lf12

November 20, 2018

*b***1**

1 Exercise set 12

Exercise 1:

a) We search for an approximation of the form

$$f'(x) = af(x) + bf(x-h) + cf(x-2h) + e(x;h)$$

where $e(x;h)$ is the error term. Thus we have to find the constants a, b and c.

Strategy: Make a Taylor expansion of the error around x, and zero out as many terms as possible. This is (the function f and its derivatives are evaluated in x if nothing else is mentioned):

$$\begin{aligned} e(x;h) &= f'(x) - af(x) - bf(x-h) - cf(x-2h) \\ &= f' - af \\ &\quad - b\big(f - hf' + \frac{h^2}{2}f'' + \frac{h^3}{6}f''' + \dots\big) \\ &\quad - c\big(f - 2hf' + \frac{(2h)^2}{2}f'' - \frac{(2h)^3}{6}f''' + \dots\big) \\ &= (-a-b-c)f + (1 + hb + 2hc)f' + \frac{h^2}{2}(-b-4c)f'' + \frac{h^3}{6}(b+8c)f''' + \dots \end{aligned}$$

There are 3 parameters to find, so try to set the first three terms of the expansion to zero, that is

$$\begin{aligned} -a - b - c &= 0 \\ -b - 2c &= \frac{1}{h} \\ -b - 4c &= 0 \end{aligned}$$

with the solution $a = 3/(2h)$, $b = -2/h$ and $c = 1/(2h)$. We get

$$f'(x) = \frac{3f(x) - 4f(x-h) + f(x-2h)}{2h} + e(x;h)$$

where the error term $e(x,h)$ is

$$e(x;h) = \frac{h^2}{3} f'''(x) + \cdots$$

and the approximation is of order 2 in h.

b) Replace $y'(x_{n+1})$ with the difference approximation found in point **a)**. Thus

$$\frac{3y(x_{n+1}) - 4y(x_n) + y(x_{n-1})}{2h} \approx y'(x_{n+1}) = f(x_{n+1}, y(x_{n+1})).$$

By replacing $y(x_n)$ with the approximation y_n we get the difference equation

$$\frac{3y_{n+1} - 4y_n + y_{n-1}}{2h} \approx f(x_{n+1}, y_{n+1}),$$

or

$$y_{n+1} = \frac{4}{3}y_n - \frac{1}{3}y_{n-1} + \frac{2}{3}hf(x_{n+1}, y_{n+1}).$$

Comment: This is the second order backward differentiation formula (BDF). BDF-methods of order k can be constructed similarly, by using the gridpoints $x_{n+1}, x_n, \ldots, x_{n+1-k}$ for making an approximation of the derivative $y'(x_{n+1}, y_{n+1})$. The BDF-methods up to order 6 are $A(0)$-stable, and they are among the most popular methods for solving stiff ordinary differential equations. MATLABs ODE-solver ODE15s as well as at least some stiff ode-solvers in Python (there are many of those) are based on the BDF schemes.

c) For this problem, the method will be

$$y_{n+1} = \frac{4}{3}y_n - \frac{1}{3}y_{n-1} + \frac{2}{3}hx_{n+1}y_{n+1}$$

or

$$y_{n+1} = \frac{4y_n - y_{n-1}}{3 - 2hx_{n+1}}.$$

In our case $h = 0.2$, $y_0 = 0.5$ and $y_1 = y(0.2) = .51010067$ and $x_2 = 2h = 0.4$ so

$$y_2 = .5423953098.$$

Comment: The exact solution of the ODE is

$$y(x) = \frac{1}{2}e^{x^2/2} \qquad \text{so} \qquad y(0.4) = .5416435340.$$

Exercise 2 The Lagrange form of the interpolation polynomial is given by

$$\begin{aligned} p_2(x) = {} & f(x_0 - h)\frac{(x - x_0)(x - x_0 - \theta h)}{(-h)(-(1+\theta)h)} \\ & + f(x_0)\frac{(x - x_0 + h)(x - x_0 - \theta h)}{h(-\theta h)} \\ & + f(x_0 + \theta h)\frac{(x - x_0 + h)(x - x_0)}{h(1+\theta)(\theta h)} \end{aligned}$$

We will use $f'(x_0) \approx p_2'(x_0)$ and $f''(x_0) \approx p_2(x_0)$. The derivatives of $p_2(x)$ are given by:

$$p_2'(x) = f(x_0-h)\frac{2x-2x_0-\theta h}{(1+\theta)h^2} + f(x_0)\frac{2x-2x_0+(1-\theta)h}{-\theta h^2} + f(x_0+\theta h)\frac{2x-2x_0+h}{\theta(1+\theta)h^2},$$
$$p_2''(x) = f(x_0-h)\frac{2}{(1+\theta)h^2} + f(x_0)\frac{2}{-\theta h^2} + f(x_0+\theta h)\frac{2}{\theta(1+\theta)h^2}.$$

The approximations becomes:

$$f'(x_0) \approx p_2'(x_0) = \frac{1}{h}\left(-\frac{\theta}{(1+\theta)}f(x_0-h) - \frac{1-\theta}{\theta}f(x_0) + \frac{1}{\theta(1+\theta)}f(x_0+\theta h)\right),$$
$$f''(x_0) \approx p_2''(x_0) = \frac{2}{h^2}\left(\frac{1}{1+\theta}f(x_0-h) - \frac{1}{\theta}f(x_0) + \frac{1}{\theta(1+\theta)}f(x_0+\theta h)\right).$$

Reality check: If $\theta = 1$ these are the central difference approximations.
To check the order, make a Taylor expansion of the errors, and the results are

$$f'(x_0) - p_2'(x_0) = -\frac{h^2}{6}f'''(x_0) + \cdots$$
$$f''(x_0) - p_2''(x_0) = \frac{h}{3}(1-\theta)f'''(x_0) - \frac{h^2}{12}(\theta^2-\theta+1)f^{(4)}(x_0) + \cdots$$

So $p_2'(x_0)$ is an approximation to $f'(x_0)$ of order 2, and $p_2''(x_0)$ is an appoximation to $f''(x_0)$ of order 1 if $\theta \neq 1$, but of order 2 if $\theta = 1$.

Comment: Using interpolation polynomials to construct difference formulas for approximations to the derivatives is an alternative to the technique used in Problem 1, as well as in the notes from the lectures. However, you should not use derivatives of interpolation polynomials for points which are not interpolation points. Such polynomials tends to oscillate somewhat, which can make the $p_n'(x)$ to a quite unreliable approximation for $f'(x)$ for a general x.

Exercise 3

a) For all inner points $i = 1, \ldots, N-1$ the difference formula is:

$$\frac{U_{i+1} - 2U_i + U_{i-1}}{\Delta x^2} - 2\frac{U_{i+1} - U_{i-1}}{2\Delta x} + U_i = 0$$

which after a bit reorganizing becomes

$$(1+\Delta x)U_{i-1} - (2-\Delta x^2)U_i + (1-\Delta x)U_{i+1} = 0 \qquad i = 1, 2, \ldots, N-1.$$

For the left boundary, $x = 0$ we have $U_0 = u_0 = 2$.

On the right boundary ($i = N$) we use the idea of a false boundary: Extend the solution to $x_{N+1} = 1 + \Delta x$, we then have two equations for the boundary: the difference formula above with $i = N$ and a central difference for $u_x(1) = 0$:

$$\frac{U_{N+1} - U_{N-1}}{2\Delta x} = 0 \qquad \Rightarrow \qquad U_{N+1} = U_{N-1}.$$

Inserting this into the scheme for $i = N$ results in

$$2U_{N-1} - (2-\Delta x^2)U_N = 0.$$

So altogether, using $x_1 = \Delta x$ and $x_N = 1$ we get the system:

$$\begin{aligned} U_0 &= 2 \\ (1+\Delta x)U_{i-1} - (2-\Delta x^2)U_i + (1-\Delta x)U_{i+1} &= 0 \qquad i = 1, 2, \ldots, M-1, \\ 2U_{N-1} - (2-\Delta x^2)U_N &= 0. \end{aligned}$$

b) Here $\Delta x = 0.5$, so the equations are after inserting $U_0 = 2$ into the first equation:

$$\begin{aligned} -1.75\,U_1 + 0.5\,U_2 &= -3 \\ 2\,U_1 - 1.75\,U_2 &= 0 \end{aligned}$$

with solutions $U_1 = 2.54545$, $U_2 = 2.90909$. For comparision, $u(0.5) = 2.47308$ and $u(1) = 2.71828$.

c)

In [12]:
```
%matplotlib inline

from numpy import *
from scipy.sparse import diags                # Greate diagonal matrices
from scipy.linalg import solve                # Solve linear systems
from matplotlib.pyplot import *
from mpl_toolkits.mplot3d import Axes3D  # For 3-d plot
from matplotlib import cm
newparams = {'figure.figsize': (8.0, 4.0), 'axes.grid': True,
             'lines.markersize': 8, 'lines.linewidth': 2,
             'font.size': 14}
rcParams.update(newparams)
from numpy import *
```

In [13]:
```
def tridiag(v, d, w, N):
    # Help function
    # Returns a tridiagonal matrix A=tridiag(v, d, w) of dimension N x N.
    e = ones(N)         # array [1,1,...,1] of length N
    A = v*diag(e[1:],-1)+d*diag(e)+w*diag(e[1:],1)
    return A
```

In [14]:
```
# Define the equation
# u'' + p*u' + q*u = r(x) on the interval [a,b]
# Boundary condition: u(a)=ua and u(b)=ub

p = -2
q = 1
def r(x):
    return 0*x

a, b = 0, 1
ua, ub = 2, 0
```

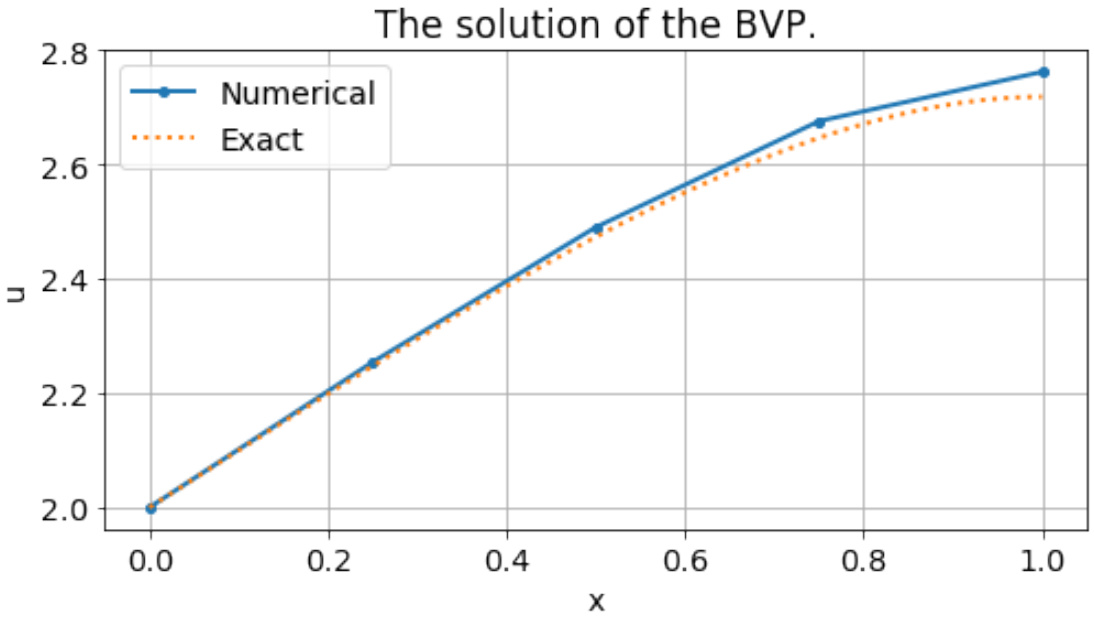
```
        # The exact solution (if known)
        def u_eksakt(x):
            return (2-x)*exp(x)
        # -- end of modified --

        # Set up the discrete system
        N = 4                       # Number of intervals

        # Start the discretization
        h = (b-a)/N                 # Stepsize
        x = linspace(a, b, N+1)     # The gridpoints x_0=a, x_1=a+h, .... , x_N=b

        # Make a draft of the A-matrix (first and last row have to be adjusted)
        v = 1-0.5*h*p               # Subdiagonal
        d = -2+h**2*q               # Diagonal
        w = 1+0.5*h*p               # Superdiagonal
        A = tridiag(v, d, w, N+1)

        # Make a draft of the b-vector
        b = h**2*r(x)

        # Modify the first equation (left boundary)
        A[0,0] = 1
        A[0,1] = 0
        b[0] = ua

        # Modify the last equation (right boundary)
        A[N,N-1] = 2          # Point d)
        # A[N,N-1] = 1            # Point e)
        # A[N,N] = -1             # Point e)

        U = solve(A, b)      #  Solve the equation
```

In [15]:
```
        # Print the matrix A, the right hand side b the numerical and exact solution
        print('A =\n', A)
        print('\nb =\n ', b)
        print('\nU =\n ', U)
        print('\nu(x)=\n', u_eksakt(x))
```

```
A =
 [[ 1.      0.      0.      0.      0.    ]
 [ 1.25   -1.9375  0.75    0.      0.    ]
 [ 0.      1.25   -1.9375  0.75    0.    ]
 [ 0.      0.      1.25   -1.9375  0.75  ]
 [ 0.      0.      0.      2.     -1.9375]]

b =
```

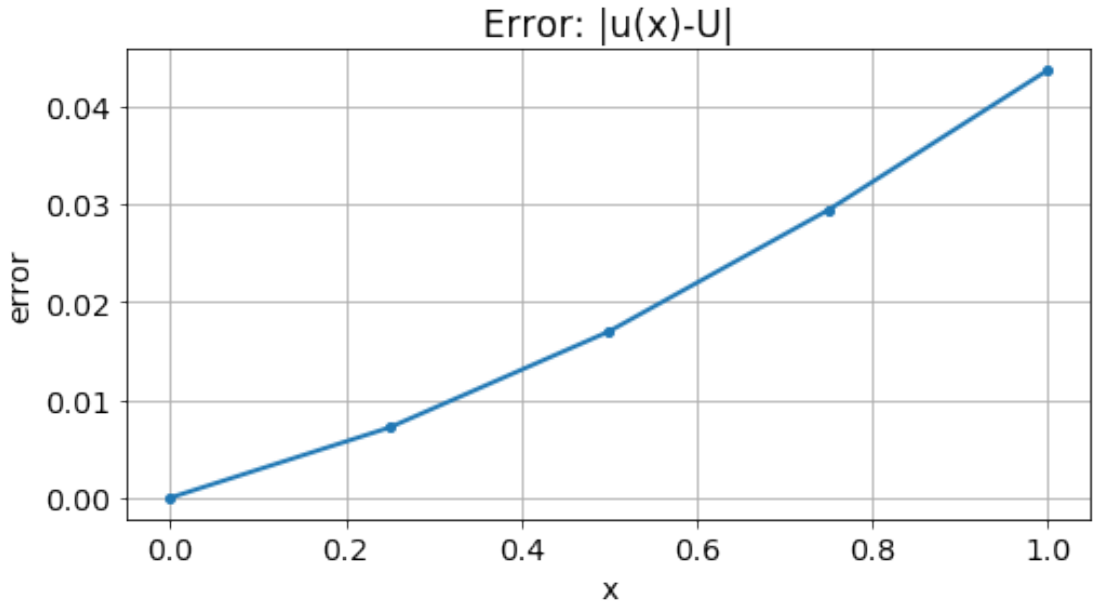
```
  [2. 0. 0. 0. 0.]

U =
  [2.         2.25421708 2.4900608  2.67562858 2.76193918]

u(x)=
 [2.         2.24704448 2.47308191 2.64625002 2.71828183]
```

In [16]:
```
# Plot the solution of the BVP
xe = linspace(0,1,101)
plot(x,U,'.-')
plot(xe, u_eksakt(xe),':')
xlabel('x')
ylabel('u')
legend(['Numerical','Exact'])
title('The solution of the BVP.');
```



In [17]:
```
# Plot the error |u(x)-U| in the gridpoints
error = abs(u_eksakt(x)-U)
plot(x, error,'.-')
xlabel('x')
ylabel('error')
title('Error: |u(x)-U|');
print('Max error = {:.3e}'.format(max(abs(error))))
```

```
Max error = 4.366e-02
```



We get the following result

N	Δx	$e(\Delta x)$	$e(2\Delta x)/e(\Delta x)$
10	0.1	$6.825 \cdot 10^{-3}$	
20	0.25	$1.701 \cdot 10^{-3}$	4.013
40	0.125	$4.248 \cdot 10^{-4}$	4.003

Since the error is reduced by a factor of approximately $1/4$ every time ΔX is reduced by a factor of $1/2$, the approximations seems to be of order 2.

d) If we use a simple backward difference to approximate the boundary condition $u_x(1) = 0$ the last equation will be replaced by:

$$U_{N-1} - U_N = 0.$$

The corresponding table becomes:

N	Δx	$e(\Delta x)$	$e(2\Delta x)/e(\Delta x)$
10	0.1	$6.644 \cdot 10^{-2}$	
20	0.25	$3.358 \cdot 10^{-2}$	1.987
40	0.125	$1.689 \cdot 10^{-2}$	1.979

So the order is reduced to 2 in this case. Using a false boundary and a central difference for the right boundary really pays off.

e) If we still use a backward difference to approximate the derivative, we get

$$U_{N-1} - U_N + U_N = 0.$$

If we use a central difference as in (a) we get

$$\frac{U_{N+1} - U_{N-1}}{2\Delta x} + U_N = 0 \qquad \Rightarrow \qquad U_{N+1} = U_{N-1} - \Delta x U_N.$$

Inserting this into the scheme for $i = N$ results in

$$2U_{N-1} + (2\Delta x^2 - \Delta x - 2)U_N = 0.$$

Excercise 4

a) Let $\Delta x = 1/M$, $x_i = ih$ for $i = 0, 1, \ldots, M$ and $U_i(t) \approx u(x_i, t)$. Notice that the right hand side of this equation is exactly the same as the left hand side of the equation in Exercise 3, so the same discretization in the x-direction applies. We have that $U_0(t) = 2$, this is now included in the first equation. The right hand side value $U_M(t) \approx u(1, t)$ is still unknown, and we use central differences and the false boundary as in Exercise 3 to get an equation for the right boundary.

The semidiscrete system becomes:

$$\begin{aligned}
U_1' &= \frac{1}{\Delta x^2}\Big((1+\Delta x)\cdot 2 - (2-\Delta x^2)U_i + (1-\Delta x)U_{i+1}\Big) \\
U_i' &= \frac{1}{\Delta x^2}\Big((1+\Delta x)U_{i-1} - (2-\Delta x^2)U_i + (1-\Delta x)U_{i+1}\Big) \qquad i = 1, 2, \ldots, M-1, \\
U_M' &= \frac{1}{\Delta x^2}\Big(2U_{M-1} - (2-\Delta x^2)U_M\Big)
\end{aligned}$$

with initial values $U_i(0) = 2(1 - x_i)^2$.

Notice that this system can be written as

$$U' = \frac{1}{\Delta x^2} AU + \frac{1}{\Delta x^2} b$$

where $U(t) = [U_1(t), \ldots, U_M(t)]^T$, and $b = [(1+\Delta x)\cdot 2, 0, 0, \ldots, 0]^T$.

For $M = 2$ this becomes:

$$\begin{aligned}
U_1' &= -7U_1 + 2U_2 + 12, \quad & U_1(0) &= 0.5, \\
U_2' &= 8U_1 - 7U_2, & U_2(0) &= 0
\end{aligned}$$

b) The trapezoidal rule for the system above is given by:

$$U^{n+1} = U^n + \frac{r}{2} A(U^n + U^{n+1}) + rb$$

with $r = \Delta t/\Delta x^2$.

For $M = 2$, with $\Delta x = 0.5$ and $\Delta t = 0.5$, the factor $r = 2$ and the numerical scheme becomes:

$$\begin{aligned}
U_1^{n+1} &= U_1^n - 1.75U_1^n + 0.5U_2^n - 1.75U_1^{n+1} + 0.5U_2^{n+1} + 6 \\
U_2^{n+1} &= U_2^n + 2U_1^n - 1.75U_2^n + 2U_1^{n+1} - 1.75U_2^{n+1}
\end{aligned}$$

Which for $n = 0$ and the initial values from point a) becomes

$$2.75U_1^1 - 0.5U_2^1 = 5.625$$
$$-2U_1^n + 2.75U_2^1 = 1$$

with solution $U_1^1 = 2.43333333$ and $U_1^2 = 2.13333333$.

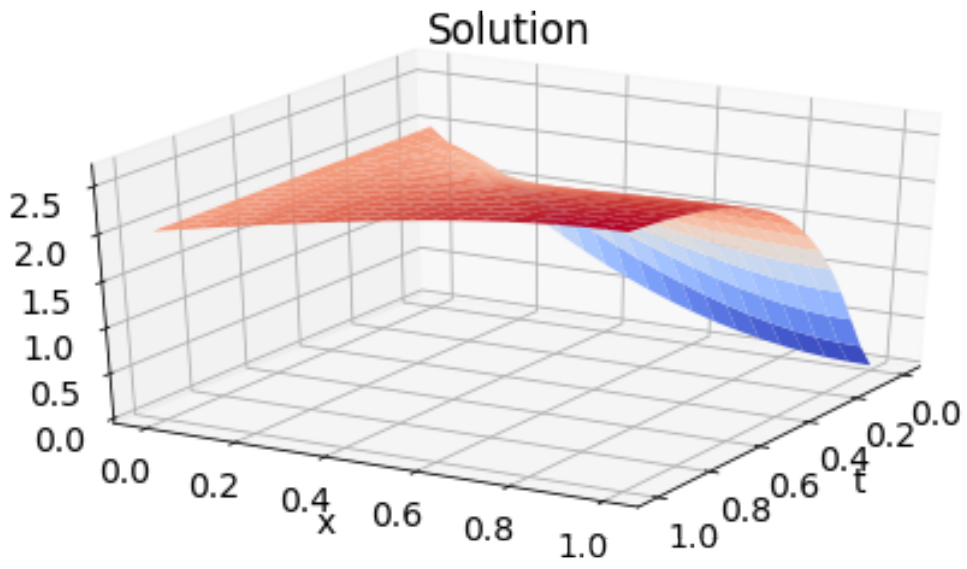
c)

In [18]:
```
# Apply Crank-Nicolson on
# the equation u_t=u_{xx}+p*u_x+q*u

# Define the problem
p = -2
q = 1

def f(x):
    return 2*(1-x)**2

# Boundary values
def g0(t):
    return 2

# Choose method
# method = 'iEuler'
method = 'CrankNicolson'


M = 20                    # Number of intervals in the x-direction
Dx = 1/M
x = linspace(0,1,M+1)    # Gridpoints in the x-direction

tend = 1.0
N = 20                   #  Number of intervals in the t-direction
Dt = tend/N
t = linspace(0,tend,N+1) # Gridpoints in the t-direction

# Array to store the solution
U = zeros((M+1,N+1))
U[:,0] = f(x)               # Initial condition U_{i,0} = f(x_i)

# Set up the matrix:
v = 1-0.5*Dx*p              # Subdiagonal
d = -2+Dx**2*q              # Diagonal
w = 1+0.5*Dx*p              # Superdiagonal
A = tridiag(v, d, w, M)

# Modify the last equation
A[M-1,M-2] = 2
```

```
        r = Dt/Dx**2
        print('r = ', r)
        if method is 'iEuler':
            K = eye(M) - r*A
        elif method is 'CrankNicolson':
            K = eye(M) - 0.5*r*A

        Utmp = U[1:,0]          # Temporary solution for the unknown gridpoints.

        # Main loop over the time steps.
        for n in range(N):
            # Set up the right hand side of the equation KU=b:
            if method is 'iEuler':
                b = copy(Utmp)                  # NB! Copy the array
                b[0] = b[0] + r*v*g0(t[n+1])
            elif method is 'CrankNicolson':
                b = dot(eye(M)+0.5*r*A, Utmp)
                b[0] = b[0] + 0.5*r*v*(g0(t[n])+g0(t[n+1]))

            Utmp = solve(K,b)         # Solve the equation K*Utmp = b

            U[1:,n+1] = Utmp          # Store the solution
            U[0, n+1] = g0(t[n+1])    # Include the boundary
```

```
r =  19.999999999999996
```

In [19]:
```
        # Plot the solution of the heat equation
        fig = figure()
        ax = fig.gca(projection='3d')
        T, X = meshgrid(t,x)
        # ax.plot_wireframe(T, X, U)
        ax.plot_surface(T, X, U, cmap=cm.coolwarm)
        ax.view_init(azim=30)              # Rotate the figure
        xlabel('t')
        ylabel('x')
        title('Solution');
```


Solution
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
x
0.6
0.8
1.0
0.0
0.2
0.4
t
0.6
0.8
1.0