x} = \Delta x \left| \frac{\widetilde{\Delta f}}{\Delta f} \right|^{\frac{1}{5}} \tag{29}$$

7. Innen két lehetőség van:
   (a) $\widetilde{\Delta x} \geq \Delta x$. Ekkor a következő lépés már lehet a nagyobb $\widetilde{\Delta x}$ méretű
   (b) $\widetilde{\Delta x} < \Delta x$. Ekkor az éppen elvégzett lépést az új, $\widetilde{\Delta x}$ lépésközzel újra el kell végezni, és a következőnél is ezt használni.
8. Vissza az első lépésre, amíg el nem értük a szükséges x távolságot.

Ezt az eljárást úgy programozhatjuk be, hogy definiálunk egy „lépő” függvényt, és azzal lépegetünk. Ez a következőképpen néz ki mondjuk az

$$f'(x) = -f(x) * x \tag{30}$$

diffegyenlet esetére:

```
double step(double x0, double f0, double dx)
{
  k1=dx*(-f0*x0);
  k2=dx*(-(f0+k1/2)*(x0+dx/2));
  k3=dx*(-(f0+k2/2)*(x0+dx/2));
  k4=dx*(-(f0+k3)*(x0+dx));

  return f0+k1/6+k2/3+k3/3+k4/6;
}
```

A `main()` függvénybe pedig a következők kerülnek:

```
int main()
{
  double f0, f1, f11, f12, df, dx;        //változók deklarálása
  double df0=0.01;                        //az elvárt pontosság
  double dx =0.01;                        //a kezdeti lépésköz
  for(int i=0;i++;i<imax)
    {
       f1 = step(x0, f0, dx);             //egész lépés
       f11= step(x0, f0, dx/2);           //első fél lépés
       f12= step(x0+dx0/2, f0+f11, dx/2); //második fél lépés
       df = fabs(f1-f12);                 //becslés a hibára
```

```
        dx0= dx*pow(fabs(df0/df),0.2);      //a df0 hibához
                                              szükséges lépésköz
        if(dx0<dx)
          f1 = step(x0, f0, dx);            //az előző lépés megismétlése,
                                              már a kisebb lépésközzel
        dx = dx0;
        f0 = f1;
        x0+= dx;
        outfile << x0 << " " << f0 << endl;//írás a kimeneti fájlba
      }
  }
```

Még hasznosabb, ha még a diffegyenlet jobb oldalán álló függvényt is külön definiáljuk, azaz bevezetjük a

```
double h(double f, double x)
{
  return -f*x;
}
```

függvényt, és ekkor a `step(double,double,double)` függvénybe az alábbiakat írhatjuk:

```
k1=dx*h(f0,x0);
k2=dx*h(f0+k1/2,x0+dx/2);
k3=dx*h(f0+k2/2,x0+dx/2);
k4=dx*h(f0+k3,x0+dx);
```

Sőt, lehet két lépést is definiálni, egy `rk_step(double,double,double)`-t és egy `euler_step(double,double,double)`-t, ahol az előbbi a fentieknek megfelelően, az utóbbit pedig egyszerűen az Euler-módszer alapján egy

```
return f0+dx*h(f0,x0);
```

sorral definiálhatjuk.

2. Parciális differenciálegyenletek

2.1. Diszkretizáció

Egy egyszerű parciális diffegyenlet a következő például:

$$\frac{\partial \rho}{\partial t} + v\frac{\partial \rho}{\partial x} = 0 \tag{31}$$

Ebben kétféle derivált is szerepel, és mindkettőt diszkretizálnunk kell. Ezt megtehetjük az Euler-módszerrel. Itt viszont két indexe lesz a diszkretizált változónak, egy az idő-rácson, egy pedig a tér-rácson vett helyzetét adja meg. Ennek az egyenletnek egyetlen paramétere van, méghozzá $v\Delta t/\Delta x$, ezt elnevezzük majd C-nek, ez a Courant-szám. A diszkretizát egyenlet a következőképpen fog kinézni:

$$\rho_j^{n+1} - \rho_j^n + v\frac{\Delta t}{\Delta x}\left(\rho_{j+1}^n - \rho_j^n\right) = 0 \tag{32}$$

Igen egyszerű megoldást kaptunk így, ugyanis kifejezzük ρ_j^{n+1}-et a fenti egyenletből:

$$\rho_j^{n+1} = \rho_j^n - C\left(\rho_{j+1}^n - \rho_j^n\right) \tag{33}$$

Így $\forall i$-re ismert ρ_i^n esetén kiszámíthatjuk ρ_i^{n+1}-t $\forall i$-re. Természetesen szükségünk van valami peremfeltételre, hogy a térbeli határokon hogyan viselkedjen a diszkretizált változó, ugyanis amikor a határon vagyunk, azaz $j = j_{max}$, a $j+1$-edig indexnek nincs értelme. Két tipikus választás van: vagy a szélső ponton fixálom nullára, egyre vagy bármely értékre a megoldást, vagy peridódikus határfeltételt veszek, azaz a j_{max+1}-edik pontot azonosítom az elsővel. Természetesen mindkét esetben a kezdeti feltételnek, azaz ρ_i^0-nek is teljesítenie kell a peremfeltételt.

2.2. Stabilitás

Sajnos a parciális differenciálegyenleteknél azonban nem ilyen egyszerű az élet, a fenti diszkretizáció ritkán működik. Ugyanis a „valódi" megoldáshoz mindig hozzáadhatunk egy olyan megoldást, ami a rácspontokon nulla, de közbe oszcillál. Ez a rész-megoldás minden határon túl felerősödhet, és elnyomhatja az eredeti megoldást. Ezért egy speciális diszkretizáció csak a Δx, Δt és az egyéb paraméterek speciális megválasztása esetén működik. Azt úgy vizsgáljuk, hogy bevezetjük a

$$\rho_j^n = G^n \mathrm{e}^{\mathrm{i}k\Delta x j} \tag{34}$$

megoldást. Ennek hullámszáma k, amelynek értékétől függő a megoldás periódicitása. G a hullám amplitúdóját jelöli, és ha ez (vagyis ennek abszolútértéke) nagyobb egynél, akkor a megoldás az időben előrehaladva minden határon túl nő.

Vizsgáljuk meg, az (33). egyenletben szereplő diszkretizációnak milyen feltételekkel megoldása (34)! Ehhez egymásba helyettesítjük őket, egyszerűsítünk $G^n \exp^{\mathrm{i}k\Delta x j}$-vel, és ami marad:

$$G = 1 - C(\mathrm{e}^{\mathrm{i}k\Delta x} - 1) \tag{35}$$

Az ennek az egyenletnek megfelelő G, C és Δx értékek esetén a fenti, (34)-ban szereplő függvény megoldás lesz. Most lássuk a $|G| < 1$ feltételt!

$$|G| = (1 + C - C\cos(k\Delta x))^2 + C^2 \sin^2(k\Delta x) \tag{36}$$

Ezen láthatjuk, hogy G abszolútértéke mindig nagyobb mint egy, azaz ez a diszkretizáció mindig instabil. Most lássunk speciális egyenltekre speciális diszkretizációkat!

2.3. Kontinuitási egyenlet

A fent említett parciális differenciálegyenlet a kontinuitási egyenlet. Erre nézünk most speciális diszkretizációkat.

2.3.1. FTCS

Az FTCS (Forward in Time, Centered in Space) módszer a következőképpen néz ki:

$$\frac{\rho_j^{n+1} - \rho_j^n}{\Delta t} + v\frac{\rho_{j+1}^n - \rho_{j-1}^n}{2\Delta x} = 0 \tag{37}$$

Ezt az 2.2-ben említett módszerrel megvizsgálva a következőre jutunk:

$$G = 1 - C\mathrm{i}\sin(k\Delta x) \tag{38}$$

Tehát ez feltétel nélkül instabil.

2.3.2. Upwind

Az upwind módszerben a térbeli deriváltat nem centráljuk, hanem áramlásirányból vesszük, azaz:

$$\frac{\rho_j^{n+1} - \rho_j^n}{\Delta t} + v\frac{\rho_j^n - \rho_{j-1}^n}{\Delta x} = 0 \tag{39}$$

vagy

$$\frac{\rho_j^{n+1} - \rho_j^n}{\Delta t} + v\frac{\rho_{j+1}^n - \rho_j^n}{\Delta x} = 0 \tag{40}$$

ha $v < 0$. Ekkor a stabilitásra a következőt kapjuk:

$$G = 1 - |C| + |C|\mathrm{e}^{\mathrm{i}k\Delta x}. \tag{41}$$

Ez a komplex szám az egységkörön belül van, ha $|C| < 1$, azaz ekkor stabil a diszkretizáció.

2.3.3. Leapfrog

Itt

$$\frac{\rho_j^{n+1} - \rho_j^{n-1}}{2\Delta t} + v\frac{\rho_{j+1}^n - \rho_{j-1}^n}{2\Delta x} = 0. \tag{42}$$

Ez a módszer szintén $|C| < 1$ esetén stabil, viszont térben és időben is másodrendűen pontos, szemben az előzőekkel.

2.3.4. Lax-Wendorff

Itt a kontinuitási egyenlet kétszer alkalmazzuk:

$$\frac{\partial^2\rho}{\partial t^2} + v^2\frac{\partial^2\rho}{\partial x^2} = 0. \tag{43}$$

Mivel a ρ Taylor-sorából az

$$\frac{\rho_j^{n+1} - \rho_j^n}{\Delta t} = \frac{\partial\rho}{\partial t} + \frac{\Delta t}{2}v^2\frac{\partial^2\rho}{\partial x^2} + ... \tag{44}$$

következik, a (43) egyenletet kihasználva a következőt írjuk fel:

$$\frac{\partial\rho}{\partial t} = \frac{\rho_j^{n+1} - \rho_j^n}{\Delta t} - \frac{\Delta t}{2}v^2\frac{\partial^2\rho}{\partial x^2}. \tag{45}$$

Így a következő, pontosabb diszkretizációt kapjuk:

$$\rho_j^{n+1} = \rho_j^n - \frac{C}{2}(\rho_{j+1}^n - \rho_{j-1}^n) + \frac{C^2}{2}(\rho_{j+1}^n - 2\rho_j^n + \rho_{j-1}^n). \tag{46}$$

Ez a módszer térben harmadrendig pontos, és szintén $|C| < 1$ esetén stabil.

2.3.5. McCormack

Ennél a módszernél két segéd-változót vezetünk be:

$$\rho_j^{\#} = \rho_j^n - C(\rho_j^n - \rho_{j-1}^n) \tag{47}$$

$$\rho_j^{\#\#} = \rho_j^n - C(\rho_{j+1}^n - \rho_j^n), \tag{48}$$

azaz egyszer jobbról, egyszer balról vesszük a térbeli deriváltat. Itt ismét bevezettük C-t, a Courant-számot. Ezekkel aztán kifejezzük az $n+1$-edik időpontban vett ρ-t:

$$\rho_j^{n+1} = \frac{\rho_j^{\#} + \rho_j^{\#\#}}{2} \tag{49}$$

Ha az a (47)-(47) egyenleteket behelyettesítjük (49)-ba, akkor a következőt kapjuk:

$$\rho_j^{n+1} = \rho_j^n - \frac{C}{2}(\rho_{j+1}^n - \rho_{j-1}^n) + \frac{C^2}{2}(\rho_{j+1}^n - 2\rho_j^n + \rho_{j-1}^n) \tag{50}$$

Ez a módszer is $|C| < 1$ esetén stabil, viszont egyszerűen számolható. A kontinuitási egyenlet esetén ez a módszer megegyezik az 2.3.4-ben tárgyalt Lax-Wendorff módszerrel.

2.3.6. Crank-Nicholson

Ennél a módszernél az FTCS-et úgy módosítjuk, hogy a térbeli deriváltat az n-edik és az $n+1$-edik időpillanat átlagából számoljuk, azaz:

$$\frac{\rho_j^{n+1} - \rho_j^n}{\Delta t} + v\frac{\rho_{j+1}^n - \rho_{j-1}^n + \rho_{j+1}^{n+1} - \rho_{j-1}^{n+1}}{4\Delta x} = 0 \tag{51}$$

Ennél a módszer amplitúdóban pontos, és feltétel nélkül stabil.

2.4. Diffúziós egyenlet

A diffúziós egyenlet a következőképpen néz ki:

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2} \tag{52}$$

Itt az

$$s = \alpha \frac{\Delta t}{\Delta x^2} \tag{53}$$

változót vezetjük be.

2.4.1. FTCS

Ugye ez a módszer a Forward in Time, Centered in Space, és a diszkretizációja a következőképpen néz ki:

$$\frac{T_j^{n+1} - T_j^n}{\Delta t} = \alpha \frac{T_{j+1}^n - 2T_j^n + T_{j-1}^n}{\Delta x^2}. \tag{54}$$

Ezt átrendezve:

$$T_j^{n+1} = T_j^n + s(T_{j+1}^n - 2T_j^n + T_{j-1}^n) \tag{55}$$

Felírjuk a stabilitási egyenletet:

$$G = 1 + s\left(\mathrm{e}^{\mathrm{i}k\Delta x} - 2 + \mathrm{e}^{-\mathrm{i}k\Delta x}\right) \tag{56}$$

Ezt átalakítva:

$$G = 1 - 4s \sin^2 \frac{k\Delta x}{2} < 1, \tag{57}$$

azaz ez a módszer az $s < 1/2$ feltétellel stabil.

2.4.2. Richardson

Ez a módszer a kontinuitási egyenletnél említett Leapfrog módszernek (lásd 2.3.3) felel meg. Itt az időbeli deriváltat is szimmetrikusan vesszük:

$$\frac{T_j^{n+1} - T_j^{n-1}}{2\Delta t} = \alpha \frac{T_{j+1}^n - 2T_j^n + T_{j-1}^n}{\Delta x^2}. \tag{58}$$

Ez a módszer itt azonban instabil.

2.4.3. Du Fort – Frankel

Ez az előzőn javít, a térbeli deriváltat módosítjuk csak kicsit:

$$\frac{T_j^{n+1} - T_j^{n-1}}{2\Delta t} = \alpha \frac{T_{j+1}^n - (T_j^{n+1} + T_j^{n-1}) + T_{j-1}^n}{\Delta x^2}. \tag{59}$$

Ez a módszer feltétel nélkül stabil!

2.4.4. Implicit módszer

Ez ugyanaz, mint az FTCS, csak a térbeli deriváltat az „új” időpillanatban vesszük:

$$\frac{T_j^{n+1} - T_j^n}{\Delta t} = \alpha \frac{T_{j+1}^{n+1} - 2T_j^{n+1} + T_{j-1}^{n+1}}{\Delta x^2}. \tag{60}$$

Ez a módszer is feltétel nélkül stabil, ugyanakkor implicit, tehát egyenletrendszert kell megoldani az időben való előrelépéshez. Ez megtehető különféle algoritmusokkal, pl a Thomas-algoritmus ilyen. Ezt most nem részletezném inkább.

2.4.5. Crank-Nicholson

Itt a térbeli deriváltat a két időpillanat átlagaként vesszük:

$$\frac{T_j^{n+1} - T_j^n}{\Delta t} = \alpha \left[\frac{T_{j+1}^{n+1} - 2T_j^{n+1} + T_{j-1}^{n+1}}{\Delta x^2} + \frac{T_{j+1}^n - 2T_j^n + T_{j-1}^n}{\Delta x^2} \right]. \tag{61}$$

Ez a módszer is feltétel nélkül stabil, és másodrendben pontos is.

2.5. Transzport egyenlet

A transzport egyenlet a kontinuitási és a diffúziós egyenletek keveréke:

$$\frac{\partial T}{\partial t} + v\frac{\partial T}{\partial x} = \alpha \frac{\partial^2 T}{\partial x^2} \tag{62}$$

Íme egy összefoglalás a fent említett módszerek ezen az egyenleten való alkalmazhatóságáról, azaz arról, hogy melyik módszer mikor stabil és mikor pontos. Itt használjuk a korábban bevezetett C és s konstansokat.

Név	Stabilitás	Pontosság
FTCS	$C^2 < 2s < 1$	$C^2 \ll 2s$
Richardson-Leapfrog	$s = 0$	
Du Fort – Frankel	$\|C\| < 1$	$C^2 \ll 1$
Upwind	$\|C\| + 2s < 1$	$\Delta t \Delta x \ll 2/(1 - \|C\|)$
Lax-Wendorff	$C^2 + 2s < 1$	$\Delta t \Delta x \ll 2$
Crank-Nicholson	Stabil!!	$\Delta t \Delta x \ll 2$

3. Függelék

3.1. Paraméterek beolvasása fájlból

Ha van egy

```
egy 1
ketto 2
harom 3
```

tartalmú `in.dat` fájlunk, és lefuttatjuk a

```
#include <iostream>
#include <string>
#include <fstream>

int main()
{
  ifstream infile;
  infile.open("in.dat",ios::in);
  string nev;
  double ertek;
  while(infile >> nev >> ertek)
  {
    if(nev=="harom")
      cout << "Es a harmadik:" << endl;
    cout << nev << "=" << ertek << endl;
  }
  cout << "Ennyi." << endl;
```

```
    return 0;
  }
```
proba.cc programot, akkor az a következőt írja ki:
```
  csanad@login03:~/temp> g++ proba.cc -o proba.exe
  csanad@login03:~/temp> ./proba.exe
  egy=1
  ketto=2
  Es a harmadik:
  harom=3
  Ennyi.
  csanad@login03:~/temp>
```

3.2. Példa osztályok használatára

Egy osztályt a következőképpen definiálhatunk:
```
  class diffegy
  {
    private:
      int N;
      double a, b, x0, f0;

    public:
      string input, output;
      ifstream infile;
      ofstream outfile;
      diffegy(string input2, string output2);
      double func(double xx, double ff);
      double step(double xx, double ff, double ddx);
      void advance();
      ~diffegy();
  };
```
Ekkor lesznek belső (`private`) és külső (`public`) változói. Az osztály tetszőleges tagfüggvénye (pl. a `double func(double xx, double ff)`) elérheti ezeket a változókat, azonban a belsőket *csak* a tagfüggvények érhetik el.

Az osztállyal azonos nevű függvény az osztály *konstruktora*, az osztály egy objektumát inicializálva ez a függvény fut le. A `~diffegy()` függvény az osztály destruktora, a program végén fut le.

Most definiáljuk a konstruktort! A két változója a ki- és bemeneti fájl neve, benne inicializáljuk a változók értékeit és a két fájlt.
```
  diffegy::diffegy(string input2, string output2)
  {
    output=output2;
    input=input2;
    double ertek;
    string tipus;

    infile.open(input,ios::in);
    if(!infile)
    {
      cout<<input<<" nem megnyithato!"<<endl;
      exit(0);
    }
```

```
    else
      cout<<"Parameterek beolvasasa "<<input<<"-bol..."<<endl;
    while(infile>>tipus>>ertek)
    {
      if(tipus=="#a") a = ertek;
      if(tipus=="#b") b = ertek;
      if(tipus=="#x0") x0 = ertek;
      if(tipus=="#f0") f0 = ertek;
      if(tipus=="#N") N = (int)ertek;
      if(tipus=="#dx0") dx = ertek;
      if(tipus=="#df0") df = ertek;
    };
    infile.close();
    cout<<"Parameterek beolvasva."<<endl;

    outfile.open(output,ios::out);
    if(!outfile)
    {
      cout<<output<<"nem megnyithato!"<<endl;
      exit(0);
    }
    else
      cout<<output<<" megnyitasa irasra..."<<endl;

    outfile<<"#Kimeneti fajl. Oszlopok: x, f"<<endl;
    outfile.close();
  };
```

Ezek után definiáljuk a „lépő” függvényt és a diffegyenlet jobb oldalát:

```
  double step(double xx, double ff, double ddx)
  {
    // pl Runge-Kutta lepes kerulhet ide, lásd 1.3
  };

  double func(double xx, double ff)
  {
    // itt kell definiálni a 'jobb oldalt', lásd 1.4
  };
```

Most pedig definiáljunk egy szubrutint, ami már a kimeneti fájlba írja az eredményt:

```
  void calculate()
  {
    //ide kerul a korábbi fő-függvény, lásd 1.4
  };
```

Végül a destruktor:

```
  diffegy::~diffegy()
  {
    cout<<"Desctructing done."<<endl;
  };
```

És már jöhet is a fő-függvény. Ennek most van két argumentuma, azaz

```
  csanad@itl84:~/temp>./diffegy.exe input.dat output.dat
```

módon lehet meghívni. Ebben inicializálunk egy `A` nevű, `diffegy` típusú objektumot, majd meghívjuk ennek a `calculate()` tagfüggvényét, és kész is!

```
  int main(int argc, char *argv[])
  {
```

```
  int tip;
  if(argc!=3)
  {
    cerr<<"args: inputfile outputfile"<<endl;
    return 0;
  };

  cout<<"Inputfile: " <<argv[1]<<"; ";
  cout<<"Outputfile: "<<argv[2]<<endl;

  diffegy A(argv[1],argv[2]);
  cout<<"Constructing done."<<endl;

  A.calculate();

  return 0;
};
```

3.3. Animációk készítése gnuplottal

Animációt a következőképpen készíthetünk gnuplot segítségével. Először is létrehozzuk az adatfájlt úgy, hogy az egyes időpillanatoknak megfelelő adatok külön oszlopokba vannak írva. Ezek után készítünk egy plot fájlt, és elmentjük mondjuk `animate.plt` néven:

```
i=i+1
plot "$0" u i w l
pause $1
if(i<N) reread
```

Itt az `i` változó fog futni `N`-ig. A `$0` és `$1` változók az `animate.plt` program betöltésekor adott argumentumok, a második azt határozza meg, hogy mekkora szünet legyen az egyes frame-ek között. A fenti plot fájlt egy `main.plt` script fogja meghívni, ez a következőképpen néz ki:

```
i=0
N=80
unset key
set xrange [0:100]
set yrange [0:20]
call 'animate.plt' 'data.txt'
```

A fenti kis scriptben szereplő számokat természetesen szintén argumentumként is megadhatjuk, a `$0`, `$1` ... használatával. A scriptet gnuplotban a

```
gnuplot> load 'main.plt'
```

paranccsal hívhatjuk meg. Ha viszont a `main.plt` legelejére beírjuk a

```
#!/usr/bin/gnuplot
```

sort, akkor közvetlenül shellből is futtatható lesz (amennyiben a gnuplot valóban a fenti helyen van – ez a `which gnuplot` paranccsal ellenőrizhető).

3.4. Oszlopok írása fájlba

A `C++` nyelvben nincs közvetlen lehetőség arra, hogy egy fájl n. sorához hozzáfűzzünk egy karaktert. Azonban létezik például egy `getline()` parancs, amellyel ki lehet olvasni a teljes aktuális sort egy karakter-állományba. Ehhez aztán hozzáfűzhetünk egy számot, egy stringet, bármit, és beírhatjuk egy új fájlba.

Erre példa a következő szubrutin. Négy argumentuma van, az eredeti és az új fájl neve,

illetve egy vektor, amelynek elemit a sorok végéhez hozzáfűzi, végül a sorok maximális mérete, ez utóbbit tárolja a `size` integer. A szubrutin tehát:

```
int append(char * inf, char * outf, double * v, int size)
{
  char * temp;
  ifstream infile;          //a bemeneti fájl
  infile.open(inf,ios::in);
  ofstream outfile;         //a kimeneti fájl
  outfile.open(outf,ios::out);

  for(int i = 0; i < 5; i++)
  {
    temp = new char[size]; //ideiglenes karaktertömb
    infile.getline(temp,size);
    outfile << temp << setw(10) << v[i] << endl;
    delete temp;
  }
  infile.close();
  outfile.close();
  return 1;
}
```

Ezt felhasználva a `main()` függvénybe írhatjuk például a következőt:

```
int main()
{
  ofstream outfile;
  outfile.open("data.txt",ios::out);
  double * v;
  v = new double[5];
  for(int j = 0; j < 5; j++)
  {
    v[j] = 2.0*j + 2.0;
    outfile << setw(10) << v[j] << endl;
  }                         //itt beírtuk az első oszlopot,
                            //amely a kezdeti feltételt jelenti
  outfile.close();
  for(int j = 1; j < 6; j++)
  {
    calc(v);                //kiszámoljuk az új v-t
    append("data.txt","temp.txt",v,2*j*11);
    calc(v);                //megint újra kiszámoljuk
    append("temp.txt","data.txt",v,(2*j+1)*11);
  }
  return 1;
}
```

Ezt a programot lefuttatva, ha valahol definiáltuk a `calc(double * v)` függvényt, akkor 13 oszlopot írunk a `data.txt` fájlba egymás után, és csinálunk egy ideiglenes `temp.txt` fájlt is, amelyb az első 12 oszlopot tartalmazza.

Természetesen ez csak egy nulladik lépés, egy példa, hogy hogyan lehet ilyesmit csinálni, hogyan lehet oszlopokat írni egy fájlba anélkül, hogy a fájl teljes tartalmát előre ismernénk. A feladat konkrét jellegének megfelelően kell a fenti kezdeményt módosítgatni, pl. dinamikusan lefoglalni a memóriában a tömböket, osztályokat használni. A fenti leírás csak egy keretet, egy ötletet akar adni ehhez.