SCHEDULE TODAY

- reminder deadlines
- recap PDE
- stability explicit method PDE (diffusion)
- implicit method PDE
- Neumann boundary condition

REMINDER DEADLINES

- 13/5/2026: in-class test

PDE DIFFUSION EQUATION

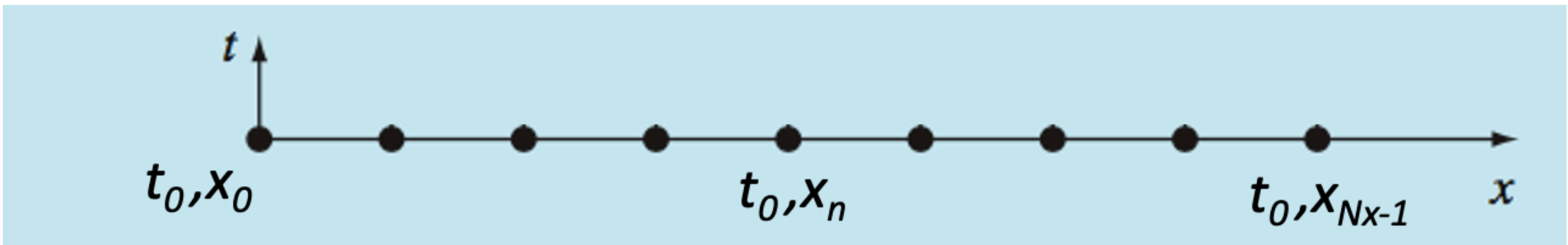
RECAP

The spatio-temporal diffusion equation, in 1 spatial dimension equals

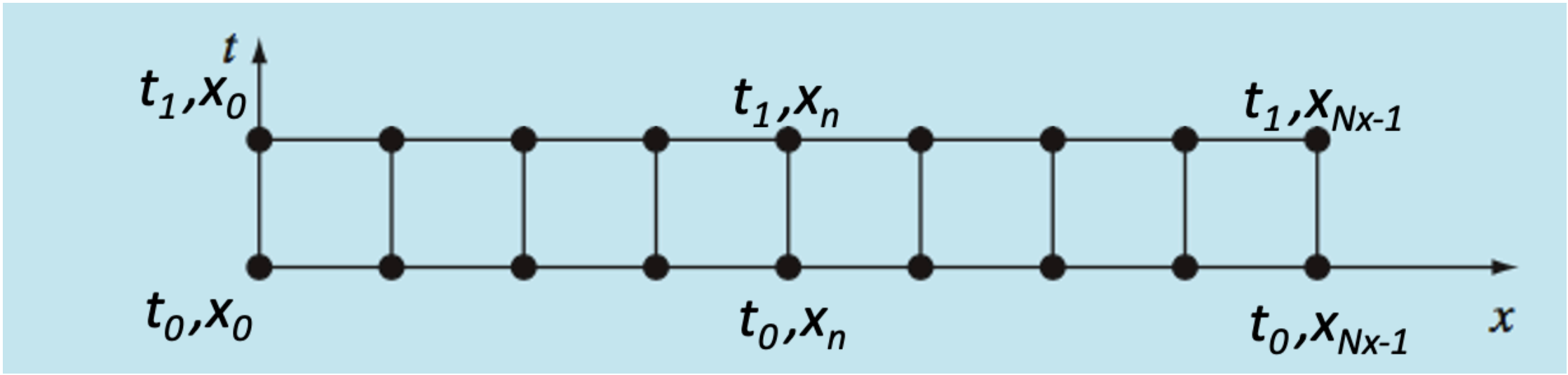
$$\frac{\partial u(t,x)}{\partial t} = \alpha \frac{\partial^2 u(t,x)}{\partial x^2}$$

To solve this equation we first need boundary ($u(t, x_L) = u_L$ and $u(t, x_R) = u_R$, where u_L and u_R are known constants) and initial ($u(t_0, x) = f_0(x)$, where $f_0(x)$ is a known function) conditions.

The equation will be solved on a grid. We start with the boundary and initial conditions at $t = t_0$. For simplicity we take $t_0 = 0$. We need to specify the function at the following grid points, where $x = x_L + n\Delta x$ and $\Delta x = (x_R - x_L)/(N_x - 1)$.



Once we used the initial condition to set the values of $u(0, x)$ at these grid points, we can determine then at the next time step, $t = \Delta t$, so $u(\Delta t, x)$.

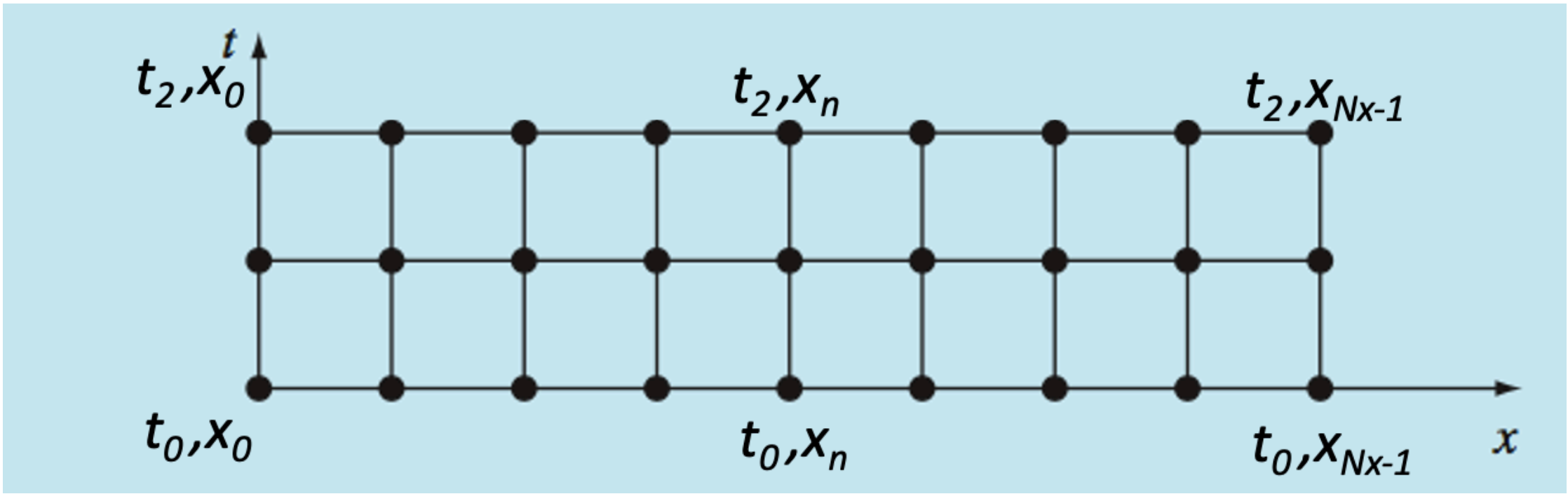


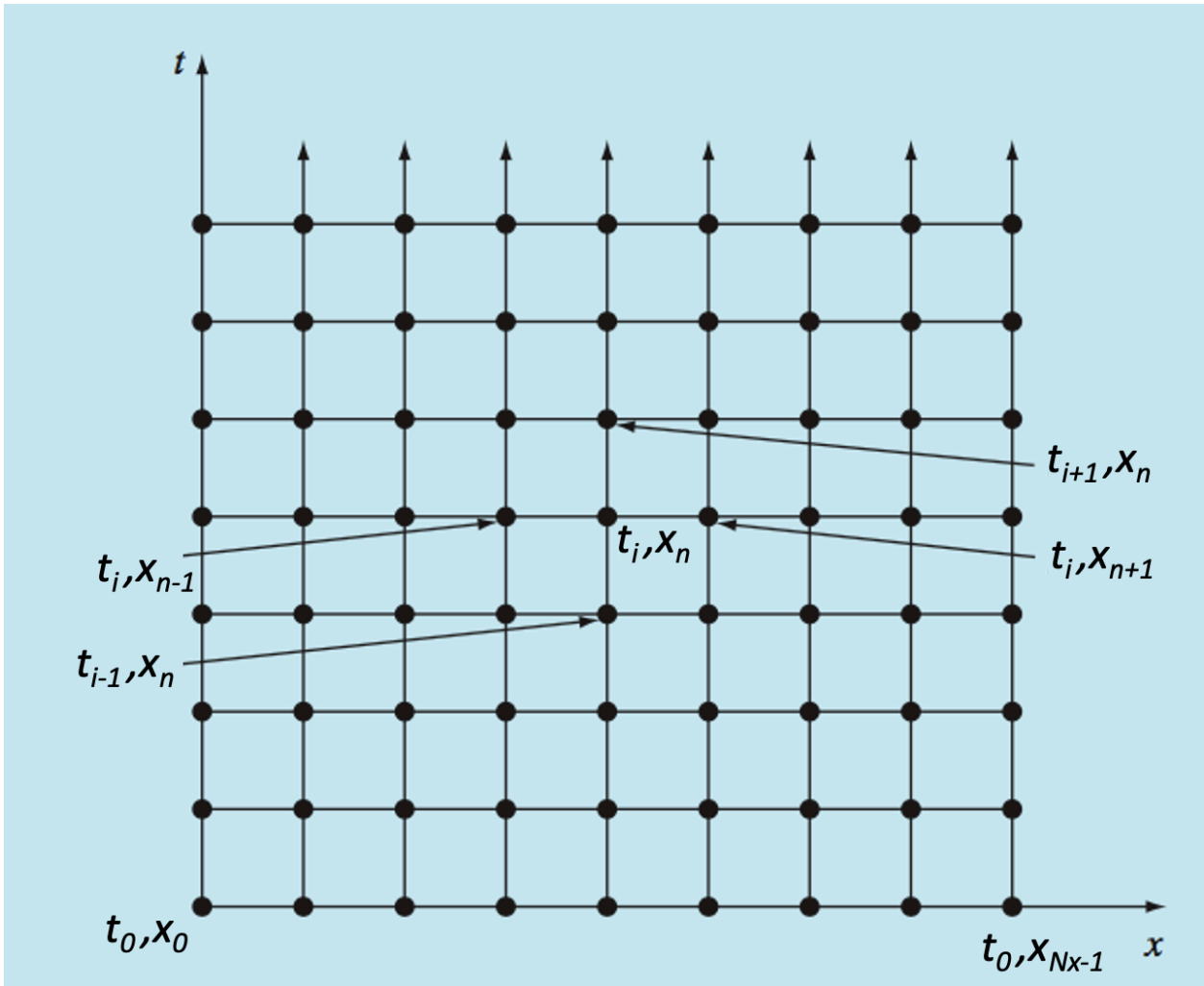
This is given by

$$
\begin{aligned}
u(t_0 + \Delta t, x) &= u(t_0, x) + \Delta t \alpha \frac{\partial^2 u(t_0, x)}{\partial x^2} \\
&\approx u(t_0, x) + \Delta t \frac{u(t_0, x + \Delta x) + u(t_0, x - \Delta x) - 2u(t_0, x)}{\Delta x^2}
\end{aligned}
$$

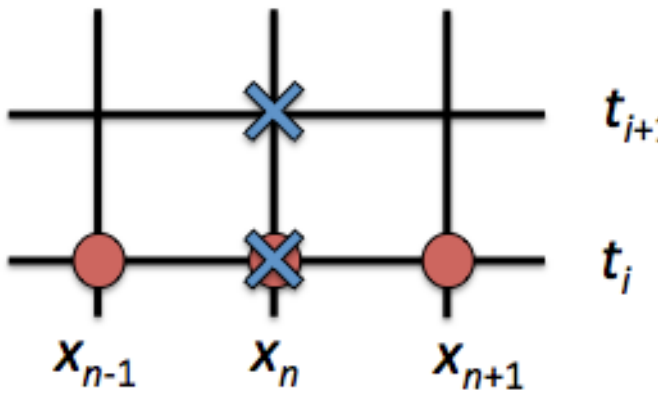
where for simplicity $\alpha = 1$ is used.

We repeat the algorithm




t
t_{i+1}, x_n
t_i, x_{n-1}
t_i, x_n
t_i, x_{n+1}
t_{i-1}, x_n
t_0, x_0
t_0, x_{Nx-1}
x

For the explicit Euler method, discussed previously, we have the following stencil

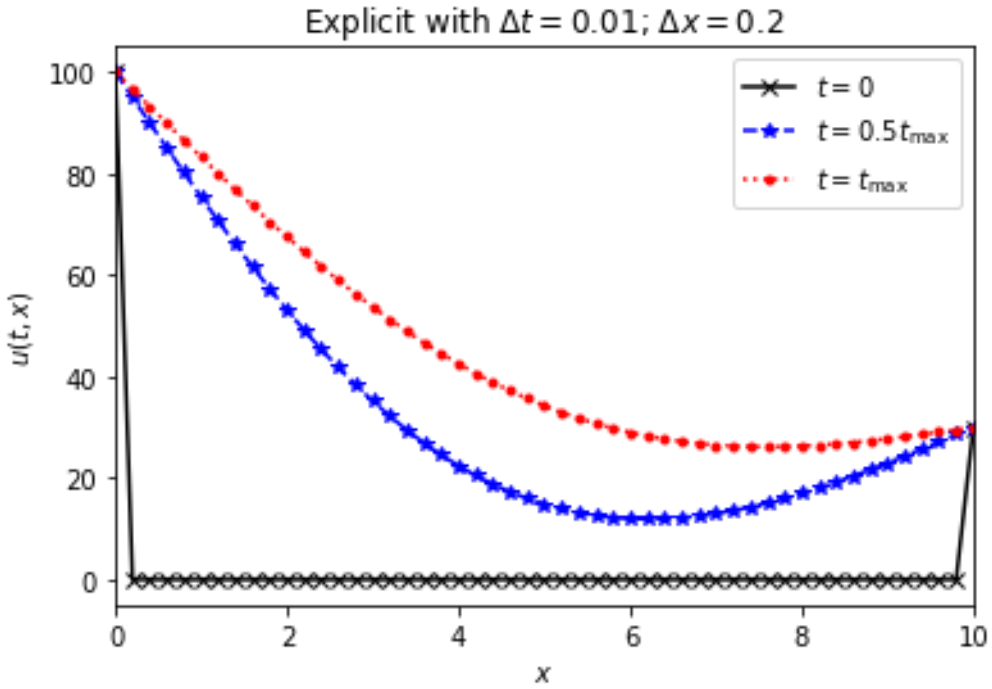


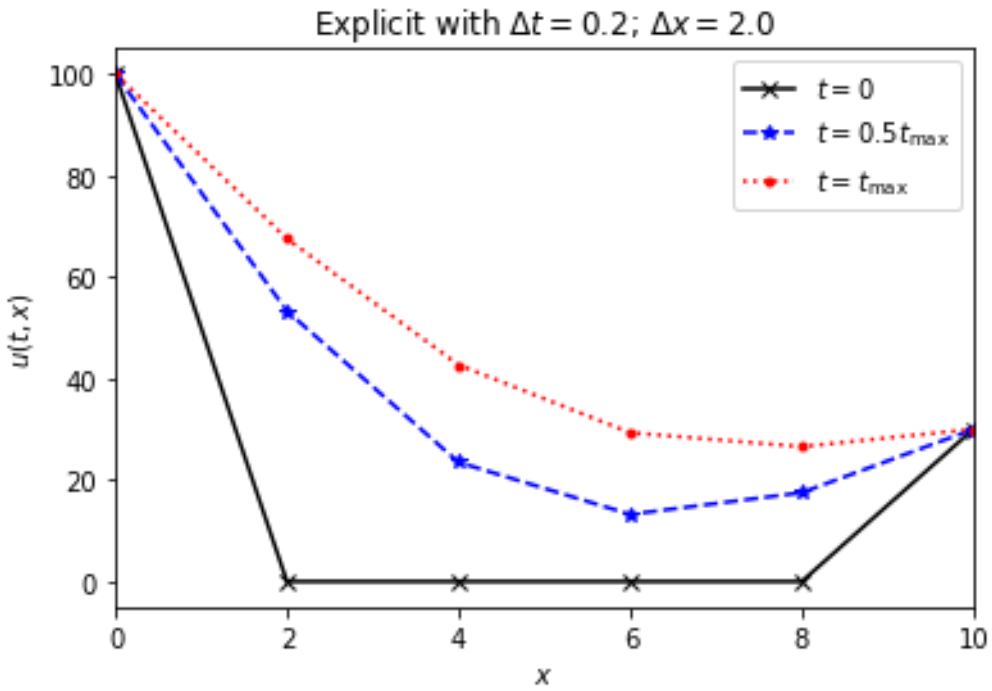
where red dots are involved in spatial differences and blue crosses in temporal differences.

SKETCH SOLUTION IN PYTHON

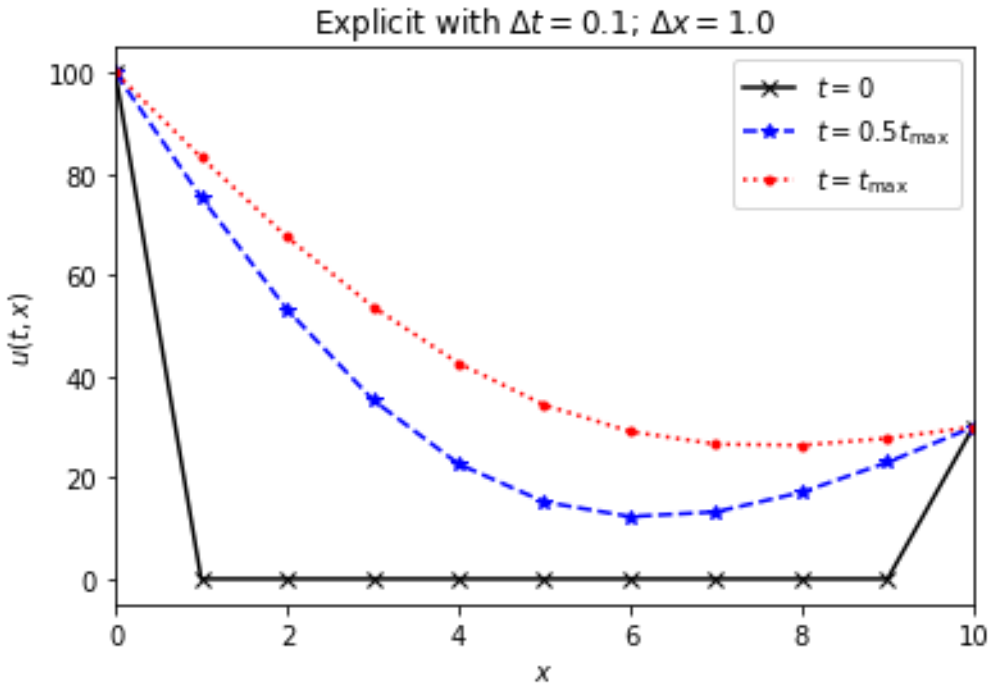
```
# Solving PDE (1D diffusion equation) using explicit scheme
import numpy as np
L=10.0
xmax=L
dx=0.2
dt=0.01
tmax=12.0
a=0.735

# Number of spatial points
nx=int(round(xmax/dx))+1 # 51

# Number of time points
nt=int(round(tmax/dt))+1 # 1201


u_tx=np.zeros((nt, nx))

u0=0
# initial condition: u(t=0,x)=u0
for ix in range(nx):
    u_tx[0,ix]=u0

# boundary conditions
u_L=100.0
u_R=30.0
for it in range(nt):
    u_tx[it,0]=u_L
    u_tx[it,nx-1]=u_R
```

See `.py` file on blackboard

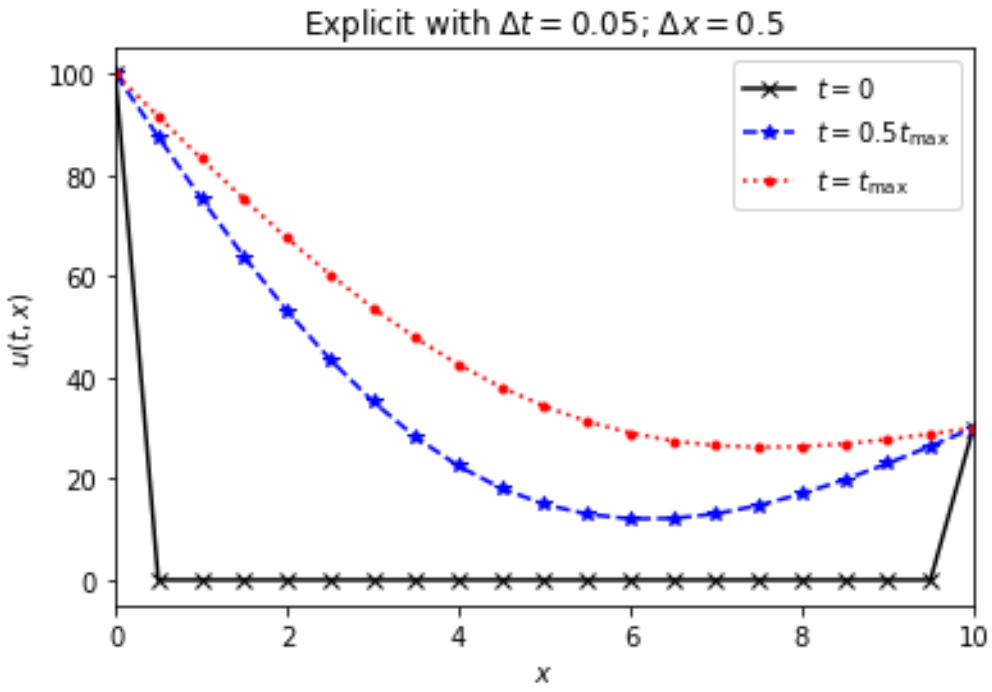

Explicit with $\Delta t = 0.01$; $\Delta x = 0.2$
$t = 0$
$t = 0.5t_{max}$
$t = t_{max}$
$u(t, x)$
x
0
20
40
60
80
100
0
2
4
6
8
10


Explicit with $\Delta t = 0.2$; $\Delta x = 2.0$
$t = 0$
$t = 0.5t_{max}$
$t = t_{max}$
$u(t, x)$
x
100
80
60
40
20
0
0
2
4
6
8
10

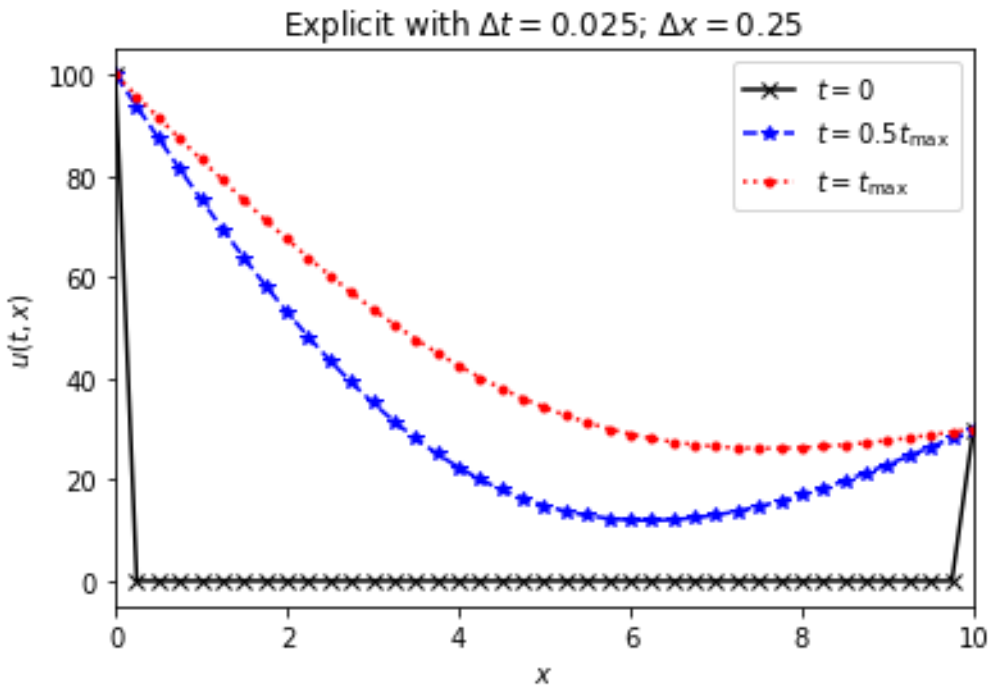
Decreasing integration steps by factor 2



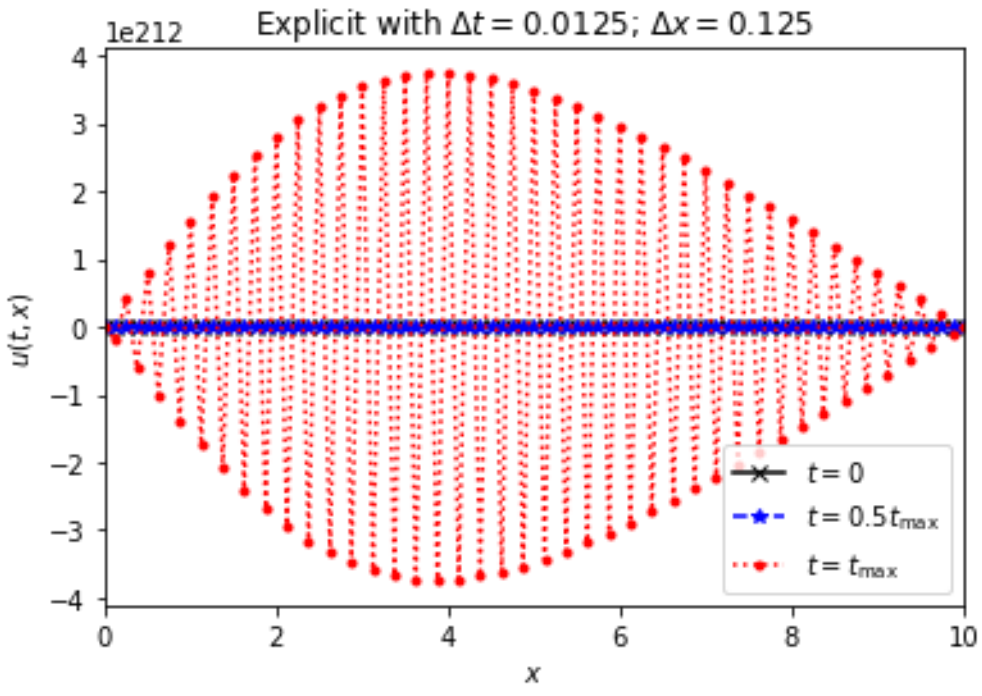
Decreasing integration steps by factor 2



Decreasing integration steps by factor 2



Decreasing integration steps by factor 2



So decreasing the integration steps in time and space leads to an unstable method?

STABILITY EXPLICIT METHOD DIFFUSION EQUATION

One can proof that the explicit method for the diffusion equation is both stable and convergent if

$$\Delta t \leq \frac{\Delta x^2}{2\alpha}$$

or, equivalently,

$$\Delta x \geq \sqrt{\frac{2\Delta t}{\alpha}}$$

This can be seen from the update scheme for the explicit method:

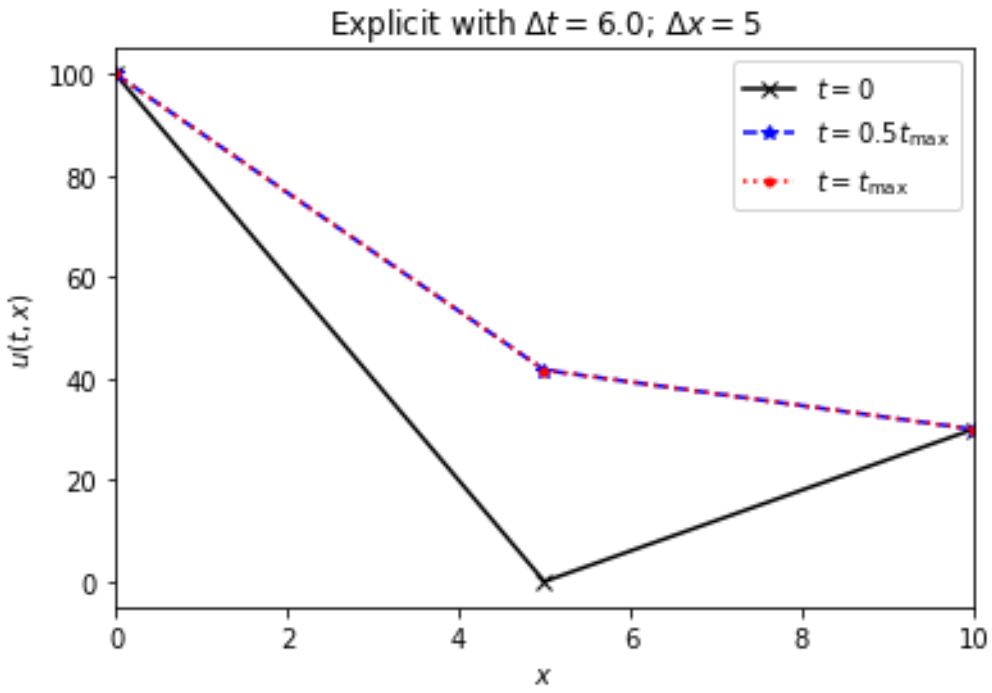
$$u_{i+1,n} = \left(1 - \frac{2\alpha\Delta t}{\Delta x^2}\right) u_{i,n} + \frac{\alpha\Delta t}{\Delta x^2}\left(u_{i,n+1} + u_{i,n-1}\right)$$

The absolute value of the prefactor to $u_{i,n}$ for a stable method is then smaller than or equal to 1. Details of the proof are in, e.g., *Hoffman* (2001).

Hence, if Δx is decreased by a factor 2, then Δt has to decrease by a factor of 4 (a factor of 2 is not enough!). This will then eventually converge to the exact solution.

EXAMPLE

This does mean that even for *very large spatial integration steps* the explicit method can be stable (although not very accurate).


Explicit with $\Delta t = 6.0$; $\Delta x = 5$
$t = 0$
$t = 0.5t_{max}$
$t = t_{max}$
$u(t, x)$
x
100
80
60
40
20
0
0
2
4
6
8
10

IMPLICIT SCHEME

In the explicit scheme we had

$$\frac{\partial u(t,x)}{\partial t} \approx \frac{u_{i+1,n} - u_{i,n}}{\Delta t}$$

We can derive an implicit scheme, by taking the backward difference for the time derivative:

$$\frac{\partial u(t,x)}{\partial t} \approx \frac{u_{i,n} - u_{i-1,n}}{\Delta t}$$

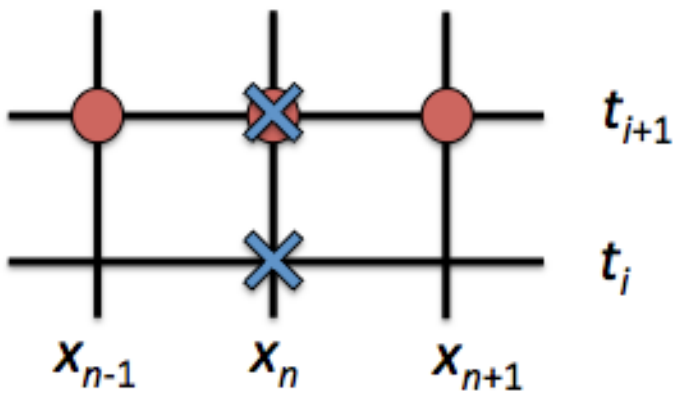
FINITE DIFFERENCE IMPLICIT SCHEME

The resulting finite difference equation for the PDE is then

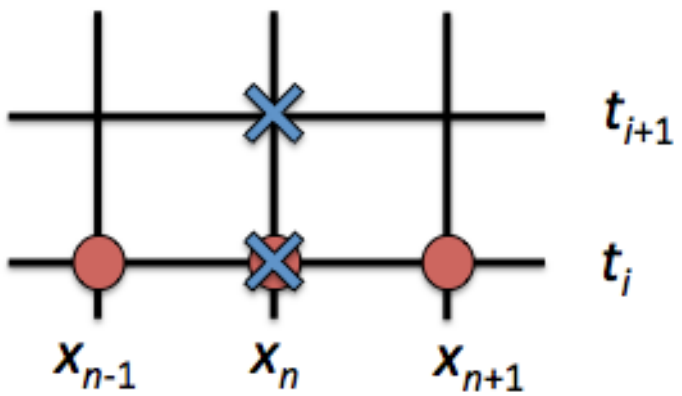
$$\frac{u_{i,n} - u_{i-1,n}}{\Delta t} \approx \frac{u_{i,n+1} - 2u_{i,n} + u_{i,n-1}}{\Delta x^2}$$

or, equivalently

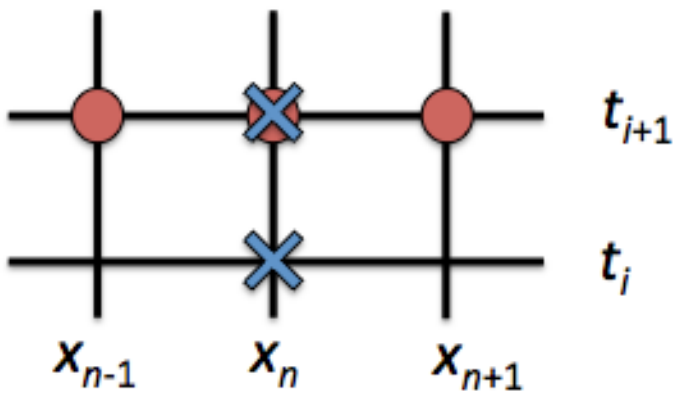
$$\frac{u_{i+1,n} - u_{i,n}}{\Delta t} \approx \frac{u_{i+1,n+1} - 2u_{i+1,n} + u_{i+1,n-1}}{\Delta x^2}$$



COMPARISON STENCILS



Explicit: unknown $u(t_{i+1}, x_n)$ at time t_{i+1} solely determined by information at previous time, t_i.



Implicit: relation between the unknowns $u(t_{i+1}, x_{n-1})$, $u(t_{i+1}, x_n)$, $u(t_{i+1}, x_{n+1})$ and the known $u(t_i, x_n)$

HOW TO SOLVE IMPLICIT?

This will be similar to the 1D boundary value problem.

$$\frac{u_{i+1,n} - u_{i,n}}{\Delta t} \approx \alpha \frac{u_{i+1,n+1} - 2u_{i+1,n} + u_{i+1,n-1}}{\Delta x^2}$$

Note that all terms containing $i + 1$ are unknown. Rewrite this equation by bringing all the unknown u's to the left-hand side and all known u's to the right-hand side:

$$u_{i+1,n}(1 + 2c) - cu_{i+1,n+1} - cu_{i+1,n-1} = u_{i,n}$$

Here a shorthand for the coefficients is introduced, $c = \alpha \Delta t / \Delta x^2$.

We can write such an equation for each spatial grid point at time point $i+1$.

There are two different types: the inner points and the boundary points.

If we have the boundary contions $u_{i,0} = u_L$ and $u_{i,N_x-1} = u_R$, then the system of equations become

$$
\begin{aligned}
u_{i+1,0} &= u_L \\
-cu_{i+1,2} + (1+2c)u_{i+1,1} - cu_{i+1,0} &= u_{i,1} \\
-cu_{i+1,3} + (1+2c)u_{i+1,2} - cu_{i+1,1} &= u_{i,2} \\
&\vdots \\
-cu_{i+1,n+1} + (1+2c)u_{i+1,n} - cu_{i+1,n-1} &= u_{i,n} \\
&\vdots \\
-cu_{i+1,N_x-1} + (1+2c)u_{i+1,N_x-2} - cu_{i+1,N_x-3} &= u_{i,N_x-2} \\
u_{i+1,N_x-1} &= u_R
\end{aligned}
$$

The system of equations can be written as a matrix equation in the unknowns $\mathbf{u}_{i+1}$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ -c & 1+2c & -c & 0 & 0 & 0 & \cdots & 0 \\ 0 & -c & 1+2c & -c & 0 & 0 & \cdots & 0 \\ 0 & 0 & -c & 1+2c & -c & 0 & \cdots & 0 \\ & & & \ddots & \ddots & \ddots & & \\ 0 & 0 & 0 & \ddots & \ddots & \ddots & \vdots & 0 \\ 0 & 0 & 0 & 0 & 0 & -c & 1+2c & -c \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_{i+1,0} \\ u_{i+1,1} \\ u_{i+1,2} \\ u_{i+1,3} \\ \vdots \\ u_{i+1,N_x-2} \\ u_{i+1,N_x-1} \end{pmatrix} = \begin{pmatrix} u_L \\ u_{i,1} \\ u_{i,2} \\ u_{i,3} \\ \vdots \\ u_{i,N_x-2} \\ u_R \end{pmatrix}$$

or

$$\mathsf{A}\mathbf{u}_{i+1} = \mathbf{u}_i \qquad (*)$$

with A an $N_x \times N_x$ matrix and $\mathbf{u}_i$ a vector of N_x elements denoting the solution at time t.

The matrix equation

$$\mathrm{A}\mathbf{u}_{i+1} = \mathbf{u}_i \tag{1}$$

can be solved using Gaussian elimination (or Matrix inversion: $\mathbf{u}_{i+1} = \mathrm{A}^{-1}\mathbf{u}_i$), where $\mathbf{u}_0$ is given by the initial condition for $t = 0$.

We therefore start with $\mathbf{u}_0$.

To get the next time step, $t = \Delta t$,

$$\mathbf{u}_1 = \mathrm{A}^{-1}\mathbf{u}_0$$

The next step, $t = 2\Delta t$ is then

$$\mathbf{u}_2 = \mathrm{A}^{-1}\mathbf{u}_1$$

etc.

Hence the implicit scheme is very similar to solving the one-dimensional boundary value problems discussed by Matt during the first term.

The only difference is that one has to solve the system of equations *for each point in time*.

CODE FOR GAUSSIAN ELIMINATION

A python code for Gaussian elimination is

```
def Gauss_elim(M):
    """
    Solves the matrix equation
    Ax=b
    where the last column of the numpy array M (augmented matrix) contains b,
and the other columns contain A,
    by using Gaussian elimination with in-place triangularization of the matrix
M
    """
    rows=M.shape[0]
    cols=M.shape[1]

    # Triangularization
    for i in range(0, rows-1):
        for j in range(i+1, rows):
            coeff=M[j,i]/(1.0*M[i,i])
            for k in range(i, cols):
                M[j,k]-=M[i,k]*coeff

    # Back substitution
    x=np.zeros(rows)
    for i in range(rows-1, -1, -1):
        x[i]=M[i, cols-1]
        for j in range(i+1, cols-1):
            x[i] -= M[i,j]*x[j]

        x[i]/=M[i,i]*1.0
```

LIBRARY CODE FOR MATRIX INVERSION

To use the numpy library for inverting a matrix do the following:

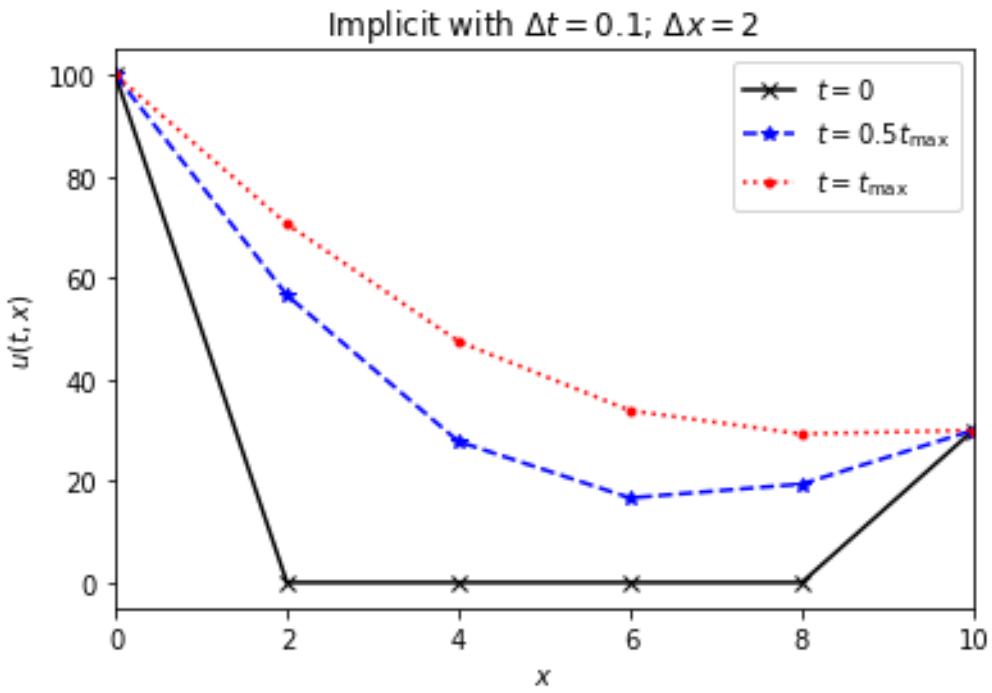
```
import numpy as np

# TODO: set Nx
A=np.zeros((Nx,Nx))
# TODO: fill the 3 diagonals of A with appropriate values

# inverting the matrix A
Ainv=np.linalg.inv(A)

# Matrix multiplication can be achieved using
unext=np.matmul(Ainv, unow)
```

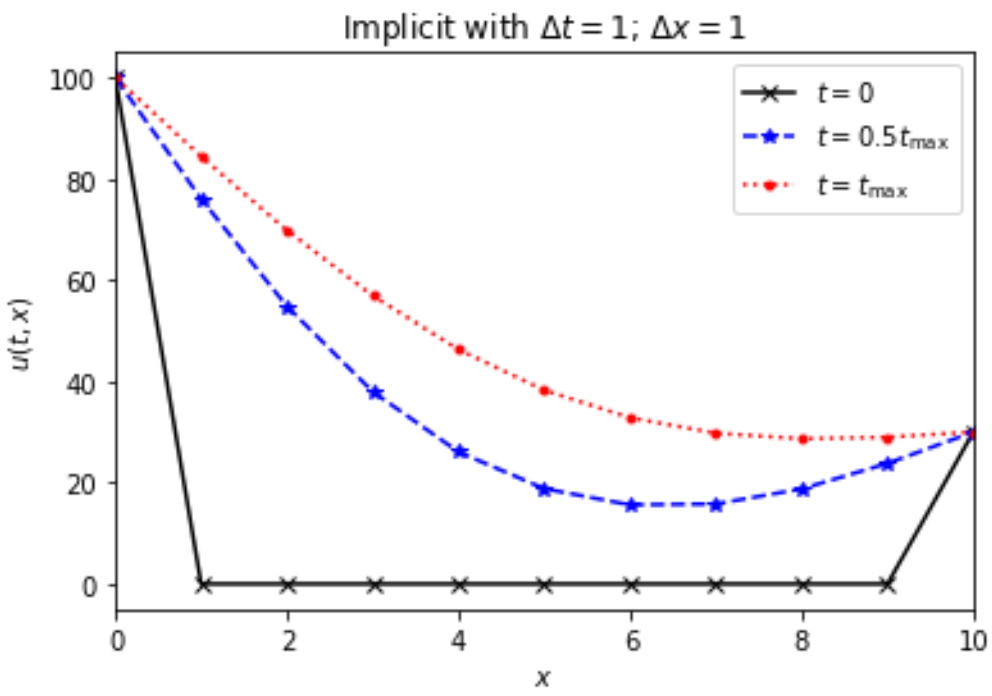
THOMAS ALGORITHM

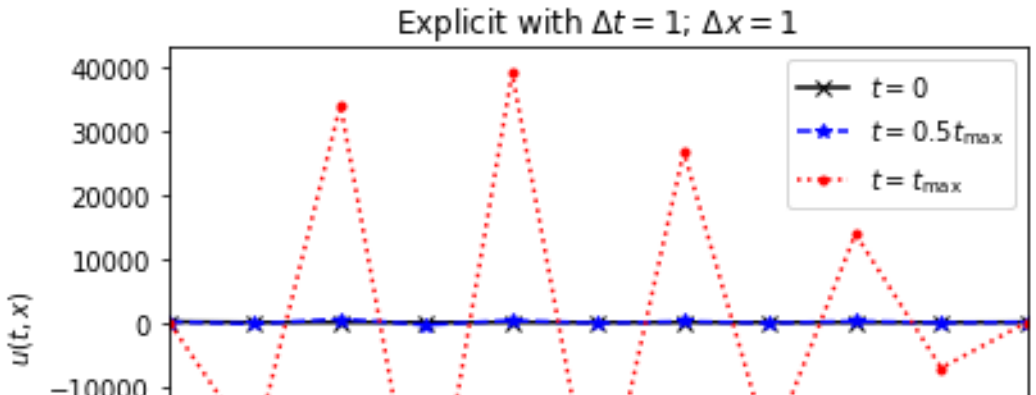
Note: to solve such a system of equations involving a tri-diagonal matrix (matrix with only non-zero elements on the main diagonal and the two off-diagonal ones) an algorithm exist which is much faster than general Gaussian elimination: the **Thomas algorithm**.

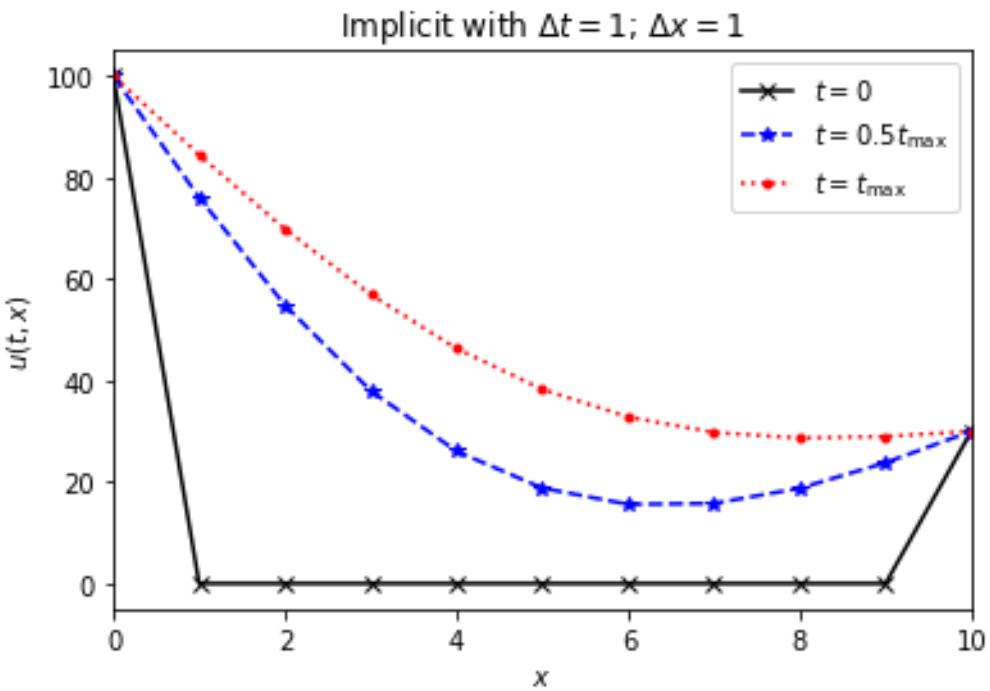
This requires only $O(n)$ operations instead of $O(n^3)$ as with Gaussian elimination, since one can use the fact that there are a lot of zeros in the matrix.

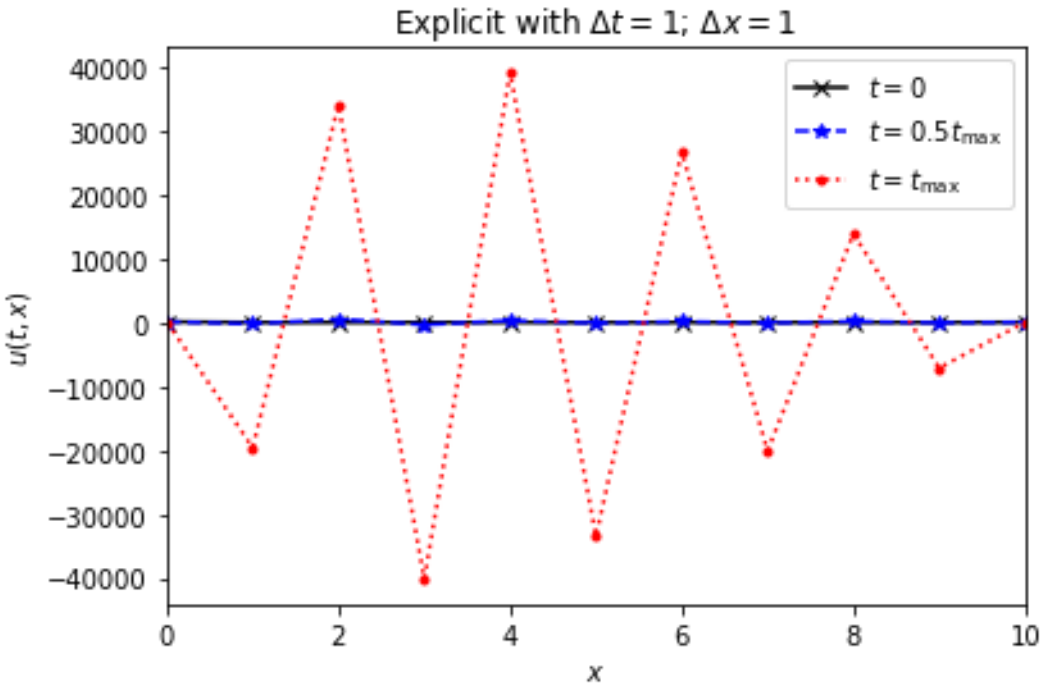
We won't discuss the algorithm. For the interested reader: see, e.g., section 11.1.1 of Chapra et al., or https://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm. It is also part of Python (scipy.linalg.lapack.dgtsv, https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lapack.dgtsv.html).


Implicit with $\Delta t = 0.1$; $\Delta x = 2$
$t = 0$
$t = 0.5t_{max}$
$t = t_{max}$
$u(t, x)$
x
0
20
40
60
80
100
0
2
4
6
8
10


Implicit with $\Delta t = 1$; $\Delta x = 1$
$t = 0$
$t = 0.5t_{max}$
$t = t_{max}$
$u(t, x)$
x
0
20
40
60
80
100
0
2
4
6
8
10


Explicit with $\Delta t = 1$; $\Delta x = 1$
$t = 0$
$t = 0.5t_{max}$
$t = t_{max}$
$u(t, x)$
0
10000
20000
30000
40000


Implicit with $\Delta t = 1$; $\Delta x = 1$
$t = 0$
$t = 0.5t_{max}$
$t = t_{max}$
$u(t, x)$
x
100
80
60
40
20
0
0
2
4
6
8
10


Explicit with $\Delta t = 1$; $\Delta x = 1$
$t = 0$
$t = 0.5t_{max}$
$t = t_{max}$
$u(t, x)$
x
40000
30000
20000
10000
0
−10000
−20000
−30000
−40000
0
2
4
6
8
10

The implicit algorithm is unconditionally stable!

However, one drawback is that its global truncation error is $O(\Delta t + \Delta x^2)$. This means second order in Δx, but only first order in Δt.

NEUMANN BOUNDARY CONDITIONS

Let us return to the ordinary differential equation

$$\frac{d^2}{dx^2}u(x) = 0$$

The associated finite difference equation reads

$$\frac{-2u_n + u_{n-1} + u_{n+1}}{\Delta x^2} = 0$$

Imagine we have the *Neumann boundary condition*

$$\frac{d}{dx}u(x)\Big|_{x=x_L} = d$$

which means that the derivative at the boundary x_L is a constant d.

An associated finite difference scheme would be

$$\frac{u_1 - u_{-1}}{2\Delta x} = d$$

where x_0 corresponds to x_L and hence x_{-1} is outside the boundary and is called an **imaginary point**. This point will be substituted away.

Solving for u_{-1} gives

$$u_{-1} = u_1 - 2\Delta x d$$

Hence the difference equation for $n = 0$,

$$\frac{u_1 - 2u_0 + u_{-1}}{\Delta x^2} \approx 0$$

becomes

$$\frac{2u_1 - 2u_0 - 2\Delta x d}{\Delta x^2} \approx 0$$

This can be rewritten as

$$-2u_0 + 2u_1 = 2\Delta x d$$

If we leave the boundary condition $u_{N-1} = u_R$, the resulting set of equations become

$$\begin{aligned}
-2u_0 + 2u_1 &= 2\Delta x d \\
u_2 - 2u_1 + u_0 &= 0 \\
u_3 - 2u_2 + u_1 &= 0 \\
&\vdots \\
u_{n+1} - 2u_n + u_{n-1} &= 0 \\
&\vdots \\
u_{N-1} - 2u_{N-2} + u_{N-3} &= 0 \\
u_{N-1} &= u_R
\end{aligned}$$

This can be cast into a matrix equation, which can be solved as before.

EXERCISES SESSION 9

1. What is the stability criterion for the explicit Euler method for the diffusion equation (PDE)

2. What is the stability criterion for the implicit Euler method for the diffusion equation (PDE)

3. See Assessment section on Blackboard