

Modelação Numérica 2017

Aula 11, 22/Mar

- Métodos de Lax, Upstream, Leapfrog
- Condições fronteira periódicas

<http://modnum.ucs.ciencias.ulisboa.pt>

Diferenças finitas

- Série de Taylor:

$$f(x_0 + \Delta x) = f(x_0) + \left(\frac{\partial f}{\partial x}\right)_{x=x_0} \Delta x + \frac{1}{2} \left(\frac{\partial^2 f}{\partial x^2}\right)_{x=x_0} \Delta x^2 + \frac{1}{3!} \left(\frac{\partial^3 f}{\partial x^3}\right)_{x=x_0} \Delta x^3 + \dots$$



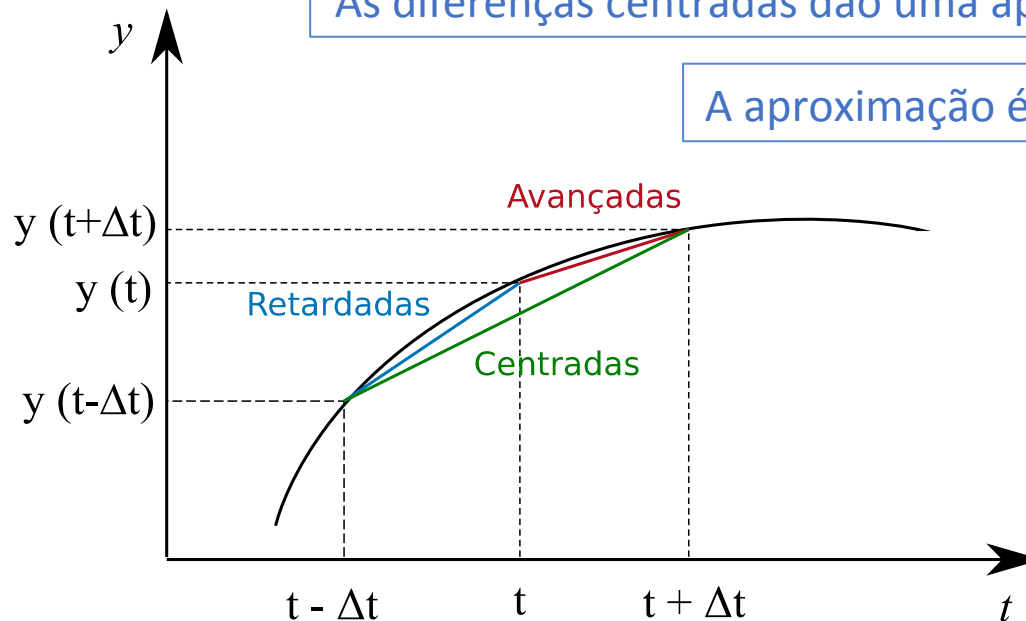
$$f(x) = e^x$$

Diferenças finitas

- Diferenças avançadas: $\left(\frac{\partial f}{\partial x}\right)_{x=x_0} = \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x} + \mathcal{O}(\Delta x)$
- Diferenças retardadas: $\left(\frac{\partial f}{\partial x}\right)_{x=x_0} = \frac{f(x_0) - f(x_0 - \Delta x)}{\Delta x} + \mathcal{O}(\Delta x)$
- Diferenças centradas: $\left(\frac{\partial f}{\partial x}\right)_{x=x_0} = \frac{f(x_0 + \Delta x) - f(x_0 - \Delta x)}{2\Delta x} + \mathcal{O}(\Delta x^2)$

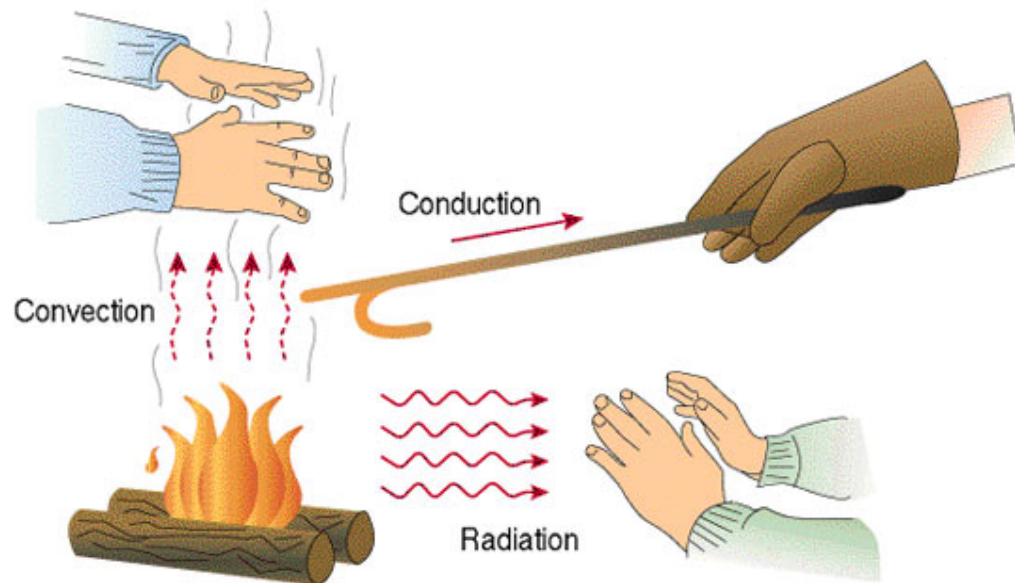
As diferenças centradas dão uma aproximação mais exacta

A aproximação é melhor quando $\Delta x \rightarrow 0$

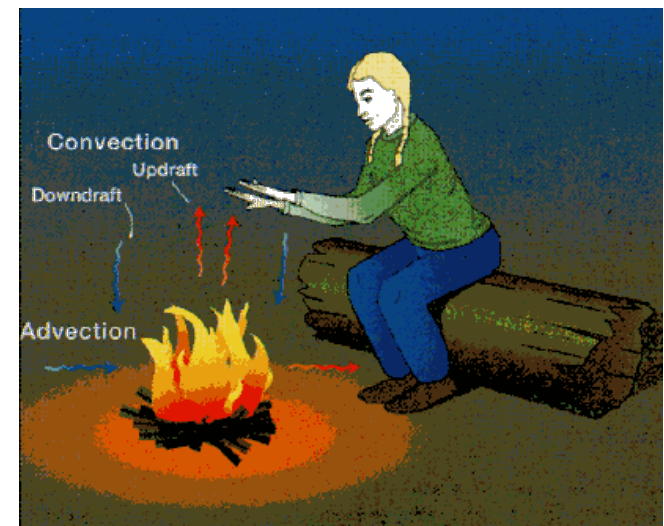


Equação de advecção (linear, 1D)

$$\frac{\partial T}{\partial t} = -u \frac{\partial T}{\partial x}, u = \text{const}$$



https://en.wikipedia.org/wiki/Taylor_series



Equação de advecção (linear, 1D)

$$\frac{\partial T}{\partial t} = -u \frac{\partial T}{\partial x}, u = \text{const}$$

- A equação é linear se $u = \text{const}$ e tem, nesse caso, **solução analítica**.
- Trata-se de um **problema de valores iniciais**. I.e., dada a distribuição inicial $T(x, t=0)$ calcular $T(x, t>0)$.
- Vamos **discretizar** a função $T(x, t) \approx T^{n\Delta t}_{k\Delta x} \equiv T^n_k$

(O índice superior representa tempo, o inferior o espaço). Vamos experimentar uma solução por **diferenças finitas usando o método de Euler**, com **diferenças avançadas no tempo e centradas no espaço**:

$$\frac{T_k^{n+1} - T_k^n}{\Delta t} = -u \frac{T_{k+1}^n - T_{k-1}^n}{2\Delta x} \Rightarrow T_k^{n+1} = T_k^n - u\Delta t \frac{T_{k+1}^n - T_{k-1}^n}{2\Delta x}$$

- Trata-se de um método com **1 nível** (o cálculo da solução no passo de tempo n só depende de 1 passo anterior $n - 1$).
- Trata-se de um **método explícito**: T_k^{n+1} depende do campo no passo de tempo anterior (e não do seu valor noutros pontos em $t = n\Delta t$).

Equação de advecção (linear, 1D)

$$\frac{\partial T}{\partial t} = -u \frac{\partial T}{\partial x}, u = \text{const}$$

$$T_k^{n+1} = T_k^n - u \Delta t \frac{T_{k+1}^n - T_{k-1}^n}{2\Delta x}$$

- Trata-se de um método de 1ª ordem no tempo e 2ª ordem no espaço.
- A solução depende de condições fronteira espaciais. Vamos definir a solução num domínio espacial finito:

$$x \in [0, L_x] \implies x_k = (k - 1)\Delta x, k = 1, \dots, N$$

- Vamos considerar dois casos:
 - Condições cíclicas (periódicas): $x_0 = x_N, x_{N+1} = x_1$
 - Condições “abertas”: $\frac{\partial T}{\partial x} = 0$, em $x = x_1, x = x_N$

```

import matplotlib.pyplot as plt
import numpy as np

plt.rcParams['figure.figsize'] = 10, 6

#%% Parâmetros

nx=1000; dx=5.           # número de pontos no espaço, intervalo entre pontos no espaço
nt=5000; dt=1.          # número de pontos no tempo, intervalo entre pontos no tempo
u=2.
x=np.arange(0,nx*dx,dx) # vector de posições

#%% Condições iniciais

x0=dx*nx/2              # ponto onde a temperatura inicial é máxima
L=100                   # largura da anomalia inicial de temperatura
Ti=np.exp(-((x-x0)/L)**2) # vector de temperaturas iniciais
T=np.zeros(len(x))      # inicializar o vector de temperaturas presentes
Tp=np.zeros(len(x))     # inicializar o vector de temperaturas futuras (próximas)
T[:]=Ti[:]

# Evolução do sistema

isp=1
for it in range(1,nt):
    for ix in range(1,nx-1):
        Tp[ix] = T[ix] - u*dt/(2*dx)* (T[ix+1] - T[ix-1]) # próxima temperatura

    Tp[nx-1] = T[nx-1] - u*dt/(2*dx)* (T[0] - T[nx-2]) # fronteira cíclica
    Tp[0] = T[0] - u*dt/(2*dx)* (T[1] - T[nx-1]) # fronteira cíclica
    T[:]=Tp[:]

    if (it+1)%250==0 and isp<=5:
        plt.subplot(5,1,isp)
        plt.plot(x,Ti,'b', x,T,'r')
        plt.xlabel('x')
        plt.ylabel('T')
        plt.title('t='+str(it*dt))
        plt.grid()
        isp += 1

    if max(T) > 10:
        print('it=' + str(it)+ ', T=' + str(T))
        break

plt.tight_layout()

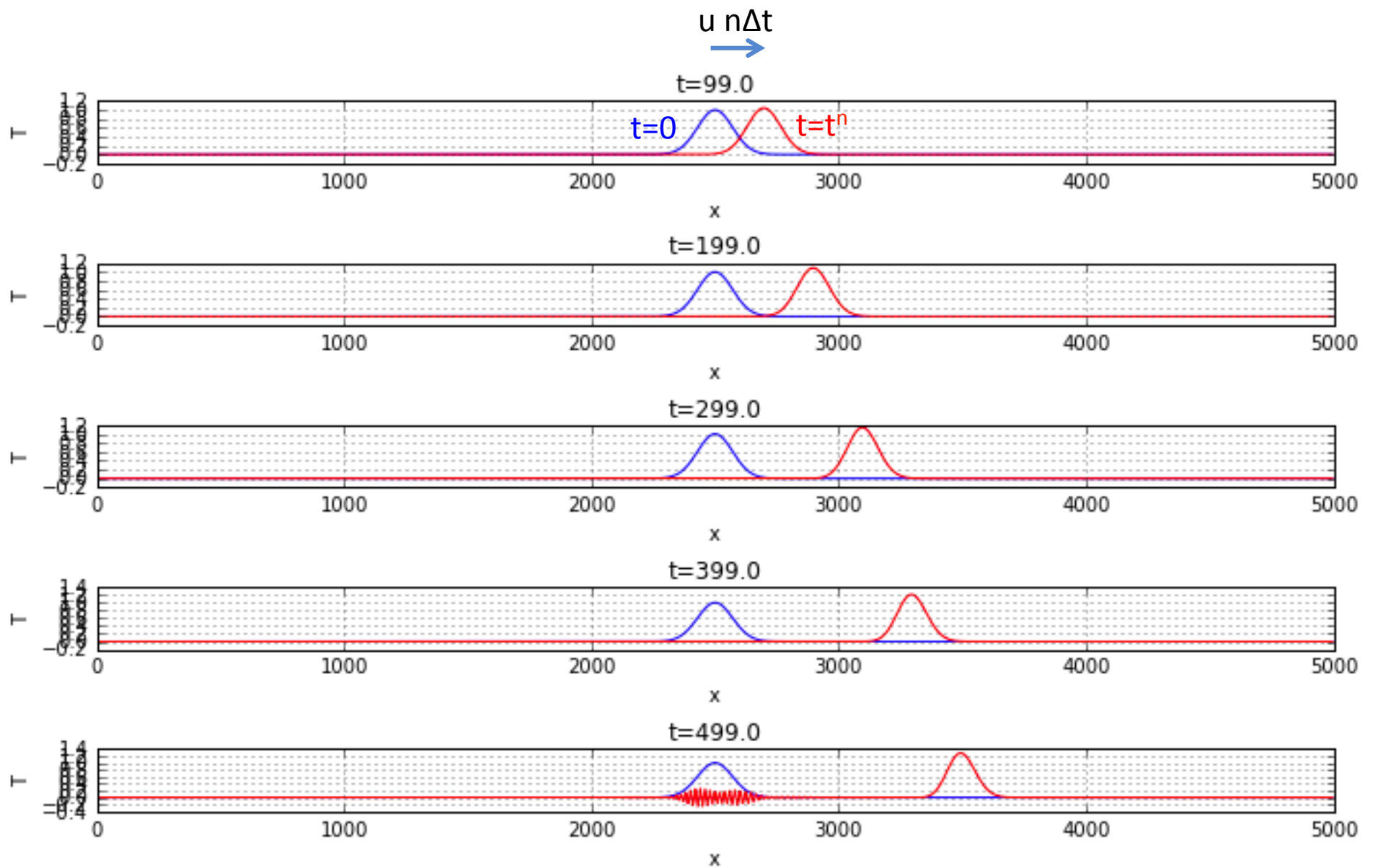
```

$$T_k^{n+1} = T_k^n - u\Delta t \frac{T_{k+1}^n - T_{k-1}^n}{2\Delta x}$$

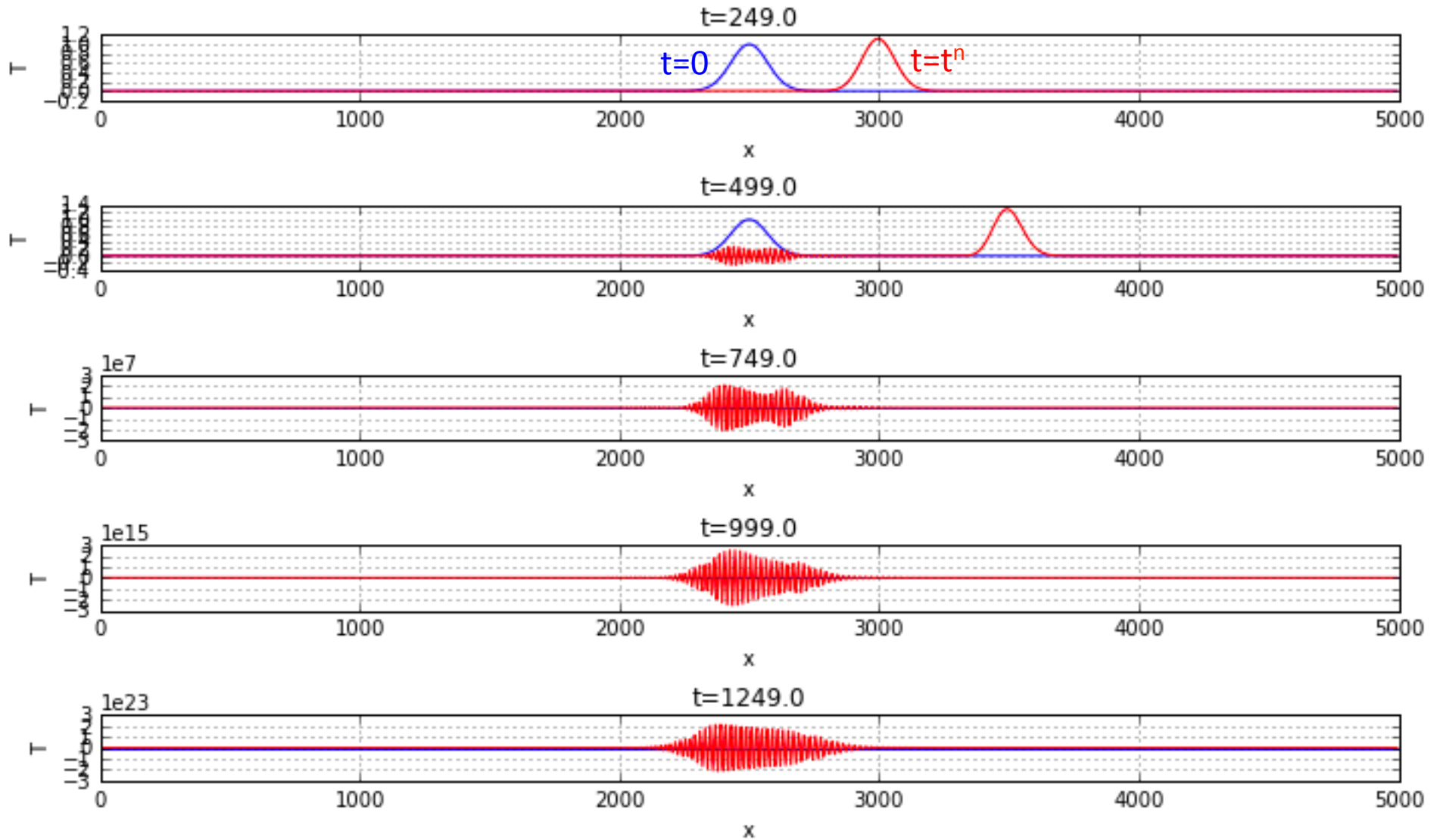
$$x_0 = x_N, x_{N+1} = x_1$$

T^{n+1} : Tp
 T^n : T
 T^{n-1} : Tm

FTCS – Forward-time, central space (método instável)



FTCS – Forward-time, central space (método instável)



O método de Euler é incondicionalmente instável

$$T_k^{n+1} = T_k^n - u\Delta t \frac{T_{k+1}^n - T_{k-1}^n}{2\Delta x}$$

- A instabilidade é, neste caso, independente da escolha dos parâmetros de discretização Δt , Δx e não é uma consequência da ordem da aproximação. É possível definir esquemas de 1ª ordem (ou ordem mais elevada) condicionalmente estáveis.

Aproximação de Lax–Friedrichs

- Em vez de: $T_k^{n+1} = \underline{T_k^n} - u\Delta t \frac{T_{k+1}^n - T_{k-1}^n}{2\Delta x}$
- Fazemos: $T_k^{n+1} = \underline{\frac{1}{2}(T_{k-1}^n + T_{k+1}^n)} - u\Delta t \frac{T_{k+1}^n - T_{k-1}^n}{2\Delta x}$
- Continua a ser um método com 1 nível temporal e de primeira ordem no tempo e segunda no espaço.

```

%% Parâmetros
nx=1000; dx=5. # número de pontos no espaço, intervalo entre pontos no espaço
nt=5000; dt=1.5 # número de pontos no tempo, intervalo entre pontos no tempo
u=4. # vector de posições
x=np.arange(0,nx*dx,dx)
courant=u*dt/dx

%% Condições iniciais
x0=1000 # ponto onde a temperatura inicial é máxima
L=100 # largura da anomalia inicial de temperatura
Ti=np.exp(-((x-x0)/L)**2) # vector de temperaturas iniciais
T=np.zeros(len(x)) # inicializar o vector de temperaturas presentes
Tp=np.zeros(len(x)) # inicializar o vector de temperaturas futuras (próximas)
T[:]=Ti[:]

# Evolução do sistema
isp=1
for it in range(1,nt):
    for ix in range(1,nx-1):
        Tp[ix] = .5*(T[ix-1]+T[ix+1]) - u*dt/(2*dx)* (T[ix+1] - T[ix-1]) # próxima temperatura

    Tp[nx-1] = .5*(T[nx-2]+T[0]) - u*dt/(2*dx)* (T[0] - T[nx-2]) # fronteira cíclica
    Tp[0] = .5*(T[nx-1]+T[1]) - u*dt/(2*dx)* (T[1] - T[nx-1]) # fronteira cíclica
    T[:]=Tp[:]

    if (it+1)%250==0 and isp<=5:
        plt.subplot(5,1,isp)
        plt.plot(x,Ti,'b', x,T,'r')
        plt.xlabel('x')
        plt.ylabel('T')
        plt.title('Lax, t='+str(it*dt) + ', u='+str(u) + ', dx='+str(dx) + ', dt='+str(dt) + ', co
        plt.grid()
        isp += 1

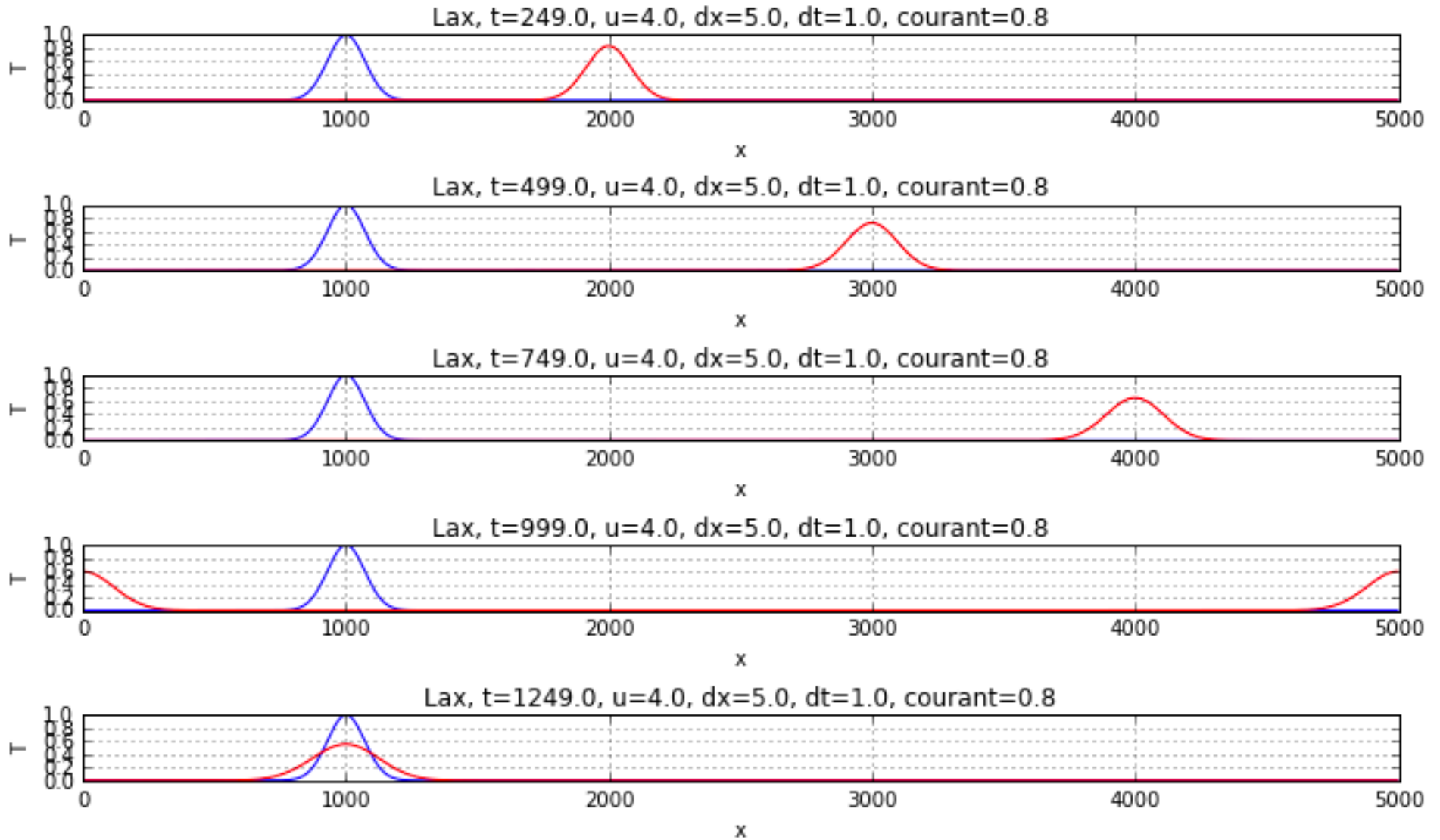
# if max(T) > 10:
#     print('it=' + str(it)+ ', T=' + str(T))
#     break

plt.tight_layout()

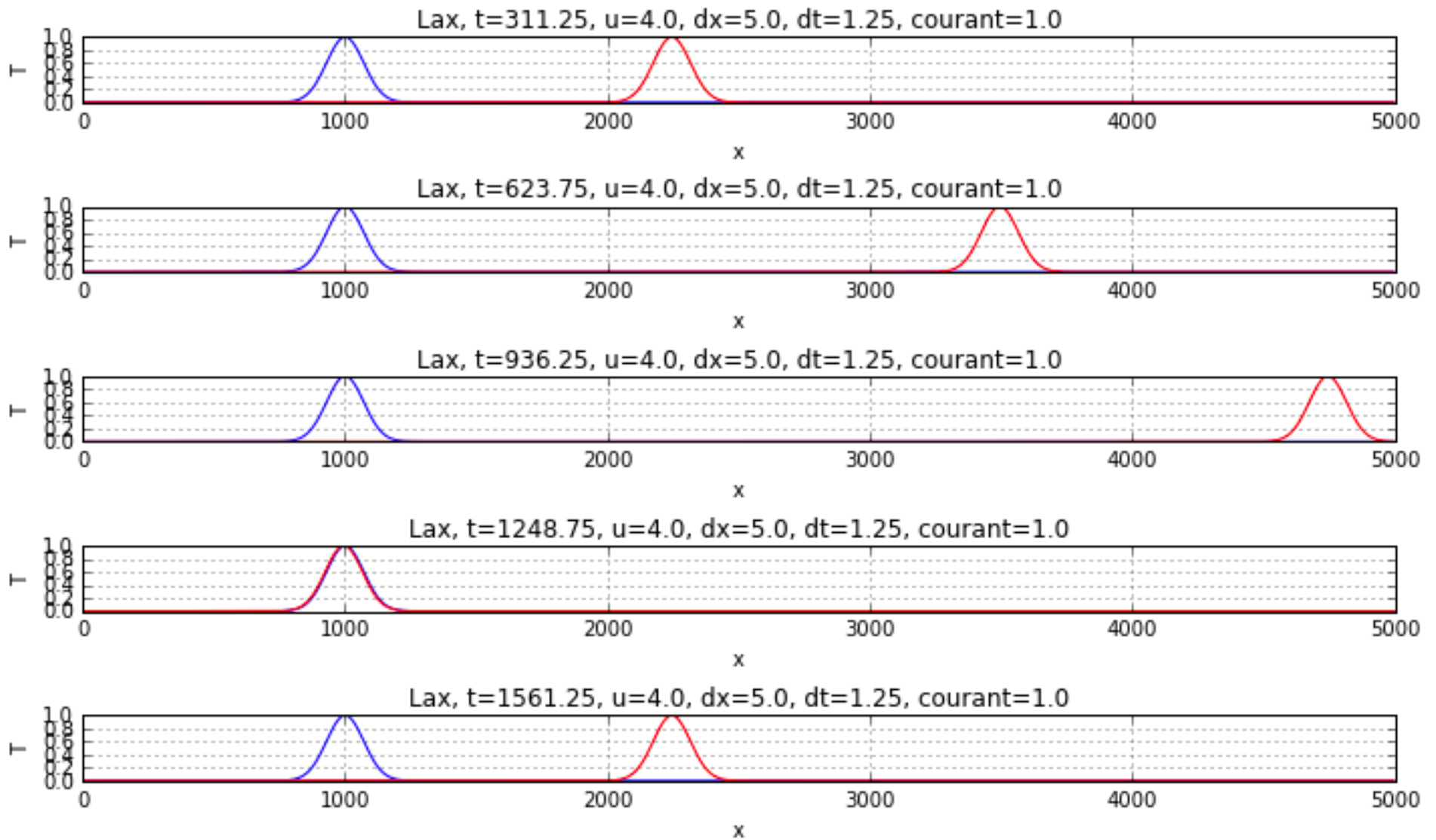
```

$$T_k^{n+1} = \frac{1}{2}(T_{k-1}^n + T_{k+1}^n) - u\Delta t \frac{T_{k+1}^n - T_{k-1}^n}{2\Delta x}$$

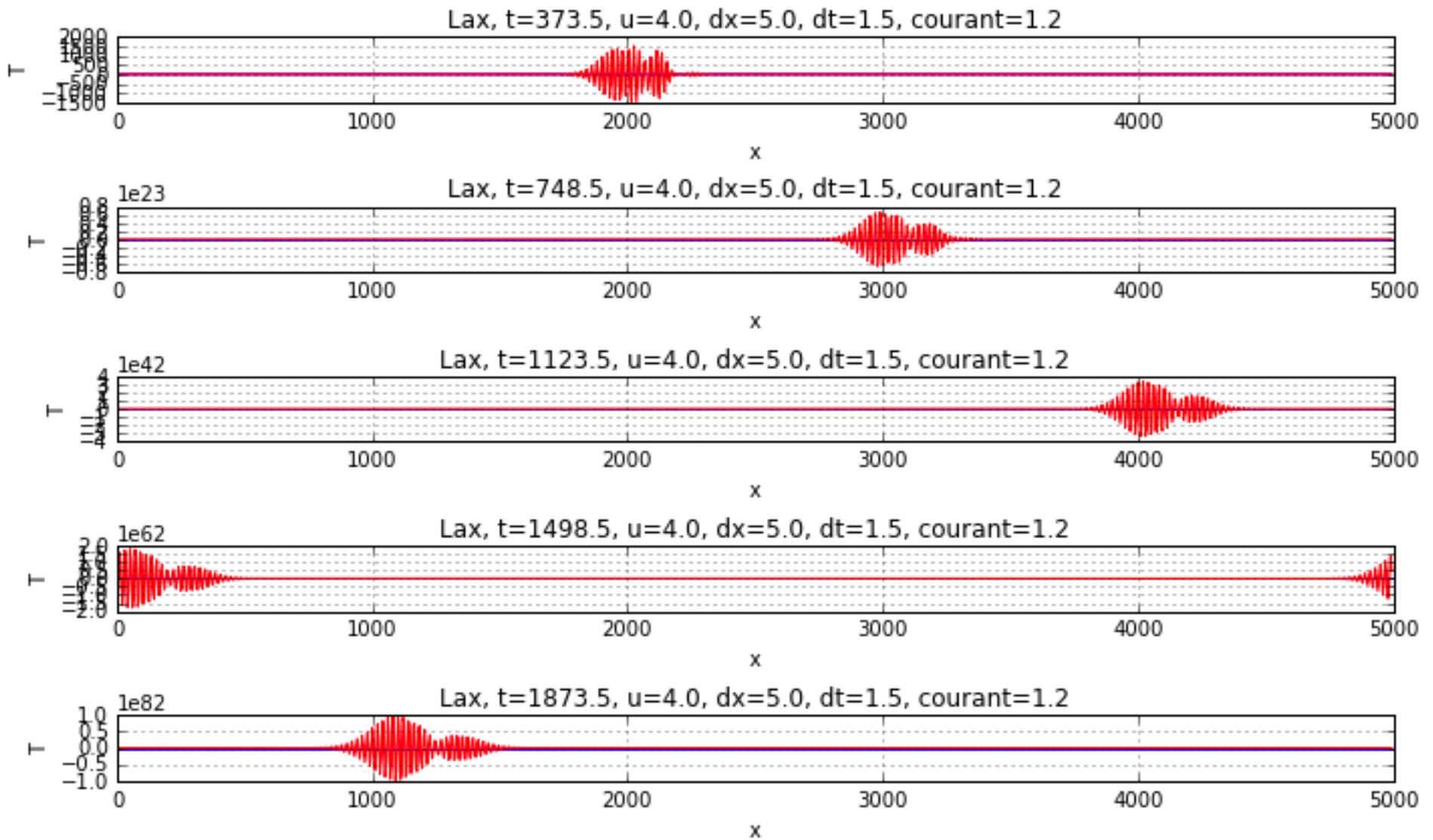
Comportamento do método Lax



Comportamento do método Lax

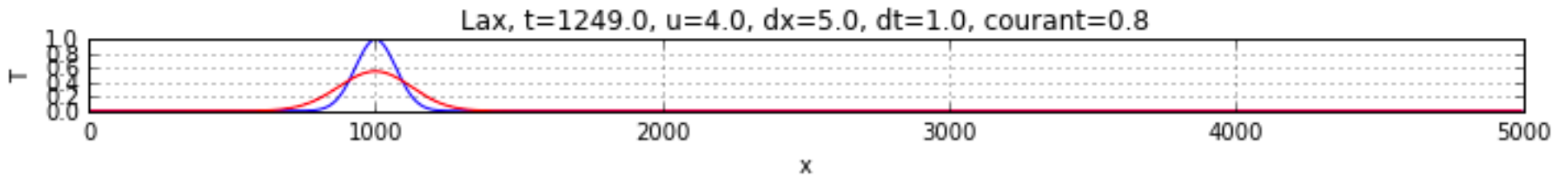


Comportamento do método Lax

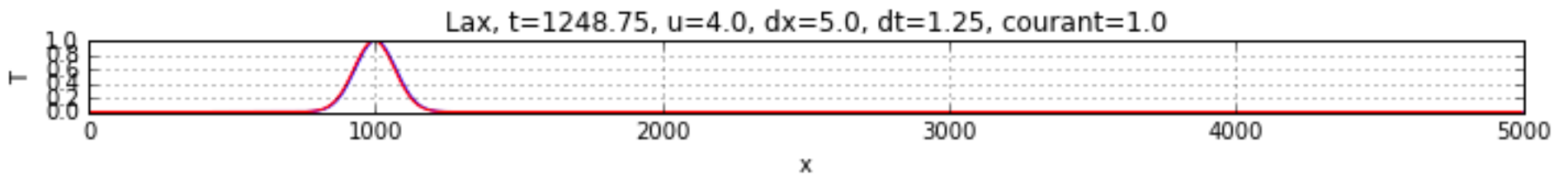


Comportamento do método Lax

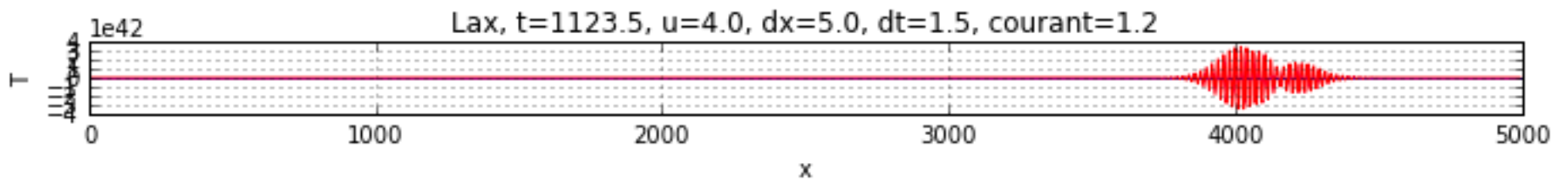
- Estável, difusivo:



- Estável (quase perfeito!):



- Instável



- Número de Courant: $\frac{u\Delta t}{\Delta x} \begin{cases} \leq 1, \text{ estável} \\ > 1, \text{ instável} \end{cases}$

Upstream differencing

$$\frac{\partial T}{\partial t} = -u \frac{\partial T}{\partial x}, u = \text{const}$$

- FTCS:
$$T_k^{n+1} = T_k^n - u\Delta t \frac{T_{k+1}^n - T_{k-1}^n}{2\Delta x}$$

- Se $u > 0$:
$$T_k^{n+1} = T_k^n - u\Delta t \frac{T_k^n - T_{k-1}^n}{\Delta x}$$

- Se $u < 0$:
$$T_k^{n+1} = T_k^n - u\Delta t \frac{T_{k+1}^n - T_k^n}{\Delta x}$$

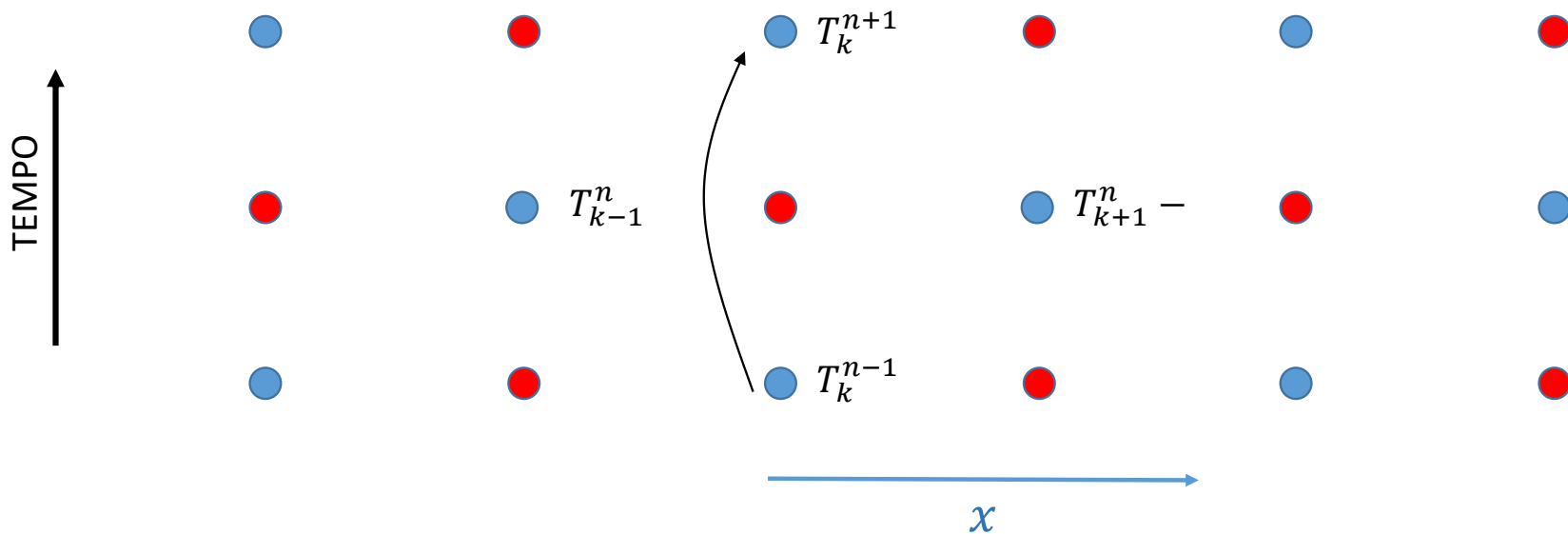
Método de primeira ordem tanto no espaço como no tempo, explícito, de 1 nível.

Leapfrog (2ª ordem)

$$\frac{\partial T}{\partial t} = -u \frac{\partial T}{\partial x}, u = \text{const}$$

$$\frac{T_k^{n+1} - T_k^{n-1}}{2\Delta t} = -u \frac{T_{k+1}^n - T_{k-1}^n}{2\Delta x}$$

Problema: A malha computacional fica dividida em dois conjuntos desacoplados...



Parâmetros

$$\frac{\partial T}{\partial t} = -u \frac{\partial T}{\partial x}, u = \text{const}$$

```
import matplotlib.pyplot as plt
import numpy as np

plt.rcParams['figure.figsize'] = 10, 6

### Parâmetros

nx=1000; dx=5.           # número de pontos no espaço, intervalo entre pontos no espaço
dt=1                     # número de pontos no tempo, intervalo entre pontos no tempo
u=1.                     # velocidade
L=100                    # largura da anomalia inicial de temperatura
x0=1000                  # ponto onde a temperatura inicial é máxima
x=np.arange(0, nx*dx, dx) # vector de posições
courant=u*dt/dx          # número de Courant
voltas=10                # número de voltas até ao final do modelo
nt = int(((max(x)-min(x))/u/dt+1) * voltas) # número de pontos no tempo
Ti=np.exp(-((x-x0)/L)**2) # vector de temperaturas iniciais
```

Lax:
$$T_k^{n+1} = \frac{1}{2}(T_{k-1}^n + T_{k+1}^n) - u\Delta t \frac{T_{k+1}^n - T_{k-1}^n}{2\Delta x}$$

$$\frac{\partial T}{\partial t} = -u \frac{\partial T}{\partial x}, u = \text{const}$$

```

%% Lax
%% Condições iniciais

T=np.zeros(len(x))           # inicializar o vector de temperaturas presentes
Tp=np.zeros(len(x))         # inicializar o vector de temperaturas futuras (próximas)
T[:]=Ti[:]

# Evolução do sistema
for it in range(1,nt):
    for ix in range(1,nx-1):
        Tp[ix] = .5*(T[ix-1]+T[ix+1]) - u*dt/(2*dx)* (T[ix+1] - T[ix-1])    # próxima temperatura

    Tp[nx-1] = .5*(T[nx-2]+T[0]) - u*dt/(2*dx)* (T[0] - T[nx-2])           # fronteira cíclica
    Tp[0] = .5*(T[nx-1]+T[1]) - u*dt/(2*dx)* (T[1] - T[nx-1])             # fronteira cíclica
    T[:]=Tp[:]

Tlax=T

plt.plot(x,Ti,'b', x,T,'r')
plt.xlabel('x')
plt.ylabel('T')
plt.title('Lax, t='+str(it*dt) + ', u='+str(u) + ', dx='+str(dx) + ', dt='+str(dt) + ', courant='+
plt.grid()

```

Upstream:
$$T_k^{n+1} = T_k^n - u\Delta t \frac{T_k^n - T_{k-1}^n}{\Delta x}$$

$$\frac{\partial T}{\partial t} = -u \frac{\partial T}{\partial x}, u = \text{const}$$

```

#%% Upstream
#%% Condições iniciais

T=np.zeros(len(x))          # inicializar o vector de temperaturas presentes
Tp=np.zeros(len(x))        # inicializar o vector de temperaturas futuras (próximas)
T[:]=Ti[:]

# Evolução do sistema
for it in range(1,nt):
    for ix in range(1,nx-1):
        Tp[ix] = T[ix] - u*dt/(dx)* (T[ix] - T[ix-1])    # próxima temperatura

    Tp[nx-1] = T[nx-1] - u*dt/(dx)* (T[nx-1] - T[nx-2])    # fronteira cíclica
    Tp[0] = T[0] - u*dt/(dx)* (T[0] - T[nx-1])             # fronteira cíclica
    T[:]=Tp[:]

Tupstream=T

plt.plot(x,Ti,'b', x,T,'r')
plt.xlabel('x')
plt.ylabel('T')
plt.title('Upstream, t='+str(it*dt) + ', u='+str(u) + ', dx='+str(dx) + ', dt='+str(dt) + ', cou
plt.grid()

```

Leapfrog:
$$\frac{T_k^{n+1} - T_k^{n-1}}{2\Delta t} = -u \frac{T_{k+1}^n - T_{k-1}^n}{2\Delta x}$$

$$\frac{\partial T}{\partial t} = -u \frac{\partial T}{\partial x}, u = \text{const}$$

```

T=np.zeros(len(x))           # inicializar o vector de temperaturas presentes (N)
Tm=np.zeros(len(x))         # inicializar o vector de temperaturas anteriores (M = N-1)
Tp=np.zeros(len(x))         # inicializar o vector de temperaturas futuras (P = N+1)
T[:]=Ti[:]
Tm[:]=T[:]
Tp[:]=T[:]

# Evolução do sistema

# 1o passo, Euler:
for ix in range(1,nx-1):
    T[ix] = Tm[ix] - u*dt/(2*dx)* (Tm[ix+1] - Tm[ix-1]) # próxima temperatura
T[nx-1] = Tm[nx-1] - u*dt/(2*dx)* (Tm[0] - Tm[nx-2])
T[0] = Tm[0] - u*dt/(2*dx)* (Tm[1] - Tm[nx-2])

# passos seguintes:
for it in range(2,nt):
    for ix in range(1,nx-1):
        Tp[ix] = Tm[ix] - u*dt/(2*dx)* (T[ix+1] - T[ix-1]) # temperatura futura (P)

    Tp[nx-1] = Tm[nx-1] - u*dt/(2*dx)* (T[0] - T[nx-2]) # fronteira cíclica
    Tp[0] = Tm[0] - u*dt/(2*dx)* (T[1] - T[nx-1]) # fronteira cíclica
    Tm[:]=T[:]
    T[:]=Tp[:]

Tleapfrog=T

plt.plot(x,Ti,'b', x,T,'r')
plt.xlabel('x')
plt.ylabel('T')
plt.title('Leapfrog, t='+str(it*dt) + ', u='+str(u) + ', dx='+str(dx) + ', dt='+str(dt) + ', coura
plt.grid()

```

Plot all

$$\frac{\partial T}{\partial t} = -u \frac{\partial T}{\partial x}, u = \text{const}$$

```
#%% plot all
```

```
plt.rcParams['figure.figsize'] = 10, 6  
plt.close()
```

```
plt.subplot(3,1,1)  
plt.plot(x,Ti,'b', x,Tlax,'r')  
plt.xlabel('x')  
plt.ylabel('T')  
plt.title('Lax, t='+str(it*dt) + ', u='+str(u) + ', dx='+str(dx) + ', dt='+str(dt) + ', courant='+  
plt.grid()
```

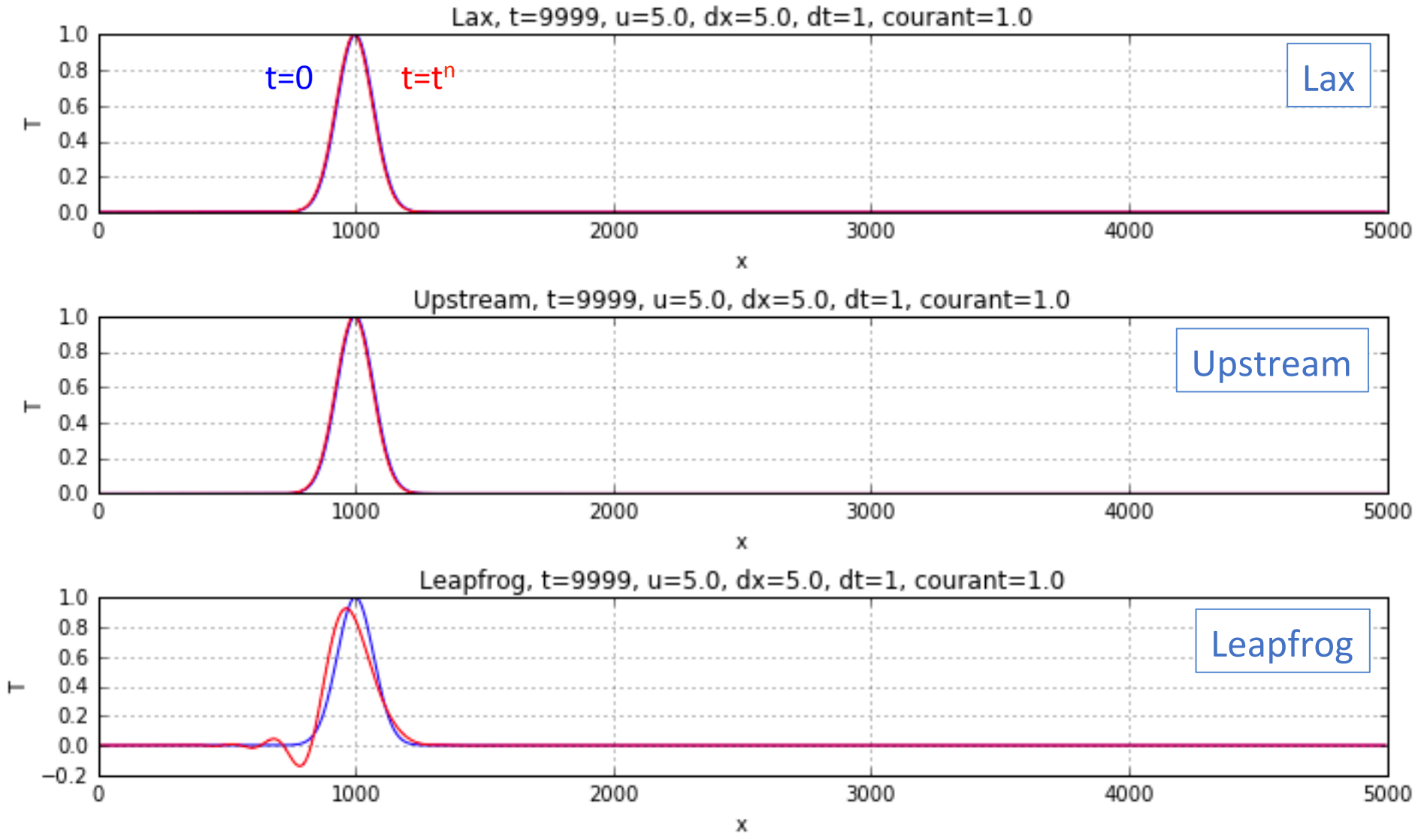
```
plt.subplot(3,1,2)  
plt.plot(x,Ti,'b', x,Tupstream,'r')  
plt.xlabel('x')  
plt.ylabel('T')  
plt.title('Upstream, t='+str(it*dt) + ', u='+str(u) + ', dx='+str(dx) + ', dt='+str(dt) + ', coura  
plt.grid()
```

```
plt.subplot(3,1,3)  
plt.plot(x,Ti,'b', x,Tleapfrog,'r')  
plt.xlabel('x')  
plt.ylabel('T')  
plt.title('Leapfrog, t='+str(it*dt) + ', u='+str(u) + ', dx='+str(dx) + ', dt='+str(dt) + ', coura  
plt.grid()
```

```
plt.tight_layout()
```

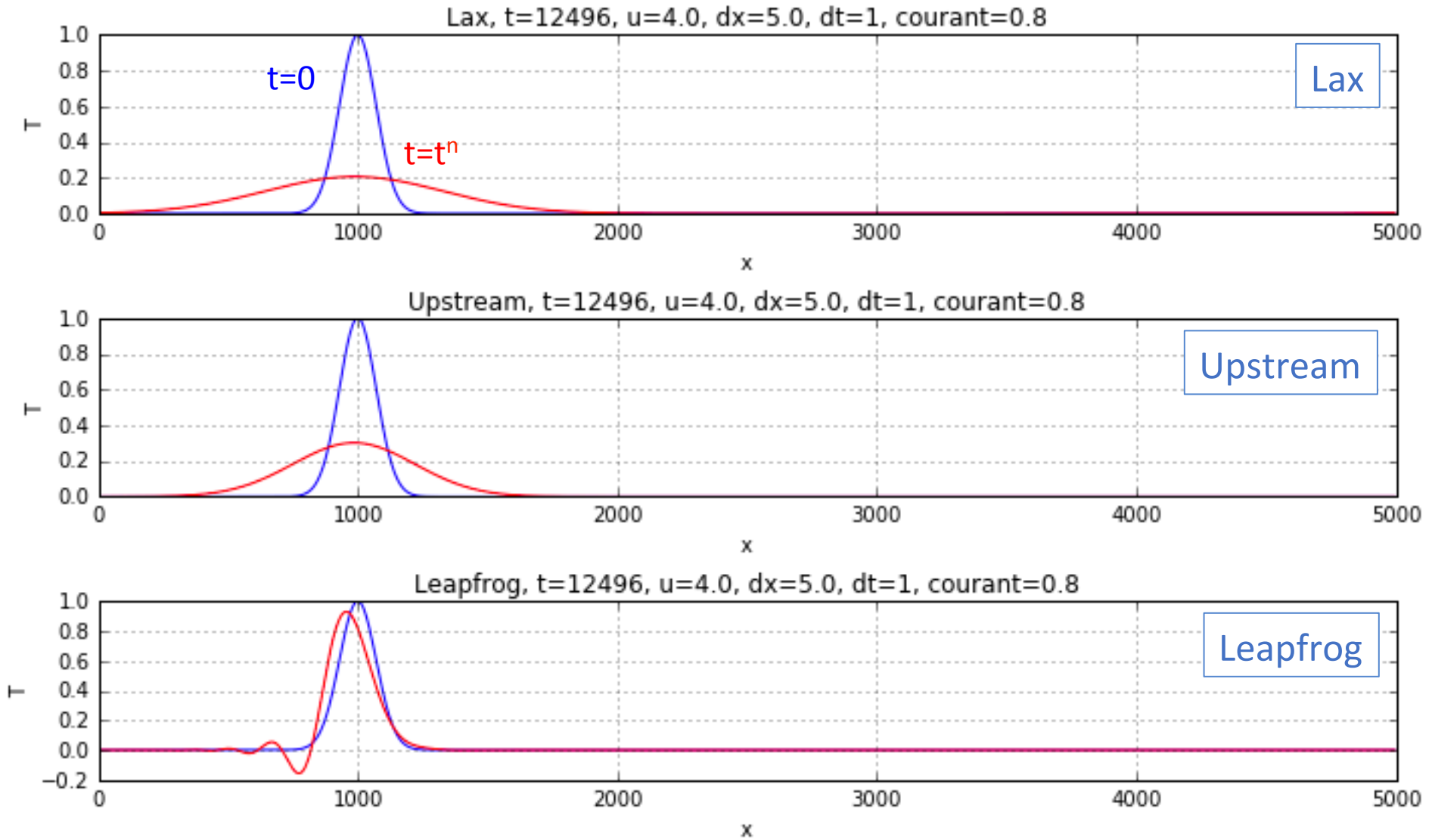
Courant = 1.0

$$\frac{\partial T}{\partial t} = -u \frac{\partial T}{\partial x}, u = \text{const}$$



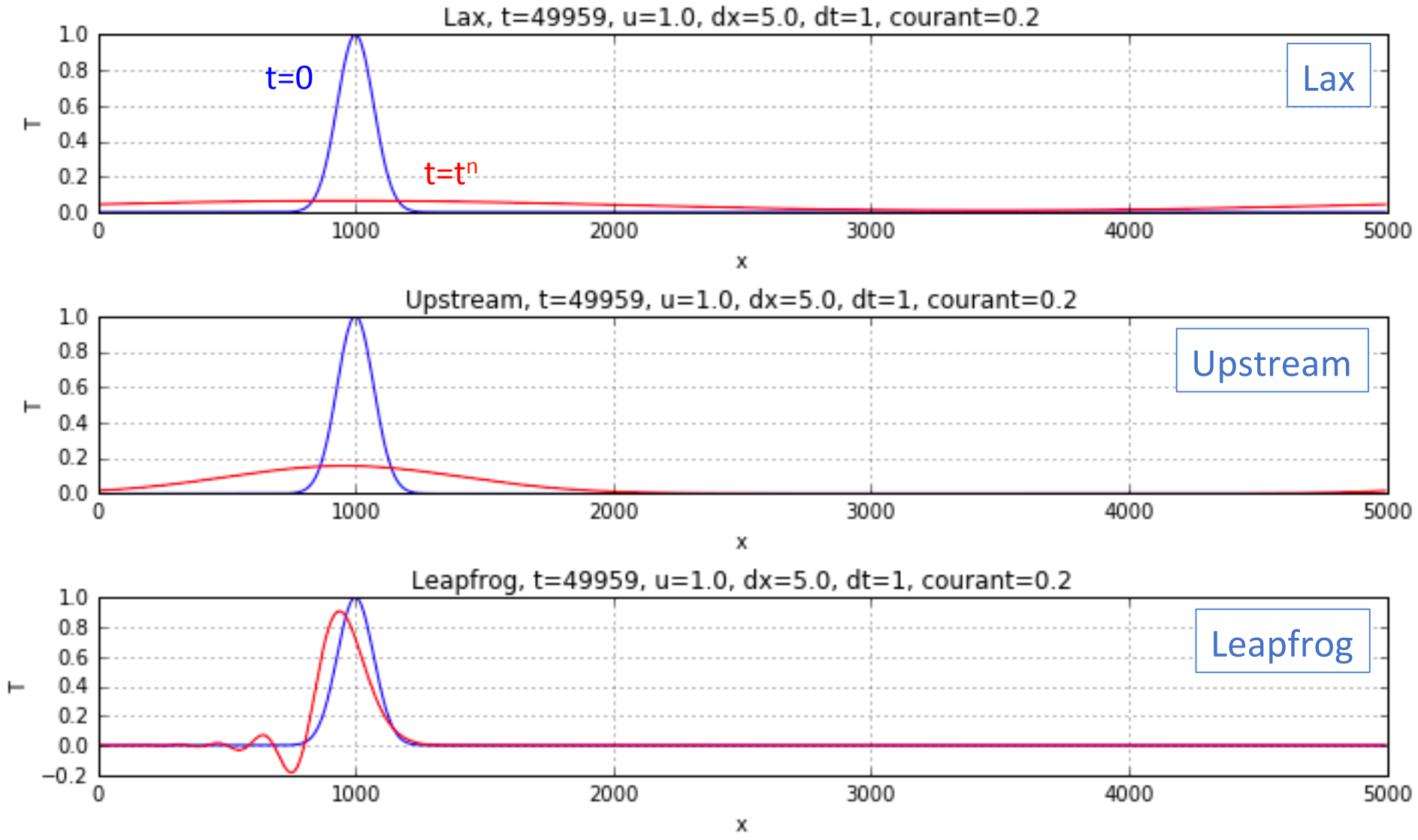
Courant = 0.8

$$\frac{\partial T}{\partial t} = -u \frac{\partial T}{\partial x}, \quad u = \text{const}$$

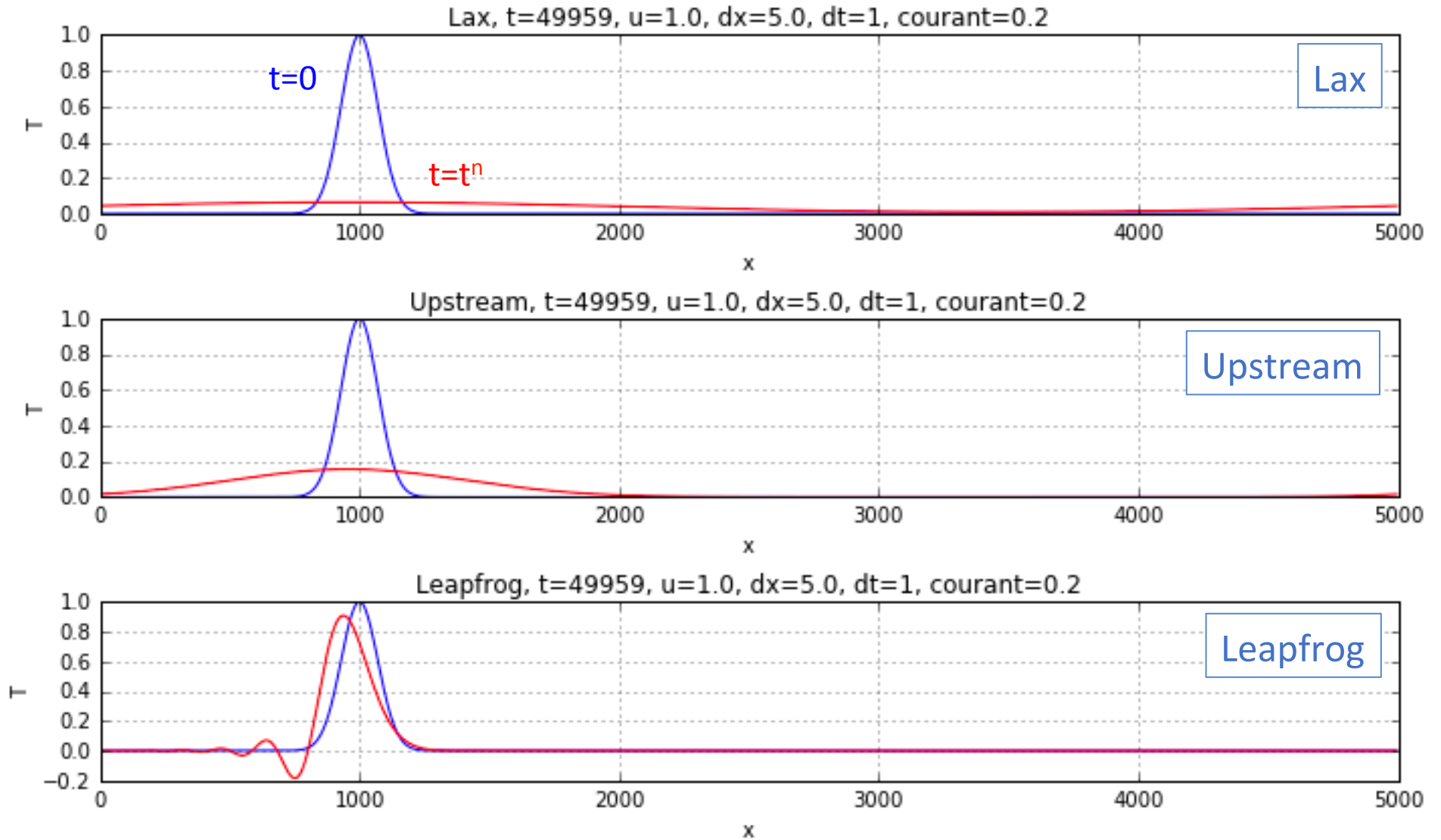


Courant = 0.2

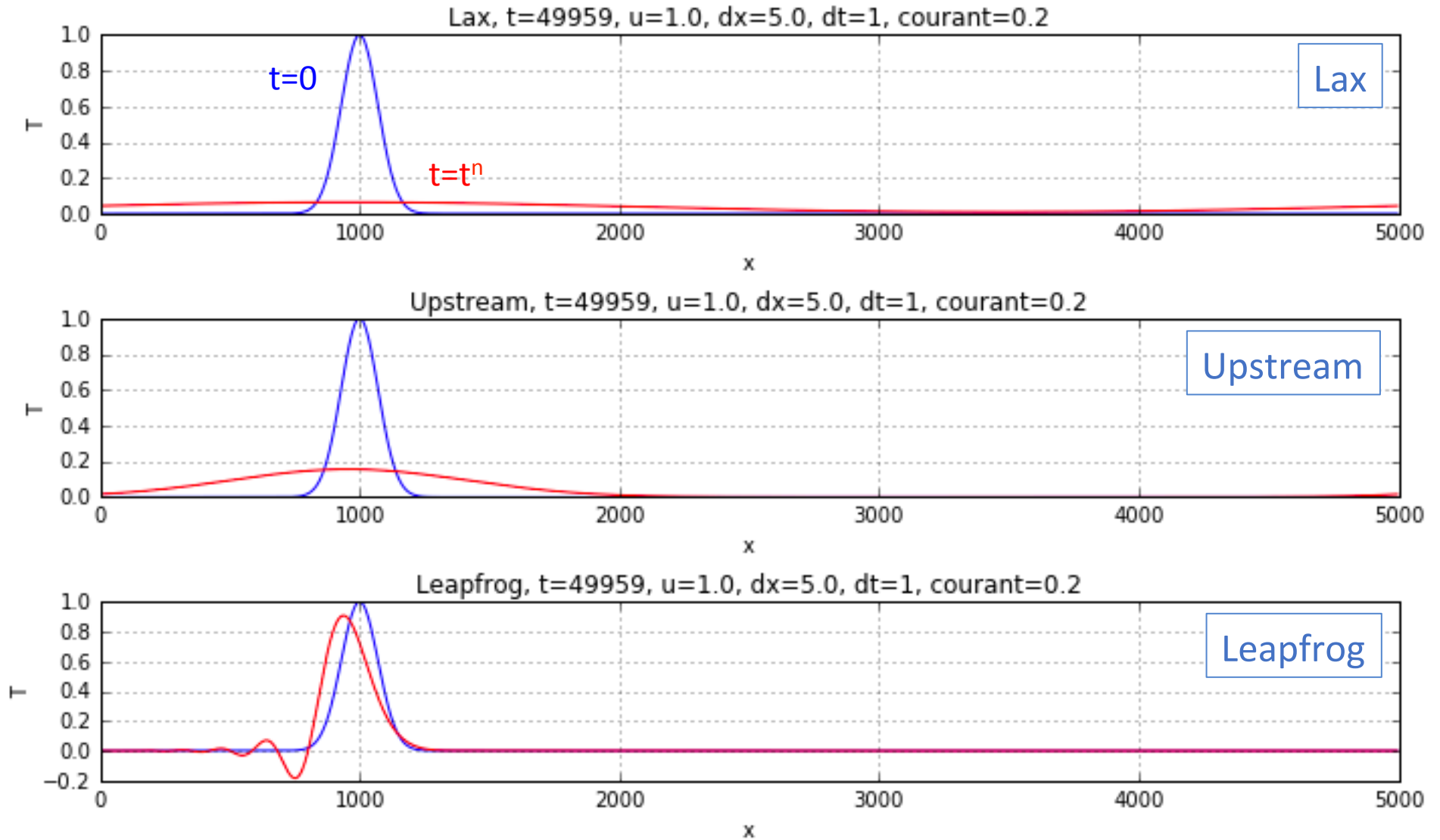
$$\frac{\partial T}{\partial t} = -u \frac{\partial T}{\partial x}, \quad u = \text{const}$$



A dissipação dos métodos de Lax e Upstream depende do Número de Courant

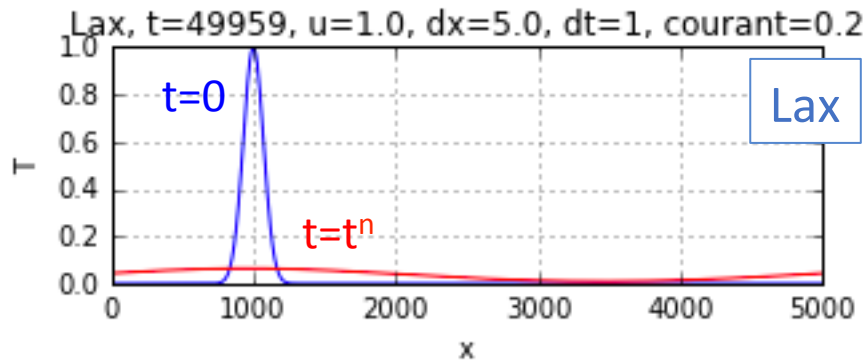


Leapfrog é pouco dissipativo (mantém amplitude) mas é dispersivo (a velocidade de fase depende do comprimento de onda).

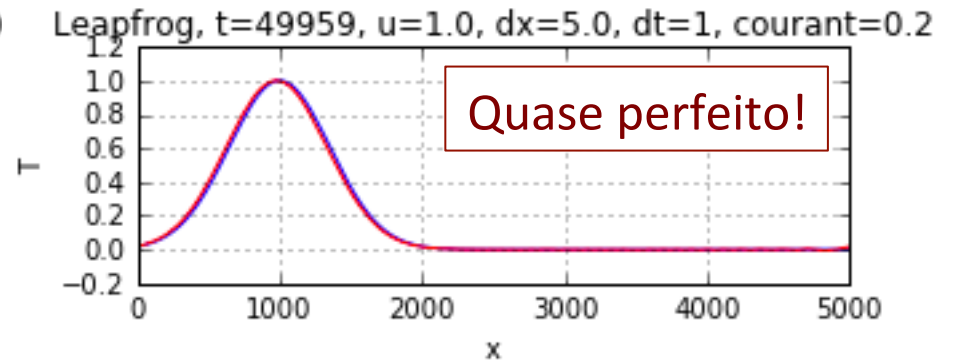
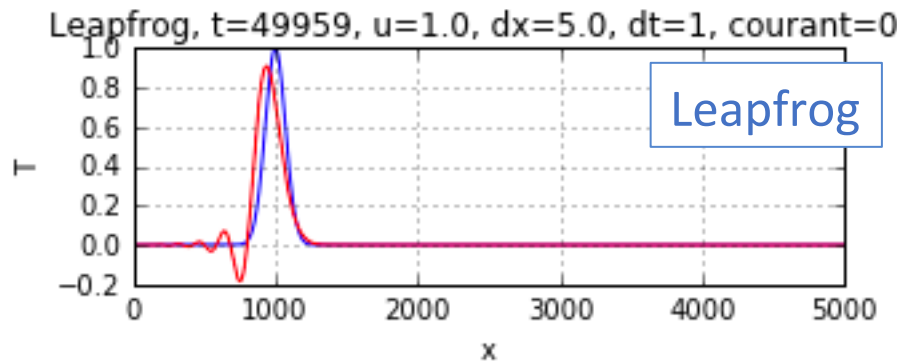
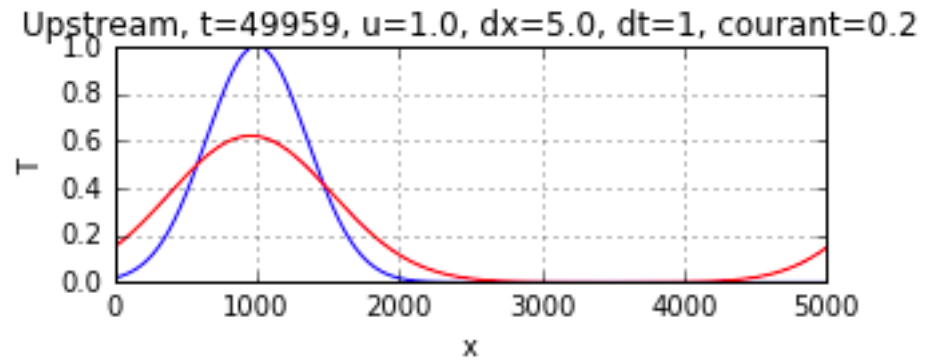
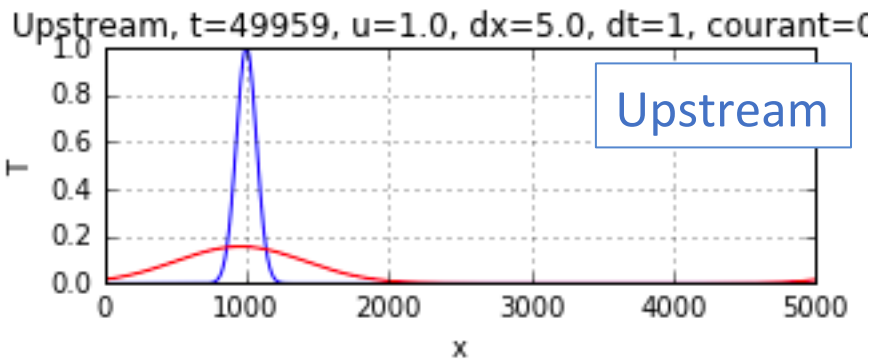
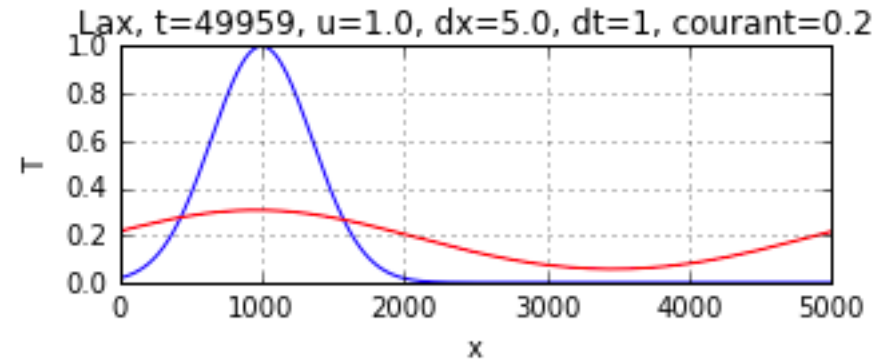


Impacto do espectro da perturbação a ser advectada (Courant=0.2, L=500)

L=100

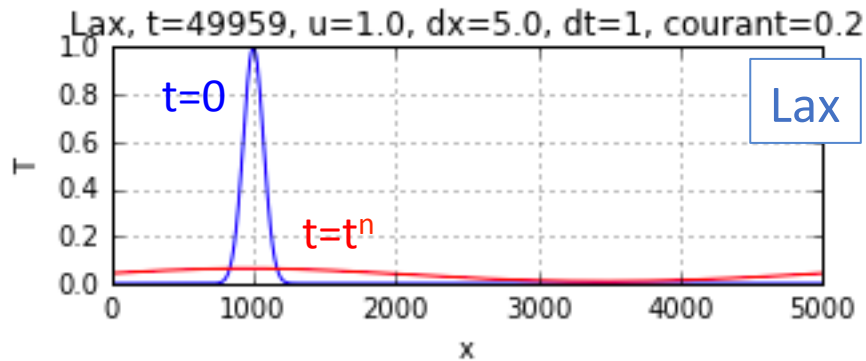


L=500

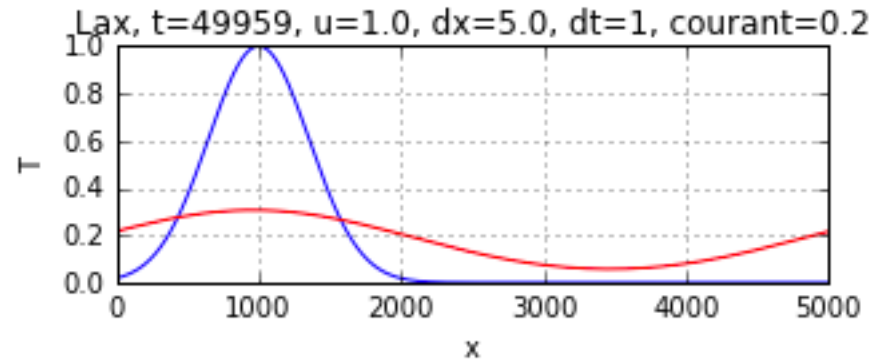


Impacto do espectro da perturbação a ser advectada (Courant=0.2, L=500)

L=100



L=500



=0.2

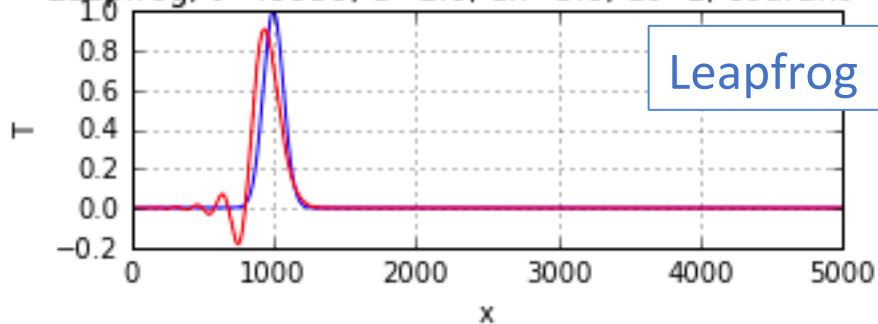
Leapfrog:

- Cauda de pequenos comprimentos de onda.
- Mas pouca atenuação, fase da perturbação principal só ligeiramente atrasada.

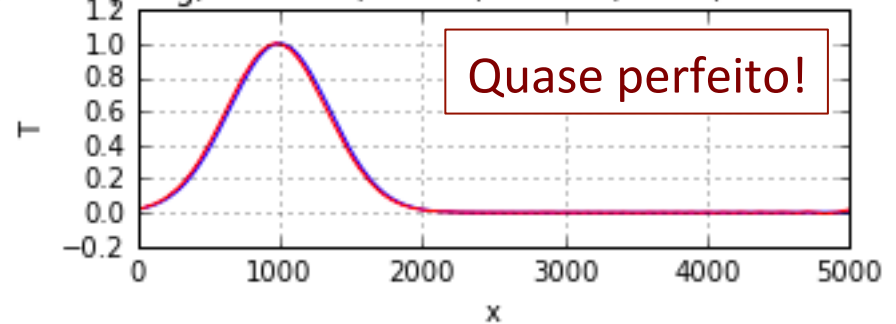
x

x

Leapfrog, t=49959, u=1.0, dx=5.0, dt=1, courant=0



Leapfrog, t=49959, u=1.0, dx=5.0, dt=1, courant=0.2



Condições fronteira abertas

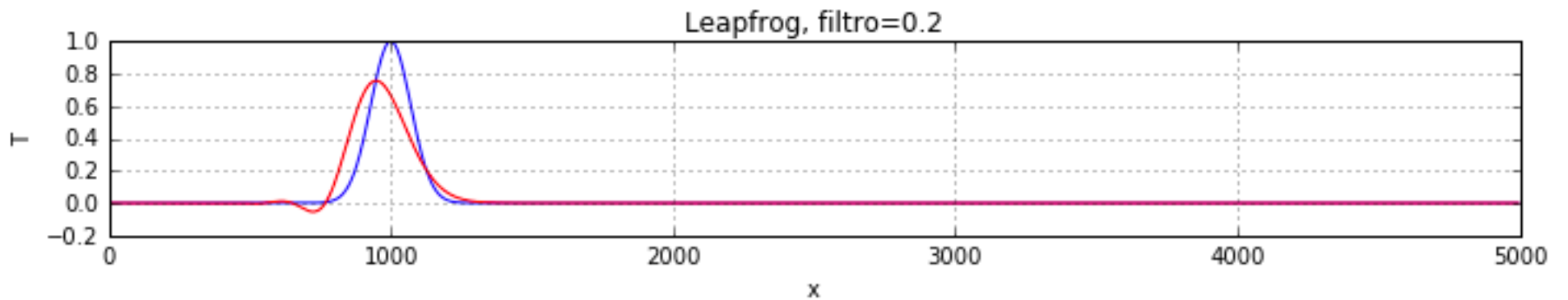
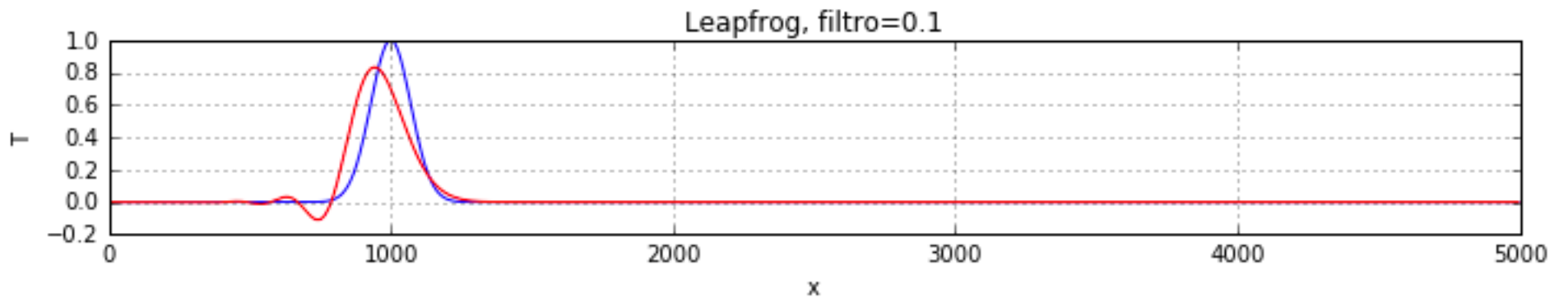
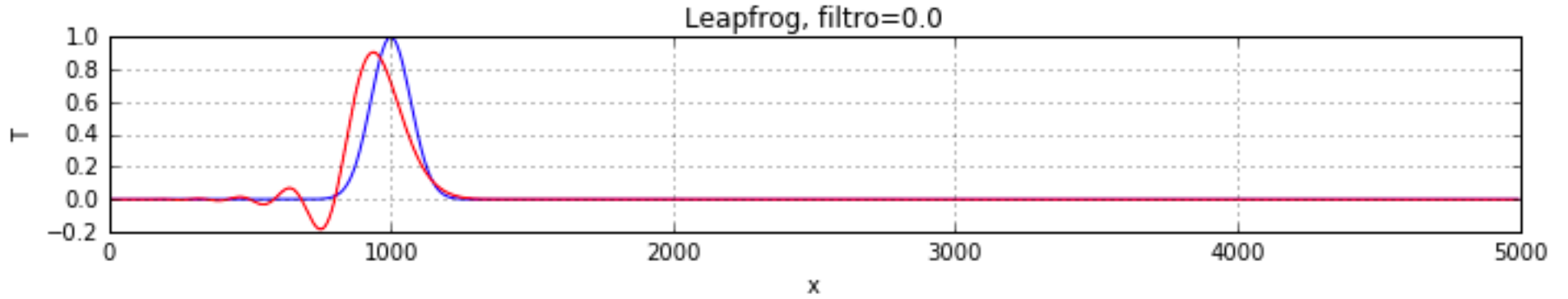
No caso da equação da advecção linear a condição fronteira aberta é muito simples:

$$\left(\frac{\partial T}{\partial x}\right)_{x=x_{max}} = \left(\frac{\partial T}{\partial x}\right)_{x=0} = 0$$

Em geral, é mais complicado...

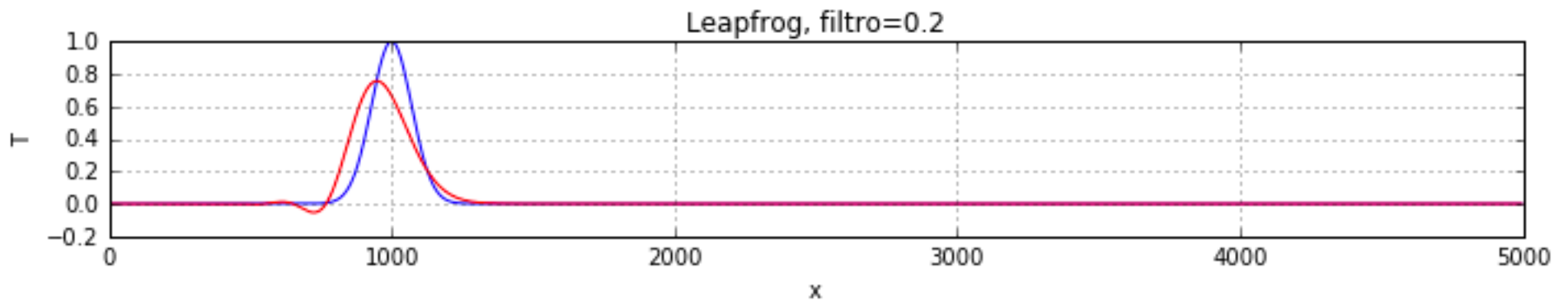
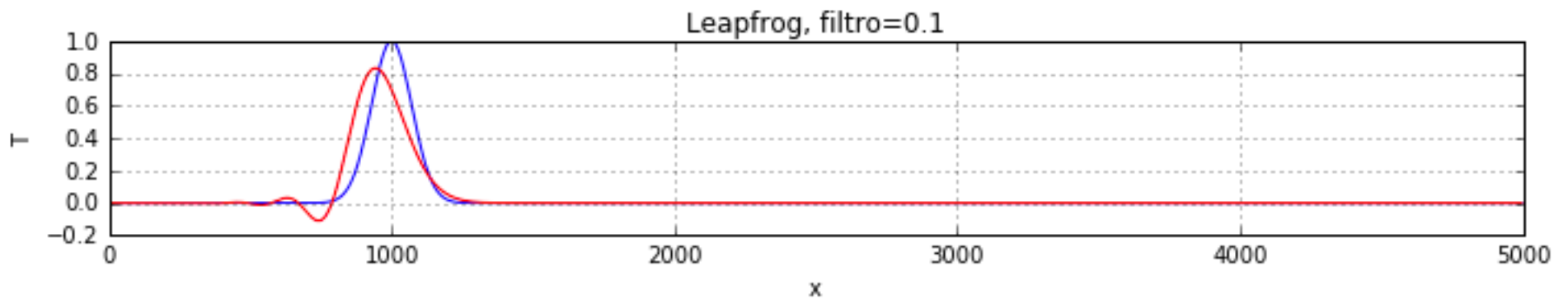
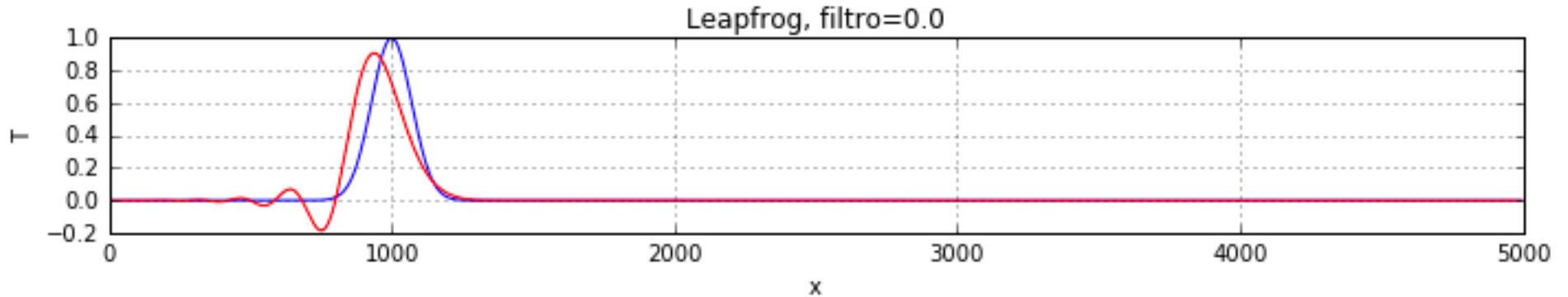
Filtro temporal no Leapfrog

$$T[:, :] = T[:, :] + \text{filtro} * (T_m + T_p - 2 * T)$$



Reduz a dispersão; Evita a instabilidade; Tem um efeito difusivo (atenua) .

$$T[:, :] = T[:, :] + \text{filtro} * (T_m + T_p - 2 * T)$$



```

# %% Leapfrog com filtro
# %% Condições iniciais

isp=1
plt.rcParams['figure.figsize'] = 10, 6
plt.close()

for filtro in [0., 0.1, 0.2]:
    T=np.zeros(len(x))           # inicializar o vector de temperaturas presentes (N)
    Tm=np.zeros(len(x))         # inicializar o vector de temperaturas anteriores (M = N-1)
    Tp=np.zeros(len(x))         # inicializar o vector de temperaturas futuras (P = N+1)
    T[:]=Ti[:]
    Tm[:]=T[:]
    Tp[:]=T[:]

    # Evolução do sistema

    # 1o passo, Euler:
    for ix in range(1,nx-1):
        T[ix] = Tm[ix] - u*dt/(2*dx)* (Tm[ix+1] - Tm[ix-1])   # próxima temperatura
    T[nx-1] = Tm[nx-1] - u*dt/(2*dx)* (Tm[0] - Tm[nx-2])
    T[0] = Tm[0] - u*dt/(2*dx)* (Tm[1] - Tm[nx-2])

    # passos seguintes:
    for it in range(2,nt):
        for ix in range(1,nx-1):
            Tp[ix] = Tm[ix] - u*dt/(2*dx)* (T[ix+1] - T[ix-1])           # temperatura futura (

            Tp[nx-1] = Tm[nx-1] - u*dt/(2*dx)* (T[0] - T[nx-2])           # fronteira cíclica
            Tp[0] = Tm[0] - u*dt/(2*dx)* (T[1] - T[nx-1])                 # fronteira cíclica
        T[:]=T[:]+filtro*(Tm+Tp-2*T)
        Tm[:]=T[:]
        T[:]=Tp[:]

    plt.subplot(3,1,isp)
    plt.plot(x,Ti,'b', x,T,'r')
    plt.xlabel('x')
    plt.ylabel('T')
    plt.title('Leapfrog, filtro='+str(filtro))
    plt.grid()
    isp+=1
plt.tight_layout()

```