13

Solução numérica de equações diferenciais parciais

13.1 – Advecção pura: a onda cinemática

Considere a equação

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0, \qquad u(x,0) = g(x). \tag{13.1}$$

A sua solução pode ser obtida pelo método das características, e é

$$u(x,t) = g(x - ct). \tag{13.2}$$

Seja então o problema

$$\frac{\partial u}{\partial t} + 2\frac{\partial u}{\partial x} = 0, \tag{13.3}$$

$$u(x,0) = 2x(1-x). \tag{13.4}$$

A condição inicial, juntamente com $u(x,1)$, $u(x,2)$ e $u(x,3)$ estão mostrados na figura 13.1. Observe que a solução da equação é uma simples onda cinemática.

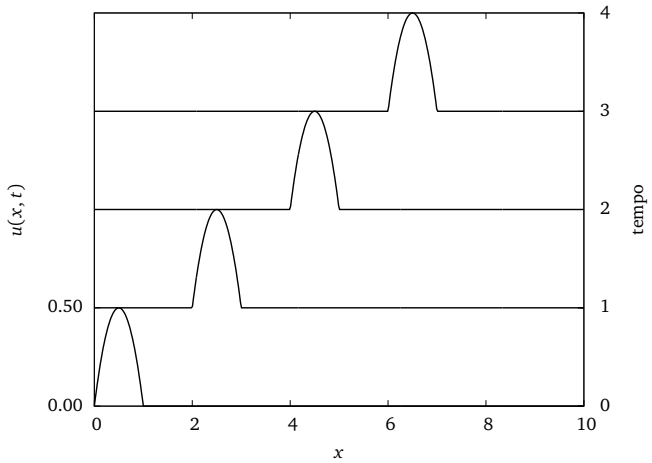

Figura 13.1: Condição inicial da equação 13.3.

Vamos adotar a notação

$$u_i^n \equiv u(x_i, t_n), \tag{13.5}$$
$$x_i = i\Delta x, \tag{13.6}$$
$$t_n = n\Delta t, \tag{13.7}$$

com

$$\Delta x = L/N_x, \tag{13.8}$$
$$\Delta t = T/N_t \tag{13.9}$$

onde L, T são os tamanhos de grade no espaço e no tempo, respectivamente, e N_x, N_t são os números de divisões no espaço e no tempo.

Uma maneira simples de transformar as derivadas parciais em diferenças finitas na equação (13.3) é fazer

$$\left.\frac{\partial u}{\partial t}\right|_{i,n} = \frac{u_i^{n+1} - u_i^n}{\Delta t} + O(\Delta t), \tag{13.10}$$
$$\left.\frac{\partial u}{\partial x}\right|_{i,n} = \frac{u_{i+1}^n - u_{i-1}^n}{\Delta x} + O(\Delta x^2). \tag{13.11}$$

Substituindo na equação (13.3), obtemos o esquema de diferenças finitas *explícito*:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = -c\left(\frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x}\right),$$
$$u_i^{n+1} = u_i^n - \frac{c\Delta t}{2\Delta x}(u_{i+1}^n - u_{i-1}^n), \tag{13.12}$$

(com $c = 2$ no nosso caso). Esse é um esquema incondicionalmente *instável*, e vai fracassar. Vamos fazer uma primeira tentativa, já conformados com o fracasso antecipado. Ela vai servir para desenferrujar nossas habilidades de programação de métodos de diferenças finitas.

O programa que implementa o esquema instável é o `onda1d-ins.py`, mostrado na listagem 13.1. Por motivos que ficarão mais claros na sequência, nós escolhemos $\Delta x = 0{,}01$, e $\Delta t = 0{,}0005$.

O programa gera um arquivo de saída binário, que por sua vez é lindo pelo próximo programa na sequência, `surf1d-ins.py`, mostrado na listagem 13.2. O único trabalho deste programa é selecionar algumas "linhas" da saída de `onda1d-ins.py`; no caso, nós o rodamos com o comando

```
[ surf1d-ins.py 3 250 ] ,
```

o que significa selecionar 3 saídas (além da condição inicial), de 250 em 250 intervalos de tempo Δt. Observe que para isto nós utilizamos uma lista (`v`), cujos elementos são `arrays`.

O resultado dos primeiros 750 intervalos de tempo de simulação é mostrado na figura 13.2. Repare como a solução se torna rapidamente instável. Repare também como a solução numérica, em $t = 750\Delta t = 0{,}375$, ainda está bastante distante dos tempos mostrados na solução analítica da figura 13.1 (que vão até $t = 4$). Claramente, o esquema explícito que nós programamos jamais nos levará a uma solução numérica satisfatória para tempos da ordem de $t = 1$!

Listagem 13.1: `onda1d-ins.py` — Solução de uma onda 1D com um método explícito instável

```
#!/usr/bin/python
# -*- coding: iso-8859-1 -*-
# ------------------------------------------------------------
# onda1d-ins resolve uma equação de onda com um método
# explícito
#
# uso: ./onda1d-ins.py
# ------------------------------------------------------------
from __future__ import print_function
from __future__ import division
fou = open('onda1d-ins.dat','wb')
dx = 0.01
dt = 0.0005
print('# dx = %9.4f' % dx)
print('# dy = %9.4f' % dt)
from numpy import zeros
nx = int(10.0/dx)             # número de pontos em x
nt = int(1.0/dt)              # número de pontos em t
print('# nx = %9d' % nx)
print('# nt = %9d' % nt)
u = zeros((2,nx+1),float)     # apenas 2 posições no tempo
                              # são necessárias!
def CI(x):                    # define a condição inicial
   if 0 <= x <= 1.0:
      return 2.0*x*(1.0-x)
   else:
      return 0.0
for i in range(nx+1):         # monta a condição inicial
   xi = i*dx
   u[0,i] = CI(xi)
u[0].tofile(fou)              # imprime a condição inicial
old = False
new = True
c = 2.0                      # celeridade da onda
couhalf = c*dt/(2.0*dx)      # metade do número de Courant
for n in range(nt):          # loop no tempo
   for i in range(1,nx):     # loop no espaço
      u[new,i] = u[old,i] - couhalf*(u[old,i+1] - u[old,i-1])
   u[new,0] = 0.0
   u[new,nx] = 0.0
   u[new].tofile(fou)        # imprime uma linha com os novos dados
   (old,new) = (new,old)     # troca os índices
fou.close()
```

Listagem 13.2: `surf1d-ins.py` — Seleciona alguns intervalos de tempo da solução numérica para plotagem

```
#!/usr/bin/python
# -*- coding: iso-8859-1 -*-
# ------------------------------------------------------------
# surf1d-ins.py: imprime em <arq> <m>+1 saídas de
# onda1d-ins a cada <n> intervalos de tempo
#
# uso: ./surf1d-ins.py <m> <n>
# ------------------------------------------------------------
from __future__ import print_function
from __future__ import division
from sys import argv
dx = 0.01
dt = 0.0005
print('# dx = %9.4f' % dx)
print('# dy = %9.4f' % dt)
nx = int(10.0/dx)             # número de pontos em x
print('# nx = %9d' % nx)
m = int(argv[1])              # m saídas
n = int(argv[2])              # a cada n intervalos de  tempo
print('#  m = %9d' % m)
print('#  n = %9d' % n)
fin = open('onda1d-ins.dat',
         'rb')                # abre o arquivo com os dados
from numpy import fromfile
u = fromfile(fin,float,nx+1)  # lê a condição inicial
v = [u]                       # inicializa a lista da "transposta"
for it in range(m):           # para <m> instantes:
   for ir in range(n):        # lê <ir> vezes, só guarda a última
      u = fromfile(fin,float,nx+1)
   v.append(u)                # guarda a última
founam = 'surf1d-ins.dat'
print(founam)
fou = open(founam,'wt')       # abre o arquivo de saída
for i in range(nx+1):
   fou.write('%10.6f' % (i*dx))    # escreve o "x"
   fou.write('%10.6f' % v[0][i])   # escreve a cond inicial
   for k in range(1,m+1):
      fou.write('%10.6f' % v[k][i])# escreve o k-ésimo
   fou.write('\n')
fou.close()
```

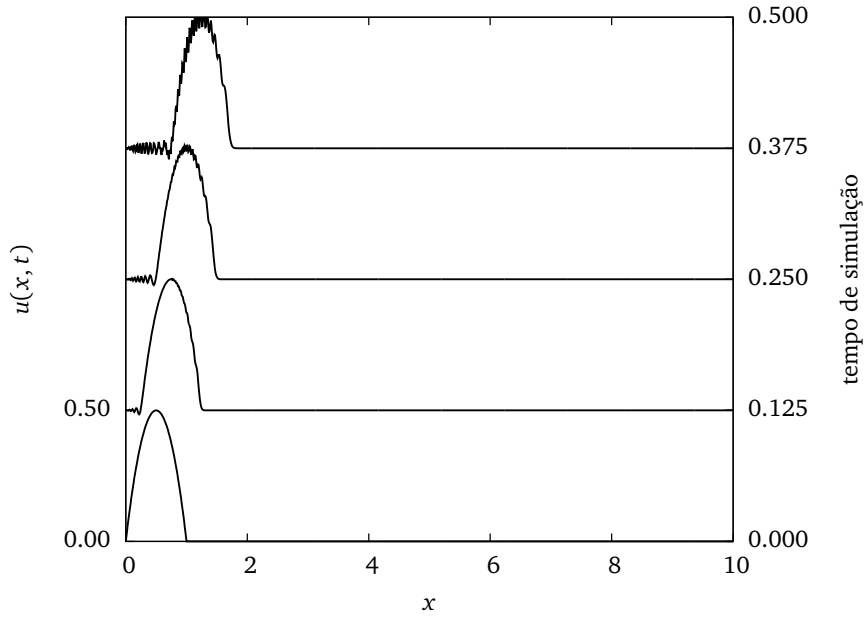

Figura 13.2: Solução numérica produzida por `onda1d-ins.py`, para $t = 250\Delta t$, $500\Delta t$ e $750\Delta t$.

Por que o esquema utilizado em (13.12) fracassa? Uma forma de obter a resposta é fazer uma *análise de estabilidade de von Neumann*.

A análise de estabilidade de von Neumann consiste primeiramente em observar que, em um computador real, (13.12) jamais será calculada com precisão infinita. O que o computador realmente calcula é um valor *truncado* $\tilde{u}_i^n$. Por enquanto, nós só vamos fazer essa distinção de notação, entre $\tilde{u}$ e u, aqui, onde ela importa. O *erro de truncamento* é

$$\epsilon_i^n \equiv \tilde{u}_i^n - u_i^n. \tag{13.13}$$

Note que (13.12) se aplica tanto para u quanto para $\tilde{u}$; subtraindo as equações resultantes para $\tilde{u}_i^{n+1}$ e u_i^{n+1}, obtém-se a *mesma* equação para a evolução de ϵ_i^n:

$$\epsilon_i^{n+1} = \epsilon_i^n - \frac{\mathrm{Co}}{2}(\epsilon_{i+1}^n - \epsilon_{i-1}^n), \tag{13.14}$$

onde

$$\mathrm{Co} \equiv \frac{c\Delta t}{\Delta x} \tag{13.15}$$

é o *número de Courant*. Isso só foi possível porque (13.12) é uma equação *linear* em u. Mesmo para equações não-lineares, entretanto, sempre será possível fazer pelo menos uma análise *local* de estabilidade.

O próximo passo da análise de estabilidade de von Neumman é escrever uma série de Fourier para ϵ_i^n, na forma

$$\begin{aligned} t_n &= n\Delta t, \\ x_i &= i\Delta x, \\ \epsilon_i^n &= \sum_{l=1}^{N/2} \xi_l \mathrm{e}^{at_n} \mathrm{e}^{\mathrm{i}k_l x_i}, \end{aligned} \tag{13.16}$$

onde e é a base dos logaritmos naturais, $\mathrm{i} = \sqrt{-1}$, $N = L/\Delta x$ é o número de pontos da discretização em x, e L é o tamanho do domínio em x.

Argumentando novamente com a linearidade, desta vez de (13.14), ela vale para cada *modo* l de (13.16), donde

$$\xi_l \mathrm{e}^{a(t_n+\Delta t)}\mathrm{e}^{\mathrm{i}k_l i\Delta x} = \xi_l \mathrm{e}^{at_n}\mathrm{e}^{\mathrm{i}k_l i\Delta x} - \frac{\mathrm{Co}}{2}\left(\xi_l \mathrm{e}^{at_n}\mathrm{e}^{\mathrm{i}k_l(i+1)\Delta x} - \xi_l \mathrm{e}^{at_n}\mathrm{e}^{\mathrm{i}k_l(i-1)\Delta x}\right); \quad (13.17)$$

eliminando o fator comum $\xi_l \mathrm{e}^{at_n+\mathrm{i}k_l i\Delta x}$,

$$\begin{aligned}\mathrm{e}^{a\Delta t} &= 1 - \frac{\mathrm{Co}}{2}\left(\mathrm{e}^{+\mathrm{i}k_l\Delta x} - \mathrm{e}^{-\mathrm{i}k_l\Delta x}\right)\\ &= 1 - \mathrm{iCo}\,\mathrm{sen}\,k_l\Delta x. \end{aligned} \quad (13.18)$$

O lado direito é um número complexo, de maneira que o lado esquerdo também tem que ser! Como conciliá-los? Fazendo $a = \alpha + \mathrm{i}\beta$, e substituindo:

$$\begin{aligned}\mathrm{e}^{(\alpha-\mathrm{i}\beta)\Delta t} &= 1 - \mathrm{iCo}\,\mathrm{sen}\,k_l\Delta x;\\ \mathrm{e}^{\alpha\Delta t}\left[\cos(\beta\Delta t) - \mathrm{i}\,\mathrm{sen}(\beta\Delta t)\right] &= 1 - \mathrm{iCo}\,\mathrm{sen}\,k_l\Delta x;\ \Rightarrow\\ e^{\alpha\Delta t}\cos(\beta\Delta t) &= 1, \quad (13.19)\\ e^{\alpha\Delta t}\,\mathrm{sen}(\beta\Delta t) &= \mathrm{Co}\,\mathrm{sen}(k_l\Delta x). \quad (13.20)\end{aligned}$$

As duas últimas equações formam um sistema não-linear nas incógnitas α β. O sistema pode ser resolvido:

$$\mathrm{tg}(\beta\Delta t) = \mathrm{Co}\,\mathrm{sen}(k_l\Delta x) \Rightarrow \beta\Delta t = \mathrm{arctg}\left(\mathrm{Co}\,\mathrm{sen}(k_l\Delta x)\right).$$

Note que $\beta \neq 0$, donde $e^{\alpha\Delta t} > 1$ via (13.19), e o esquema de diferenças finitas é *incondicionalmente instável*.

O método de Lax Uma alternativa que produz um esquema estável é o método de Lax:

$$u_i^{n+1} = \frac{1}{2}\left[(u_{i+1}^n + u_{i-1}^n) - \mathrm{Co}(u_{i+1}^n - u_{i-1}^n)\right]. \quad (13.21)$$

Agora que nós já sabemos que esquemas numéricos podem ser instáveis, devemos fazer uma análise de estabilidade *antes* de tentar implementar (13.21) numericamente. Vamos a isto: utilizando novamente (13.16) e substituindo em (13.21), temos

$$\begin{aligned}\xi_l \mathrm{e}^{a(t_n+\Delta t)}\mathrm{e}^{\mathrm{i}k_l i\Delta x} &= \frac{1}{2}\left[\left(\xi_l \mathrm{e}^{at_n}\mathrm{e}^{\mathrm{i}k_l(i+1)\Delta x} + \xi_l \mathrm{e}^{at_n}\mathrm{e}^{\mathrm{i}k_l(i-1)\Delta x}\right)\right.\\ &\quad \left. - \mathrm{Co}\left(\xi_l \mathrm{e}^{at_n}\mathrm{e}^{\mathrm{i}k_l(i+1)\Delta x} - \xi_l \mathrm{e}^{at_n}\mathrm{e}^{\mathrm{i}k_l(i-1)\Delta x}\right)\right];\\ e^{a\Delta t} &= \frac{1}{2}\left[\left(\mathrm{e}^{+\mathrm{i}k_l\Delta x} + \mathrm{e}^{-\mathrm{i}k_l\Delta x}\right) - \mathrm{Co}\left(\mathrm{e}^{+\mathrm{i}k_l\Delta x} - \mathrm{e}^{-\mathrm{i}k_l\Delta x}\right)\right];\\ e^{a\Delta t} &= \cos(k_l\Delta x) - \mathrm{iCo}\,\mathrm{sen}(k_l\Delta x) \quad (13.22)\end{aligned}$$

Nós podemos, é claro, fazer $a = \alpha - \mathrm{i}\beta$, mas há um caminho mais rápido: o truque é perceber que se o fator de amplificação $\mathrm{e}^{a\Delta t}$ for um número complexo com módulo maior que 1, o esquema será instável. Desejamos, portanto, que $|\mathrm{e}^{a\Delta t} \leq 1|$, o que só é possível se

$$\mathrm{Co} \leq 1, \quad (13.23)$$

que é o critério de estabilidade de Courant-Friedrichs-Lewy.

A "mágica" de (13.21) é que ela introduz um pouco de *difusão numérica*; de fato, podemos reescrevê-la na forma

$$\begin{aligned}\frac{u_i^{n+1}-u_i^n}{\Delta t} &= -c\frac{u_{i+1}^n-u_{i-1}^n}{2\Delta x}+\frac{u_{i+1}^n-2u_i^n+u_{i-1}^n}{2\Delta t}\\ &= -c\frac{u_{i+1}^n-u_{i-1}^n}{2\Delta x}+\left(\frac{\Delta x^2}{2\Delta t}\right)\frac{u_{i+1}^n-2u_i^n+u_{i-1}^n}{\Delta x^2}.\end{aligned} \tag{13.24}$$

Não custa repetir: (13.24) é *idêntica* a (13.21). Porém, comparando-a com (13.12) (nosso esquema instável inicialmente empregado), nós vemos que ela também é equivalente a esta última, *com o termo adicional* $\left(\Delta x^2/2\Delta t\right)\left(u_{i+1}^n-2u_i^n+u_{i-1}^n\right)/\Delta x^2$. O que este termo adicional significa? A resposta é *uma derivada numérica de ordem 2*. De fato, considere as expansões em série de Taylor

$$\begin{aligned}u_{i+1} &= u_i+\left.\frac{du}{dx}\right|_i\Delta x+\frac{1}{2}\left.\frac{d^2u}{dx^2}\right|_i+O(\Delta x^2),\\ u_{i-1} &= u_i-\left.\frac{du}{dx}\right|_i\Delta x+\frac{1}{2}\left.\frac{d^2u}{dx^2}\right|_i+O(\Delta x^2),\end{aligned}$$

e some:

$$\begin{aligned}u_{i+1}+u_{i-1} &= 2u_i+\left.\frac{d^2u}{dx^2}\right|_i\Delta x^2+O(\Delta x^2),\\ \left.\frac{d^2u}{dx^2}\right|_i &= \frac{u_{i+1}-2u_i+u_{i-1}}{\Delta x^2}+O(\Delta x^2).\end{aligned} \tag{13.25}$$

Portanto, a equação (13.24) — ou seja: o esquema de Lax (13.21) — pode ser interpretada *também* como uma solução aproximada da equação de advecção-difusão

$$\frac{\partial u}{\partial t}+c\frac{\partial u}{\partial c}=D\frac{\partial^2 u}{\partial x^2},$$

com

$$D=\left(\frac{\Delta x^2}{2\Delta t}\right).$$

Note que D tem dimensões de *difusividade*: $[\![D]\!]=\mathbb{L}^2\mathbb{T}^{-1}$. No entanto: não estamos então resolvendo a equação *errada*? De certa forma, sim: estamos introduzindo um pouco de difusão na equação para amortecer as oscilações que aparecerão em decorrência da amplificação dos erros de truncamento.

O quanto isto nos prejudica? Não muito, *desde que o efeito da difusão seja muito menor que o da advecção que estamos tentando simular*. Como a velocidade de advecção ("física"; "real") que estamos simulando é c, precisamos comparar isto com (por exemplo) a magnitude das velocidades introduzidas pela difusão numérica; devemos

portanto verificar se

$$\begin{aligned}
\frac{D\frac{\partial^2 u}{\partial x^2}}{c\frac{\partial u}{\partial x}} &\ll 1,\\
\frac{D\frac{u}{\Delta x^2}}{c\frac{u}{\Delta x}} &\ll 1,\\
\frac{D}{\Delta x} &\ll c,\\
\frac{\Delta x^2}{2\Delta t \Delta x} &\ll c,\\
\frac{c\Delta t}{\Delta x} = \mathrm{Co} &\gg \frac{1}{2}
\end{aligned}$$

Em outras palavras, nós descobrimos que o critério para que o esquema seja acurado do ponto de vista físico é conflitante com o critério de estabilidade: enquanto que estabilidade demandava $\mathrm{Co} < 1$, o critério de que a solução seja *também* fisicamente acurada demanda que $\mathrm{Co} \gg 1/2$. Na prática, isto significa que, para $c = 2$, *ou* o esquema é estável com muita difusão numérica, ou ele é instável. Isto praticamente elimina a possibilidade de qualquer uso sério de (13.21).

Mesmo assim, vamos programá-lo! O programa `onda1d-lax.py` está mostrado na listagem 13.3. Ele usa os mesmos valores $\Delta t = 0{,}0005$ e $\Delta x = 0{,}01$, ou seja, $\mathrm{Co} = 0{,}10$.

O programa gera um arquivo de saída binário, que por sua vez é lido pelo próximo programa na sequência, `surf1d-lax.py`, mostrado na listagem 13.4. O único trabalho deste programa é selecionar algumas "linhas" da saída de `onda1d-lax.py`; no caso, nós o rodamos com o comando

```
[ surf1d-lax.py 3 500 ] ,
```

o que significa selecionar 3 saídas (além da condição inicial), de 500 em 500 intervalos de tempo Δt. Com isto, nós conseguimos chegar até o instante 0,75 da simulação.

O resultado dos primeiros 1500 intervalos de tempo de simulação é mostrado na figura 13.3. Observe que agora não há oscilações espúrias: o esquema é estável no tempo. No entanto, a solução está agora "amortecida" pela difusão numérica!

Upwind Um esquema que é conhecido na literatura como indicado por representar melhor o termo advectivo em (13.1) é o esquema de diferenças regressivas; neste esquema, chamado de esquema *upwind* — literalmente, "corrente acima" — na literatura de língua inglesa, a discretização utilizada é

$$\begin{aligned}
\frac{u_i^{n+1} - u_i^n}{\Delta t} &= -c\frac{u_i^n - u_{i-1}^n}{\Delta x},\\
u_i^{n+1} &= u_i^n - \mathrm{Co}\left[u_i^n - u_{i-1}^n\right].
\end{aligned} \tag{13.26}$$

Claramente, estamos utilizando um esquema de $O(\Delta x)$ para a derivada espacial. Ele é um esquema menos acurado que os usados anteriormente, mas se ele ao mesmo tempo for condicionalmente estável e não introduzir difusão numérica, o resultado pode ser melhor para tratar a advecção.

Antes de "colocarmos as mãos na massa", sabemos que devemos analisar analiticamente a estabilidade do esquema. Vamos a isto:

Listagem 13.3: `onda1d-lax.py` — Solução de uma onda 1D com um método explícito laxtável

```
#!/usr/bin/python
# -*- coding: iso-8859-1 -*-
# ------------------------------------------------------------
# onda1d-lax resolve uma equação de onda com um método
# explícito
#
# uso: ./onda1d-ins.py
# ------------------------------------------------------------
from __future__ import print_function
from __future__ import division
fou = open('onda1d-lax.dat','wb')
dx = 0.01
dt = 0.0005
print('# dx = %9.4f' % dx)
print('# dy = %9.4f' % dt)
from numpy import zeros
nx = int(10.0/dx)             # número de pontos em x
nt = int(1.0/dt)              # número de pontos em t
print('# nx = %9d' % nx)
print('# nt = %9d' % nt)
u = zeros((2,nx+1),float)     # apenas 2 posições no tempo
                              # são necessárias!
def CI(x):                    # define a condição inicial
   if 0 <= x <= 1.0:
      return 2.0*x*(1.0-x)
   else:
      return 0.0
for i in range(nx+1):         # monta a condição inicial
   xi = i*dx
   u[0,i] = CI(xi)
u[0].tofile(fou)              # imprime a condição inicial
old = False
new = True
c = 2.0                      # celeridade da onda
cou = c*dt/(dx)              # número de Courant
print("Co = %10.6f" % cou)
for n in range(nt):          # loop no tempo
   for i in range(1,nx):     # loop no espaço
      u[new,i] = 0.5*(    (u[old,i+1] + u[old,i-1]) -
                      cou*(u[old,i+1] - u[old,i-1])    )
   u[new,0] = 0.0
   u[new,nx] = 0.0
   u[new].tofile(fou)        # imprime uma linha com os novos dados
   (old,new) = (new,old)     # troca os índices
fou.close()
```

Listagem 13.4: `surf1d-lax.py` — Seleciona alguns intervalos de tempo da solução numérica para plotagem

```
#!/usr/bin/python
# -*- coding: iso-8859-1 -*-
# ------------------------------------------------------------
# surf1d-lax.py: imprime em <arq> <m>+1 saídas de
# onda1d-lax a cada <n> intervalos de tempo
#
# uso: ./surf1d-lax.py <m> <n>
# ------------------------------------------------------------
from __future__ import print_function
from __future__ import division
from sys import argv
dx = 0.01
dt = 0.0005
print('# dx = %9.4f' % dx)
print('# dy = %9.4f' % dt)
nx = int(10.0/dx)             # número de pontos em x
print('# nx = %9d' % nx)
m = int(argv[1])              # m saídas
n = int(argv[2])              # a cada n intervalos de  tempo
print('#  m = %9d' % m)
print('#  n = %9d' % n)
fin = open('onda1d-lax.dat',
         'rb')                # abre o arquivo com os dados
from numpy import fromfile
u = fromfile(fin,float,nx+1)  # lê a condição inicial
v = [u]                       # inicializa a lista da "transposta"
for it in range(m):           # para <m> instantes:
   for ir in range(n):        # lê <ir> vezes, só guarda a última
      u = fromfile(fin,float,nx+1)
   v.append(u)                # guarda a última
founam = 'surf1d-lax.dat'
print(founam)
fou = open(founam,'wt')       # abre o arquivo de saída
for i in range(nx+1):
   fou.write('%10.6f' % (i*dx))    # escreve o "x"
   fou.write('%10.6f' % v[0][i])   # escreve a cond inicial
   for k in range(1,m+1):
      fou.write('%10.6f' % v[k][i])# escreve o k-ésimo
   fou.write('\n')
fou.close()
```

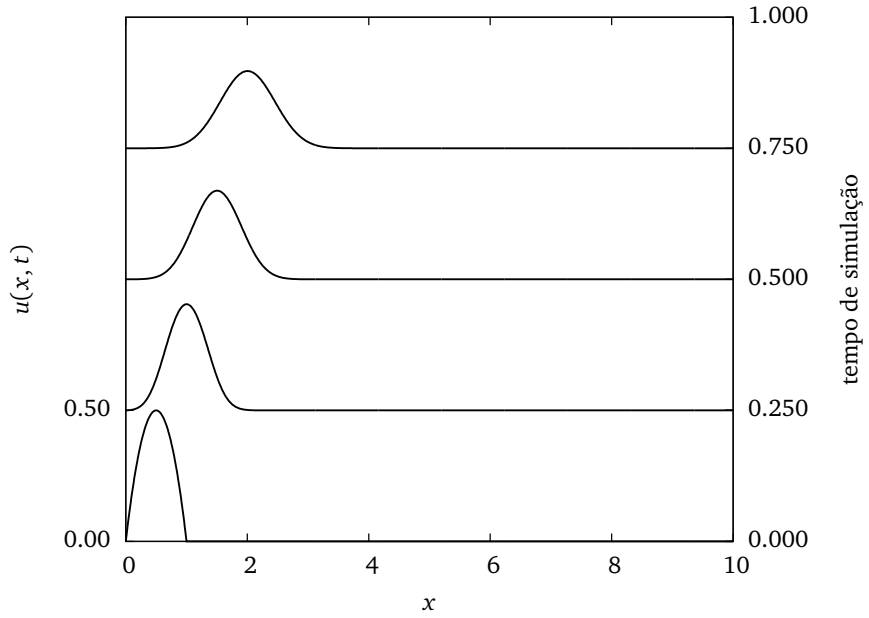

Figura 13.3: Solução numérica produzida por `onda1d-lax.py`, para $t = 500\Delta t$, $1000\Delta t$ e $1500\Delta t$.

$$\begin{aligned}
\xi_l e^{a(t_n+\Delta t)} e^{ik_l i\Delta x} &= \xi_l e^{at_n} e^{ik_l i\Delta x} - \mathrm{Co}\left[\xi_l e^{at_n} e^{ik_l i\Delta x} - \xi_l e^{at_n} e^{ik_l (i-1)\Delta x}\right] \\
e^{a\Delta t} e^{ik_l i\Delta x} &= e^{ik_l i\Delta x} - \mathrm{Co}\left[e^{ik_l i\Delta x} - e^{ik_l (i-1)\Delta x}\right] \\
e^{a\Delta t} &= 1 - \mathrm{Co}\left[1 - e^{-ik_l \Delta x}\right] \\
e^{a\Delta t} &= 1 - \mathrm{Co} + \mathrm{Co}\cos(k_l\Delta x) - i\mathrm{Co}\,\mathrm{sen}(k_l\Delta x). \qquad (13.27)
\end{aligned}$$

Desejamos que o módulo do fator de amplificação $e^{a\Delta t}$ seja menor que 1. O módulo (ao quadrado) é

$$|e^{a\Delta t}|^2 = \left(1 - \mathrm{Co} + \mathrm{Co}\cos(k_l\Delta x)\right)^2 + \left(\mathrm{Co}\,\mathrm{sen}(k_l\Delta x)\right)^2.$$

Para aliviar a notação, façamos

$$\begin{aligned}
C_k &\equiv \cos(k_l\Delta x), \\
S_k &\equiv \mathrm{sen}(k_l\Delta x).
\end{aligned}$$

Então,

$$\begin{aligned}
|e^{a\Delta t}|^2 &= (\mathrm{Co}S_k)^2 + (\mathrm{Co}C_k - \mathrm{Co} + 1)^2 \\
&= \mathrm{Co}^2 S_k^2 + (\mathrm{Co}^2 C_k^2 + \mathrm{Co}^2 + 1) + 2(-\mathrm{Co}^2 C_k + \mathrm{Co}C_k - \mathrm{Co}) \\
&= \mathrm{Co}^2(S_k^2 + C_k^2 + 1 - 2C_k) + 2\mathrm{Co}(C_k - 1) + 1 \\
&= 2\mathrm{Co}^2(1 - C_k) + 2\mathrm{Co}(C_k - 1) + 1.
\end{aligned}$$

A condição para que o esquema de diferenças finitas seja estável é, então,

$$\begin{aligned}
2\mathrm{Co}^2(1 - C_k) + 2\mathrm{Co}(C_k - 1) + 1 &\leq 1, \\
2\mathrm{Co}\left[\mathrm{Co}(1 - C_k) + (C_k - 1)\right] &\leq 0, \\
\left(1 - \cos(k_l\Delta x)\right)\left[\mathrm{Co} - 1\right] &\leq 0, \\
\mathrm{Co} &\leq 1 \blacksquare
\end{aligned}$$

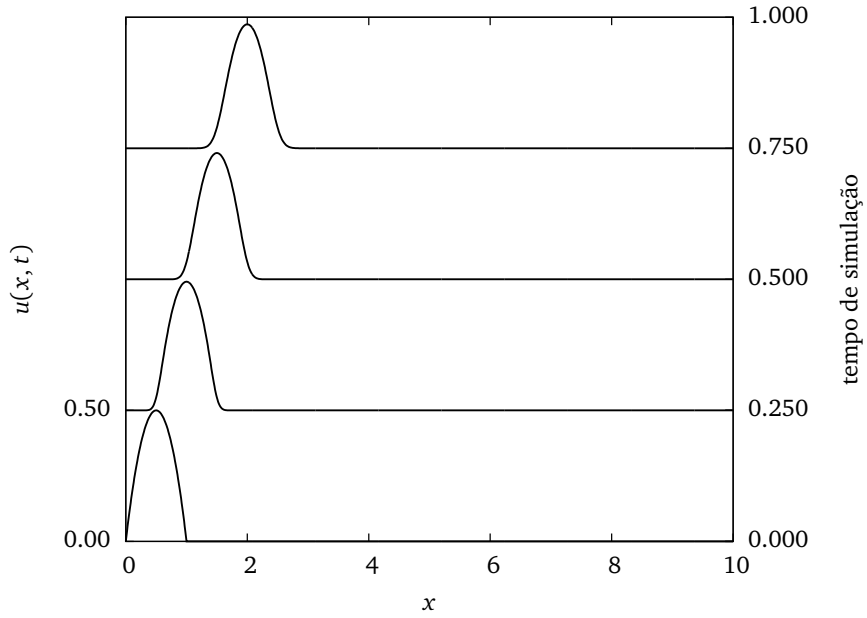

Figura 13.4: Solução numérica produzida pelo esquema *upwind*, para $t = 500\Delta t$, $1000\Delta t$ e $1500\Delta t$.

Reencontramos, portanto, a condição (13.23), *mas em um outro esquema de diferenças finitas*. A lição não deve ser mal interpretada: longe de supor que (13.23) vale sempre, é a análise de estabilidade que deve refeita para cada novo esquema de diferenças finitas!

O esquema *upwind*, portanto, é condicionalmente estável, e tudo indica que podemos agora implementá-lo computacionalmente, e ver no que ele vai dar. Nós utilizamos os mesmos valores de Δt e de Δx de antes. As mudanças necessárias nos códigos computacionais são óbvias, e são deixadas a cargo do(a) leitor(a).

A figura 13.4 mostra o resultado do esquema *upwind*. Note que ele é *muito melhor* (para *esta* equação diferencial) que o esquema de Lax. No entanto, a figura sugere que algum amortecimento também está ocorrendo, embora em grau muito menor.

Exercícios Propostos

13.1 Escreva o programa `onda1d-upw` e `surfa1d-upw`, que implementam o esquema *upwind*. Reproduza a figura 13.4.

13.2 Calcule a difusividade numérica introduzida pelo esquema *upwind*.

13.2 – Difusão pura

Considere agora a equação da difusão,

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}, \tag{13.28}$$

com condições iniciais e de contorno

$$u(x,0) = f(x) \tag{13.29}$$

$$u(0,t) = u(L,t) = 0. \tag{13.30}$$

Esta solução será vista no capítulo 17:

$$u(x,t) = \sum_{n=1}^{\infty} A_n \mathrm{e}^{-\frac{n^2\pi^2\alpha^2}{L^2}t} \operatorname{sen} \frac{n\pi x}{L}, \tag{13.31}$$

$$A_n = \frac{2}{L}\int_0^L f(x) \operatorname{sen} \frac{n\pi x}{L}\,\mathrm{d}x. \tag{13.32}$$

Em particular, se

$$\begin{aligned} D &= 2, \\ L &= 1, \\ f(x) &= 2x(1-x), \\ A_n &= 2\int_0^1 2x(1-x)\operatorname{sen}(n\pi x)\,\mathrm{d}x = \frac{8}{\pi^3 n^3}\left[1-(-1)^n\right]. \end{aligned}$$

Todos os A_n's pares se anulam. Fique então apenas com os ímpares:

$$\begin{aligned} A_{2n+1} &= \frac{16}{\pi^3(2n+1)^3}, \\ u(x,t) &= \sum_{n=0}^{\infty} \frac{16}{\pi^3(2n+1)^3} \mathrm{e}^{-(2(2n+1)^2\pi^2)t} \operatorname{sen}\left((2n+1)\pi x\right) \end{aligned} \tag{13.33}$$

O programa `difusao1d-ana.py`, mostrado na listagem 13.5, implementa a solução analítica para $\Delta t = 0{,}0005$ e $\Delta x = 0{,}001$.

Da mesma maneira que os programas `surf1d*.py`, o programa `divisao1d-ana.py`, mostrado na listagem 13.6, seleciona alguns instantes de tempo da solução analítica para visualização:

[`divisao1d-ana.py 3 100`] .

A figura 13.5 mostra o resultado da solução numérica para $t = 0$, $t = 0{,}05$, $t = 0{,}10$ e $t = 0{,}15$. Este é praticamente o "fim" do processo difusivo, com a solução analítica tendendo rapidamente para zero.

Esquema explícito Talvez o esquema explícito mais óbvio para discretizar (13.28) seja

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = D\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}. \tag{13.34}$$

A derivada parcial em relação ao tempo é de $O(\Delta t)$, enquanto que a derivada segunda parcial em relação ao espaço é, como vimos em (13.25), de $O(\Delta x^2)$. Mas não nos preocupemos muito, ainda, com a acurácia do esquema numérico. Nossa primeira preocupação, como você já sabe, é outra: o esquema (13.34) é *estável*?

Explicitamos u_i^{n+1} em (13.34):

$$u_i^{n+1} = u_i^n + \mathrm{Fo}\left[u_{i+1}^n - 2u_i^n + u_{i-1}^n\right], \tag{13.35}$$

onde

$$\mathrm{Fo} = \frac{D\Delta t}{\Delta x^2} \tag{13.36}$$

Listagem 13.5: `difusao1d-ana.py` — Solução analítica da equação da difusão

```
#!/usr/bin/python
# -*- coding: iso-8859-1 -*-
# ------------------------------------------------------------
# difusao1d-ana: solução analítica de
#
# du/dt = D du^2/dx^2
#
# u(x,0) = 2x(1-x)
# u(0,t) = 0
# u(1,t) = 0
#
# uso: ./difusao1d-ana.py
# ------------------------------------------------------------
from __future__ import print_function
from __future__ import division
fou = open('difusao1d-ana.dat','wb')
dx = 0.001
dt = 0.0005
print('# dx = %9.4f' % dx)
print('# dy = %9.4f' % dt)
nx = int(1.0/dx)              # número de pontos em x
nt = int(1.0/dt)              # número de pontos em t
print('# nx = %9d' % nx)
print('# nt = %9d' % nt)
from math import pi, sin, exp
epsilon = 1.0e-6              # precisão da solução analítica
dpiq = 2*pi*pi                # 2pi^2
dzpic = 16/(pi*pi*pi)         # 16/pi^3
def ana(x,t):
   s = 0.0
   ds = epsilon
   n = 0
   while abs(ds) >= epsilon:
      dnm1 = 2*n + 1          # (2n+1)
      dnm1q = dnm1*dnm1       # (2n+1)^2
      dnm1c = dnm1q*dnm1      # (2n+1)^3
      ds = exp(-dnm1q*dpiq*t)
      ds *= sin(dnm1*pi*x)
      ds /= dnm1c
      s += ds
      n += 1
   return s*dzpic
from numpy import zeros
u = zeros(nx+1,float)         # um array para conter a solução
for n in range(nt+1):         # loop no tempo
   t = n*dt
   print(t)
   for i in range(nx+1):      # loop no espaço
      xi = i*dx
      u[i] = ana(xi,t)
   u.tofile(fou)              # imprime uma linha com os novos dados
fou.close()
```

Listagem 13.6: `divisao1d-ana.py` — Seleciona alguns instantes de tempo da solução analítica para visualização

```
#!/usr/bin/python
# -*- coding: iso-8859-1 -*-
# ------------------------------------------------------------
# divisao1d-ana.py: imprime em <arq> <m>+1 saídas de
# difusao1d-ana a cada <n> intervalos de tempo
#
# uso: ./divisao1d-ana.py <m> <n>
# ------------------------------------------------------------
from __future__ import print_function
from __future__ import division
from sys import argv
dx = 0.001
dt = 0.0005
print('# dx = %9.4f' % dx)
print('# dt = %9.4f' % dt)
nx = int(1.0/dx)             # número de pontos em x
print('# nx = %9d' % nx)
m = int(argv[1])              # m saídas
n = int(argv[2])              # a cada n intervalos de  tempo
print('#  m = %9d' % m)
print('#  n = %9d' % n)
fin = open('difusao1d-ana.dat',
         'rb')                # abre o arquivo com os dados
from numpy import fromfile
u = fromfile(fin,float,nx+1)  # lê a condição inicial
v = [u]                       # inicializa a lista da "transposta"
for it in range(m):           # para <m> instantes:
   for ir in range(n):        # lê <ir> vezes, só guarda a última
      u = fromfile(fin,float,nx+1)
   v.append(u)                # guarda a última
founam = 'divisao1d-ana.dat'
print(founam)
fou = open(founam,'wt')       # abre o arquivo de saída
for i in range(nx+1):
   fou.write('%10.6f' % (i*dx))    # escreve o "x"
   fou.write('%10.6f' % v[0][i])   # escreve a cond inicial
   for k in range(1,m+1):
      fou.write('%10.6f' % v[k][i])# escreve o k-ésimo
   fou.write('\n')
fou.close()
```

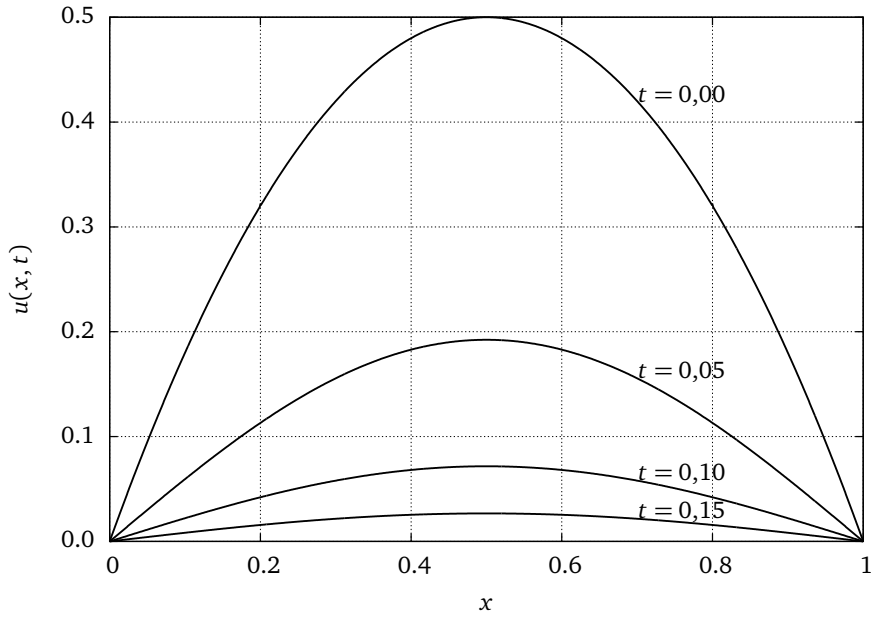

Figura 13.5: Solução analítica da equação de difusão para $t = 0$, $t = 0{,}05$, $t = 0{,}10$ e $t = 0{,}15$.

é o *número de Fourier de grade* (El-Kadi e Ling, 1993). A análise de estabiliade de von Neumann agora produz

$$\begin{aligned}
\xi_l e^{a(t_n+\Delta t)} e^{ik_l i\Delta x} &= \xi_l e^{at_n} e^{ik_l i\Delta x} + \\
&\quad \mathrm{Fo}\left[\xi_l e^{at_n} e^{ik_l(i+1)\Delta x} - 2\xi_l e^{at_n} e^{ik_l i\Delta x} + \xi_l e^{at_n} e^{ik_l(i-1)\Delta x}\right], \\
e^{a\Delta t} &= 1 + \mathrm{Fo}\left[e^{+ik_l\Delta x} - 2 + e^{-ik_l\Delta x}\right] \\
&= 1 + 2\mathrm{Fo}\left[\cos(k_l\Delta x) - 1\right] \\
&= 1 - 4\mathrm{Fo}\,\mathrm{sen}^2\left(\frac{k_l\Delta x}{2}\right)
\end{aligned} \tag{13.37}$$

A análise de estabilidade requer que $|e^{a\Delta t}| < 1$:

$$|e^{a\Delta t}|^2 = 1 - 8\mathrm{Fo}\,\mathrm{sen}^2\left(\frac{k_l\Delta x}{2}\right) + 16\mathrm{Fo}^2\,\mathrm{sen}^4\left(\frac{k_l\Delta x}{2}\right) < 1$$

ou

$$\begin{aligned}
-8\mathrm{Fo}\,\mathrm{sen}^2\left(\frac{k_l\Delta x}{2}\right) + 16\mathrm{Fo}^2\,\mathrm{sen}^4\left(\frac{k_l\Delta x}{2}\right) &< 0, \\
8\mathrm{Fo}\,\mathrm{sen}^2\left(\frac{k_l\Delta x}{2}\right)\left[-1 + 2\mathrm{Fo}\,\mathrm{sen}^2\left(\frac{k_l\Delta x}{2}\right)\right] &< 0, \\
\mathrm{Fo} &< \frac{1}{2}.
\end{aligned} \tag{13.38}$$

Podemos agora calcular o número de Fourier que utilizamos para plotar a solução analítica (verifique nas listagens 13.5 e 13.6):

$$\mathrm{Fo} = \frac{2 \times 0{,}0005}{(0{,}001)^2} = 1000.$$

Listagem 13.7: `difusao1d-exp.py` — Solução numérica da equação da difusão: método explícito.

```
#!/usr/bin/python
# -*- coding: iso-8859-1 -*-
# ------------------------------------------------------------
# difusao1d-exp resolve uma equação de difusão com um método
# explícito
#
# uso: ./difusao1d-exp.py
# ------------------------------------------------------------
from __future__ import print_function
from __future__ import division
fou = open('difusao1d-exp.dat','wb')
dx = 0.01
dt = 0.00001
print('# dx = %9.4f' % dx)
print('# dy = %9.4f' % dt)
from numpy import zeros
nx = int(round(1.0/dx,0))     # número de pontos em x
nt = int(round(1.0/dt,0))     # número de pontos em t
print('# nx = %9d' % nx)
print('# nt = %9d' % nt)
u = zeros((2,nx+1),float)     # apenas 2 posições no tempo
                              # são necessárias!
def CI(x):                    # define a condição inicial
   if 0 <= x <= 1.0:
      return 2.0*x*(1.0-x)
   else:
      return 0.0
for i in range(nx+1):         # monta a condição inicial
   xi = i*dx
   u[0,i] = CI(xi)
u[0].tofile(fou)              # imprime a condição inicial
old = False
new = True
D = 2.0                      # celeridade da onda
Fon = D*dt/((dx)**2)         # número de Fourier
print("Fo = %10.6f" % Fon)
for n in range(nt):          # loop no tempo
   print(n)
   for i in range(1,nx):     # loop no espaço
      u[new,i] = u[old,i] + Fon*(u[old,i+1] - 2*u[old,i] + u[old,i-1])
   u[new,0] = 0.0            # condição de contorno, x = 0
   u[new,nx] = 0.0           # condição de contorno, x = 1
   u[new].tofile(fou)        # imprime uma linha com os novos dados
   (old,new) = (new,old)     # troca os índices
fou.close()
```

Utilizar os valores $\Delta x = 0{,}0005$ e $\Delta x = 0{,}001$ levaria a um esquema instável. Precisamos *diminuir* Δt e/ou *aumentar* Δx. Com $\Delta t = 0{,}00001$ e $\Delta x = 0{,}01$,

$$\mathrm{Fo} = \frac{2 \times 0{,}00001}{(0{,}01)^2} = 0{,}2 < 0{,}5 \qquad \text{(OK)}.$$

Repare que $\mathrm{Fo} < 1/2$ é um critério de estabilidade muito mais exigente do que $\mathrm{Co} < 1/2$ (para $D = 2$). Nós esperamos que nosso esquema explícito agora rode muito lentamente. Mas vamos implementá-lo. O programa que implementa o esquema é o `difusao1d-exp.py`, mostrado na listagem 13.7.

O programa `divisao1d-exp.py`, mostrado na listagem 13.8, seleciona alguns instantes de tempo da solução analítica para visualização:

[`divisao1d-exp 3 5000`] .

Listagem 13.8: `divisao1d-exp.py` — Seleciona alguns instantes de tempo da solução analítica para visualização

```
#!/usr/bin/python
# -*- coding: iso-8859-1 -*-
# ------------------------------------------------------------
# divisao1d-exp.py: imprime em <arq> <m>+1 saídas de
# difusao1d-exp a cada <n> intervalos de tempo
#
# uso: ./divisao1d-exp.py <m> <n>
# ------------------------------------------------------------
from __future__ import print_function
from __future__ import division
from sys import argv
dx = 0.01
dt = 0.00001
print('# dx = %9.4f' % dx)
print('# dt = %9.4f' % dt)
nx = int(round(1.0/dx,0))     # número de pontos em x
nt = int(round(1.0/dt,0))     # número de pontos em t
print('# nx = %9d' % nx)
m = int(argv[1])              # m saídas
n = int(argv[2])              # a cada n intervalos de  tempo
print('#  m = %9d' % m)
print('#  n = %9d' % n)
fin = open('difusao1d-exp.dat',
         'rb')                # abre o arquivo com os dados
from numpy import fromfile
u = fromfile(fin,float,nx+1)  # lê a condição inicial
v = [u]                       # inicializa a lista da "transposta"
for it in range(m):           # para <m> instantes:
   for ir in range(n):        # lê <ir> vezes, só guarda a última
      u = fromfile(fin,float,nx+1)
   v.append(u)                # guarda a última
founam = 'divisao1d-exp.dat'
fou = open(founam,'wt')       # abre o arquivo de saída
for i in range(nx+1):
   fou.write('%10.6f' % (i*dx))    # escreve o "x"
   fou.write('%10.6f' % v[0][i])   # escreve a cond inicial
   for k in range(1,m+1):
      fou.write('%10.6f' % v[k][i])# escreve o k-ésimo
   fou.write('\n')
fou.close()
```

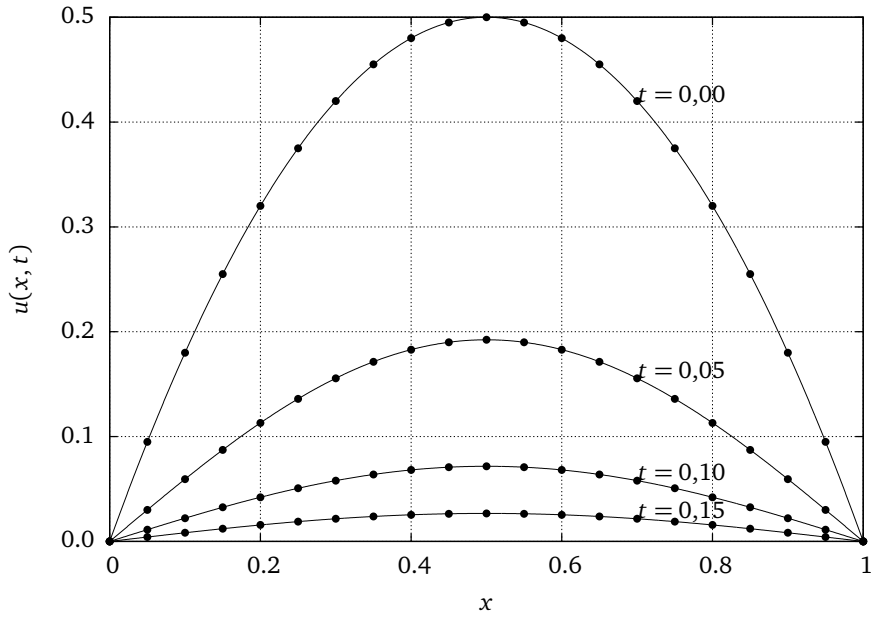

Figura 13.6: Solução numérica com o método explícito (13.35) (círculos) *versus* a solução analítica (linha cheia) da equação de difusão para $t = 0$, $t = 0{,}05$, $t = 0{,}10$ e $t = 0{,}15$. Apenas 1 a cada 5 pontos da grade numérica são mostrados, para facilitar a comparação com a solução analítica.

O resultado da solução numérica com o método explícito está mostrado na figura 13.6: ele é impressionantemente bom, embora seja computacionalmente muito caro. A escolha judiciosa de Δt e Δx para obeder ao critério (13.38) foi fundamental para a obtenção de um bom resultado "de primeira", sem a necessidade dolorosa de ficar tentando diversas combinações até que o esquema se estabilize e produza bons resultados.

Esquemas implícitos Embora o esquema explícito que nós utilizamos acima seja acurado, ele é *lento* — se você programou e rodou `difusao1d-exp.py`, deve ter notado *alguma* demora para o programa rodar. Embora nossos computadores estejam ficando a cada dia mais rápidos, isso não é desculpa para utilizar mal nossos recursos computacionais (é claro que, ao utilizarmos uma linguagem interpretada — Python — para programar, nós *já* estamos utilizando muito mal nossos recursos; no entanto, nosso argumento é didático: com uma linguagem mais simples, podemos aprender mais rápido e errar menos. Além disso, todos os ganhos *relativos* que obtivermos se manterão em qualquer outra linguagem)

Vamos portanto fazer uma mudança fundamental nos nossos esquemas de diferenças finitas: vamos calcular a derivada espacial no instante $n+1$:

$$\begin{aligned}
\frac{u_i^{n+1} - u_i^n}{\Delta t} &= D \frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{\Delta x^2}, \\
u_i^{n+1} - u_i^n &= \mathrm{Fo}(u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}), \\
-\mathrm{Fo}u_{i-1}^{n+1} + (1 + 2\mathrm{Fo})u_i^{n+1} - \mathrm{Fo}u_{i+1}^{n+1} &= u_i^n. \qquad (13.39)
\end{aligned}$$

Reveja a discretização (13.5)–(13.9): para $i = 1, \ldots, N_x - 1$, (13.39) acopla 3 valores das incógnitas u^{n+1} no instante $n+1$. Quando $i = 0$, e quando $i = N_x$, não

podemos utilizar (13.39), porque não existem os índices $i = -1$, e $i = N_x + 1$. Quando $i = 1$ e $i = N_x - 1$, (13.39) precisa ser modificada, para a introdução das *condições de contorno*: como $u_0^n = 0$ e $u_{N_x}^n = 0$ para qualquer n, teremos

$$(1 + 2\text{Fo})u_1^{n+1} - \text{Fo}u_2^{n+1} = u_1^n, \tag{13.40}$$

$$-\text{Fo}u_{N_x-2}^{n+1} + (1 + 2\text{Fo})u_{N_x-1}^{n+1} = u_{N_x-1}^n. \tag{13.41}$$

Em resumo, nossas incógnitas são $u_1^{n+1}, u_2^{n+1}, \ldots u_{N_x-1}^{n+1}$ ($N_x - 1$ incógnitas), e seu cálculo envolve a solução do sistema de equações

$$\begin{bmatrix} 1+2\text{Fo} & -\text{Fo} & 0 & \ldots & 0 & 0 \\ -\text{Fo} & 1+2\text{Fo} & \text{Fo} & 0 & \ldots & 0 \\ \vdots & & & & & \vdots \\ 0 & \ldots & 0 & -\text{Fo} & 1+2\text{Fo} & -\text{Fo} \\ 0 & 0 & \ldots & 0 & -\text{Fo} & 1+2\text{Fo} \end{bmatrix} \begin{bmatrix} u_1^{n+1} \\ u_2^{n+1} \\ \vdots \\ u_{N_x-2}^{n+1} \\ u_{N_x-1}^{n+1} \end{bmatrix} = \begin{bmatrix} u_1^n \\ u_2^n \\ \vdots \\ u_{N_x-2}^n \\ u_{N_x-1}^n \end{bmatrix} \tag{13.42}$$

A análise de estabilidade de von Neumann procede agora da maneira usual:

$$\begin{aligned}
\epsilon_i^{n+1} &= \epsilon_i^n + \text{Fo}(\epsilon_{i+1}^{n+1} - 2\epsilon_i^{n+1} + \epsilon_{i-1}^{n+1}) \\
\xi_l e^{a(t_n+\Delta t)} e^{\mathrm{i}k_l i\Delta x} &= \xi_l e^{at_n} e^{\mathrm{i}k_l i\Delta x} \\
&\quad + \text{Fo}\left(\xi_l e^{a(t_n+\Delta t)} e^{\mathrm{i}k_l(i+1)\Delta x} - 2\xi_l e^{a}(t_n+\Delta t) e^{\mathrm{i}k_l i\Delta x}\right. \\
&\quad \left. + \xi_l e^{a(t_n+\Delta t)} e^{\mathrm{i}k_l(i-1)\Delta x}\right), \\
e^{a\Delta t} &= 1 + e^{a\Delta t}\text{Fo}\left(e^{\mathrm{i}k_l\Delta x} - 2 + e^{-\mathrm{i}k_l\Delta x}\right), \\
e^{a\Delta t} &= 1 + e^{a\Delta t} 2\text{Fo}\left(\cos(k_l\Delta x) - 1\right), \\
e^{a\Delta t} &= 1 - e^{a\Delta t} 4\text{Fo}\sin^2\left(\frac{k_l\Delta x}{2}\right), \\
e^{a\Delta t}\left[1 + 4\text{Fo}\sin^2\left(\frac{k_l\Delta x}{2}\right)\right] &= 1, \\
|e^{a\Delta t}| &= \frac{1}{1 + 4\text{Fo}\sin^2\left(\frac{k_l\Delta x}{2}\right)} \leq 1 \qquad \textit{sempre}.
\end{aligned} \tag{13.43}$$

Portanto, o esquema implícito (13.39) é incondicionalmente estável, e temos confiança de que o programa correspondente não se instabilizará.

Existem várias coisas atraentes para um programador em (13.42). Em primeiro lugar, a matriz do sistema é uma matriz banda tridiagonal; sistemas lineares com este tipo de matriz são particularmente simples de resolver, e estão disponíveis na literatura (por exemplo: Press et al., 1992, seção 2.4, subrotina `tridag`). Em segundo lugar, a matriz do sistema é *constante*: ela só precisa ser montada uma vez no programa, o que torna a solução numérica potencialmente muito rápida.

Nós vamos começar, então, construindo um pequeno módulo, convenientemente denominado `alglin.py`, que exporta a função `tridag`, que resolve um sistema tridiagonal, mostrado na listagem 13.9.

Em seguida, o programa `difusao1d-imp.py` resolve o problema com o método implícito. Ele está mostrado na listagem 13.10. A principal novidade está nas linhas 42–46, e depois novamente na linha 56. Em Python e Numpy, é possível especificar sub-listas, e sub-`arrays`, com um dispositivo denominado *slicing*, que torna a

Listagem 13.9: `alglin.py` — Exporta uma rotina que resolve um sistema tridiagonal, baseado em Press et al. (1992)

```
1  # -*- coding: iso-8859-1 -*-
2  # ------------------------------------------------------------------------
3  # alglin.py implementa uma solução de um sistema linear com matriz tridiagonal
4  # ------------------------------------------------------------------------
5  from numpy import zeros
6  def tridag(A,y):              # A,y têm que ser arrays!
7     m = A.shape[0]             # garante que A representa uma
8     n = A.shape[1]             # matriz tridiagonal
9     assert(m == 3)             # garante que todos os tamanhos estão OK
10    o = y.shape[0]
11    assert (n == o)
12    x = zeros(n,float)         # vetor de trabalho: vai retornar a solução
13    gam = zeros(n,float)       # vetor de trabalho: vai ficar por aqui
14    if A[1,0] == 0.0 :
15       exit("Erro 1 em tridag")
16    bet = A[1,0]
17    x[0] = y[0]/bet
18    for j in range(1,n):
19       gam[j] = A[2,j-1]/bet
20       bet = A[1,j] - A[0,j]*gam[j]
21       if (bet == 0.0):
22          exit("Erro 2 em tridag")
23       x[j] = (y[j] - A[0,j]*x[j-1])/bet
24    for j in range(n-2,-1,-1):
25       x[j] -= gam[j+1]*x[j+1]
26    return x
```

programação mais compacta e clara. Por exemplo, na linha 43, todos os elementos `A[0,1]...A[0,nx-1]` recebem o valor `-Fon`.

Existe um programa `divisao1d-imp.py`, mas ele não precisa ser mostrado aqui, porque as modificações, por exemplo a partir de `divisao1d-exp.py`, são demasiadamente triviais para justificarem o gasto adicional de papel. Para $\Delta t = 0{,}001$, e $\Delta x = 0{,}01$, o resultado do método implícito está mostrado na figura 13.7

Nada mal, para uma economia de 100 vezes (em relação ao método explícito) em passos de tempo! (Note entretanto que a solução, em cada passo de tempo, é um pouco mais custosa, por envolver a solução de um sistema de equações acopladas, ainda que tridiagonal.)

Crank Nicholson A derivada espacial em (13.28) é aproximada, no esquema implícito (13.39), por um esquema de $O(\Delta x^2)$. A derivada temporal, por sua vez, é apenas de $O(\Delta t)$. Mas é possível consertar isso! A idéia é substituir (13.39) por

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{D}{2}\left[\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} + \frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{\Delta x^2}\right],$$

$$u_i^{n+1} = u_i^n + \frac{\mathrm{Fo}}{2}\left[u_{i+1}^n - 2u_i^n + u_{i-1}^n + u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}\right]. \qquad (13.44)$$

Com esta mudança simples, a derivada espacial agora é uma média das derivadas em n e $n+1$, ou seja: ela está *centrada* em $n+1/2$. Com isto, a derivada temporal do lado esquerdo torna-se, na prática, um esquema de ordem 2 centrado em $n+1/2$!

Como sempre, nosso trabalho agora é verificar a estabilidade do esquema numérico. Para isto, fazemos

$$\epsilon_i^{n+1} - \frac{\mathrm{Fo}}{2}\left[\epsilon_{i+1}^{n+1} - 2\epsilon_i^{n+1} + \epsilon_{i-1}^{n+1}\right] = \epsilon_i^n + \frac{\mathrm{Fo}}{2}\left[\epsilon_{i+1}^n - 2\epsilon_i^n + \epsilon_{i-1}^n\right],$$

Listagem 13.10: `difusao1d-imp.py` — Solução numérica da equação da difusão: método implícito.

```
#!/usr/bin/python
# -*- coding: iso-8859-1 -*-
# ------------------------------------------------------------
# difusao1d-imp resolve uma equação de difusão com um método
# implícito
#
# uso: ./difusao1d-imp.py
# ------------------------------------------------------------
from __future__ import print_function
from __future__ import division
fou = open('difusao1d-imp.dat','wb')
dx = 0.01                     # define a discretização em x
dt = 0.001                    # define a discretização em t
print('# dx = %9.4f' % dx)
print('# dy = %9.4f' % dt)
nx = int(round(1.0/dx,0))     # número de pontos em x
nt = int(round(1.0/dt,0))     # número de pontos em t
print('# nx = %9d' % nx)
print('# nt = %9d' % nt)
from numpy import zeros
u = zeros((2,nx+1),float)     # apenas 2 posições no tempo
                              # são necessárias!
def CI(x):                    # define a condição inicial
   if 0 <= x <= 1.0:
      return 2.0*x*(1.0-x)
   else:
      return 0.0
for i in range(nx+1):         # monta a condição inicial
   xi = i*dx
   u[0,i] = CI(xi)
u[0].tofile(fou)              # imprime a condição inicial
old = False
new = True
D = 2.0                      # difusividade
Fon = D*dt/((dx)**2)         # número de Fourier
print("Fo = %10.6f" % Fon)
A = zeros((3,nx-1),float)    # cria a matriz do sistema
# ------------------------------------------------------------------------------
# cuidado, "linha" e "coluna" abaixo não significam as reais linhas e colunas
# do sistema de equações, mas sim a forma de armazenar uma matriz tridiagonal
# ------------------------------------------------------------------------------
A[0,0] = 0.0                 # zera A[0,0]
A[0,1:nx-1] = -Fon           # preenche o fim da 1a linha
A[1,0:nx-1] = 1.0 + 2*Fon    # preenche a segunda linha
A[2,0:nx-2] = -Fon           # preenche o início da 2a linha
A[2,nx-2] = 0.0              # zera A[2,nx-2]
# ------------------------------------------------------------------------------
# importa uma tradução de tridag de Numerical Recipes para Python
# ------------------------------------------------------------------------------
from alglin import tridag
for n in range(nt):          # loop no tempo
   print(n)
# ------------------------------------------------------------------------------
# atenção: calcula apenas os pontos internos de u!
# ------------------------------------------------------------------------------
   u[new,1:nx] = tridag(A,u[old,1:nx])
   u[new,0] = 0.0            # condição de contorno, x = 0
   u[new,nx] = 0.0           # condição de contorno, x = 1
   u[new].tofile(fou)        # imprime uma linha com os novos dados
   (old,new) = (new,old)     # troca os índices
fou.close()                  # fecha o arquivo de saída, e fim.
```

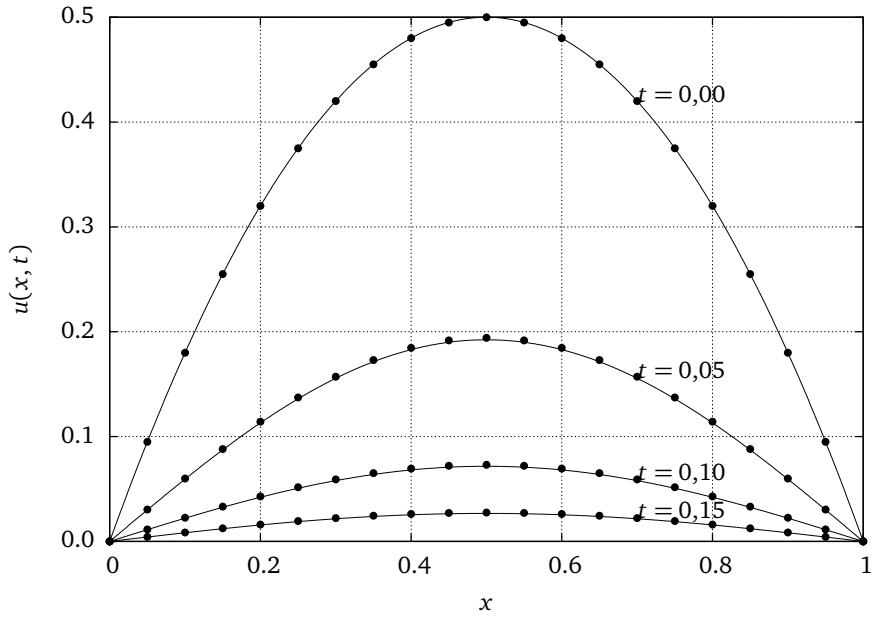

Figura 13.7: Solução numérica com o método implícito (13.39) (círculos) *versus* a solução analítica (linha cheia) da equação de difusão para $t = 0$, $t = 0{,}05$, $t = 0{,}10$ e $t = 0{,}15$. Apenas 1 a cada 5 pontos da grade numérica são mostrados, para facilitar a comparação com a solução analítica.

e substituímos um modo de Fourier:

$$\xi_l e^{a(t_n+\Delta t)}\left[e^{\mathrm{i}k_l i\Delta x} - \frac{\mathrm{Fo}}{2}\left(e^{\mathrm{i}k_l(i+1)\Delta x} - 2e^{\mathrm{i}k_l i\Delta x} + e^{\mathrm{i}k_l(i-1)\Delta x}\right)\right] = \xi_l e^{at_n}\left[e^{\mathrm{i}k_l i\Delta x} + \frac{\mathrm{Fo}}{2}\left(e^{\mathrm{i}k_l(i+1)\Delta x} - 2e^{\mathrm{i}k_l i\Delta x} + e^{\mathrm{i}k_l(i-1)\Delta x}\right)\right]$$

$$\begin{aligned}
e^{a\Delta t}\left[1 - \frac{\mathrm{Fo}}{2}\left(e^{\mathrm{i}k_l\Delta x} - 2 + e^{-\mathrm{i}k_l\Delta x}\right)\right] &= \left[1 + \frac{\mathrm{Fo}}{2}\left(e^{\mathrm{i}k_l\Delta x} - 2 + e^{-\mathrm{i}k_l\Delta x}\right)\right] \\
e^{a\Delta t}\left[1 - \mathrm{Fo}\left(\cos(k_l\Delta x) - 1\right)\right] &= \left[1 + \mathrm{Fo}\left(\cos(k_l\Delta x) - 1\right)\right] \\
e^{a\Delta t}\left[1 + 2\mathrm{Fo}\sin^2\left(\frac{k_l\Delta x}{2}\right)\right] &= \left[1 - 2\mathrm{Fo}\sin^2\left(\frac{k_l\Delta x}{2}\right)\right] \\
e^{a\Delta t} &= \frac{1 - 2\mathrm{Fo}\sin^2\left(\frac{k_l\Delta x}{2}\right)}{1 + 2\mathrm{Fo}\sin^2\left(\frac{k_l\Delta x}{2}\right)}.
\end{aligned}$$

É fácil notar que $|e^{a\Delta t}| < 1$, e o esquema numérico de Crank-Nicholson é incondicionalmente estável. O esquema numérico de Crank-Nicholson é similar a (13.39):

$$-\frac{\mathrm{Fo}}{2}u_{i-1}^{n+1} + (1+\mathrm{Fo})u_i^{n+1} - \frac{\mathrm{Fo}}{2}u_{i+1}^{n+1} = \frac{\mathrm{Fo}}{2}u_{i-1}^{n} + (1-\mathrm{Fo})u_i^{n} + \frac{\mathrm{Fo}}{2}u_{i+1}^{n} \tag{13.45}$$

Para as condições de contorno de (13.30), as linhas correspondentes a $i = 1$ e $i = N_x - 1$ são

$$(1+\mathrm{Fo})u_1^{n+1} - \frac{\mathrm{Fo}}{2}u_2^{n+1} = (1-2\mathrm{Fo})u_1^{n} + \frac{\mathrm{Fo}}{2}u_2^{n}, \tag{13.46}$$

$$-\frac{Fou}{2}u_{N_x-2}^{n+1} + (1+\mathrm{Fo})u_{N_x-1}^{n+1} = \frac{\mathrm{Fo}}{2}u_{N_x-2}^{n} + (1-\mathrm{Fo})u_{N_x-1}^{n} \tag{13.47}$$

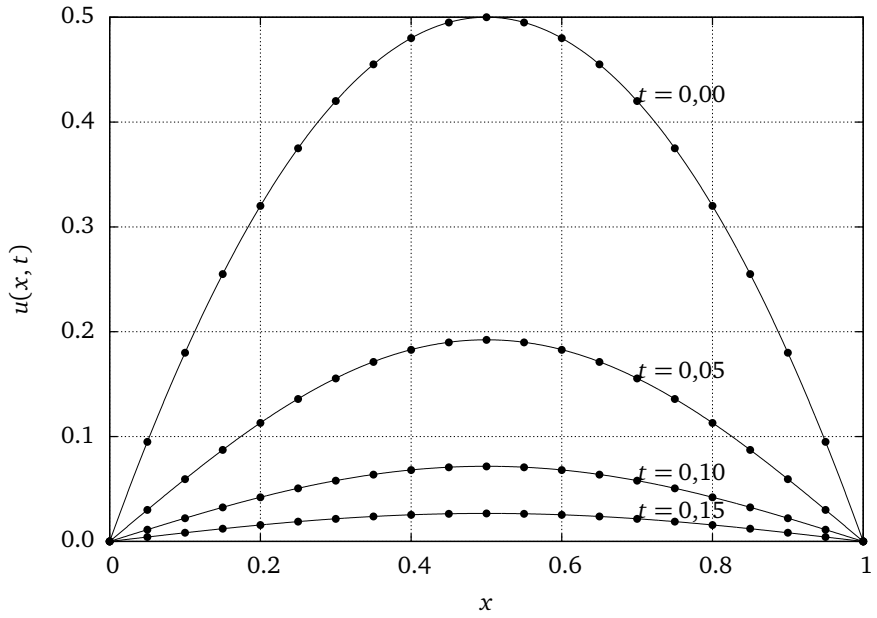

Figura 13.8: Solução numérica com o método de Crank-Nicholson ((13.45)) (círculos) *versus* a solução analítica (linha cheia) da equação de difusão para $t = 0$, $t = 0{,}05$, $t = 0{,}10$ e $t = 0{,}15$. Apenas 1 a cada 5 pontos da grade numérica são mostrados, para facilitar a comparação com a solução analítica.

As mudanças no código de `difusao-imp.py` são relativamente fáceis de se identificar. O código do programa que implementa o esquema numérico de Crank-Nicholson, `difusao1d-ckn.py`, é mostrado na listagem 13.11.

A grande novidade computacional de `difusao1d-ckn.py` é a linha 56: com os `arrays` proporcionados por Numpy, é possível escrever (13.45) *vetorialmente*: note que não há necessidade de fazer um *loop* em x para calcular cada elemento `b[i]` individualmente. O mesmo tipo de facilidade está disponível em FORTRAN90, FORTRAN95, etc.. Com isso, a implementação computacional dos cálculos gerada por Numpy (ou pelo compilador FORTRAN) também é potencialmente mais eficiente.

O método de Crank-Nicholson possui acurácia $O(\Delta t)^2$, portanto ele deve ser capaz de dar passos ainda mais largos no tempo que o método implícito (13.39); no programa `difusao1d-ckn.py`, nós especificamos um passo de tempo 5 vezes maior do que em `difusao1d-imp.py`.

O resultado é uma solução cerca de 5 vezes mais rápida (embora, novamente, haja mais contas agora para calcular o vetor de carga `b`), e é mostrado na figura 13.8.

Exemplo 13.1 Dada a equação da difusão unidimensional

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}, \qquad 0 \leq x \leq L,$$

Listagem 13.11: `difusao1d-ckn.py` — Solução numérica da equação da difusão: esquema de Crank-Nicholson.

```
#!/usr/bin/python
# -*- coding: iso-8859-1 -*-
# ------------------------------------------------------------
# difusao1d-ckn resolve uma equação de difusão com o método
# de Crank-Nicholson
#
# uso: ./difusao1d-ckn.py
# ------------------------------------------------------------
from __future__ import print_function
from __future__ import division
fou = open('difusao1d-ckn.dat','wb')
dx = 0.01                     # define a discretização em x
dt = 0.005                    # define a discretização em t
print('# dx = %9.4f' % dx)
print('# dt = %9.4f' % dt)
nx = int(round(1.0/dx,0))     # número de pontos em x
nt = int(round(1.0/dt,0))     # número de pontos em t
print('# nx = %9d' % nx)
print('# nt = %9d' % nt)
from numpy import zeros
u = zeros((2,nx+1),float)     # apenas 2 posições no tempo
                              # são necessárias!
def CI(x):                    # define a condição inicial
   if 0 <= x <= 1.0:
      return 2.0*x*(1.0-x)
   else:
      return 0.0
for i in range(nx+1):         # monta a condição inicial
   xi = i*dx
   u[0,i] = CI(xi)
u[0].tofile(fou)              # imprime a condição inicial
old = False
new = True
D = 2.0                      # difusividade
Fon = D*dt/((dx)**2)         # número de Fourier
print("Fo = %10.6f" % Fon)
A = zeros((3,nx-1),float)    # cria a matriz do sistema
# ------------------------------------------------------------------------------
# cuidado, "linha" e "coluna" abaixo não significam as reais linhas e colunas
# do sistema de equações, mas sim a forma de armazenar uma matriz tridiagonal
# ------------------------------------------------------------------------------
A[0,0] = 0.0                 # zera A[0,0]
A[0,1:nx-1] = -Fon/2.0       # preenche o fim da 1a linha
A[1,0:nx-1] = 1.0 + Fon      # preenche a segunda linha
A[2,0:nx-2] = -Fon/2.0       # preenche o início da 2a linha
A[2,nx-2] = 0.0              # zera A[2,nx-2]
# ------------------------------------------------------------------------------
# importa uma tradução de tridag de Numerical Recipes para Python
# ------------------------------------------------------------------------------
from alglin import tridag
for n in range(nt):          # loop no tempo
   print(n)
# ------------------------------------------------------------------------------
# recalcula o vetor de carga vetorialmente
# ------------------------------------------------------------------------------
   b = (Fon/2)*u[old,0:nx-1] + (1 - Fon)*u[old,1:nx] + (Fon/2)*u[old,2:nx+1]
# ------------------------------------------------------------------------------
# atenção: calcula apenas os pontos internos de u!
# ------------------------------------------------------------------------------
   u[new,1:nx] = tridag(A,b)
   u[new,0] = 0.0            # condição de contorno, x = 0
   u[new,nx] = 0.0           # condição de contorno, x = 1
   u[new].tofile(fou)        # imprime uma linha com os novos dados
   (old,new) = (new,old)     # troca os índices
fou.close()                  # fecha o arquivo de saída, e fim.
```

e o esquema de discretização

$$\begin{aligned}
\Delta x &= L/N_x, \\
x_i &= i\Delta x, \qquad i = 0, \ldots, N_x, \\
t_n &= t\Delta t, \\
u_i^n &= u(x_i, t_n), \\
\frac{1}{2\Delta t}\left[(u_{i+1}^{n+1} - u_{i+1}^n) + (u_{i-1}^{n+1} - u_{i-1}^n)\right] &= D\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2},
\end{aligned}$$

obtenha o critério de estabilidade por meio de uma análise de estabilidade de von Neumann.

SOLUÇÃO

Suponha que a equação diferencial se aplique ao erro:

$$\epsilon_i^n = \sum_l \xi_l e^{at_n} e^{ik_l i\Delta x} \Rightarrow$$

Então

$$\frac{1}{2\Delta t}\left[(\xi_l e^{a(t_n+\Delta t)} e^{ik_l(i+1)\Delta x} - \xi_l e^{at_n} e^{ik_l(i+1)\Delta x}) + (\xi_l e^{a(t_n+\Delta t)} e^{ik_l(i-1)\Delta x} - \xi_l e^{at_n} e^{ik_l(i-1)\Delta x}\right] =$$
$$D\frac{\xi_l e^{at_n} e^{ik_l(i+1)\Delta x} - 2\xi_l e^{at_n} e^{ik_l i\Delta x} + \xi_l e^{at_n} e^{ik_l(i-1)\Delta x}}{\Delta x^2};$$
$$\frac{1}{2\Delta t}\left[(\xi_l e^{a\Delta t} e^{ik_l(i+1)\Delta x} - \xi_l e^{ik_l(i+1)\Delta x}) + (\xi_l e^{a\Delta t} e^{ik_l(i-1)\Delta x} - \xi_l e^{ik_l(i-1)\Delta x}\right] =$$
$$D\frac{\xi_l e^{ik_l(i+1)\Delta x} - 2\xi_l e^{ik_l i\Delta x} + \xi_l e^{ik_l(i-1)\Delta x}}{\Delta x^2};$$
$$\frac{1}{2\Delta t}\left[(\, e^{a\Delta t} e^{ik_l(i+1)\Delta x} - \, e^{ik_l(i+1)\Delta x}) + (\, e^{a\Delta t} e^{ik_l(i-1)\Delta x} - \, e^{ik_l(i-1)\Delta x}\right] =$$
$$D\frac{e^{ik_l(i+1)\Delta x} - 2\, e^{ik_l i\Delta x} + \, e^{ik_l(i-1)\Delta x}}{\Delta x^2};$$
$$\left[(\, e^{a\Delta t} e^{ik_l(i+1)\Delta x} - \, e^{ik_l(i+1)\Delta x}) + (\, e^{a\Delta t} e^{ik_l(i-1)\Delta x} - \, e^{ik_l(i-1)\Delta x}\right] =$$
$$\frac{2D\Delta t}{\Delta x^2}\left[e^{ik_l(i+1)\Delta x} - 2\, e^{ik_l i\Delta x} + \, e^{ik_l(i-1)\Delta x}\right].$$

Segue-se que

$$e^{a\Delta t}\left[e^{ik_l(i+1)\Delta x} + e^{ik_l(i-1)\Delta x}\right] = e^{ik_l(i+1)\Delta x} + e^{ik_l(i-1)\Delta x} + 2\mathrm{Fo}\left[e^{ik_l(i+1)\Delta x} - 2\, e^{ik_l i\Delta x} + \, e^{ik_l(i-1)\Delta x}\right]$$
$$e^{a\Delta t}\left[e^{ik_l\Delta x} + e^{-ik_l\Delta x}\right] = e^{ik_l\Delta x} + e^{-ik_l\Delta x} + 2\mathrm{Fo}\left[e^{ik_l\Delta x} - 2 + e^{-ik_l\Delta x}\right]$$
$$\begin{aligned}
e^{a\Delta t} &= 1 + 2\mathrm{Fo}\frac{2\cos(k_l\Delta x) - 2}{2\cos(k_l\Delta x)} \\
&= 1 + 2\mathrm{Fo}\frac{\cos(k_l\Delta x) - 1}{\cos(k_l\Delta x)} \\
&= 1 - 4\mathrm{Fo}\frac{\mathrm{sen}^2(k_l\Delta x/2)}{\cos(k_l\Delta x)}.
\end{aligned}$$

A função

$$f(x) = \frac{\mathrm{sen}^2(x/2)}{\cos(x)}$$

possui singularidades em $\pi/2 + k\pi$, e muda de sinal em torno destas singularidades: não é possível garantir que $|e^{a\Delta t}| < 1$ uniformemente, e o esquema é incondicionalmente instável.

Exercícios Propostos

13.3 Considere a equação diferencial parcial

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}, \qquad 0 \leq x \leq L,$$

com condições iniciais e de contorno

$$u(x,0) = 0, \qquad u(0,t) = c, \qquad \frac{\partial u}{\partial x}(L,t) = 0,$$

onde c é uma constante. Dado o esquema de discretização implícito clássico,

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = D\frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{\Delta x^2}$$

para $N_x = 8$, obtenha o sistema de equações lineares

$$[A][u]^{n+1} = [b]$$

onde os $A_{i,j}$'s dependem do número de grade de Fourier, e os b_i's dependem dos u_i^n's. Em outras palavras, **escreva explicitamente a matriz quadrada** $[A]$ **e a matriz-coluna** $[b]$ **para** $N_x = 8$.

13.4 Dada a equação diferencial não-linear de Boussinesq,

$$\begin{aligned} \frac{\partial h}{\partial t} &= \frac{\partial}{\partial x}\left[h\frac{\partial h}{\partial x}\right], \\ h(x,0) &= H, \\ h(0,t) &= H_0, \\ \left.\frac{\partial h}{\partial x}\right|_{x=1} &= 0, \end{aligned}$$

obtenha uma discretização linearizada da mesma em diferenças finitas do tipo

$$h(x_i, t_n) = h(i\Delta x, n\Delta t) = h_i^n$$

da seguinte forma:

- discretize a derivada parcial em relação ao tempo com um esquema progressivo no tempo entre n e $n+1$;
- aproxime h dentro do colchete por h_i^n (este é o truque que lineariza o esquema de diferenças finitas) **e mantenha-o assim**;
- utilize esquemas de diferenças finitas implícitos centrados no espaço para as derivadas parciais em relação a x, **exceto no termo** h_i^n **do item anterior**.

NÃO MEXA COM AS CONDIÇÕES DE CONTORNO.

13.3 – Difusão em 2 Dimensões: ADI, e equações elíticas

Considere a equação da difusão em 2 dimensões,

$$\frac{\partial u}{\partial t} = D\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right). \tag{13.48}$$

Como sempre, nós queremos ser muito concretos, e trabalhar com um problema que possua solução analítica. Considere então a condição inicial

$$u(x,y,0) = u_0 \exp\left(-\frac{(x^2+y^2)}{L^2}\right); \tag{13.49}$$

a solução analítica é

$$u(x,y,t) = \frac{u_0}{1+4tD/L^2} \exp\left(-\frac{(x^2+y^2)}{L^2+4Dt}\right). \tag{13.50}$$

Na verdade esta solução se "espalha" por todo o plano xy, mas nós podemos trabalhar com um problema finito em x e y, por exemplo, fazendo $-L \leq x \leq L$, $-L \leq y \leq L$, e impondo condições de contorno que se ajustem *exatamente* à solução analítica:

$$u(-L,y,t) = \frac{u_0}{1+4tD/L^2} \exp\left(-\frac{(L^2+y^2)}{L^2+4Dt}\right), \tag{13.51}$$

$$u(L,y,t) = \frac{u_0}{1+4tD/L^2} \exp\left(-\frac{(L^2+y^2)}{L^2+4Dt}\right), \tag{13.52}$$

$$u(x,-L,t) = \frac{u_0}{1+4tD/L^2} \exp\left(-\frac{(x^2+L^2)}{L^2+4Dt}\right), \tag{13.53}$$

$$u(x,L,t) = \frac{u_0}{1+4tD/L^2} \exp\left(-\frac{(x^2+L^2)}{L^2+4Dt}\right). \tag{13.54}$$

Agora, nós vamos fazer $D = 2$ (como antes) e $L = 1$, e resolver o problema numericamente. Nossa escolha recairá sobre um método simples, e de $O(\Delta t)^2$, denominado ADI (*alternating-direction implicit*). Este método nos proporcionará um exemplo de uma técnica denominada *operator splitting* ou *time splitting*, que nós vamos traduzir como "separação de operadores" Esta técnica consiste em marchar implicitamente em uma dimensão espacial de cada vez, mantendo a outra dimensão "explícita". Portanto, nós vamos utilizar *dois* esquemas diferentes de diferenças finitas (na prática), para resolver o problema! Ei-los

$$\frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} = D\left(\frac{u_{i+1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i-1,j}^{n+1}}{\Delta x^2} + \frac{u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n}}{\Delta y^2}\right) \tag{13.55}$$

$$\frac{u_{i,j}^{n+2} - u_{i,j}^{n+1}}{\Delta t} = D\left(\frac{u_{i+1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i-1,j}^{n+1}}{\Delta x^2} + \frac{u_{i,j+1}^{n+2} - 2u_{i,j}^{n+2} + u_{i,j-1}^{n+2}}{\Delta y^2}\right) \tag{13.56}$$

Examine cuidadosamente (13.55) e (13.56): na primeira, note que o esquema é implícito em x; na segunda, a situação se reverte, e o esquema é implícito em y. É claro

que nós vamos precisar de duas análises de estabilidade de von Neumann, uma para cada equação.

2011-09-24T17:07:04 Por enquanto, vou supor que os dois esquemas são incondicionalmente estáveis, e mandar ver.

Além disto, por simplicidade vamos fazer $\Delta x = \Delta y = \Delta$, de maneira que só haverá um número de Fourier de grade no problema,

$$\mathrm{Fo} = \frac{D\Delta t}{\Delta^2}, \tag{13.57}$$

e então teremos, para x:

$$\begin{aligned} u_{i,j}^{n+1} - u_{i,j}^{n} &= \mathrm{Fo}\left(u_{i-1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i+1,j}^{n+1} + u_{i,j-1}^{n} - 2u_{i,j}^{n} + u_{i,j+1}^{n}\right), \\ u_{i,j}^{n+1} - \mathrm{Fo}\left(u_{i-1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i+1,j}^{n+1}\right) &= u_{i,j}^{n} + \mathrm{Fo}\left(u_{i,j-1}^{n} - 2u_{i,j}^{n} + u_{i,j+1}^{n}\right), \\ -\mathrm{Fo}u_{i-1,j}^{n+1} + (1+2\mathrm{Fo})u_{i,j}^{n+1} - \mathrm{Fo}u_{i+1,j}^{n+1} &= \mathrm{Fo}u_{i,j-1}^{n} + (1-2\mathrm{Fo})u_{i,j}^{n} + \mathrm{Fo}u_{i,j+1}^{n} \end{aligned} \tag{13.58}$$

Na dimensão y,

$$\begin{aligned} u_{i,j}^{n+2} - u_{i,j}^{n+1} &= \mathrm{Fo}\left(u_{i-1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i+1,j}^{n+1} + u_{i,j-1}^{n+2} - 2u_{i,j}^{n+2} + u_{i,j+1}^{n+2}\right), \\ u_{i,j}^{n+2} - \mathrm{Fo}\left(u_{i,j-1}^{n+2} - 2u_{i,j}^{n+2} + u_{i,j+1}^{n+2}\right) &= u_{i,j}^{n+1} + \mathrm{Fo}\left(u_{i-1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i+1,j}^{n+1}\right), \\ -\mathrm{Fo}u_{i,j-1}^{n+2} + (1+2\mathrm{Fo})u_{i,j}^{n+2} - \mathrm{Fo}u_{i,j+1}^{n+2} &= \mathrm{Fo}u_{i-1,j}^{n+1} + (1-2\mathrm{Fo})u_{i,j}^{n+1} + \mathrm{Fo}u_{i+1,j}^{n+1} \end{aligned} \tag{13.59}$$

Se nós utilizarmos (novamente por simplicidade) o mesmo número de pontos $N+1$ em x e em y, teremos o seguinte mapeamento para a nossa grade:

$$N = \frac{2L}{\Delta}; \tag{13.60}$$

$$x_i = -L + i\Delta, \qquad i = 0, \ldots, N, \tag{13.61}$$

$$y_j = -L + j\Delta, \qquad j = 0, \ldots, N, \tag{13.62}$$

e portanto $-L \le x_i \le L$ e $-L \le y_j \le L$. Lembrando que os valores de $u_{0,j}$, $u_{N,j}$, $u_{i,0}$ e $u_{i,N}$ estão especificados, há $(N-1)^2$ incógnitas para serem calculadas. A beleza de (13.58) e (13.59) é que em vez de resolver a cada passo (digamos) $2\Delta t$ um sistema de $(N-1)^2$ incógnitas, nós agora podemos resolver a cada passo Δt $N-1$ sistemas de $(N-1)$ incógnitas, alternadamente para $u_{1,\ldots,N-1;j}$ e $u_{i;1,\ldots,N_1}$.

É claro que o céu é o limite: poderíamos, por exemplo, em vez de usar um esquema totalmente implícito, usar Crank-Nicholson em cada avanço Δt; isto nos daria imediatamente um esquema com acurácia de ordem Δt^2. No entanto, assim como está o método ADI já é suficientemente sofisticado para nosso primeiro encontro com este tipo de problema. Devemos, portanto, programá-lo. Vamos, inicialmente, programar a solução analítica, na listagem 13.12.

A solução analítica do problema para os instantes de tempo $t = 0$, $t = 0{,}1$, $t = 0{,}2$ e $t = 0{,}3$ está mostrada na figura 13.9

Listagem 13.12: `difusao2d-ana.py` — Solução analítica da equação da difusão bidimensional.

```
#!/usr/bin/python
# -*- coding: iso-8859-1 -*-
# ------------------------------------------------------------
# difusao2d-ana: solução analítica de
#
# du/dt = D (du^2/dx^2 + du^2/dy^2)
#
# u(x,0) = T0 exp(-(x^2 + y^2)/L^2)
#
# com T0 = 1, D = 2, L = 1, em um domínio (-L,-L) até (L,L)
#
# uso: ./difusao2d-ana.py
# ------------------------------------------------------------
from __future__ import print_function
from __future__ import division
fou = open('difusao2d-ana.dat','wb')
dd = 0.01
dt = 0.001
print('# dd = %9.4f' % dd)
print('# dt = %9.4f' % dt)
nn = int(2.0/dd)              # número de pontos em x e em y
nt = int(1.0/dt)              # número de pontos em t
print('# nn = %9d' % nn)
print('# nt = %9d' % nt)
from math import exp
def ana(x,y,t):
   return (1.0/(1.0 + 8*t))*exp(-(x**2 + y**2))
from numpy import zeros
u = zeros((nn+1,nn+1),float)  # um array para conter a solução
for n in range(nt+1):         # loop no tempo
   t = n*dt
   print(t)
   for i in range(nn+1):      # loop no espaço
      xi = -1 + i*dd
      for j in range(nn+1):
         yj = -1 + j*dd
         u[i,j] = ana(xi,yj,t)
   u.tofile(fou)              # imprime uma matriz com os dados
fou.close()
```

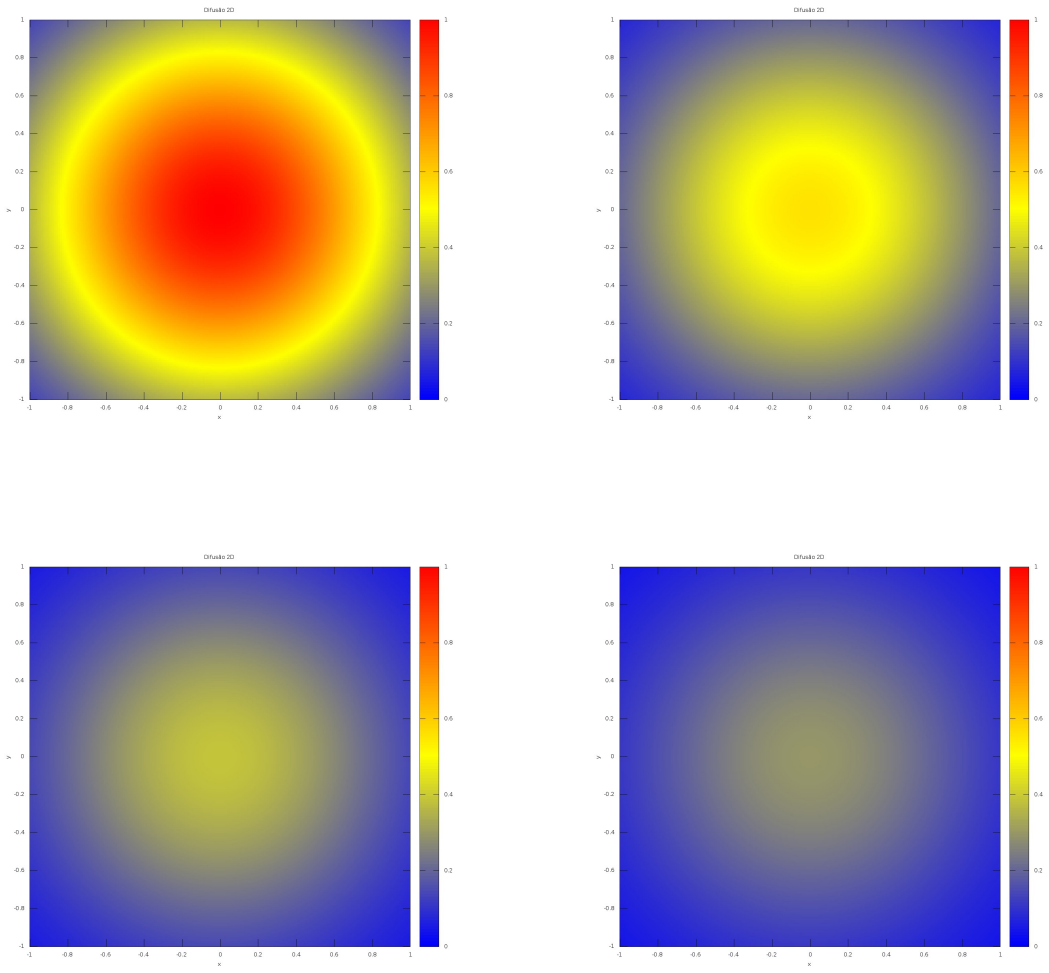

Figura 13.9: Solução analítica da equação da difusão bidimensional, para $t = 0$, $t = 0$, $t = 0{,}1$, $t = 0{,}2$ e $t = 0{,}3$

Em seguida, o esquema numérico ADI está implementado na listagem 13.13. O resultado é mostrado na figura 13.10.

Listagem 13.13: `difusao2d-adi.py` — Solução numérica da equação da difusão bi-dimensional, esquema ADI.

```
#!/usr/bin/python
# -*- coding: iso-8859-1 -*-
# ------------------------------------------------------------------------------
# difusao2d-adi resolve uma equação de difusão com um método implícito
#
# uso: ./difusao2d-adi.py
# ------------------------------------------------------------------------------
from __future__ import print_function
from __future__ import division
fou = open('difusao2d-adi.dat','wb')
dd = 0.02                               # define a discretização em x,y
dt = 0.001                              # define a discretização em t
print('# dd = %9.4f' % dd)
print('# dt = %9.4f' % dt)
nn = int(round(2.0/dd,0))               # número de pontos em x,y
nt = int(round(1.0/dt,0))               # número de pontos em t
print('# nn = %9d' % nn)
print('# nt = %9d' % nt)
from numpy import zeros
u = zeros((2,nn+1,nn+1),float)          # apenas 2 posições no tempo
                                        # são necessárias!
from math import exp
def CCy(y):
   return (1.0/aux)*exp( - (1.0 + y*y)/aux)
def CCx(x):
   return (1.0/aux)*exp( - (1.0 + x*x)/aux)
def CI(x, y):
   return exp(-(x*x + y*y))
for i in range(nn+1):                   # monta a condição inicial
   xi = -1.0 + i*dd                     # inteira, até as fronteiras
   for j in range(nn+1):                # inclusive
      yj = -1.0 + j*dd
      u[0,i,j] = CI(xi,yj)
u[0].tofile(fou)                        # imprime a condição inicial
D = 2.0                                 # difusividade
Fon = D*dt/((dd)**2)                    # número de Fourier
print("Fo = %10.6f" % Fon)
A = zeros((3,nn-1),float)               # cria a matriz do sistema
# ------------------------------------------------------------------------------
# monta a matriz do sistema
# ------------------------------------------------------------------------------
A[0,0]      = 0.0                       # zera A[0,0]
A[0,1:nn-1] = -Fon                      # preenche o fim da 1a linha
A[1,0:nn-1] = 1.0 + 2*Fon               # preenche a segunda linha
A[2,0:nn-2] = -Fon                      # preenche o início da 2a linha
A[2,nn-2]   = 0.0                       # zera A[2,nn-2]
# ------------------------------------------------------------------------------
# importa uma tradução de tridag de Numerical Recipes para Python
# ------------------------------------------------------------------------------
from alglin import tridag
old = False                             # o velho truque!
new = True
n = 0
b = zeros(nn-1,float)
while (n < nt+1):                       # loop no tempo
   n += 1
   print(n)
# ------------------------------------------------------------------------------
# varre na direção x
# ------------------------------------------------------------------------------
   t = n*dt
   aux = 1.0 + 8.0*t
   for j in range(nn+1):                # CC ao longo de x
      yj = -1.0 + j*dd
      u[new,0,j]  = CCy(yj)
      u[new,nn,j] = CCy(yj)
```

```
67     for j in range(1,nn):                # nn-1 varreduras em x (logo, loop em y)
68        yj = -1.0 + j*dd
69        b[1:nn-2] = Fon*u[old,2:nn-1,j-1] \
70                                 + (1.0 - 2*Fon)*u[old,2:nn-1,j] \
71                                 + Fon*u[old,2:nn-1,j+1]
72        b[0] = Fon*u[old,1,j-1] + (1.0 - 2*Fon)*u[old,1,j] \
73                                 + Fon*u[old,1,j+1] \
74                                 + Fon*u[new,0,j]
75        b[nn-2] = Fon*u[old,nn-1,j-1] + (1.0 - 2*Fon)*u[old,nn-1,j] \
76                                 + Fon*u[old,nn-1,j+1] \
77                                 + Fon*u[new,nn,j]
78        u[new,1:nn,j] = tridag(A,b)
79     u[new].tofile(fou)
80  # ------------------------------------------------------------------------
81  # varre na direção y
82  # ------------------------------------------------------------------------
83     (new,old) = (old,new)
84     n += 1
85     print(n)
86     t = n*dt
87     aux = 1.0 + 8.0*t
88     for i in range(nn+1):                # CC ao longo de y
89        xi = -1.0 + i*dd
90        u[new,i,0]  = CCx(xi)
91        u[new,i,nn] = CCx(xi)
92     for i in range(1,nn):                # nn-1 varreduras em y (logo, loop em x)
93        xi = -1.0 + i*dd
94        b[1:nn-2] = Fon*u[old,i-1,2:nn-1] \
95                                   + (1.0 - 2*Fon)*u[old,i,2:nn-1] \
96                                   + Fon*u[old,i+1,2:nn-1]
97        b[0] = Fon*u[old,i-1,1] + (1.0 - 2*Fon)*u[old,i,1] \
98                                 + Fon*u[old,i+1,1] \
99                                 + Fon*u[new,i,0]
100       b[nn-2] = Fon*u[old,i-1,nn-1] + (1.0 - 2*Fon)*u[old,i,nn-1] \
101                                + Fon*u[old,i+1,nn-1] \
102                                + Fon*u[new,i,nn]
103       u[new,i,1:nn] = tridag(A,b)
104    u[new].tofile(fou)
105    (new,old) = (old,new)
106 fou.close()                  # fecha o arquivo de saída, e fim.
```

Exemplo 13.2 Utilizando a análise de estabilidade de von Newmann, mostre que o esquema numérico correspondente à primeira das equações acima é incondicionalmente estável. Suponha $\Delta x = \Delta y = \Delta s$.

SOLUÇÃO

Inicialmente, rearranjamos o esquema de discretização, multiplicando por Δt e dividindo por Δx^2:

$$u_{i,j}^{n+1} - u_{i,j}^{n} = \mathrm{Fo}\left[u_{i+1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i-1,j}^{n+1} + u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n}\right],$$

onde

$$\mathrm{Fo} = \frac{D\Delta t}{\Delta s^2}.$$

Faça agora

$$\begin{aligned} t_n &= n\Delta t, \\ x_i &= i\Delta s, \\ y_j &= j\Delta s, \\ \epsilon_i^n &= \sum_{l,m} \xi_{l,m} \mathrm{e}^{at_n} \mathrm{e}^{\mathrm{i}k_l x_i} \mathrm{e}^{\mathrm{i}k_m y_j}, \end{aligned}$$

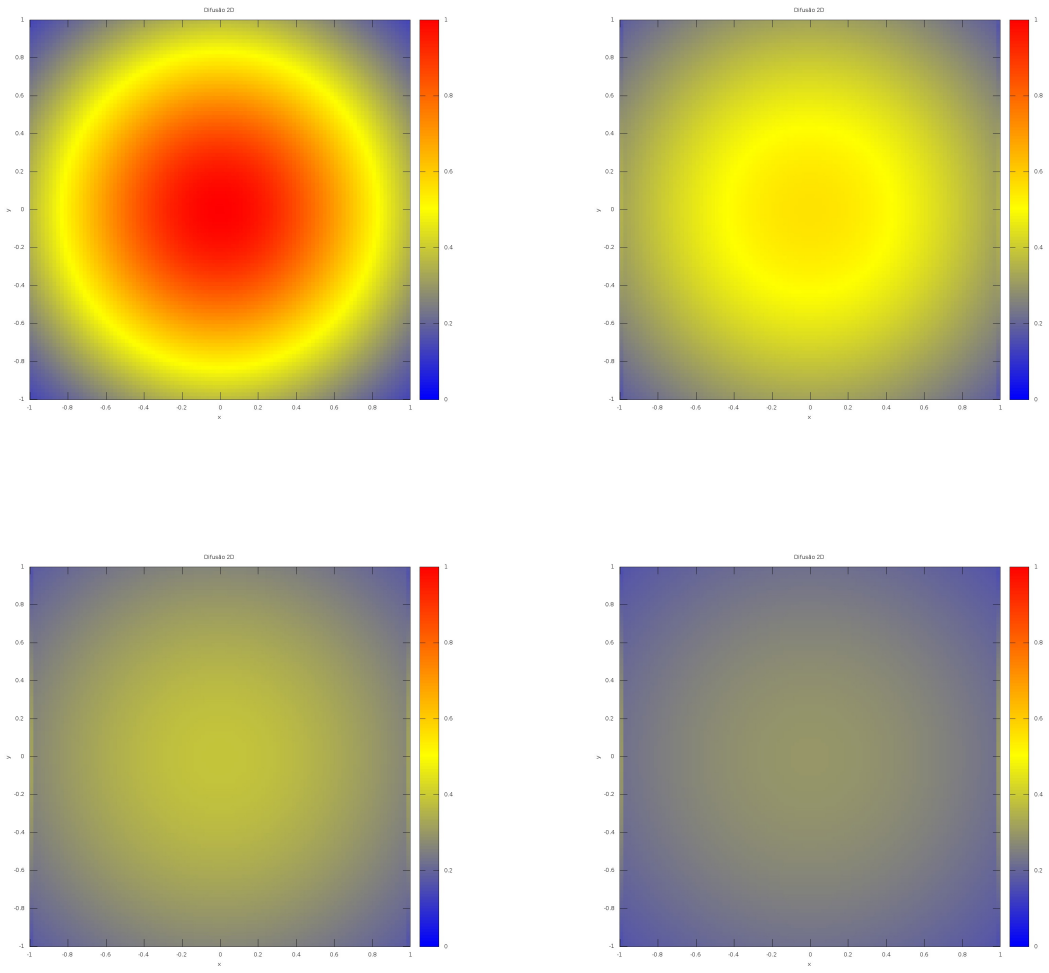

Figura 13.10: Solução numérica da equação da difusão bidimensional com o esquema ADI, para $t = 0$, $t = 0$, $t = 0{,}1$, $t = 0{,}2$ e $t = 0{,}3$

e substitua o modo (l,m) no esquema de discretização:

$$\begin{aligned}
&\xi_{l,m}e^{a(t_n+\Delta t)}e^{ik_l i\Delta s}e^{ik_m j\Delta s} - \xi_{l,m}e^{at_n}e^{ik_l i\Delta s}e^{ik_m j\Delta s} = \\
\mathrm{Fo}\Big[&\xi_{l,m}e^{a(t_n+\Delta t)}e^{ik_l(i+1)\Delta s}e^{ik_m j\Delta s} - 2\xi_{l,m}e^{a(t_n+\Delta t)}e^{ik_l i\Delta s}e^{ik_m j\Delta s} + \xi_{l,m}e^{a(t_n+\Delta t)}e^{ik_l(i-1)\Delta s}e^{ik_m j\Delta s} \\
&\xi_{l,m}e^{at_n}e^{ik_l i\Delta s}e^{ik_m(j+1)\Delta s} - 2\xi_{l,m}e^{at_n}e^{ik_l i\Delta s}e^{ik_m j\Delta s} + \xi_{l,m}e^{at_n}e^{ik_l i\Delta s}e^{ik_m(j-1)\Delta s}\Big].
\end{aligned}$$

Nós imediatamente reconhecemos o fator comum

$$\xi_{l,m}e^{at_n}e^{ik_l i\Delta s}e^{ik_m j\Delta s},$$

e simplificamos:

$$\begin{aligned}
e^{a\Delta t} - 1 &= \mathrm{Fo}\left[e^{a\Delta t}e^{+ik_l\Delta s} - 2e^{a\Delta t} + e^{a\Delta t}e^{-ik_l\Delta s} + e^{+ik_m\Delta s} - 2 + e^{-ik_m\Delta s}\right], \\
e^{a\Delta t}\left[1 - \mathrm{Fo}(e^{+ik_l\Delta s} - 2 + e^{-ik_l\Delta s})\right] &= 1 + \mathrm{Fo}\left[e^{+ik_m\Delta s} - 2 + e^{-ik_m\Delta s}\right] \\
e^{a\Delta t}\left[1 - 2\mathrm{Fo}(\cos(k_l\Delta s) - 1)\right] &= 1 + 2\mathrm{Fo}(\cos(k_m\Delta s) - 1) \\
|e^{a\Delta t}| &= \left|\frac{1 + 2\mathrm{Fo}(\cos(k_m\Delta s) - 1)}{1 - 2\mathrm{Fo}(\cos(k_l\Delta s) - 1)}\right|. \\
|e^{a\Delta t}| &= \left|\frac{1 - 4\mathrm{Fo}\,\mathrm{sen}^2\left(\frac{k_m\Delta s}{2}\right)}{1 + 4\mathrm{Fo}\,\mathrm{sen}^2\left(\frac{k_l\Delta s}{2}\right)}\right| \leq 1 \blacksquare
\end{aligned}$$