# Applied Numerical Methods - Lab 4

Goyens Florentin & Weicker David

6th November 2015

## Partial differential equation of parabolic type

### a) Reformulation of the problem

We apply the change of variable $T = T_0 u$, $x = L\xi$ and $t = t_p \tau$. This gives

$$\frac{\partial T}{\partial t} = \frac{\partial T}{\partial \tau}\frac{\partial \tau}{\partial t} = \frac{T_0}{t_p}\frac{\partial u}{\partial \tau},$$

and similarly

$$\frac{\partial T}{\partial x} = \frac{\partial T}{\partial \xi}\frac{\partial \xi}{\partial x} = \frac{T_0}{L}\frac{\partial u}{\partial \xi}.$$

So we find

$$\frac{\partial^2 T}{\partial x^2} = \frac{\partial}{\partial x}\Big(\frac{\partial T}{\partial x}\Big) = \frac{\partial}{\partial x}\Big(\frac{T_0}{L}\frac{\partial u}{\partial \xi}\Big) = \frac{\partial}{\partial \xi}\Big(\frac{T_0}{L}\frac{\partial u}{\partial \xi}\Big)\frac{\partial \xi}{\partial x} = \frac{T_0}{L^2}\frac{\partial^2 u}{\partial \xi^2}.$$

The equations then becomes

$$\rho C_p \frac{T_0}{t_p}\frac{\partial u}{\partial \tau} = k\frac{T_0}{L^2}\frac{\partial^2 u}{\partial \xi^2}.$$

The boundary conditions on $u$ are clearly what we want using the definition of the change of variable.

$$u(0,\tau) = \begin{cases} 1, & 0 \le \tau \le 1 \\ 0, & \tau > 1. \end{cases}$$

ensures that $u(0,\tau) = 1$ for $0 \le \tau \le 1$ which is gives $T(0,t) = T_0$ for the range $0 \le t \le t_p$. We also have $T(0,t) = 0$ for $t > t_p$. The right condition $\dfrac{\partial u}{\partial \xi}(1,\tau) = 0$ gives $\dfrac{L}{T_0}\dfrac{\partial T}{\partial x}(L,t) = 0$. Finally the initial condition $u(\xi,0) = 0$ for $0 < \xi \le 1$ is equivalent to $T(x,0) = 0$ for $0 < x \le L$.
For the last part of the question, writing

$$\frac{\partial u}{\partial \tau} = \underbrace{\frac{t_p k}{\rho C_p L^2}}_{a}\frac{\partial^2 u}{\partial \xi^2}.$$

We have $a = \dfrac{t_p k}{\rho C_p L^2}$ and we check that it has no dimension. Indeed,

$$\frac{s \cdot J/(m \cdot s \cdot C)}{kg/m^3 \cdot J/(kg \cdot C) \cdot m^2} = \frac{s \cdot J \cdot m^3 \cdot kg \cdot C}{m \cdot s \cdot C \cdot kg \cdot J \cdot m^2}$$

which simplifies.

## b) Discretize to a system of equation

Points go from $\xi_0 = 0$ to $\xi_N = 1$ and there is a ghost point at $\xi_{N+1}$. The boundary condition gives the value of $U_0$ for all $\tau$ and this does not need to be an unknown of the problem we solve. The right boundary condition is $\dfrac{\partial u}{\partial \xi} = 0$. We approximate with $(U_{N+1} - U_{N-1})/2h = 0$ or $U_{N+1} = U_{N-1}$. And this allows to get rid of $U_{N+1}$ and work with $U_1, .., U_N$ as the $N$ unknowns.

The equation itself is discretized to

$$\frac{du}{dt}(1) = \frac{U_2 - 2U_1 + u(0,\tau)}{h^2}$$

$$\frac{du}{dt}(2 : end - 1) = \frac{U_{i+1} - 2U_i + U_{i-1}}{h^2} \text{ for } i = 2, .., N-1.$$

$$\frac{du}{dt}(end) = \frac{2U_{N-1} - 2U_N}{h^2} \text{ using } U_{N+1} = U_{N-1}.$$

$$\frac{du}{dt} = \frac{1}{h^2} \underbrace{\begin{pmatrix} -2 & 1 & & & & & \\ 1 & -2 & 1 & & & & \\ & 1 & \ddots & \ddots & & & \\ & & \ddots & & & & \\ & & & & 1 & -2 & 1 \\ & & & & & 2 & -2 \end{pmatrix}}_{A} U + \frac{1}{h^2} \underbrace{\begin{pmatrix} u(0,\tau) \\ 0 \\ \vdots \\ 0 \end{pmatrix}}_{b(\tau)}$$

Matrix $A \in \mathbb{R}^{N \times N}$ and vector $b(\tau) \in \mathbb{R}^N$ as explained.

## c) Resolution

Let us now use Euler's explicit method to solve the equation $\dfrac{dU}{d\tau} = AU + b(\tau)$ in time. The scheme gives

$$\begin{aligned} U^{k+1} &= U^k + \Delta t\Big(\frac{1}{h^2} A U^k + \frac{1}{h^2} b(\tau)\Big) \\ &= \Big(I_N + \frac{\Delta t}{h^2} A\Big) U^k + \frac{\Delta t}{h^2} \cdot b(\tau) \end{aligned}$$

The code is available below in `tempEE.m`.

We cannot choose $\Delta x$ and $\Delta t$ as we want to obtain a stable solution.The following tests have been made :

| $\Delta x$ | $\Delta t$ | $\frac{\Delta t}{\Delta x^2}$ | **Stability** |
|---|---|---|---|
| 0.05 | 0.01 | 4 | Unstable |
| 0.1 | 0.001 | 0.1 | Stable |
| 0.1 | 0.005 | 0.5 | Stable |
| 0.1 | 0.0051 | 0.51 | Unstable |

It is possible to show that, in order to have a stable solution, the following condition must be satisfied : $\frac{\Delta t}{\Delta x^2} \leq \frac{1}{2}$.

We present a stable and an unstable solution on the figures 1,2. They were obtained respectively with the calls `tempEE(10,0.005,2)` and `tempEE(10,0.0051,2)`. This corresponds

to $\Delta x = 0.1$ and $\Delta t = \{0.005, 0.0051\}$. This gives $\dfrac{\Delta t}{\Delta x^2} = \{0.5, 0.51\}$. The unstable case breaks the condition $\dfrac{\Delta t}{\Delta x^2} \leq \dfrac{1}{2}$.

We can clearly see that the solution presented in figure 2 is unstable. Indeed, it shows large oscillations and even negative temperatures (which is not possible since the rod is heated).

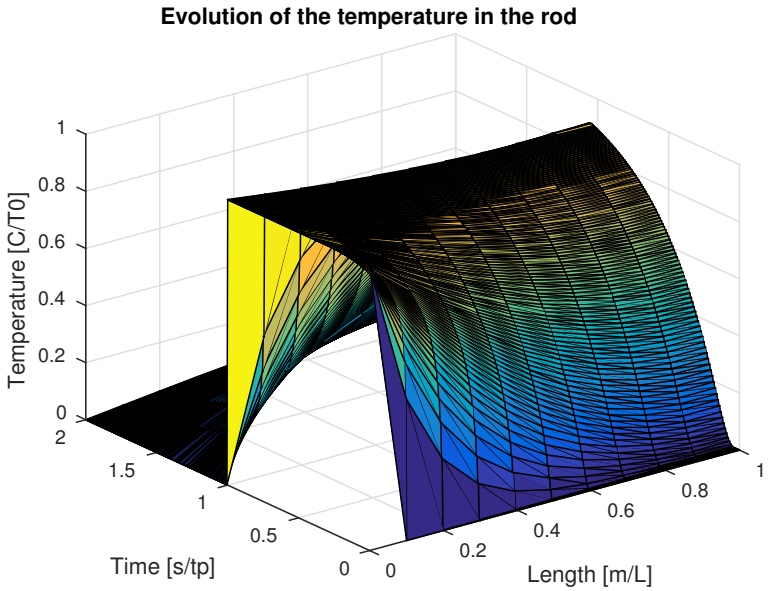


Figure 1: Stable solution

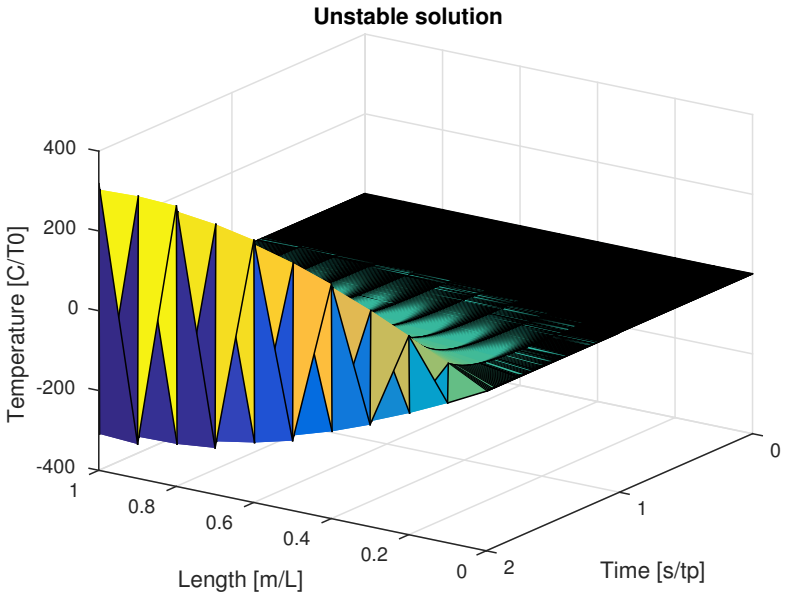


Figure 2: Unstable solution

## d) Comparison of methods

In this section, we will compare two methods for solving the system of differential equations (the one obtained in part 2) : the explicit method *ode*23 and the implicit method *ode*23*s*. As already said, an implicit method is often more suitable for stiff problems.

The two methods will be compared for differents constant stepsizes for the spatial discretization ($N \in [10, 20, 40]$) and for a determined time interval ($\tau \in [0; 2]$). We will based our analysis our three different factors : the number of steps needed to reach $\tau = 2$, the cpu-time needed to do the computations and the maximal value of the stepsize for the time discretization. The default tolerances are used for both methods.

The Matlab code used to do this analysis can be found at the end of this section. As we will reuse this code for part e, this is how the function was called :

$$[timeStep, cpuTime, hMax] = tempOde('Gaussian')$$

The returned variables are arrays containing the data. The first column is when *ode*23 is used and the second is for *ode*23*s*. The lines correspond to the different stepsizes for the spatial discretization.

The call to this function gives the following data :

| | **timesteps** | | **cpu - time** | | $\mathbf{h}_{tmax}$ | |
|---|---|---|---|---|---|---|
| $N$ | *ode*23 | *ode*23*s* | *ode*23 | *ode*23*s* | *ode*23 | *ode*23*s* |
| 10 | 345 | 104 | 0.1479 | 0.1944 | 0.0169 | 0.1473 |
| 20 | 1295 | 132 | 0.4730 | 0.4108 | 0.0043 | 0.1563 |
| 40 | 5114 | 169 | 1.9056 | 0.8883 | 0.0010 | 0.1522 |

We can easily see that, for all the factors taken into account, *ode*23*s* is better than its explicit counterpart.

First, the number of time steps. The explicit method needs much more steps to reach $\tau = 2$, this is because the problem is stiff. We can even compare this number with the theoretical number needed for stability in the case of a constant step size. For $N = 10$, the maximal stable stepsize would be $h_t = \frac{(0.1)^2}{2} = 0.005$ so the number of steps to reach $\tau = 2$ would be $n = \frac{2}{0.005} = 400$. We can see that *ode*23 is just a little better whereas *ode*23*s* divides this number by 4.

Concerning the cpu - time, *ode*23*s* is an obvious winner only when $N = 40$. For $N = 10$, *ode*23 is even a little bit better. This is mainly because the Gaussian elmination of the linear system takes time, but this will be solved in part e.

Finally, the maximal timestep looks approximately constant for *ode*23*s*. This is not the case for the explicit method. It has to reduce considerably the timestep in order to avoid instability.

## e) Improvements

In this section, we will try to improve the efficiency of *ode*23*s*. Indeed, this method has, at each timestep, to solve a system and uses a Gaussian elimination. However, our system is tridiagonal so this is not optimal. It is in fact possible to "tell" Matlab a bit more about our system so that the resolution will be more efficient. Two ways are used here.

First, we can tell him which entries of the jacobian are non zeros. It will increase the efficiency because Matlab will know what computations are not necessary (because somothing times zero is always zero). This is done by the *odeset* property called $JPattern$. We can obtain information by calling the same function as in part d but with a different argument :

$$[timeStep, cpuTime, hMax] = tempOde('sparse')$$

Second, we can tell Matlab what is the jacobian of our system. In our case, it is a constant matrix. This will greatly help Matlab in his resolution of the system. To see this, we can call :

$$[timeStep, cpuTime, hMax] = tempOde('jacobian')$$

To see the effect of the combined properties, we can call :

$$[timeStep, cpuTime, hMax] = tempOde('both')$$

The following table contains the cpu - time for $N = 10, 20, 40$ for different combinations of the *odeset* properties with the method $ode23s$. The column $'both'$ means that the two properties were set.

**CPU - time** ($ode23s$)

| N | Gaussian | sparse | jacobian | both |
|---|---|---|---|---|
| 10 | 0.1944 | 0.1097 | 0.0504 | 0.0560 |
| 20 | 0.4108 | 0.1476 | 0.0667 | 0.0697 |
| 40 | 0.8883 | 0.1803 | 0.0910 | 0.0992 |

We can see that $JPattern$ improves the compution time and $Jacobian$ enven more. Using both $Jacobian$ and $JPattern$ seems to not have a greater effect than using $Jacobian$ alone since the CPU - times are really close. But the computation time can be reduced by a almost a factor 10 for $N = 40$.

Finally, we can look at the same table as in part e and see if something other than the cpu - time changes when we set the properties. The following table is obtained when the two properties are set :

| | **timesteps** | | **cpu - time** | | $\mathbf{h}_{tmax}$ | |
|---|---|---|---|---|---|---|
| $N$ | $ode23$ | $ode23s$ | $ode23$ | $ode23s$ | $ode23$ | $ode23s$ |
| 10 | 345 | 103 | 0.1217 | 0.0560 | 0.0169 | 0.1473 |
| 20 | 1295 | 132 | 0.4383 | 0.0697 | 0.0043 | 0.1563 |
| 40 | 5114 | 173 | 1.8093 | 0.0992 | 0.0010 | 0.1523 |

We can see that, the cpu - time aside, all the factors remain almost the same. For $ode23$, this is because this is an explicit method and so those two properties will have little influence. Indeed, in an explicit method, no system has to be solved. Instead, a matrix multiplication is used. The resolution will thus be a little bit faster because the jacobian is constant but the gain will not be as large as for an implicit method.

## f) Visualization

This section is about the visualization of the results. The Matlab code can be found at the end of the report. Figure 3 shows the temperature in the rod at four time points (namely $\tau \approx 0.5, 1, 1.5, 2$).

We can notice that the boundary condition is fulfilled (as it always should be). The temperature rises until $\tau = 1$ and then starts decreasing, as it is expected.

The next plot is the same as the stable one in section c). It is shown in figure 4. This plot clearly depicts the boundary condition. As for the initial condition, the discretization of the $x$-axis has made it less obvious. We have in fact $u(0, 0) = 1$ and $u(0.1, 0) = U_1(0) = 0$. Between the two, it should be zero but Matlab uses a linear interpolation. We can also see here that until $\tau = 1$, the temperature is rising while it is decreasing afterwards.

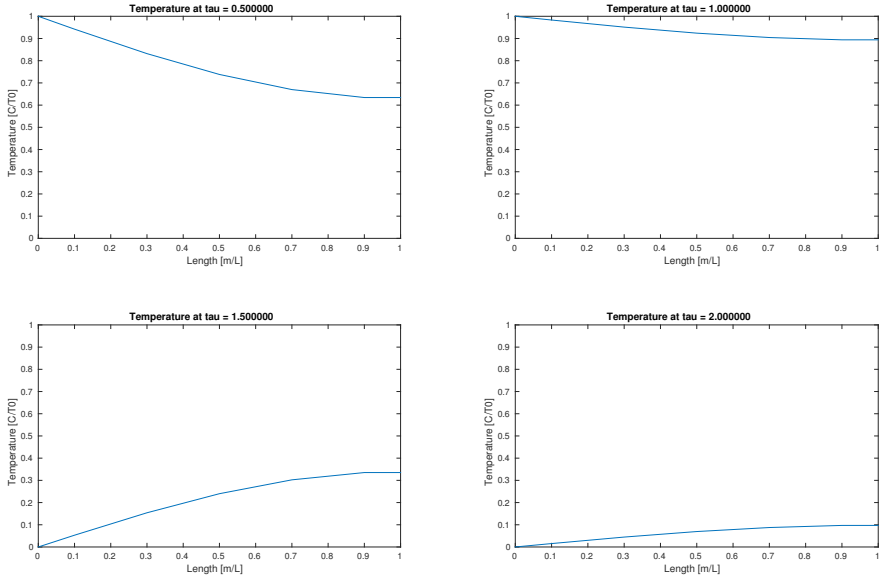


Figure 3: Temperature in the rod at different times

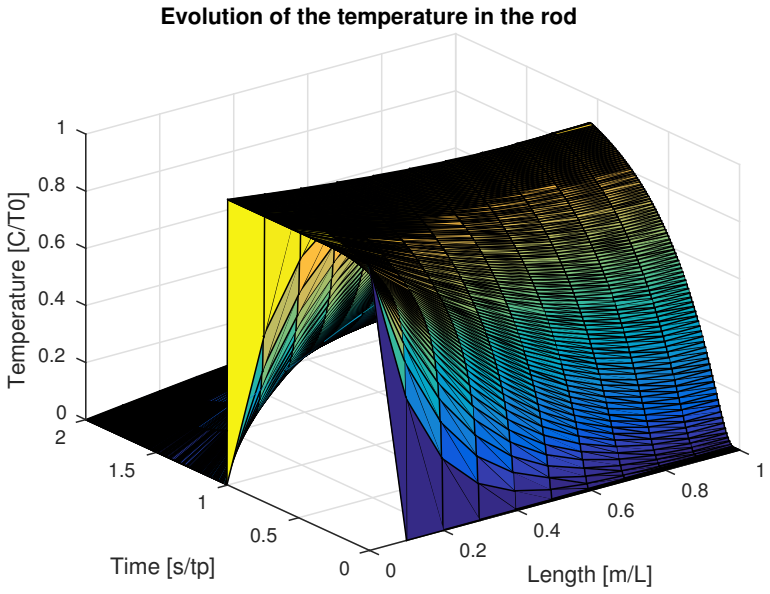


Figure 4: Temperature in the rod at any time

## Conclusion

This report presented the solution of the very practical problem : the evolution of the temperature in a rod. This can be modelled as a parabolic PDE. To solve this PDE, we used the finite difference method to arrive at a system of ODEs. This system of ODE is stiff and a stiff method should then be used to solve it efficiently.

The stucture of the jacobian of the ODE system is really particular : it is tridiagonal (thus banded). This particularity should be taken into account when using the stiff solver (as did in section e) of the report).

## MATLAB

```
%This scripts plots the variables figures needed for part f) of LAB4
%We use : hx = 0.1
%         ht = 0.005
%         tend = 2
close all;
clear all;

[u,x,t] = tempEE(10,0.005,2);

figure;
surf(x(length(x):-1:1),t,u');title('Evolution of the temperature in the rod');xlabel('↩
    Length [m/L]');
ylabel('Time [s/tp]');zlabel('Temperature [C/T0]');

figure;
T = [0.5 1 1.5 2];
ti = floor(T/0.005)+1;
for i = 1:4
    subplot(2,2,i);
    plot(x(length(x):-1:1),u(:,ti(i)));title(sprintf('Temperature at tau = %f',T(i)));
    xlabel('Length [m/L]'); ylabel('Temperature [C/T0]');axis([0 1 0 1]);
end

%We can also plot an unstable solution
%We use : hx = 0.1
%         ht = 0.0051
%         tend = 2
[uUnstable,x,t] = tempEE(10,0.0051,2);

figure;
surf(x(length(x):-1:1),t,uUnstable');title('Unstable solution');xlabel('Length [m/L]')↩
    ;
ylabel('Time [s/tp]');zlabel('Temperature [C/T0]');
```

```
function [u,x,t] = tempEE(Nx,ht,tend)
%TEMPEE Solve the temperature problem with Euler Explicit
% This function solve the problem given in LAB4 with the FDM for the
% spatial derivatives and with Euler Explicit method for the time
% derivative
% INPUT : hx is the spatial stepsize
%         ht is the time stepsize
%         tend is the final time
% OUTPUT : u is the numerical solution
%          x is the number of points in the x-axis
%          t is the discretized t-axis (from 0 to tend)

hx= 1/Nx;
coeff = ht/(hx*hx);
x = 0:hx:1; %length = Nx+1
```

```
t = 0:ht:tend;

u = zeros(Nx+1,floor(tend/ht+1));

e = ones(Nx,1);
A = spdiags([e -2*e e],-1:1,Nx,Nx);
A(Nx,Nx-1) = 2;
b = sparse(Nx,1);

i = 1;
for ti = ht:ht:tend
    b(1,1) = boundCondLeft(ti);
    u(2:Nx+1,i+1) = (speye(Nx)+coeff*A)*u(2:Nx+1,i)+coeff*b;
    i = i+1;
end
u(1,:) = boundCondLeft(0:ht:tend);


% Time animation of the temperature :)
% for i=1:length(t)
%     titre = sprintf('Time t=%f',(i-1)*ht);
%     bar = u(:,i)*[1 1];
%     contourf(x,1:2,bar',0:0.01:1);title(titre);colorbar;caxis([0 1]);
%     F(i)=getframe;
% end
% movie(F);

u = flipud(u);
end

function U = boundCondLeft(t)
%Returns the boundary condition at the given time
U = (t<=1);
end
```

```
function [timeStep,cpuTime,hMax] = tempOde(efficiency)
%TEMPODE This function compares two built-in ode solvers (ode23 and ode23s)
%for the solution of the temperature problem
N = [10 20 40];
timeStep = zeros(3,2);
cpuTime = zeros(3,2);
hMax = zeros(3,2);

for i=1:3
    u0 = zeros(N(i),1);
    %We set, if neededd, the options for ode23 and ode23s
    e = ones(N(i),1);
    switch efficiency
        case 'sparse'
            S = spdiags([e e e],-1:1,N(i),N(i));
            options = odeset('JPattern',S);
        case 'jacobian'
            A = spdiags([e -2*e e],-1:1,N(i),N(i));
            A(N(i),N(i)-1) = 2;
            options = odeset('Jacobian',N(i)*N(i)*A);
        case 'both'
            S = spdiags([e e e],-1:1,N(i),N(i));
            A = spdiags([e -2*e e],-1:1,N(i),N(i));
            A(N(i),N(i)-1) = 2;
            options = odeset('JPattern',S,'Jacobian',N(i)*N(i)*A);
        otherwise
            options = [];
    end
    %We use ode23
    tstart = tic;
    [t,~] = ode23(@(t,u) tempe(N(i),t,u),[0 2],u0,options);
    cpuTime(i,1) = toc(tstart);
    %We use ode23s
```

```
    tstart = tic;
    [tStiff,~] = ode23s(@(t,u) tempe(N(i),t,u),[0  2],u0,options);
    cpuTime(i,2) = toc(tstart);
    %We compute the time steps
    h = diff(t);
    timeStep(i,1) = length(h);
    hMax(i,1) = max(h);
    hStiff = diff(tStiff);
    timeStep(i,2) = length(hStiff);
    hMax(i,2) = max(hStiff);
end

end

function dudt = tempe(Nx,t,u)
hx = 1/Nx;

e = ones(Nx,1);
A = spdiags([e -2*e e],-1:1,Nx,Nx);
A(Nx,Nx-1) = 2;
b = sparse(Nx,1);
b(1) = (t<=1);

dudt = (A*u+b)./(hx*hx);
end
```