Intensive Course: *Elements of Scientific Computing*

Part I: The Basics

Computing Integrals

Trapezoid method

- Generally we will study how to approximate definitive integrals of the form

$$\int_a^b f(x)dx$$

- Consider e.g. the function $f(x) = e^x$ and calculate

$$\int_1^2 e^x dx \tag{1}$$

- We will in the following pretend that this integral is not analytically integrable , and later use the exact analytical solution for comparison

Trapezoid method

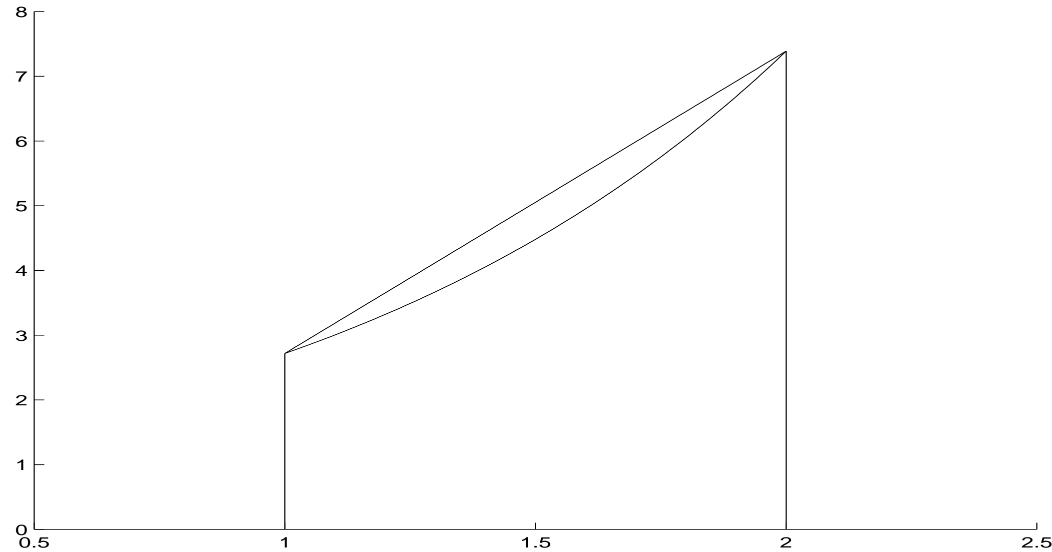

Figure 1: The figure illustrates how the integral of $f(x) = e^x$ (lower curve) may be approximated by a trapezoid on a given interval

Trapezoid method

- Let $y(x)$ be the straight line equal to f at the endpoints $x = 1$ and $x = 2$, i.e.

$$y(x) = e\left[1 + (e-1)(x-1)\right]$$

- Note that

$$y(1) = e = f(1)$$
$$y(2) = e^2 = f(2)$$

- Since $y(x) \approx f(x)$ we approximate the integral by

$$\int_1^2 e^x dx \approx \int_1^2 y(x) dx \tag{2}$$

Trapezoid method

We can now compute both integrals and compare the results

- Approximate

$$\int_1^2 y(x)dx = \int_1^2 e\left[1+(e-1)(x-1)\right]dx = \frac{1}{2}e + \frac{1}{2}e^2 \approx 5.0537$$

- Exact

$$\int_1^2 e^x dx = e(e-1) \approx 4.6708$$

Trapezoid method

The relative error is

$$\frac{5.0537 - 4.6708}{5.0537} \cdot 100\% \approx 7.6\%$$

Trapezoid method

- Generally we can approximate the integral of f by

$$\int_a^b f(x)dx \approx \int_a^b y(x)dx \tag{3}$$

where $y(x)$ is a straight line equal to f at the endpoints, i.e.

$$y(x) = f(a) + \frac{f(b) - f(a)}{b - a}(x - a) \tag{4}$$

- $y(x)$ is called the linear interpolation of f in the interval [a,b]

Trapezoid method

- Since y is linear, it is easy to compute the integral of this function

$$\int_a^b y(x)dx = \int_a^b \left[f(a) + \frac{f(b)-f(a)}{b-a}(x-a) \right] dx$$
$$= (b-a)\frac{1}{2}(f(a)+f(b))$$

- The trapezoid rule is therefore given by

$$\boxed{\int_a^b f(x)dx \approx (b-a)\frac{1}{2}(f(a)+f(b))} \qquad (5)$$

Example 1

- $f(x) = \sin(x),\ \ a = 1,\ \ b = 1.5$
- Trapezoid method

$$\int_1^{1.5} f(x)dx \approx (1.5-1)\frac{1}{2}(\sin(1)+\sin(1.5)) \approx 0.4597$$

- The exact value

$$\int_1^{1.5} f(x)dx = -\left[\cos(x)\right]_1^{1.5} = -(\cos(1.5)-\cos(1)) \approx 0.4696$$

- The relative error is

$$\frac{0.4696-0.4597}{0.4696}\cdot 100\% \approx 2.11\%$$

Trapezoid method

Now we approximate the integral using two trapezoids

- Choosing the middle point between a and b, $c = (a+b)/2$, we have that

$$\int_a^b f(x)dx = \int_a^c f(x)dx + \int_c^b f(x)dx$$

- Using (5) on each integral gives

$$\int_a^b f(x)dx \approx \left[(c-a)\frac{1}{2}(f(a)+f(c))\right] + \left[(b-c)\frac{1}{2}(f(c)+f(b))\right]$$

Trapezoid method

- By using that

$$c - a = b - c = \frac{1}{2}(b - a),$$

we get

$$\boxed{\int_a^b f(x)dx \approx \frac{1}{4}(b-a)\left[f(a) + 2f(c) + f(b)\right]} \qquad (6)$$

Example 2

- Using (6) on the problem considered in Example 1 gives

$$\int_1^{1.5} \sin(x)dx \approx \frac{1}{4} \cdot \frac{1}{2} [\sin(1) + 2\sin(1.25) + \sin(1.5)] \approx 0.4671$$

- The relative error of this approximation is

$$\frac{0.4696 - 0.4671}{0.4696} \cdot 100\% = 0.53\%$$

- This is significantly better than the approximation computed in in Example 1, where the error was 2.11%

Trapezoid method

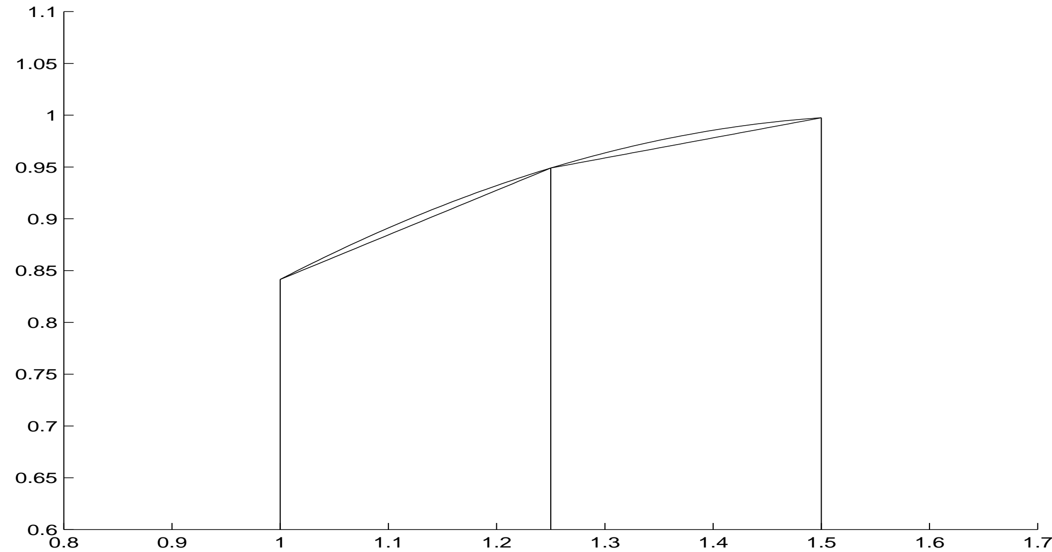

Figure 2: The figure illustrates how the integral of $f(x) = \sin(x)$ can be approximated by two trapezoids on a given interval

Trapezoid method

More generally we can approximate the integral using n trapezoids

- Let $h = \frac{b-a}{n}$
- Define $x_i = a + ih$
- The points

$$a = x_0 < x_1 < \cdots < x_{n-1} < x_n = b$$

divide the interval from a to b into n subintervals of length h

Trapezoid method

- The integral has the following additive property

$$\int_a^b f(x)dx = \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx + \cdots + \int_{x_{n-1}}^{x_n} f(x)dx$$
$$= \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)dx \quad (7)$$

- We use (5) on each integral, i.e.

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx (x_{i+1} - x_i)\frac{1}{2}[f(x_i) + f(x_{i+1})]$$

Trapezoid method

Since $h = x_{i+1} - x_i$, we get

$$\int_a^b f(x)dx = \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)dx$$

$$\approx \sum_{i=0}^{n-1} \frac{h}{2} [f(x_i) + f(x_{i+1})]$$

$$= \frac{h}{2} \Big([f(x_0) + f(x_1)] + [f(x_1) + f(x_2)] + [f(x_2) + f(x_3)]$$

$$+ \cdots + [f(x_{n-2}) + f(x_{n-1})] + [f(x_{n-1}) + f(x_n)] \Big)$$

$$= h \Big[\frac{1}{2} f(x_0) + f(x_1) + f(x_2) + \cdots$$

$$\cdots + f(x_{n-2}) + f(x_{n-1}) + \frac{1}{2} f(x_n) \Big]$$

Trapezoid method

Written more compactly

$$\int_a^b f(x)dx \approx h\left[\frac{1}{2}f(x_0) + \sum_{i=1}^{n-1} f(x_i) + \frac{1}{2}f(x_n)\right] \tag{8}$$

Example 3

The integral considered in Example 1 with $n = 100$.

- $h = \frac{b-a}{n} = \frac{0.5}{100} = 0.005$
- We get

$$\int_1^{1.5} \sin(x)dx \approx 0.005 \left[\frac{1}{2}\sin(1) + \sin(1.005) + \cdots + \frac{1}{2}\sin(1.5) \right]$$
$$= 0.469564$$

- The relative error is

$$\frac{0.469565 - 0.469564}{0.469565} \cdot 100\% = 0.0002\%$$

Example 4

Calculate $\int_0^1 f(x)dx$, where $f(x) = (1+x)e^x$

- The exact integral is

$$\int_0^1 (1+x)e^x dx = [xe^x]_0^1 = e$$

- Define $T_h = h\left[\frac{1}{2}f(0) + \sum_{i=1}^{n-1} f(x_i) + \frac{1}{2}f(1)\right]$
- where n is given and $h = \frac{1}{n}$ and $x_i = ih$ for $i = 1, \ldots, n$
- We want to study the error defined by

$$E_h = |e - T_h|$$

Example 4

n	h	E_h	E_h/h^2
1	1.0000	0.5000	0.5000
2	0.5000	0.1274	0.5096
4	0.2500	0.0320	0.5121
8	0.1250	0.0080	0.5127
16	0.0625	0.0020	0.5129
32	0.0313	0.0005	0.5129
64	0.0156	0.0001	0.5129

Table 1: The table shows the number of intervals, n, the length of the intervals, h, the error, E_h, and E_h/h^2

Example 4

- From the table it seems that

$$\frac{E_h}{h^2} \approx 0.5129$$

  for small values of h

- That is

$$E_h \approx 0.5129h^2 \tag{9}$$

- This means that we can get as accurate approximation as we want

Example 4

- Assume that you want $E_h \leq 10^{-5}$
- then $0.5129h^2 \leq 10^{-5}$
- or $h \leq 0.0044$
- This means that $n = 1/h \geq 226.47$
- n has to be an integer, so therefore we set $n = 227$ to obtain the desired accuracy

Example 5

We want to test the trapezoid method for the following three integrals:

- $\int_0^1 x^4 dx$
- $\int_0^1 x^{20} dx$
- $\int_0^1 \sqrt{x} dx$
- Let E_h denote the error for a given value of h, i.e.

$$E_h = \left| \int_a^b f(x)dx - h\left[\frac{1}{2}f(x_0) + \sum_{i=1}^{n} f(x_i) + \frac{1}{2}f(x_n)\right]\right|,$$

where $h = \frac{b-a}{n}$ and $x_i = a + ih$ for $i = 0, \ldots, n$

Example 5

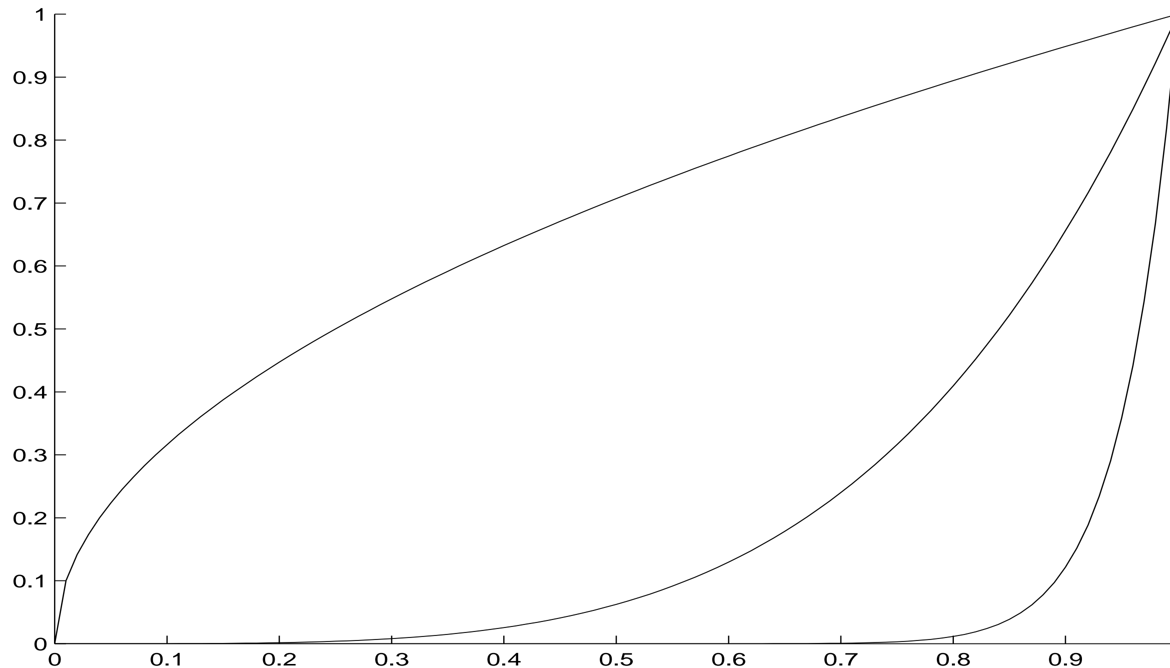

Figure 3: The figure shows the graph of $\sqrt{x}$ (upper), x^4 (middle) and x^{20} (lower)

Example 5

	$\int_0^1 x^4 dx = \frac{1}{5}$		$\int_0^1 x^{20} dx = \frac{1}{21}$		$\int_0^1 \sqrt{x} dx = \frac{2}{3}$	
h	$10^5 E_h$	E_h/h^2	$10^5 E_h$	E_h/h^2	$10^5 E_h$	E_h/h^2
0.01	3.33	0.33	16.66	1.67	20.37	2.04
0.005	0.83	0.33	4.17	1.67	7.25	2.90
0.0025	0.21	0.33	1.04	1.67	2.57	4.17
0.00125	0.05	0.33	0.26	1.67	0.91	5.84

Table 2: The table shows how accurate the trapezoidal method is for approximating three definite integrals.

Example 5

Conclusions

- In the two first integrals $\frac{E_h}{h^2}$ seems to be constant
- The constant is smaller for x^4 than for x^{20}
- The approximate integral of $\sqrt{x}$ on [0,1], seems to converge towards the correct value as $h \to 0$, but $\frac{E_h}{h^2}$ increases with decreasing h

Trapezoid method

- We have studied several examples where the exact integral is obtainable
- In practice these examples are not so interesting
- Numerical integration is more interesting on examples where analytical integration is impossible

```
import numpy as np

def integrate(a, b, n, function):

    x = np.linspace(a, b, n+1);
    value = function(x);
    value[0] = 0.5*value[0]
    value[-1] = 0.5*value[-1]
    h = (b-a)/float(n)

    return h*sum(value)

def f(x):

    return x**2;


print "The integral is approximatly ", integrate(0, 1, 100, f)
```

Differential Equations

Differential equations

- A differential equations is: an equations that relate a function to its derivatives in such a way that the function can be determined
- In practice, differential equations typically describe quantities that changes in relation to each other
- Examples of such equations arise in several disciplines of Science and Technology (e.g. physics, chemistry, biology, economy, weather forecasting,...)

Cultivation of rabbits

- A number of rabbits are placed on an isolated island with perfect environments for them
- How will the number of rabbits grow?

Note that this question can not be answered based on clever thinking only.

The simplest model

- Let $r = r(t)$ denote the number of rabbits
- Let $r_0 = r(0)$ denote the initial number of rabbits
- Assume that the change of rabbits per time is given by $f(t)$
- For a small period of time $\Delta t > 0$, we have

$$\frac{r(t+\Delta t) - r(t)}{\Delta t} = f(t) \tag{10}$$

- Assuming that $r(t)$ is continuous and differentiable and letting Δt go to zero, we obtain

$$r'(t) = f(t) \tag{11}$$

The simplest model

- From the fundamental theorem of Calculus, we get the solution

$$r(t) = r(0) + \int_0^t f(s)ds \tag{12}$$

- The integral can then be calculated as accurate as we want, with the methods presented in the previous lectures

Exponential growth

- We now assume that the growth in population is proportional to the number of rabbits, i.e

$$\frac{r(t+\Delta t)-r(t)}{\Delta t}=ar(t), \tag{13}$$

where a is a positive constant

- Letting Δt go to zero we get

$$r'(t)=ar(t) \tag{14}$$

- In practice a has to be measured

Analytical solution

- We want to solve the problem

$$r'(t) = ar(t) \tag{15}$$

with initial condition

$$r(0) = r_0$$

- Since

$$\frac{dr}{dt} = ar$$

- we have

$$\frac{1}{r}dr = a\,dt$$

Analytical solution

- by integrating we get

$$\int \frac{1}{r} dr = \int a \, dt$$

- which gives

$$\ln(r) = at + c \tag{16}$$

  where c is a constant of integration
- The right value for c is received by putting $t = 0$

$$c = \ln(r_0)$$

Analytical solution

- From (16) we get

$$\ln(r(t)) - \ln(r_0) = at$$

- or

$$\ln(\frac{r(t)}{r_0}) = at$$

- and therefore

$$r(t) = r_0 e^{at} \tag{17}$$

- Conclusion: the number of rabbits increase exponentially in time

Logistic growth

The exponential growth is not realistic, since the number of rabbits will go to infinity as the time increase.

- We assume that there is a *carrying capacity* R of the island
- This number tells how many rabbits the island can feed, host etc.
- The *logistic model* reads

$$r'(t) = ar(t)\left(1 - \frac{r(t)}{R}\right) \tag{18}$$

where $a > 0$ is the growth rate and $R > 0$ is the carrying capacity

Analytical solution

Solve

$$r'(t) = ar(t)\left(1 - \frac{r(t)}{R}\right)$$

$$r(0) = r_0.$$

We write

$$\frac{dr}{dt} = ar\left(1 - \frac{r}{R}\right),$$

or

$$\frac{dr}{r\left(1 - \frac{r}{R}\right)} = a\,dt.$$

Analytical solution

By integration we get

$$\ln \frac{r}{R-r} = at + c$$

where c is a integration constant. This constant is determined by the initial condition

$$\ln \frac{r_0}{R-r_0} = c$$

and thus

$$\ln \left[\frac{\frac{r}{R-r}}{\frac{r_0}{R-r_0}} \right] = at,$$

Analytical solution

or

$$\frac{r}{R-r} = \frac{r_0}{R-r_0} e^{at}.$$

Solving this with respect to r gives

$$r(t) = \frac{r_0}{r_0 + e^{-at}(R - r_0)} R \tag{19}$$

(see Figure 4.)

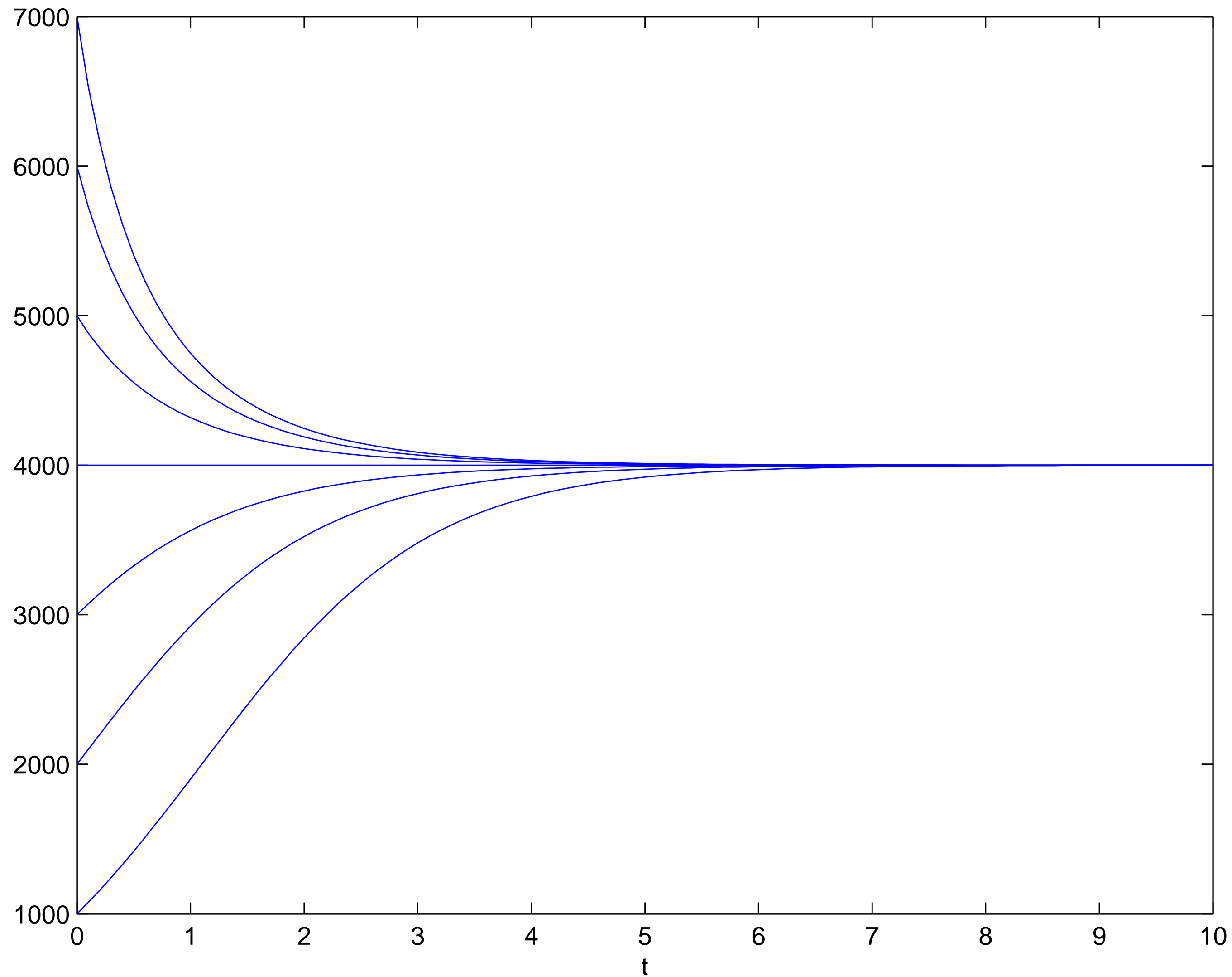

Figure 4: Different solutions of (19) using different values of r_0.

Numerical solution

- For simple examples of differential equations we can find analytical solutions
- This is not the case for most of the realistic models of nature
- Analytical solutions are still important for testing numerical methods
- Analytical insight is very important for designing good numerical methods
- An example of this is the insight we got above from the arguments about increase and decrease in rabbit population

The simplest model

We now pretend that we do not know the exact solution of

$$r'(t) = f(t)$$

with $r(0) = r_0$, and solve the problem in $t \in (0,1)$.

- Pick a positive integer N, and define the time-step

$$\Delta t = \frac{1}{N}$$

- Define time-levels $t_n = n\Delta t$
- Let r_n denote the approximation of $r(t_n)$

$$r_n \approx r(t_n)$$

The simplest model

- Remember the series expansion

$$r(t+\Delta t) = r(t) + \Delta t\, r'(t) + O(\Delta t^2)$$

- or

$$r'(t) = \frac{r(t+\Delta t) - r(t)}{\Delta t} + O(\Delta t)$$

- By setting $t = t_n$, we therefore see that

$$r'(t_n) \approx \frac{r(t_{n+1}) - r(t_n)}{\Delta t}$$

- By using the approximate solutions $r_n \approx r(t_n)$ and $r_{n+1} \approx r(t_{n+1})$, the numerical scheme is defined

$$\frac{r_{n+1} - r_n}{\Delta t} = f(t_n)$$

Exponential growth

We now want study numerical solution of the problem $r'(t) = ar(t)$, $t \in (0, T)$, where a is a given constant, and initial condition $r(0) = r_0$.

- We choose an integer $N > 0$, define the time-steps $\Delta t = T/N$ and the time-levels $t_n = n\Delta t$, r_n is the approximation of $r(t_n)$ and the derivative is approximated by

$$r'(t_n) \approx \frac{r(t_{n+1}) - r(t_n)}{\Delta t}$$

- The numerical scheme is defined by

$$\frac{r_{n+1} - r_n}{\Delta t} = ar_n \tag{20}$$

Exponential growth

- Which can be written

$$r_{n+1} = (1 + a\Delta t)r_n \tag{21}$$

- This formula gives initially

$$\begin{aligned} r_1 &= (1 + a\Delta t)r_0 \\ r_2 &= (1 + a\Delta t)r_1 = (1 + a\Delta t)^2 r_0 \end{aligned}$$

- and for general n we can see that

$$r_n = (1 + a\Delta t)^n r_0 \tag{22}$$

Example 8

We test an example where $a = 1$, $r_0 = 1$ and $T = 1$.

- The exact solution is $r(t) = e^t$ and therefore

$$r(1) = e \approx 2.718$$

- Using $N = 10$ in the numerical scheme gives

$$r(1) \approx r_{10} = (1 + \frac{1}{10})^{10} \approx 2.594$$

- Choosing $N = 100$, gives

$$r_{100} = (1 + \frac{1}{100})^{100} \approx 2.705$$

Example 8 - Convergence

- The general formula is

$$r(1) \approx r_N = (1 + \frac{1}{N})^N$$

- From Calculus we know that

$$\lim_{N \longrightarrow \infty} (1 + \frac{1}{N})^N = e = r(1)$$

- Thus the numerical scheme will converge to the right solution in this example

```
import numpy as np

def f(t, u):
    return u;

T = 1;
N = 10; #number of time steps
t = np.linspace(0,T,N+1)
dt = float(T)/N

u = np.zeros(N+1)
u[0] = 1; # set intitial condition

for i in range(N):
    u[i+1] = u[i] + dt*f(t[i], u[i])

import pylab
pylab.plot(t, u, t, np.exp(t))
pylab.show()
```

Numerical stability

Consider the initial value problem

$$\begin{aligned} y'(t) &= -100y(t), \quad t \in (0,1) \\ y(0) &= 1, \end{aligned} \tag{23}$$

with analytic solution

$$y(t) = e^{-100t}$$

- For a given N and corresponding Δt we have

$$y_{n+1} = (1 - 100\Delta t)y_n \tag{24}$$

- which gives

$$y_n = \left(1 - \frac{100}{N}\right)^n$$

Numerical stability

- Note that the analytical solution is always positive, but decreases rapidly and monotonically towards zero
- For $N = 10$ we get the formula

$$y_n = \left(1 - \frac{100}{10}\right)^n = (-9)^n$$

- which gives $y_0 = 1,\ y_1 = -9,\ y_2 = 18,\ y_3 = -729$
- This is referred to as *numerical instability*

Numerical stability

- For y_n to stay positive we get from (24) that

$$1 - 100\Delta t > 0$$

- or

$$\Delta t < \frac{1}{100} \tag{25}$$

- which means

$$N \geq 101$$

- This is referred to as *stability condition*
- A numerical scheme that is stable for all Δt is called *unconditionally stable*
- A scheme that needs a stability condition is called *conditionally stable*

An implicit scheme

We still study the exponential model, $r'(t) = ar(t)$.

- Above the observation

$$r'(t_n) = \frac{r(t_{n+1}) - r(t_n)}{\Delta t} + O(\Delta t)$$

- led to the scheme

$$\frac{r_{n+1} - r_n}{\Delta t} = ar_n$$

- Similarly we could have observed that

$$r'(t_{n+1}) = \frac{r(t_{n+1}) - r(t_n)}{\Delta t} + O(\Delta t)$$

An implicit scheme

- This leads to

$$\frac{r_{n+1} - r_n}{\Delta t} = ar(t_{n+1})$$

- which can be written

$$r_{n+1} = \frac{1}{1 - \Delta t a} r_n$$

- This leads to

$$r_n = \left(\frac{1}{1 - \Delta t a} \right)^n r_0$$

An implicit scheme

- Reconsider the initial value problem

$$
\begin{aligned}
y'(t) &= -100y(t), \\
y(0) &= 1
\end{aligned}
$$

- The implicit scheme gives

$$
\begin{aligned}
y_n &= \left(\frac{1}{1+100\Delta t}\right)^n \\
&= \left(\frac{N}{N+100}\right)^n
\end{aligned}
$$

- We see that y_n is positive for all choices of N
- The scheme is therefore unconditionally stable

An implicit scheme

N	y_N
10^1	$3.85 \cdot 10^{-11}$
10^2	$7.89 \cdot 10^{-31}$
10^3	$4.05 \cdot 10^{-42}$
10^7	$3.72 \cdot 10^{-44}$

The exact solution is $e^{-100} \approx 3.72 \cdot 10^{-44}$.

Explicit and implicit schemes

We consider problems on the form

$$v'(t) = \textit{something}(t) \tag{26}$$

The term $v'(t)$ is replaced

$$\frac{v_{n+1} - v_n}{\Delta t}$$

The right hand side can be evaluated in $t = t_n$ or $t = t_{n+1}$.

- Explicit scheme: $v_{n+1} = v_n + \Delta t\, \textit{something}(t_n)$
- Implicit scheme: $v_{n+1} = v_n + \Delta t\, \textit{something}(t_{n+1})$

Implicit schemes are often unconditionable stable, but might be harder to use. Explicit schemes are often only conditionable stable, but are very simple to implement.

Logistic equation

We study the explicit scheme for the logistic equation

$$r'(t) = ar(t)\left(1 - \frac{r(t)}{R}\right) \quad (27)$$

$$r(0) = r_0, \quad (28)$$

where $a > 0$ is the growth rate and R is the carrying capacity. The discussion above gives the properties

- If $R >> r_0$, then for small t, we have $r'(t) \approx ar(t)$ and thus exponential growth
- If $0 < r_0 < R$, then the solution satisfies $r_0 \leq r(t) \leq R$ and $r'(t) \geq 0$ for all time
- If $r_0 > R$, then the solution satisfies $R \leq r(t) \leq r_0$ and $r'(t) \leq 0$ for all time

Explicit scheme

An explicit scheme for this model reads

$$\frac{r_{n+1} - r_n}{\Delta t} = ar_n(1 - \frac{r_n}{R}),$$

or

$$r_{n+1} = r_n + ar_n\Delta t(1 - \frac{r_n}{R}). \tag{29}$$

We assume the same stability conditions for this scheme as for the exponential growth because of the exponential growth, i.e.

$$\Delta t < 1/a. \tag{30}$$

Implicit scheme

The implicit scheme for the logistic model reads

$$\frac{r_{n+1} - r_n}{\Delta t} = a r_{n+1} (1 - \frac{r_{n+1}}{R}),$$

or

$$r_{n+1} - \Delta t a r_{n+1} (1 - \frac{r_{n+1}}{R}) = r_n.$$

- For r_n given, this is a nonlinear equation in r_{n+1}
- This is easy to solve since it is only a second order polynomial equation

The scheme is unconditionally stable and it fulfills the same properties as the explicit scheme did.

Systems of Ordinary Differential Equations

Systems of ordinary differential equations

We have studied models of the form

$$y'(t) = F(y), \qquad y(0) = y_0 \tag{31}$$

this is an scalar ordinary differential equation (ODE).
We shall now study systems of ODEs. Especially we will consider numerical methods for systems of two ODEs on the form

$$\begin{aligned} y'(t) &= F(y,z), & y(0) &= y_0, \\ z'(t) &= G(y,z), & z(0) &= z_0. \end{aligned} \tag{32}$$

Here y_0 and z_0 are given initial states and F and G are smooth functions.

Rabbits and foxes

- Earlier we have studied the evolution of a rabbit population, and studied the Logistic model

$$y' = \alpha y(1 - y/\beta), \qquad y(0) = y_0 \tag{33}$$

where now y is the number of rabbits, $\alpha > 0$ denotes the growth rate and β is the carrying capacity.

- Note that this model is the same as the Exponential growth model if $\beta = \infty$
- We will consider the case where foxes are introduced to the model
- This model is called a predator-prey system, and is similar to models describing populations of fish (prey) and sharks (predators)

Fish and Sharks

The first mathematician to study predator-pray models was Vito Volterra. He studied shark-fish populations, but his results are valid for rabbit-fox populations as well.

- Let $F = F(t)$ denote the number of fishes and $S = S(t)$ the number of sharks for a given time t
- If there is no sharks we assume that the number of fishes follows the logistic model

$$F' = \alpha F(1 - F/\beta) \tag{34}$$

- Expressed with relative growth it reads

$$\frac{F'}{F} = \alpha(1 - F/\beta) \tag{35}$$

Fish and Sharks

- Introducing sharks to the model, we assume the relative growth rate of fish is reduced linearly with respect to S

$$\frac{F'}{F} = \alpha\left(1 - F/\beta - \gamma S\right), \tag{36}$$

where $\gamma > 0$

- or

$$F' = \alpha\left(1 - F/\beta - \gamma S\right) F \tag{37}$$

Fish and Sharks

- If there is no fish, we expect the number of sharks to decrease, and assume the relative change of sharks to be expressed as

$$\frac{S'}{S} = -\delta, \tag{38}$$

where $\delta > 0$ is the decay rate

- We also assume that the relative change of sharks increase linearly with the number of fish

$$\frac{S'}{S} = -\delta + \varepsilon F \tag{39}$$

Fish and Sharks

We now have a 2×2 system which predicts the development of fish- and shark- population

$$F' = \alpha(1 - F/\beta - \gamma S)F, \qquad F(0) = F_0, \qquad (40)$$

$$S' = (\varepsilon F - \delta)S, \qquad S(0) = S_0. \qquad (41)$$

- In practice the parameters α, β, γ and ε, and initial values F_0 and S_0 must be determined with some estimation methods

Numerical method; Unlimited resources

- First we study the system (40)-(41) with $\beta = \infty$, i.e. unlimited resources of food and space for the fish
- For the other parameters we choose $\alpha = 2, \quad \gamma = 1/2, \quad \varepsilon = 1 \quad$ and $\quad \delta = 1$, which gives the system

$$
\begin{aligned}
F' &= (2 - S)F, \qquad F(0) = F_0, &(42)\\
S' &= (F - 1)S, \qquad S(0) = S_0. &(43)
\end{aligned}
$$

- We introduce $\Delta t > 0$ and define $t_n = n\Delta t$, and let F_n and S_n denote approximations of $F(t_n)$ and $S(t_n)$ respectively

Numerical method

- The derivatives, F' and S', are approximated with

$$\frac{F(t_{n+1}) - F(t_n)}{\Delta t} \approx F'(t_n) \quad \text{and} \quad \frac{S(t_{n+1}) - S(t_n)}{\Delta t} \approx S'(t_n),$$

  which correspond to the explicit scheme
- The numerical scheme can then be written

$$\frac{F_{n+1} - F_n}{\Delta t} = (2 - S_n)F_n \qquad (44)$$

$$\frac{S_{n+1} - S_n}{\Delta t} = (F_n - 1)S_n \qquad (45)$$

Numerical method

- This can then be rewritten on an explicit form

$$
\begin{aligned}
F_{n+1} &= F_n + \Delta t (2 - S_n) F_n \qquad (46) \\
S_{n+1} &= S_n + \Delta t (F_n - 1) S_n \qquad (47)
\end{aligned}
$$

- When F_0 and S_0 are given, this formula gives us F_1 and S_1 by setting $n = 0$, and then we can compute F_2 and S_2 by putting $n = 1$ in the formula, and so on
- In Figure 5 we have tested the explicit scheme (46)-(47) with $F_0 = 1.9$, $S_0 = 0.1$ and $\Delta t = 1/1000$

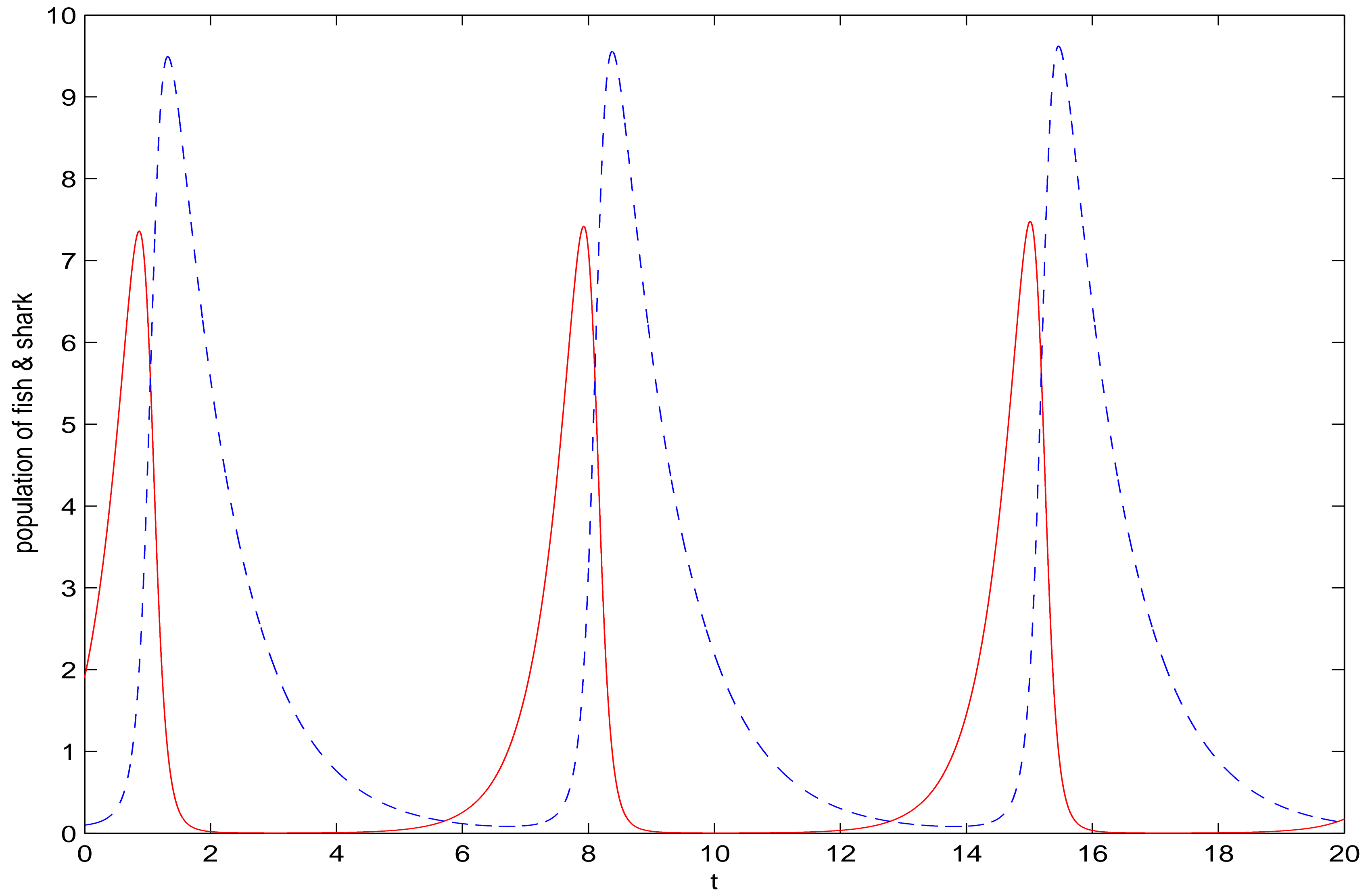

Figure 5: The solid curve is the solution for F, and the dashed curve is the solution for S.

Numerical methods; limited resources

- We do the same as above, but use $\beta = 2$, which corresponds to quite limited resources
- The system now reads

$$
\begin{aligned}
F' &= (2 - F - S)F, \quad F(0) = F_0, & (48)\\
S' &= (F - 1)S, \quad S(0) = S_0 & (49)
\end{aligned}
$$

- Similar to above we can define an explicit numerical scheme

$$
\begin{aligned}
F_{n+1} &= F_n + \Delta t(2 - F_n - S_n)F_n, & (50)\\
S_{n+1} &= S_n + \Delta t(F_n - 1)S_n & (51)
\end{aligned}
$$

- The results for $F_0 = 1.9$, $S_0 = 0.1$ and $\Delta t = 1/1000$ are shown in Figure 6

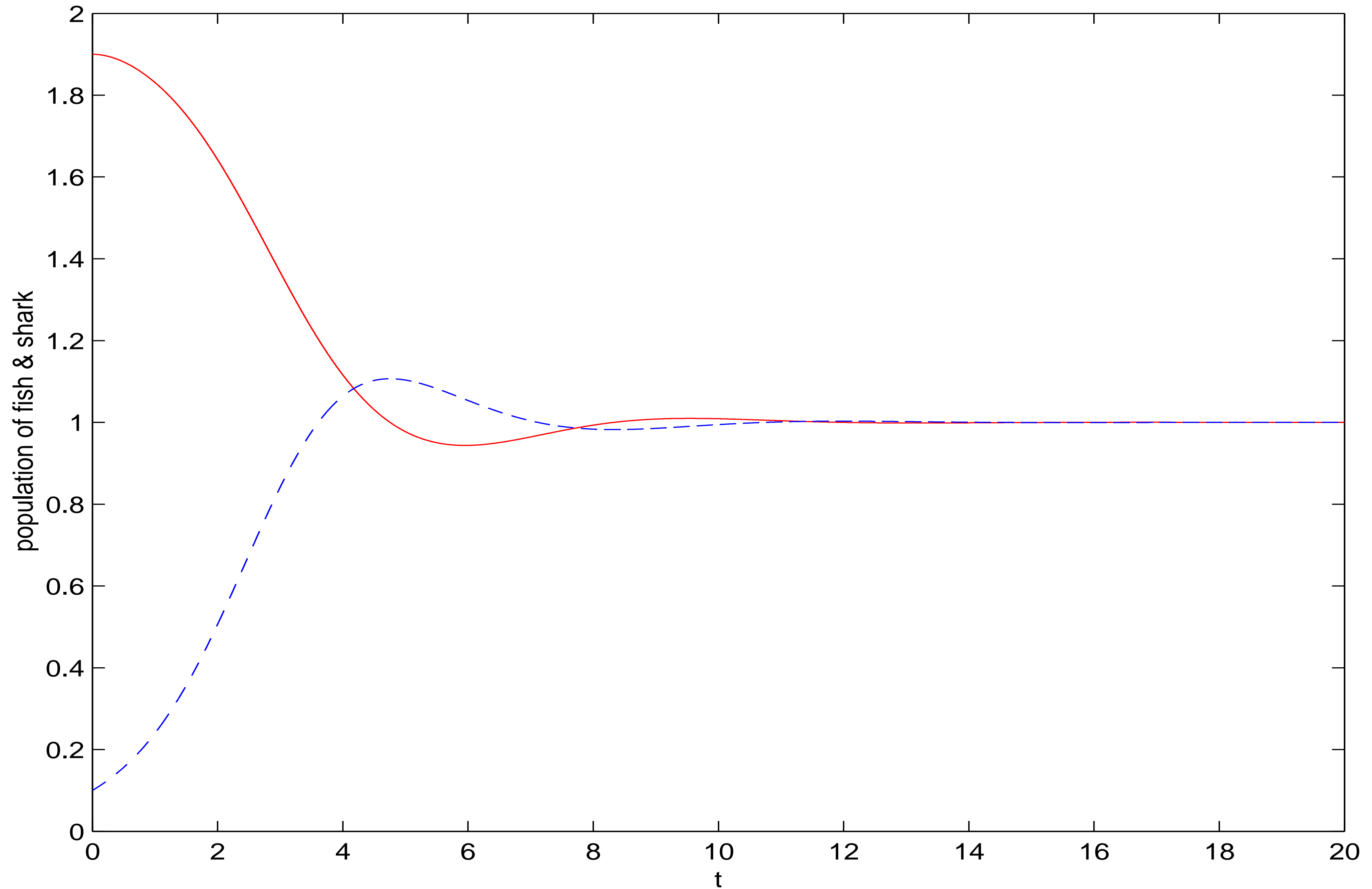

Figure 6: The solution for F is the solid curve, whereas the solution for S is the dashed curve.

Numerical methods

- We see from Figure 5 that the solutions for both $F(t)$ and $S(t)$ seem to be periodic
- From Figure 6 it seems that the solutions converge to an equilibrium solution represented by $S = F = 1$
- Therefore it is interesting to notice that, different parameter values can give different quantitative behavior of the solution

Phase plane analysis

We shall now study a simplified version of the fish-shark model

$$F'(t) = 1 - S(t), \quad F(0) = F_0,$$
$$S'(t) = F(t) - 1, \quad S(0) = S_0. \tag{52}$$

- Using the notation as above an explicit numerical scheme for this problem reads

$$F_{n+1} = F_n + \Delta t (1 - S_n),$$
$$S_{n+1} = S_n + \Delta t (F_n - 1), \tag{53}$$

  where F_0 and S_0 are given initial states
- Figure 7 show a solution of this scheme when $F_0 = 0.9$, $S_0 = 0.1$ and $\Delta t = 1/1000$

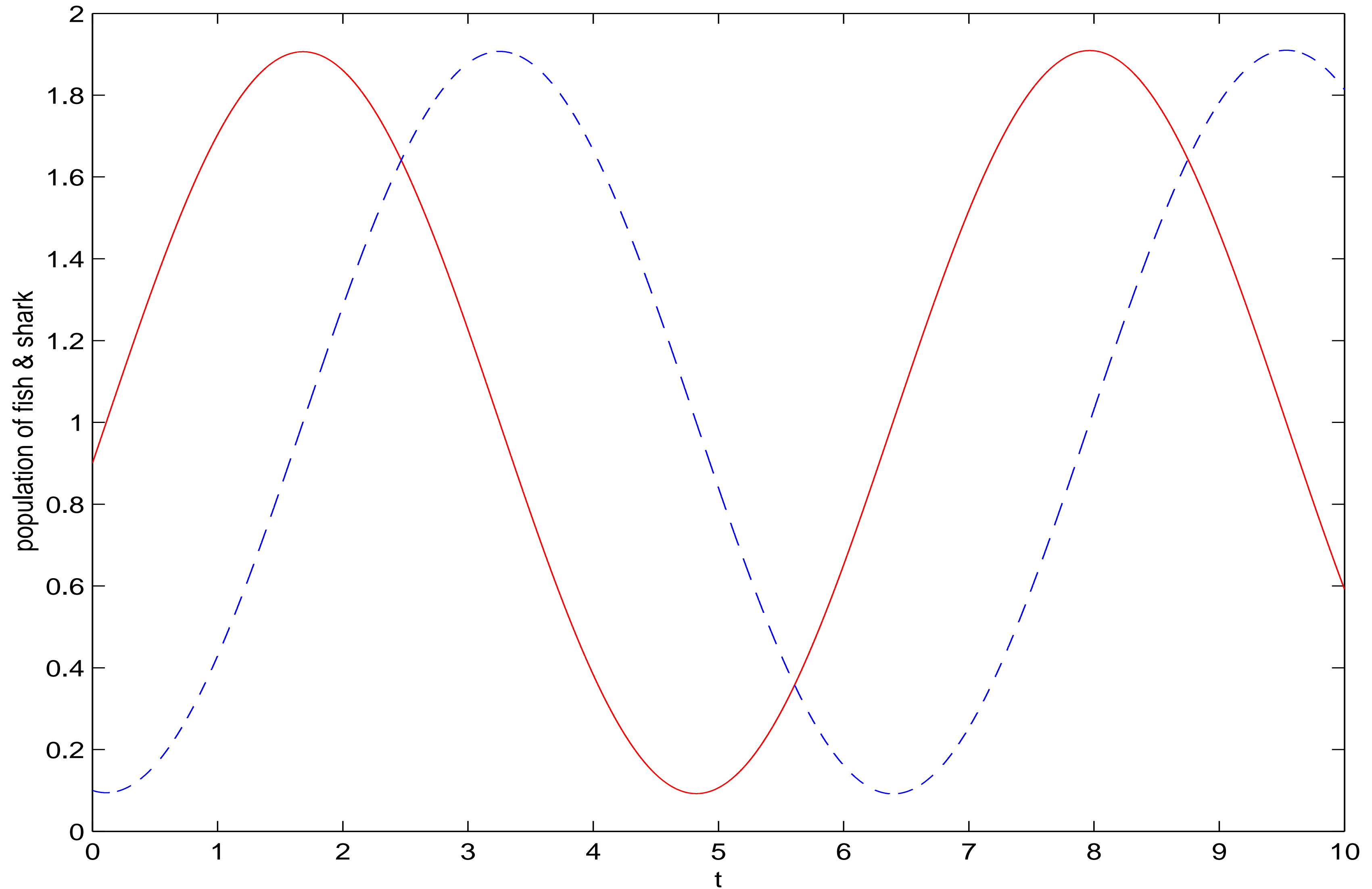

Figure 7: The solution for F is the solid curve, whereas the solution for S is the dashed curve.

Phase plane analysis

- The solution of (52) seems to be periodic like the solution of (42)-(43)
- In order to study how F and S interact we will plot the solution in the $F-S$ coordinate system, i.e. we plot the points (F_n, S_n) for all n-values
- In Figure 8 we plot the solution of (53) in the $F-S$ coordinate system, with the same specifications as above ($F_0 = 0.9$, $S_0 = 0.1$, $\Delta t = 1/1000$)
- In Figure 9 we do the same, but $\Delta t = 1/100$

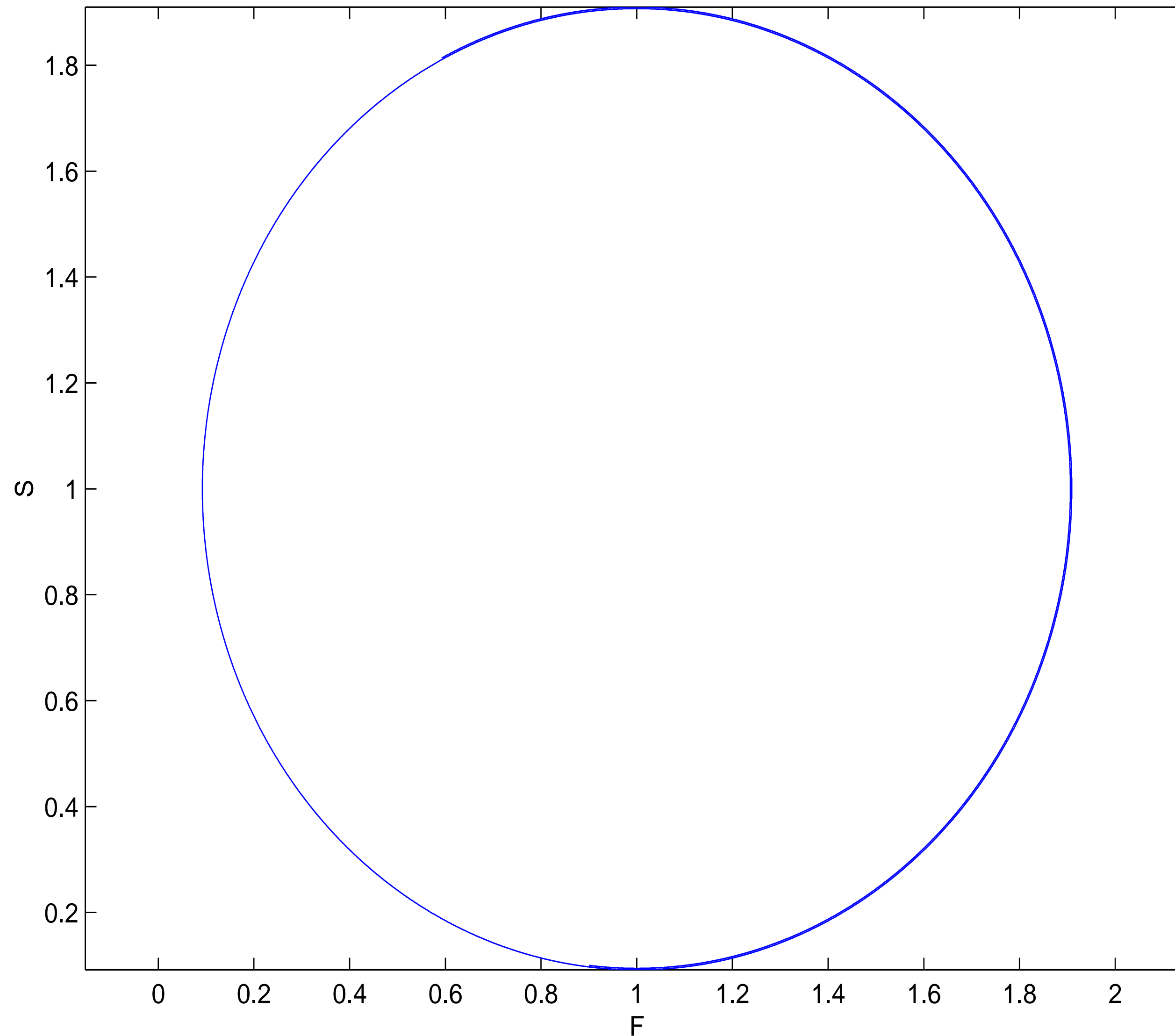

Figure 8: Explicit scheme (53) using $\Delta t = 1/1000$, $F_0 = 0.9$ and $S_0 = 0.1$, plotted in the F-S coordinate system

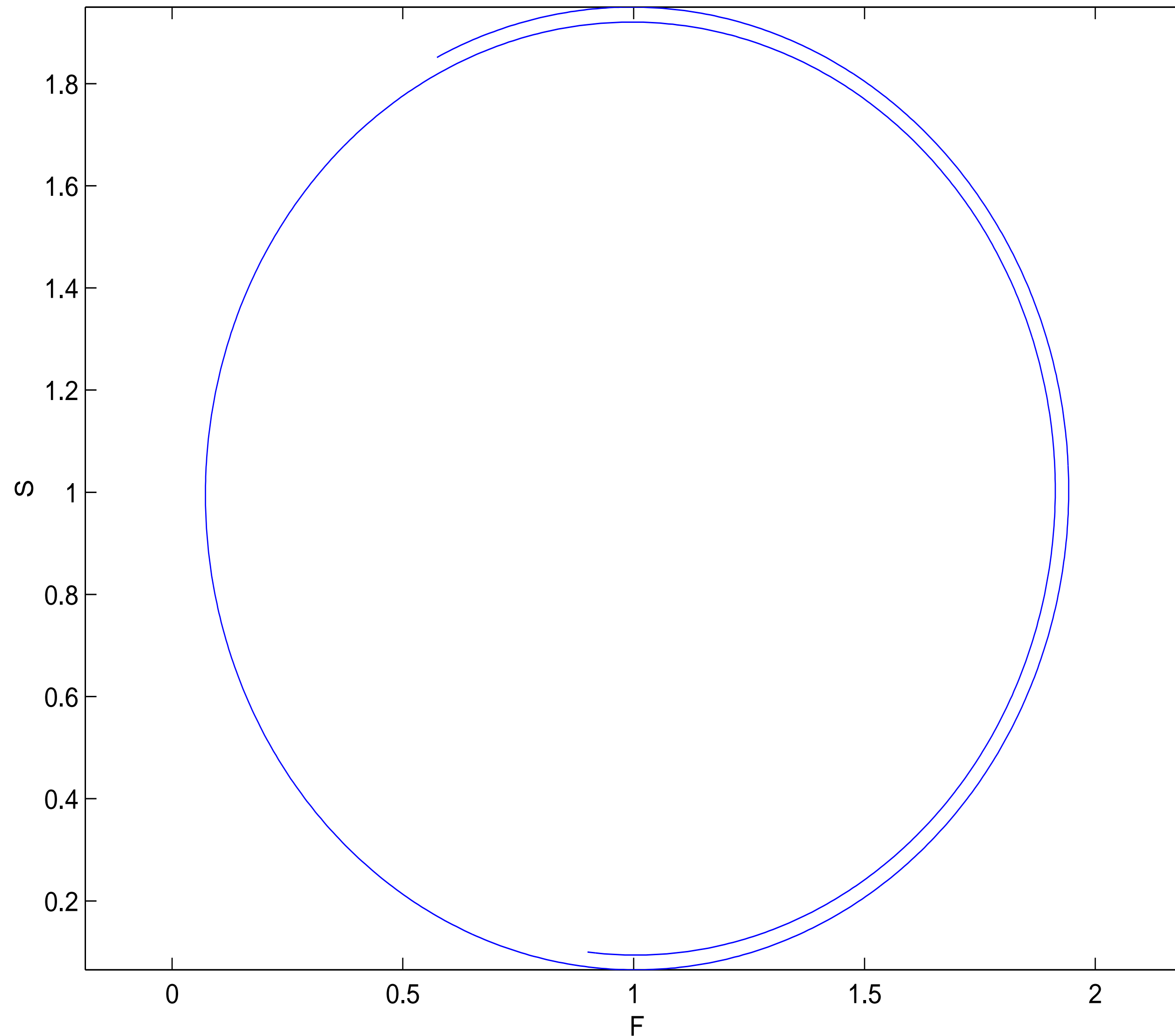

Figure 9: Explicit scheme (53) using $\Delta t = 1/100$, $F_0 = 0.9$ and $S_0 = 0.1$, plotted in the F-S coordinate system

```
def explicit_euler(dt, T, ic):
    # Store solutions in list of tuples: one tuple for each time
    times = [0, ]
    solutions = [ic, ]

    # Extract initial conditions
    (F0, S0) = ic

    # Start time-loop: let F_, S_ be the previous solutions and F, S
    # be the current solutions
    t = dt
    (F_, S_) = ic
    while (t <= T):
        # Define the new solutions from the old solutions
        F = F_ + dt*(1 - S_)
        S = S_ + dt*(F_ - 1)
        # Store the new solutions
        solutions += [(F, S)]
        times += [t]
        # Prepare for next iteration by updating the previous values
        (F_, S_) = (F, S)
        t += dt

    return times, solutions
```

Crank-Nicolson scheme

The Crank-Nicolson scheme for the system

$$\begin{aligned} F'(t) &= 1 - S(t), \quad F(0) = F_0, \\ S'(t) &= F(t) - 1, \quad S(0) = S_0. \end{aligned} \tag{54}$$

reads

$$\begin{aligned} \frac{F_{n+1} - F_n}{\Delta t} &= \frac{1}{2}\left[(1 - S_n) + (1 - S_{n+1})\right], \\ \frac{S_{n+1} - S_n}{\Delta t} &= \frac{1}{2}\left[(F_n - 1) + (F_{n+1} - 1)\right]. \end{aligned} \tag{55}$$

Crank-Nicolson scheme

The Crank-Nicolson scheme can be rewritten as

$$\begin{aligned} F_{n+1}+\tfrac{\Delta t}{2}S_{n+1} &= F_n+\Delta t-\tfrac{\Delta t}{2}S_n, \\ -\tfrac{\Delta t}{2}F_{n+1}+S_{n+1} &= S_n-\Delta t+\tfrac{\Delta t}{2}F_n. \end{aligned} \tag{56}$$

- We see that when F_n and S_n are given, we have to solve a 2×2 system of linear equations, to find F_{n+1} and S_{n+1}

Crank-Nicolson scheme

Define

$$\mathbf{A} = \begin{bmatrix} 1 & \Delta t/2 \\ -\Delta t/2 & 1 \end{bmatrix}, \tag{57}$$

and

$$\mathbf{b}_n = \begin{pmatrix} F_n + \Delta t - \frac{\Delta t}{2} S_n \\ S_n - \Delta t + \frac{\Delta t}{2} F_n \end{pmatrix}. \tag{58}$$

Crank-Nicolson scheme

Solving (56) for one time-step can now be done by:

- Solve

$$\mathbf{A}\mathbf{x}_{n+1} = \mathbf{b}_n, \tag{59}$$

  where x_{n+1} is the unknown vector with two components
- The new solution for F and S is then

$$\begin{pmatrix} F_{n+1} \\ S_{n+1} \end{pmatrix} = \mathbf{x}_{n+1} \tag{60}$$

Crank-Nicolson scheme

In general, a 2×2 matrix

$$\mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \tag{61}$$

is non-singular if $ad \neq cb$. And when $ad \neq cb$ the inverse is given by

$$\mathbf{B}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}. \tag{62}$$

Crank-Nicolson scheme

- In order for the problem to be well defined we need the matrix $\mathbf{A}$ to be non-singular
- But we have that

$$\text{det}(\mathbf{A}) \;=\; 1+\Delta t^2/4, \tag{63}$$

which ensures $\text{det}(\mathbf{A}) > 0$ for all values of Δt, and $\mathbf{A}$ is always non-singular

Crank-Nicolson scheme

- For the matrix (57), the inverse is given by

$$\mathbf{A}^{-1} = \frac{1}{1+\Delta t^2/4}\begin{bmatrix} 1 & -\Delta t/2 \\ \Delta t/2 & 1 \end{bmatrix} \tag{64}$$

- This fact together with (59) and (60) gives

$$\begin{pmatrix} F_{n+1} \\ S_{n+1} \end{pmatrix} = \frac{1}{1+\Delta t^2/4}\begin{bmatrix} 1 & -\Delta t/2 \\ \Delta t/2 & 1 \end{bmatrix}\begin{pmatrix} F_n + \Delta t - \frac{\Delta t}{2} S_n \\ S_n - \Delta t + \frac{\Delta t}{2} F_n \end{pmatrix}$$

Crank-Nicolson scheme

- We get

$$\begin{aligned} F_{n+1} &= \frac{1}{1+\Delta t^2/4}\left[\left(1-\Delta t^2/4\right)F_n + \Delta t\left(\frac{\Delta t}{2}+1\right) - \Delta t S_n\right] \\ S_{n+1} &= \frac{1}{1+\Delta t^2/4}\left[\left(1-\Delta t^2/4\right)S_n + \Delta t\left(\frac{\Delta t}{2}-1\right) + \Delta t F_n\right] \end{aligned} \quad (65)$$

- Figure 10 plots the solution of this scheme for $S_0 = 0.1$, $F_0 = 0.9$ and $\Delta t = 1/1000$, t is from $t = 0$ to $t = 10$ and the solution is plotted in the F-S coordinate system

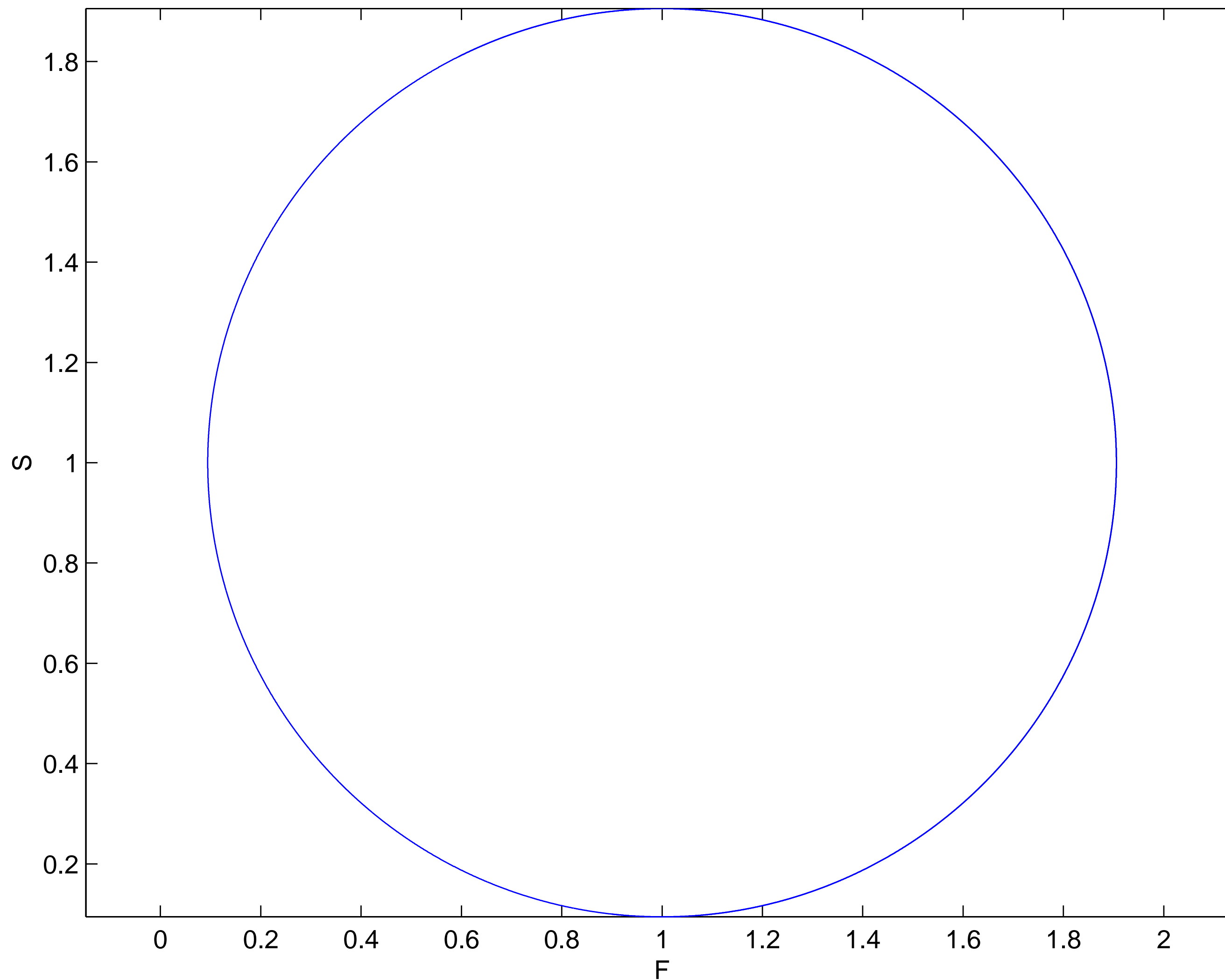

Figure 10: The numerical solution for the Crank-Nicholson scheme

Crank-Nicolson scheme

- In Figure 10 we observe that the solution again seems to form a perfect circle
- To study this closer we define, as above

$$r_n = (F_n - 1)^2 + (S_n - 1)^2 \tag{66}$$

- and study the relative change

$$\frac{r_N - r_0}{r_0} \tag{67}$$

in Table 10

Crank-Nicolson scheme

Δt	N	$\frac{r_N - r_0}{r_0}$
10^{-1}	10^2	$-2.6682 \cdot 10^{-16}$
10^{-2}	10^3	$-1.59986 \cdot 10^{-17}$
10^{-3}	10^4	$3.97982 \cdot 10^{-17}$
10^{-4}	10^5	$7.06021 \cdot 10^{-15}$

Table 3: The table shows Δt, the number of time steps N, and the "error" $\frac{r_N - r_0}{r_0}$.

Crank-Nicolson scheme

- We observe that the relative error $\frac{r_N - r_0}{r_0}$ is much smaller for the Crank-Nicolson scheme (66) than for the explicit scheme (53)
- We therefore conclude that the Crank-Nicolson scheme produces better solutions than the explicit scheme

Nonlinear Algebraic Equations

Nonlinear algebraic equations

In implicit methods we need to solve equations on the form

$$u_{n+1} - u_n = \Delta t\, g(u_{n+1}) \tag{68}$$

where Δt is a small number, we know that u_{n+1} is close to u_n. This will be a useful property later.
More generally, we want to solve eqatations on the form:

$$f(x) = 0, \tag{69}$$

where f is nonlinear. We assume that we have available a value x_0 close to the true solution x^* (, i.e. $f(x^*) = 0$).
We also assume that f has no other zeros in a small region around x^*.

The bisection method

Consider the function

$$f(x) = 2 + x - e^x \tag{70}$$

for x ranging from 0 to 3, see the graph in Figure 11.

- We want to find $x = x^*$ such that

$$f(x^*) = 0$$

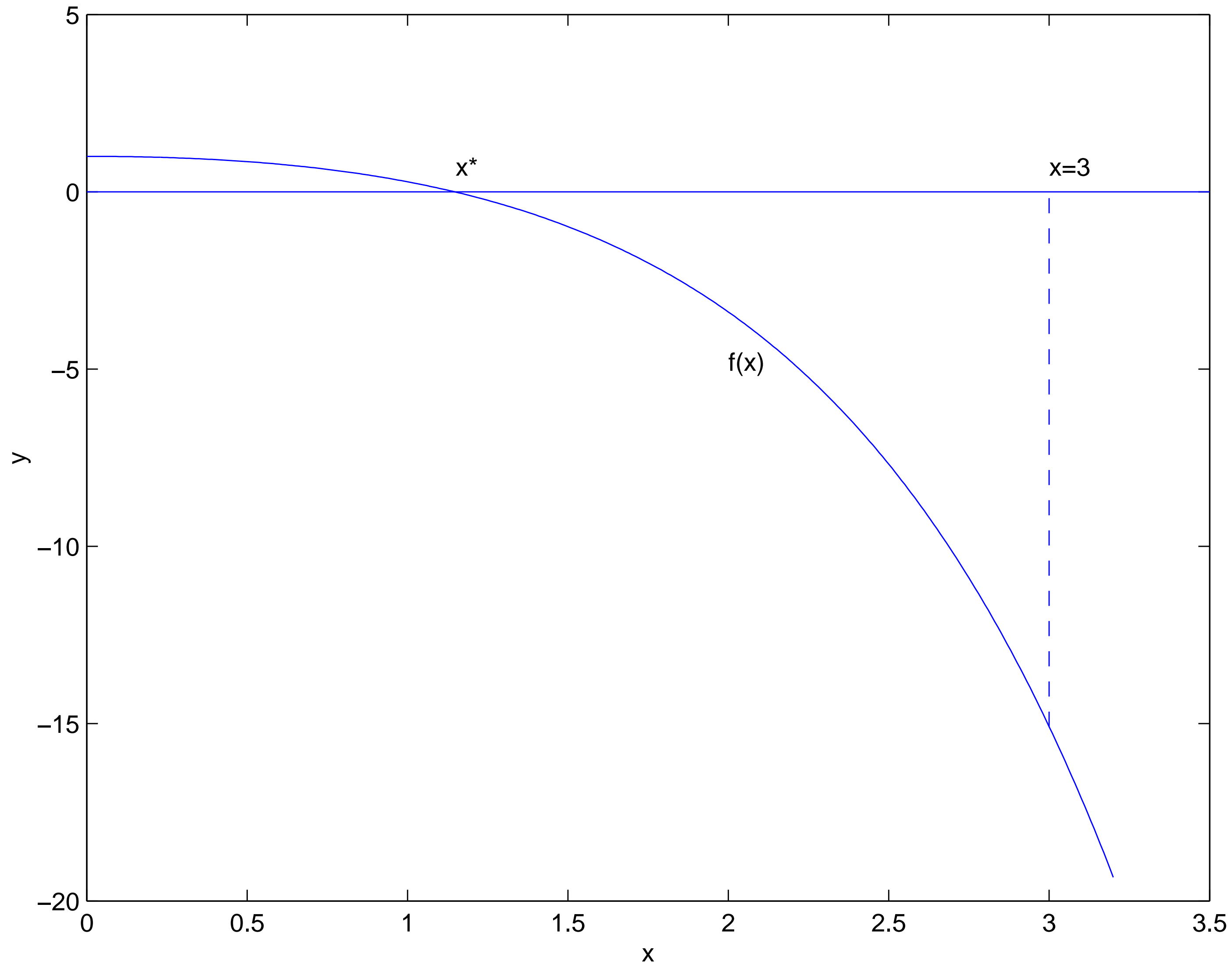

Figure 11: The graph of $f(x) = 2 + x - e^x$.

The bisection method

- An iterative method is to create a series $\{x_i\}$ of approximations of x^*, which hopefully converges towards x^*
- For the Bisection Method we choose the two first guesses x_0 and x_1 as the endpoints of the definition domain, i.e.

$$x_0 = 0 \quad \text{and} \quad x_1 = 3$$

- Note that $f(x_0) = f(0) > 0$ and $f(x_1) = f(3) < 0$, and therefore $x_0 < x^* < x_1$, provided that f is continuous
- We now define the mean value of x_0 and x_1

$$x_2 = \frac{1}{2}(x_0 + x_1) = \frac{3}{2}$$

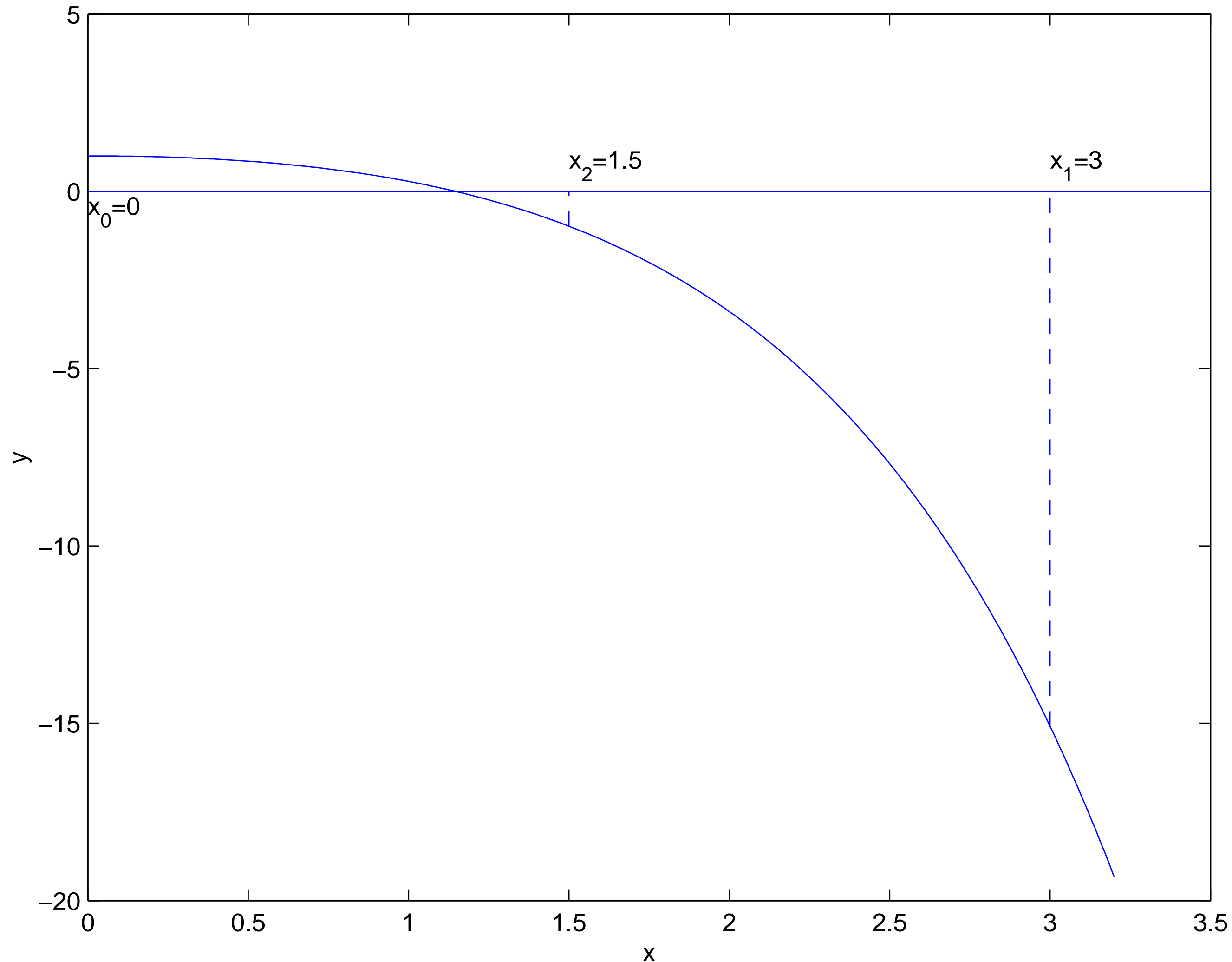

Figure 12: The graph of $f(x) = 2 + x - e^x$ and three values of f: $f(x_0)$, $f(x_1)$ and $f(x_2)$.

The bisection method

- We see that

$$f(x_2) = f(\frac{3}{2}) = 2 + 3/2 - e^{3/2} \; < \; 0,$$

- Since $f(x_0) > 0$ and $f(x_2) < 0$, we know that $x_0 < x^* < x_2$
- Therefore we define

$$x_3 = \frac{1}{2}(x_0 + x_2) = \frac{3}{4}$$

- Since $f(x_3) > 0$, we know that $x_3 < x^* < x_2$ (see Figure 13)
- This can be continued until $|f(x_n)|$ is sufficiently small

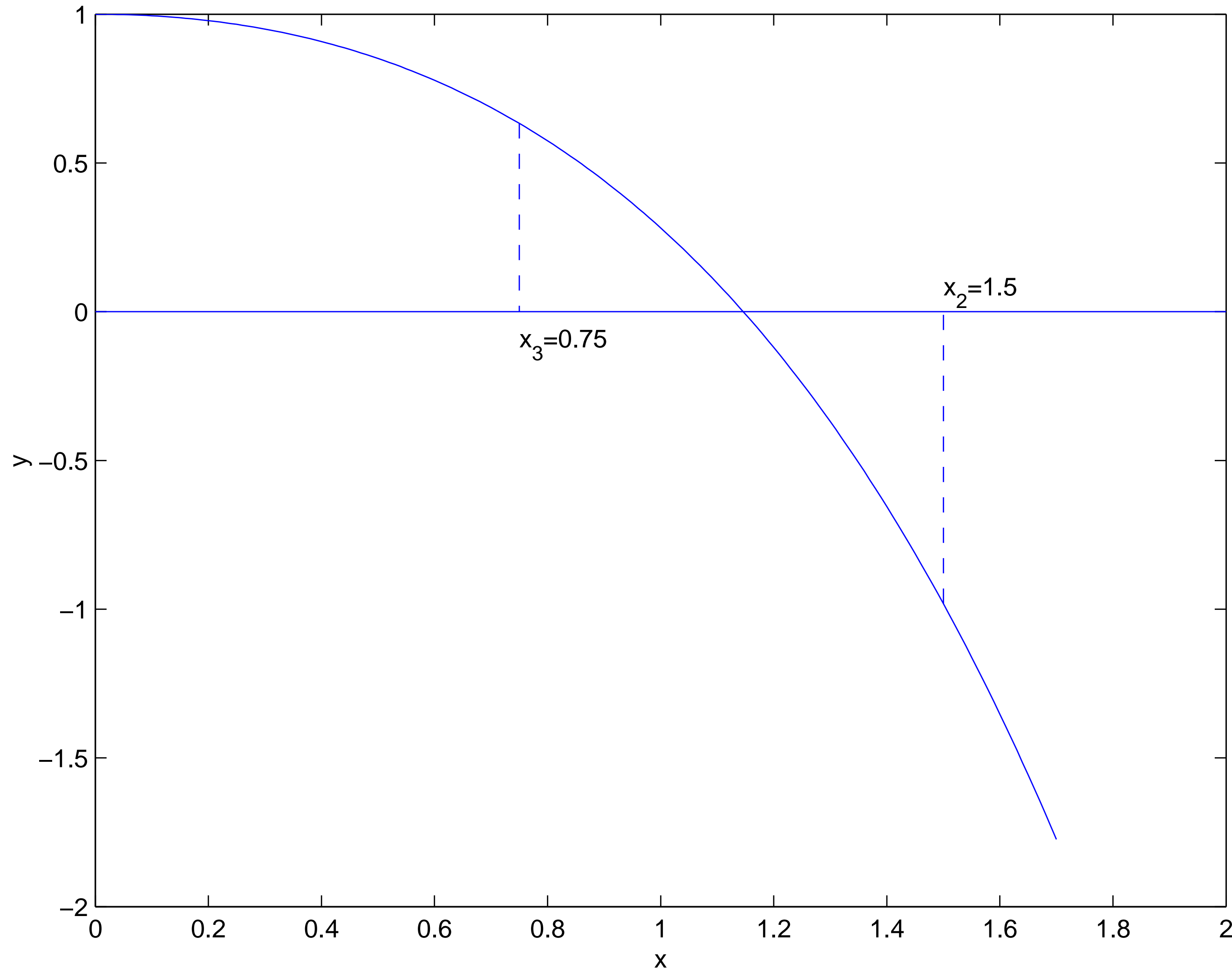

Figure 13: The graph of $f(x) = 2 + x - e^x$ and two values of f: $f(x_2)$ and $f(x_3)$.

The bisection method

Written in algorithmic form the Bisection method reads:

Algorithm 1. Given a, b such that $f(a)\cdot f(b)<0$ and given a tolerance ε. Define $c=\frac{1}{2}(a+b)$.

while $|f(c)|>\varepsilon$ **do**

  if $f(a)\cdot f(c)<0$

  then $b=c$

  else $a=c$

  $c:=\frac{1}{2}(a+b)$

end

Example 11

Find the zeros for

$$f(x) = 2 + x - e^x$$

using Algorithm 1 and choose $a = 0$, $b = 3$ and $\varepsilon = 10^{-6}$.

- In Table 4 we show the number of iterations i, c and $f(c)$
- The number of iterations, i, refers to the number of times we pass through the while-loop of the algorithm

i	c	$f(c)$
1	1.500000	-0.981689
2	0.750000	0.633000
4	1.312500	-0.402951
8	1.136719	0.0201933
16	1.146194	$-2.65567 \cdot 10^{-6}$
21	1.146193	$4.14482 \cdot 10^{-7}$

Table 4: Solving the nonlinear equation $f(x) = 2 + x - e^x = 0$ by using the bisection method; the number of iterations i, c and $f(c)$.

Example 11

- We see that we get sufficient accuracy after 21 iterations
- The next slide show the C program that is used to solve this problem
- The entire computation uses $5.82 \cdot 10^{-6}$ seconds on a Pentium III 1GHz processor
- Even if this quite fast, even faster algorithms exists.

```
#include <stdio.h>
#include <math.h>

double f (double x) { return 2.0+x-exp(x); }
/* we define function 'fabs' for calculating absolute values */
inline double fabs (double  r) { return ( (r >= 0.0) ? r : -r ); }

int main (int nargs, const char** args)
{
  double epsilon = 1.0e-6;
  double a, b, c, fa, fc;
  a = 0.; b = 3.;
  fa = f(a);
  c = 0.5*(a+b);
  while (fabs(fc=(f(c))) > epsilon) {
    if ((fa*fc) < 0) {
      b = c;
    }
    else {
      a = c;
      fa = fc;
    }
    c = 0.5*(a+b);
  }
```

```
def bisection(f, a, b, tolerance):

    assert (f(a)*f(b) < 0), "Input does not satisfy ansatz!"

    c = 0.5*(a + b)
    k = 1

    points = [c, ]
    values = [f(c), ]

    while (abs(f(c)) > tolerance):

        if f(a)*f(c) < 0:
            b = c
        else:
            a = c
        c = 0.5*(a + b)

        points += [c]
        values += [f(c)]
        k += 1

    return points, values
```

Newton's method

- Recall that we have assumed that we have a good initial guess x_0 close to x^* (where $f(x^*) = 0$)
- We will also assume that we have a small region around x^* where f has only one zero, and that $f'(x) \neq 0$
- Taylor series expansion around $x = x_0$ yields

$$f(x_0 + h) = f(x_0) + hf'(x_0) + O(h^2) \tag{71}$$

- Thus, for small h we have

$$f(x_0 + h) \approx f(x_0) + hf'(x_0) \tag{72}$$

Newton's method

- We want to choose the step h such that $f(x_0+h) \approx 0$
- By (72) this can be done by choosing h such that

$$f(x_0) + hf'(x_0) = 0$$

- Solving this gives

$$h = -\frac{f(x_0)}{f'(x_0)}$$

- We therefore define

$$x_1 \stackrel{\text{def}}{=} x_0 + h = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{73}$$

Newton's method

- We test this on the example studied above with $f(x) = 2 + x - e^x$ and $x_0 = 3$
- We have that

$$f'(x) = 1 - e^x$$

- Therefore

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = 3 - \frac{5 - e^3}{1 - e^3} = 2.2096$$

- We see that

$$|f(x_0)| = \ |f(3)| \approx 15.086 \quad \text{and} \quad |f(x_1)| = \ |f(2.2096)| \approx 4.902$$

i.e, the value of f is significantly reduced

Newton's method

We can now repeat the above procedure and define

$$x_2 \overset{\text{def}}{=} x_1 - \frac{f(x_1)}{f'(x_1)}, \tag{74}$$

and in algorithmic form Newton's method reads:

Algorithm 2. Given an initial approximation x_0 and a tolerance ε.

$k = 0$

while $|f(x_k)| > \varepsilon$ **do**

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

$k = k + 1$

end

Newton's method

In Table 5 we show the results generated by Newton's method on the above example.

k	x_k	$f(x_k)$
1	2.209583	-4.902331
2	1.605246	-1.373837
3	1.259981	-0.265373
4	1.154897	$-1.880020 \cdot 10^{-2}$
5	1.146248	$-1.183617 \cdot 10^{-4}$
6	1.146193	$-4.783945 \cdot 10^{-9}$

Table 5: Solving the nonlinear equation $f(x) = 2 + x - e^x = 0$ by using Algorithm 108 and $\varepsilon = 10^{-6}$; the number of iterations k, x_k and $f(x_k)$.

Newton's method

- We observe that the convergence is much faster for Newton's method than for the Bisection method
- Generally, Newton's method converges faster than the Bisection method
- This will be studied in more detail in Project 1

Example 12

Let

$$f(x) = x^2 - 2,$$

and find x^* such that $f(x^*) = 0$.

- Note that one of the exact solutions is $x^* = \sqrt{2}$
- Newton's method for this problem reads

$$x_{k+1} = x_k - \frac{x_k^2 - 2}{2x_k}$$

- or

$$x_{k+1} = \frac{x_k^2 + 2}{2x_k}$$

Example 12

If we choose $x_0 = 1$, we get

$$x_1 = 1.5,$$
$$x_2 = 1.41667,$$
$$x_3 = 1.41422.$$

Comparing this with the exact value

$$x^* = \sqrt{2} \approx 1.41421,$$

we see that a very accurate approximation is obtained in only 3 iterations.

An alternative derivation

- The Taylor series expansion of f around x_0 is given by

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + O((x - x_0)^2)$$

- Let $F_0(x)$ be a linear approximation of f around x_0:

$$F_0(x) = f(x_0) + (x - x_0)f'(x_0)$$

- $F_0(x)$ approximates f around x_0 since

$$F_0(x_0) = f(x_0) \quad \text{and} \quad F_0'(x_0) = f'(x_0)$$

- We now define x_1 to be such that $F(x_1) = 0$, i.e.

$$f(x_0) + (x_1 - x_0)f'(x_0) = 0$$

An alternative derivation

- Then we get

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)},$$

which is identical to the iteration obtained above

- We repeat this process, and define a linear approximation of f around x_1

$$F_1(x) = f(x_1) + (x - x_1)f'(x_1)$$

- x_2 is defined such that $F_1(x_2) = 0$, i.e.

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

An alternative derivation

- Generally we get

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

- This process is illustrated in Figure 14

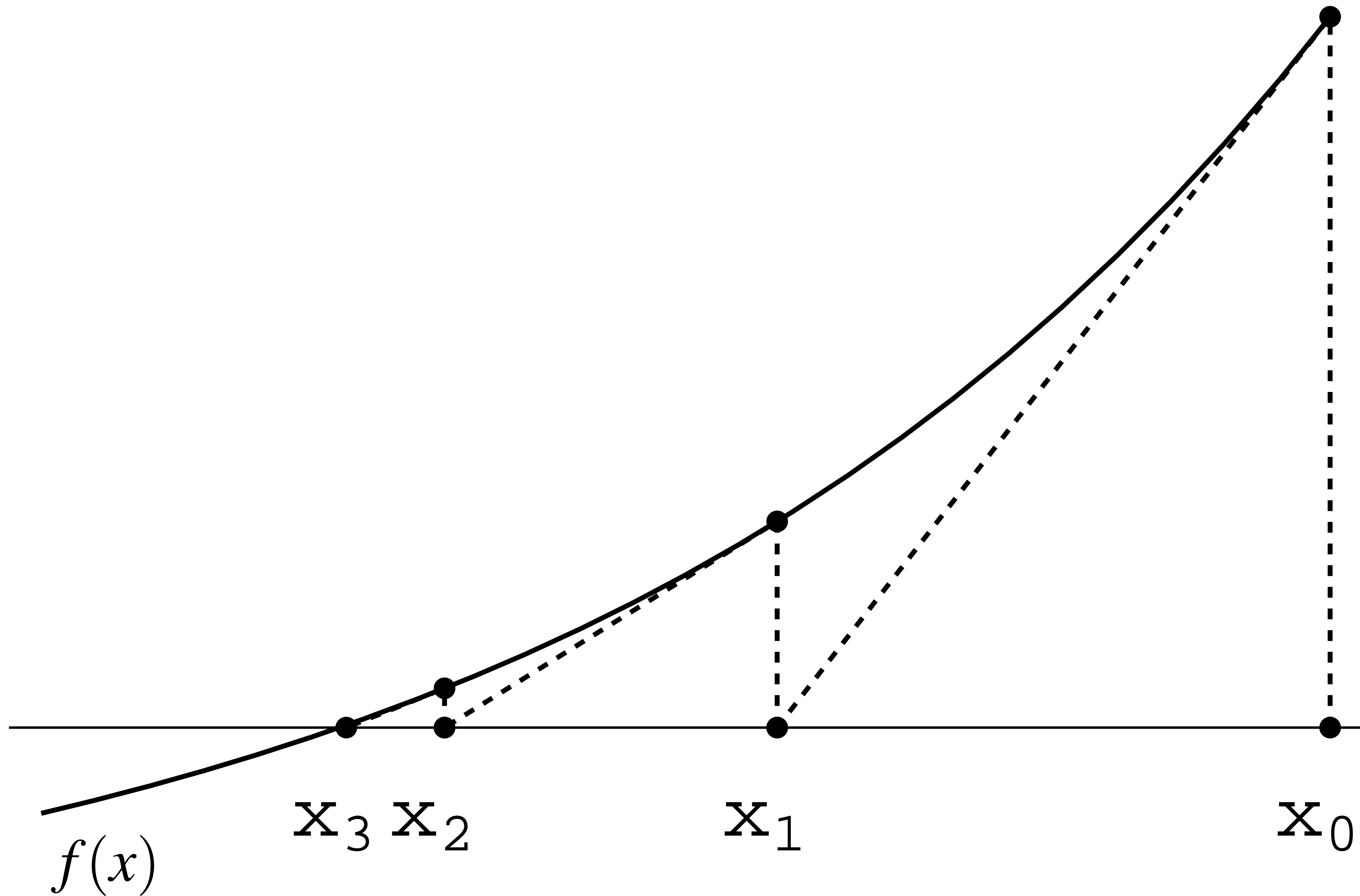

Figure 14: Graphical illustration of Newton's method.

```
def newton(f, df, y, tolerance):

    k = 1  # iteration counter
    c = y  # initial guess

    points = [c, ]
    values = [f(c), ]
    while (abs(f(c)) > tolerance):
        c = c - f(c)/df(c)

        points += [c]
        values += [f(c)]
        k += 1

    return points, values

# Define f (want f(x) == 0)
def f(x):
    return x + 0.1*x**3 - 1.

# The derivative of f
def df(x):
    return 1 + 3*0.1*x**2
```

The Secant method

- The secant method is similar to Newton's method, but the linear approximation of f is defined differently
- Now we assume that we have two values x_0 and x_1 close to x^*, and define the linear function $F_0(x)$ such that

$$F_0(x_0) = f(x_0) \quad \text{and} \quad F_0(x_1) = f(x_1)$$

- The function $F_0(x)$ is therefore given by

$$F_0(x) = f(x_1) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_1)$$

- $F_0(x)$ is called the linear interpolant of f

The Secant method

- Since $F_0(x) \approx f(x)$, we can compute a new approximation x_2 to x^* by solving the linear equation

$$F(x_2) = 0$$

- This means that we must solve

$$f(x_1) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x_2 - x_1) = 0,$$

with respect to x_2 (see Figure 15)

- This gives

$$x_2 = x_1 - \frac{f(x_1)(x_1 - x_0)}{f(x_1) - f(x_0)}$$

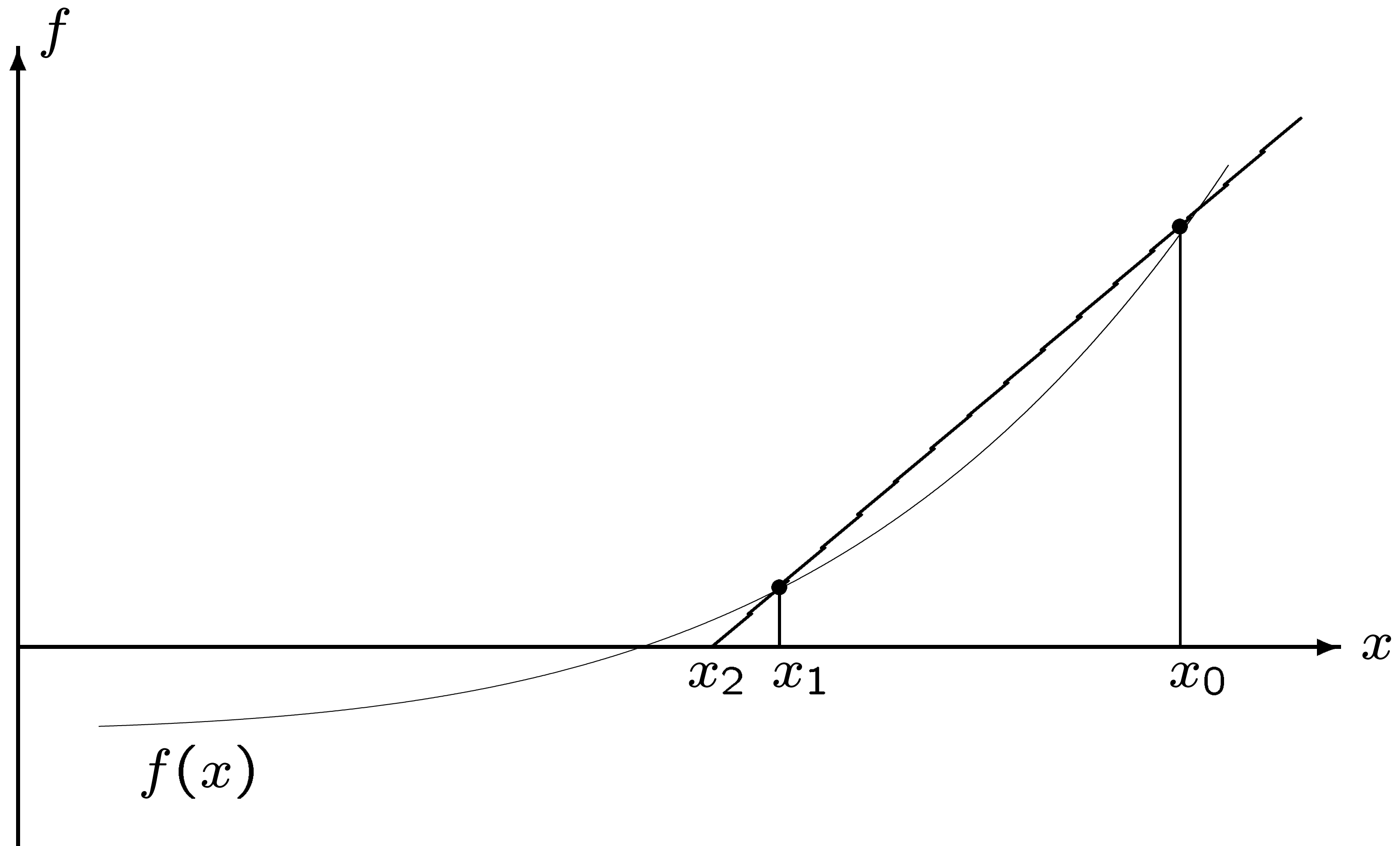

Figure 15: The figure shows a function $f = f(x)$ and its linear interpolant F between x_0 and x_1.

The Secant method

Following the same procedure as above we get the iteration

$$x_{k+1} = x_k - \frac{f(x_k)(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})},$$

and the associated algorithm reads

Algorithm 3. Given two initial approximations x_0 and x_1 and a tolerance ε.

$k = 1$

while $|f(x_k)| > \varepsilon$ **do**

$$x_{k+1} = x_k - f(x_k)\frac{(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}$$

$k = k + 1$

end

Example 13

Let us apply the Secant method to the equation

$$f(x) = 2 + x - e^x = 0,$$

studied above. The two initial values are $x_0 = 0$, $x_1 = 3$, and the stopping criteria is specified by $\varepsilon = 10^{-6}$.

- Table 6 show the number of iterations k, x_k and $f(x_k)$ as computed by Algorithm 3
- Note that the convergence for the Secant method is slower than for Newton's method, but faster than for the Bisection method

k	x_k	$f(x_k)$
2	0.186503	0.981475
3	0.358369	0.927375
4	3.304511	-21.930701
5	0.477897	0.865218
6	0.585181	0.789865
7	1.709760	-1.817874
8	0.925808	0.401902
9	1.067746	0.158930
10	1.160589	$-3.122466 \cdot 10^{-2}$
11	1.145344	$1.821544 \cdot 10^{-3}$
12	1.146184	$1.912908 \cdot 10^{-5}$
13	1.146193	$-1.191170 \cdot 10^{-8}$

Table 6: The Secant method applied with $f(x) = 2 + x - e^x$=0.

Example 14

Find a zero of

$$f(x) = x^2 - 2,$$

which has a solution $x^* = \sqrt{2}$.

- The general step of the secant method is in this case

$$\begin{aligned}
x_{k+1} &= x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} \\
&= x_k - (x_k^2 - 2)\frac{x_k - x_{k-1}}{x_k^2 - x_{k-1}^2} \\
&= x_k - \frac{x_k^2 - 2}{x_k + x_{k-1}} \\
&= \frac{x_k x_{k-1} + 2}{x_k + x_{k-1}}
\end{aligned}$$

Example 14

- By choosing $x_0 = 1$ and $x_1 = 2$ we get

$$x_2 = 1.33333$$
$$x_3 = 1.40000$$
$$x_4 = 1.41463$$

- This is quite good compared to the exact value

$$x^* = \sqrt{2} \approx 1.41421$$

- Recall that Newton's method produced the approximation 1.41422 in three iterations, which is slightly more accurate

A nonlinear system

We start our study of nonlinear equations, by considering a nonlinear system of ordinary differential equations

$$\begin{aligned} u' &= -v^3, \qquad u(0) = u_0, \\ v' &= u^3, \qquad v(0) = v_0. \end{aligned} \tag{75}$$

An implicit Euler scheme for this system reads

$$\frac{u_{n+1} - u_n}{\Delta t} = -v_{n+1}^3, \quad \frac{v_{n+1} - v_n}{\Delta t} = u_{n+1}^3, \tag{76}$$

which can be rewritten on the form

$$\begin{aligned} u_{n+1} + \Delta t\, v_{n+1}^3 - u_n &= 0, \\ v_{n+1} - \Delta t\, u_{n+1}^3 - v_n &= 0. \end{aligned} \tag{77}$$

A nonlinear system

- Observe that in order to compute (u_{n+1}, v_{n+1}) based on (u_n, v_n), we need to solve a nonlinear system of equations

We would like to write the system on the generic form

$$\begin{aligned} f(x,y) &= 0, \\ g(x,y) &= 0. \end{aligned} \tag{78}$$

This is done by setting

$$\begin{aligned} f(x,y) &= x + \Delta t\, y^3 - \alpha, \\ g(x,y) &= y - \Delta t\, x^3 - \beta, \end{aligned} \tag{79}$$

$\alpha = u_n$ and $\beta = v_n$.

Newton's method

When deriving Newton's method for solving a scalar equation

$$p(x) = 0 \qquad (80)$$

we exploited Taylor series expansion

$$p(x_0+h) = p(x_0)+hp'(x_0)+O(h^2), \qquad (81)$$

to make a linear approximation of the function p, and solve the linear approximation of (80). This lead to the iteration

$$x_{k+1} = x_k - \frac{p(x_k)}{p'(x_k)}. \qquad (82)$$

Newton's method

We shall try to extend Newton's method to systems of equations on the form

$$\begin{aligned} f(x,y) &= 0, \\ g(x,y) &= 0. \end{aligned} \tag{83}$$

The Taylor-series expansion of a smooth function of two variables $F(x,y)$, reads

$$\begin{aligned} F(x+\Delta x, y+\Delta y) &= F(x,y) + \Delta x \frac{\partial F}{\partial x}(x,y) + \Delta y \frac{\partial F}{\partial y}(x,y) \\ &\quad + O(\Delta x^2, \Delta x \Delta y, \Delta y^2). \end{aligned} \tag{84}$$

Newton's method

Using Taylor expansion on (83) we get

$$\begin{aligned} f(x_0+\Delta x, y_0+\Delta y) &= f(x_0,y_0)+\Delta x\frac{\partial f}{\partial x}(x_0,y_0)+\Delta y\frac{\partial f}{\partial y}(x_0,y_0) \\ &\quad +O(\Delta x^2,\Delta x\Delta y,\Delta y^2), \qquad (85) \end{aligned}$$

and

$$\begin{aligned} g(x_0+\Delta x, y_0+\Delta y) &= g(x_0,y_0)+\Delta x\frac{\partial g}{\partial x}(x_0,y_0)+\Delta y\frac{\partial g}{\partial y}(x_0,y_0) \\ &\quad +O(\Delta x^2,\Delta x\Delta y,\Delta y^2). \qquad (86) \end{aligned}$$

Newton's method

Since we want Δx and Δy to be such that

$$\begin{aligned} f(x_0+\Delta x, y_0+\Delta y) &\approx 0, \\ g(x_0+\Delta x, y_0+\Delta y) &\approx 0, \end{aligned} \tag{87}$$

we define Δx and Δy to be the solution of the linear system

$$\begin{aligned} f(x_0,y_0)+\Delta x\frac{\partial f}{\partial x}(x_0,y_0)+\Delta y\frac{\partial f}{\partial y}(x_0,y_0) &= 0, \\ g(x_0,y_0)+\Delta x\frac{\partial g}{\partial x}(x_0,y_0)+\Delta y\frac{\partial g}{\partial y}(x_0,y_0) &= 0. \end{aligned} \tag{88}$$

Remember here that x_0 and y_0 are known numbers, and therefore $f(x_0,y_0)$, $\frac{\partial f}{\partial x}(x_0,y_0)$ and $\frac{\partial f}{\partial y}(x_0,y_0)$ are known numbers as well. Δx and Δy are the unknowns.

Newton's method

(88) can be written on the form

$$\begin{pmatrix} \frac{\partial f_0}{\partial x} & \frac{\partial f_0}{\partial y} \\ \frac{\partial g_0}{\partial x} & \frac{\partial g_0}{\partial y} \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} = - \begin{pmatrix} f_0 \\ g_0 \end{pmatrix}. \tag{89}$$

where $f_0 = f(x_0, y_0)$, $g_0 = g(x_0, y_0)$, $\frac{\partial f_0}{\partial x} = \frac{\partial f}{\partial x}(x_0, y_0)$, etc. If the matrix

$$\mathbf{A} = \begin{pmatrix} \frac{\partial f_0}{\partial x} & \frac{\partial f_0}{\partial y} \\ \frac{\partial g_0}{\partial x} & \frac{\partial g_0}{\partial y} \end{pmatrix} \tag{90}$$

is nonsingular. Then

$$\begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} = - \begin{pmatrix} \frac{\partial f_0}{\partial x} & \frac{\partial f_0}{\partial y} \\ \frac{\partial g_0}{\partial x} & \frac{\partial g_0}{\partial y} \end{pmatrix}^{-1} \begin{pmatrix} f_0 \\ g_0 \end{pmatrix}. \tag{91}$$

Newton's method

We can now define

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} - \begin{pmatrix} \frac{\partial f_0}{\partial x} & \frac{\partial f_0}{\partial y} \\ \frac{\partial g_0}{\partial x} & \frac{\partial g_0}{\partial y} \end{pmatrix}^{-1} \begin{pmatrix} f_0 \\ g_0 \end{pmatrix}.$$

And by repeating this argument we get

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ y_k \end{pmatrix} - \begin{pmatrix} \frac{\partial f_k}{\partial x} & \frac{\partial f_k}{\partial y} \\ \frac{\partial g_k}{\partial x} & \frac{\partial g_k}{\partial y} \end{pmatrix}^{-1} \begin{pmatrix} f_k \\ g_k \end{pmatrix}, \qquad (92)$$

where $f_k = f(x_k, y_k)$, $g_k = g(x_k, y_k)$ and $\frac{\partial f_k}{\partial x} = \frac{\partial f}{\partial x}(x_k, y_k)$ etc.
The scheme (92) is Newton's method for the system (83).

A Nonlinear example

We test Newton's method on the system

$$\begin{aligned} e^x - e^y &= 0, \\ \ln(1+x+y) &= 0. \end{aligned} \tag{93}$$

The system have analytical solution $x = y = 0$. Define

$$\begin{aligned} f(x,y) &= e^x - e^y, \\ g(x,y) &= \ln(1+x+y). \end{aligned}$$

The iteration in Newton's method (92) reads

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ y_k \end{pmatrix} - \begin{pmatrix} e^{x_k} & -e^{y_k} \\ \frac{1}{1+x_k+y_k} & \frac{1}{1+x_k+y_k} \end{pmatrix}^{-1} \begin{pmatrix} e^{x_k} - e^{y_k} \\ \ln(1+x_k+y_k) \end{pmatrix} . \tag{94}$$

A Nonlinear example

The table below shows the computed results when $x_0 = y_0 = \frac{1}{2}$.

k	x_k	y_k
0	0.5	0.5
1	-0.193147	-0.193147
2	-0.043329	-0.043329
3	-0.001934	-0.001934
4	$-3.75 \cdot 10^{-6}$	$-3.75 \cdot 10^{-6}$
5	$-1.40 \cdot 10^{-11}$	$-1.40 \cdot 10^{-11}$

We observe that, as in the scalar case, Newton's method gives very rapid convergence towards the analytical solution $x = y = 0$.

The Nonlinear System Revisited

We now go back to nonlinear system of ordinary differential equations (75), presented above. For each time step we had to solve

$$\begin{aligned} f(x,y) &= 0, \\ g(x,y) &= 0, \end{aligned} \tag{95}$$

where

$$\begin{aligned} f(x,y) &= x + \Delta t\, y^3 - \alpha, \\ g(x,y) &= y - \Delta t\, x^3 - \beta. \end{aligned} \tag{96}$$

We shall now solve this system using Newton's method.

The Nonlinear System Revisited

We put $x_0 = \alpha$, $y_0 = \beta$ and iterate as follows

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ y_k \end{pmatrix} - \begin{pmatrix} \frac{\partial f_k}{\partial x} & \frac{\partial f_k}{\partial y} \\ \frac{\partial g_k}{\partial x} & \frac{\partial g_k}{\partial y} \end{pmatrix}^{-1} \begin{pmatrix} f_k \\ g_k \end{pmatrix}, \qquad (97)$$

where

$$\begin{aligned} f_k &= f(x_k, y_k), & g_k &= g(x_k, y_k), \\ \frac{\partial f_k}{\partial x} = \frac{\partial f}{\partial x}(x_k, y_k) &= 1, & \frac{\partial f_k}{\partial y} = \frac{\partial f}{\partial y}(x_k, y_k) &= 3\Delta t\, y_k^2, \\ \frac{\partial g_k}{\partial x} = \frac{\partial g}{\partial x}(x_k, y_k) &= -3\Delta t\, x_k^2, & \frac{\partial g_k}{\partial y} = \frac{\partial g}{\partial y}(x_k, y_k) &= 1. \end{aligned}$$

The Nonlinear System Revisited

The matrix

$$\mathbf{A} = \begin{pmatrix} \frac{\partial f_k}{\partial x} & \frac{\partial f_k}{\partial y} \\ \frac{\partial g_k}{\partial x} & \frac{\partial g_k}{\partial y} \end{pmatrix} = \begin{pmatrix} 1 & 3\Delta t\, y_k^2 \\ -3\Delta t\, x_k^2 & 1 \end{pmatrix} \tag{98}$$

has its determinant given by: det$(\mathbf{A}) = 1 + 9\Delta t^2 x_k^2 y_k^2 > 0$. So $\mathbf{A}^{-1}$ is well defined and is given by

$$\mathbf{A}^{-1} = \frac{1}{1 + 9\Delta t^2 x_k^2 y_k^2} \begin{pmatrix} 1 & -3\Delta t\, y_k^2 \\ 3\Delta t\, x_k^2 & 1 \end{pmatrix}. \tag{99}$$

For each time-level we can e.g. iterate until

$$|f(x_k, y_k)| + |g(x_k, y_k)| < \varepsilon = 10^{-6}. \tag{100}$$

The Nonlinear System Revisited

- We have tested this method with $\Delta t = 1/100$ and $t \in [0,1]$
- In Figure 16 the numerical solutions of u and v are plotted as functions of time, and in Figure 17 the numerical solution is plotted in the (u,v) coordinate system
- In Figure 18 we have plotted the number of Newton's iterations needed to reach the stopping criterion (100) at each time-level
- Observe that we need no more than two iterations at all time-levels

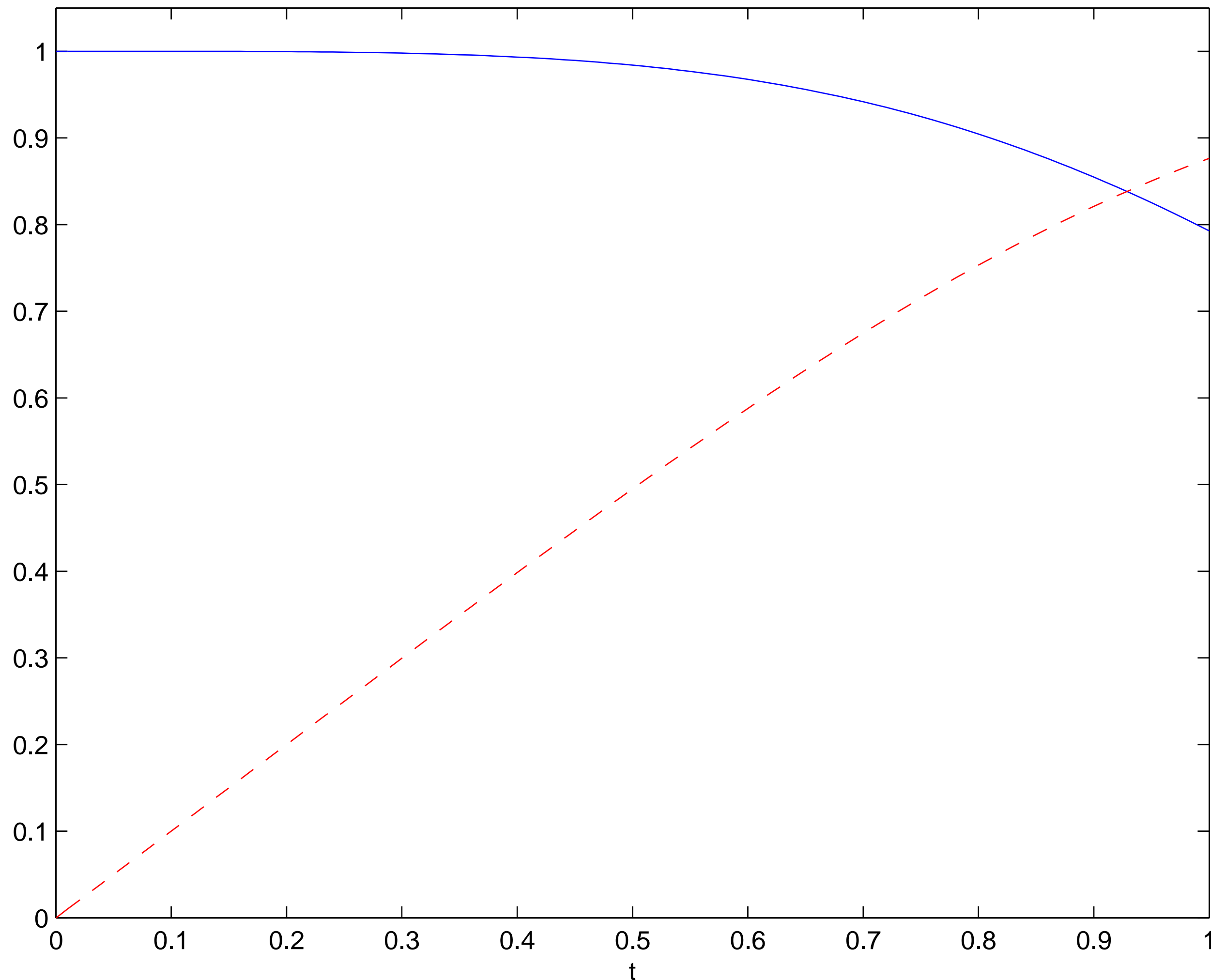

Figure 16: The numerical solutions $u(t)$ and $v(t)$ (in dashed line) of (75) produced by the implicit Euler scheme (76) using $u_0 = 1$, $v_0 = 0$ and $\Delta t = 1/100$.

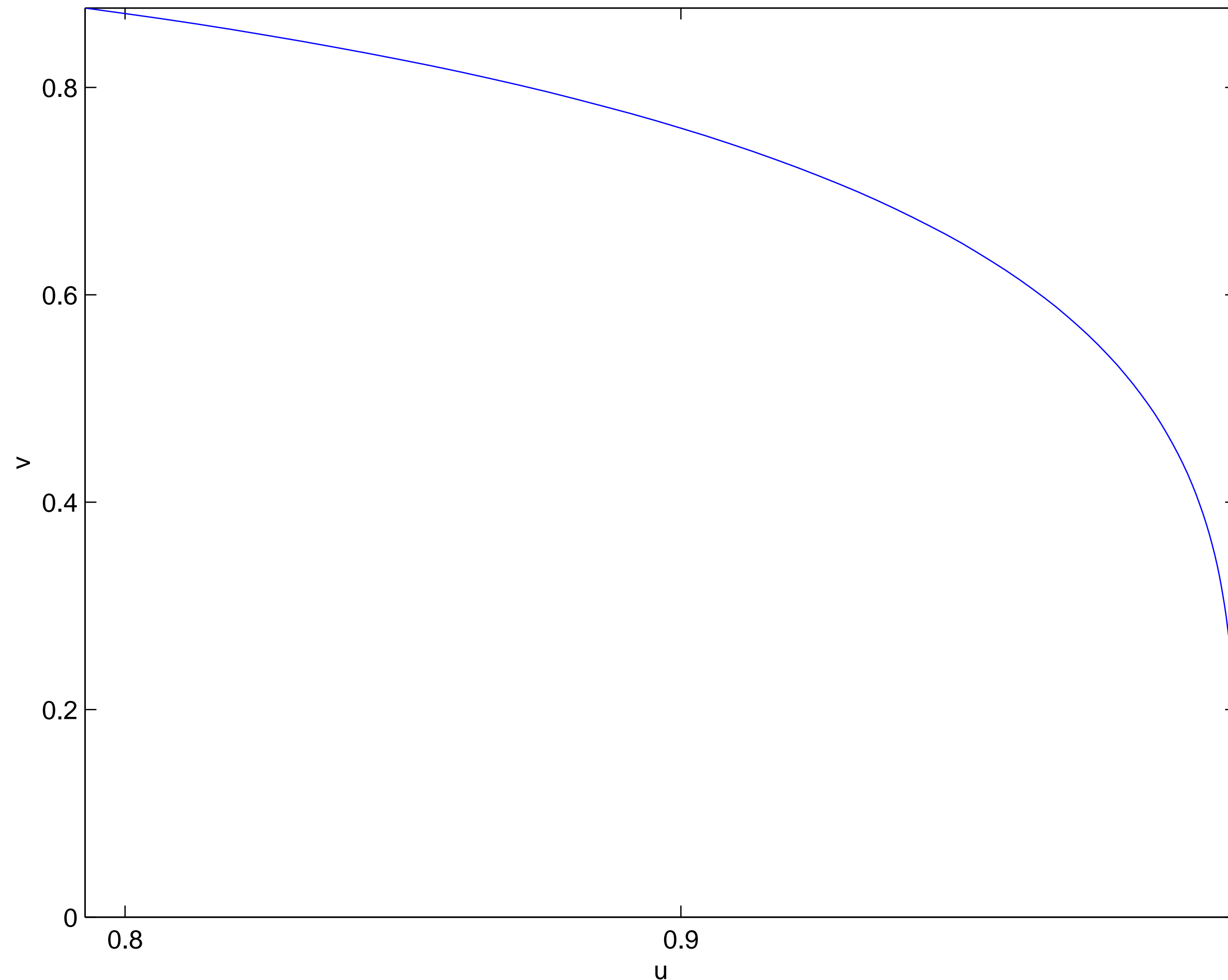

Figure 17: The numerical solutions of (75) in the (u, v)-coordinate system, arising from the implicit Euler scheme (76) using $u_0 = 1$, $v_0 = 0$ and $\Delta t = 1/100$.

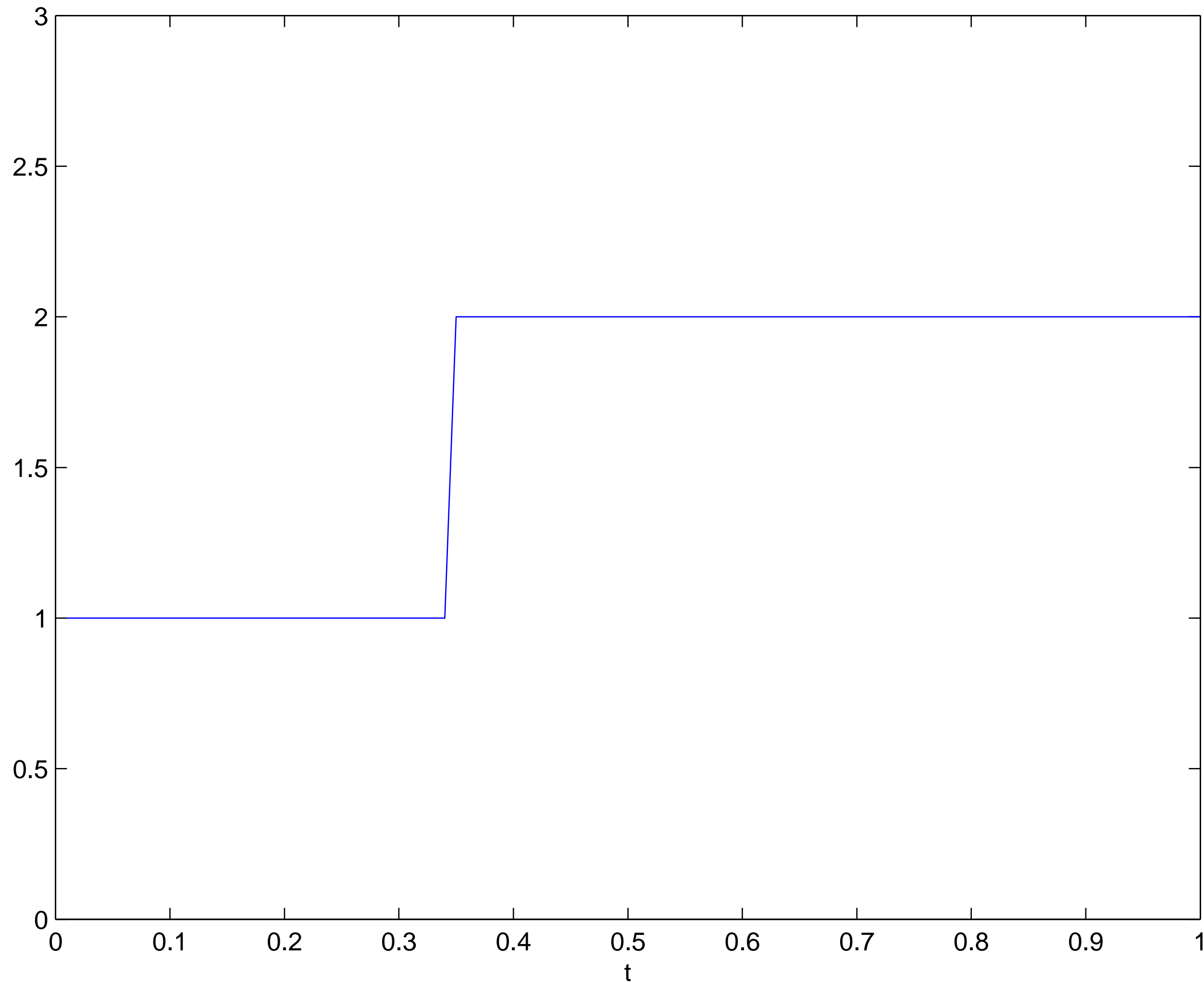

Figure 18: The graph shows the number of iterations used by Newton's method to solve the system (77) at each time-level.

The Method of Least Squares

The method of least squares

We study the following problem:
Given n points (t_i, y_i) for $i = 1, \ldots, n$ in the (t, y)-plane. How can we determine a function $p(t)$ such that

$$p(t_i) \approx y_i, \quad \text{for} \quad i = 1, \ldots, n? \tag{101}$$

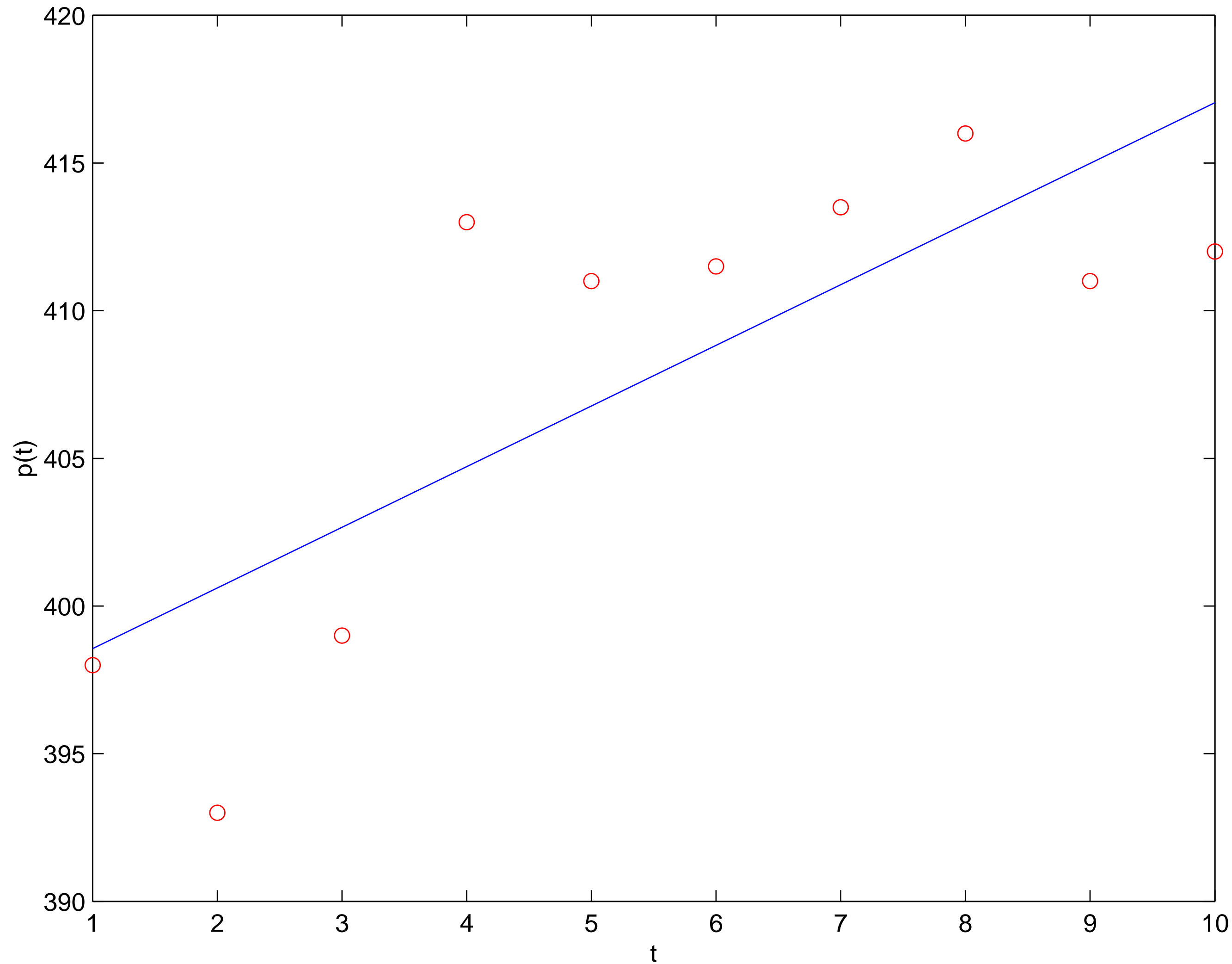

Figure 19: A set of discrete data marked by small circles is approximated with a linear function $p = p(t)$ represented by the solid line.

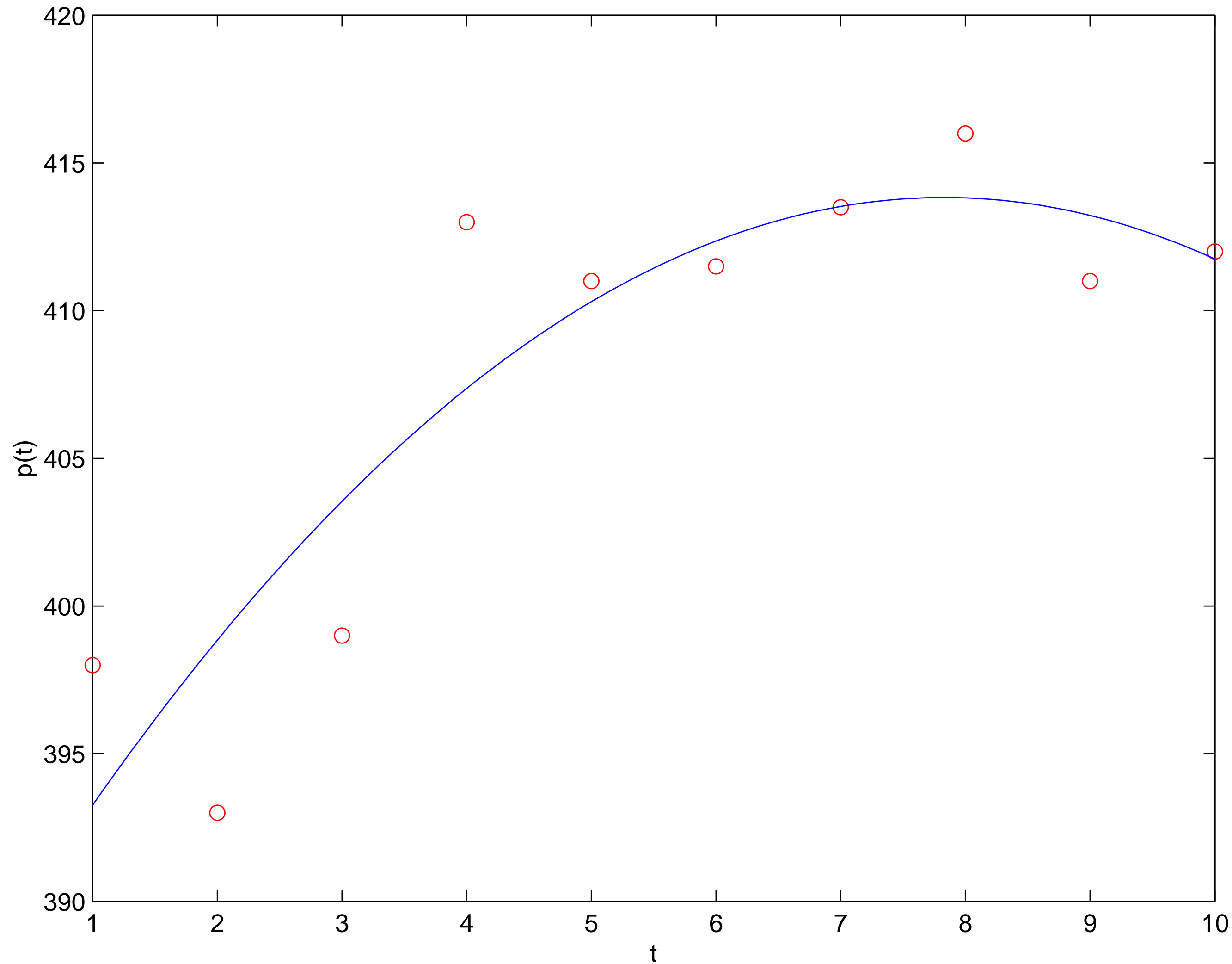

Figure 20: A set of discrete data marked by small circles is approximated with a quadratic function $p = p(t)$ represented by the solid curve.

The method of least square

- Above we saw a discrete data set being approximated by a continuous function
- We can also approximate continuous functions by simpler functions, see Figure 21 and Figure 22

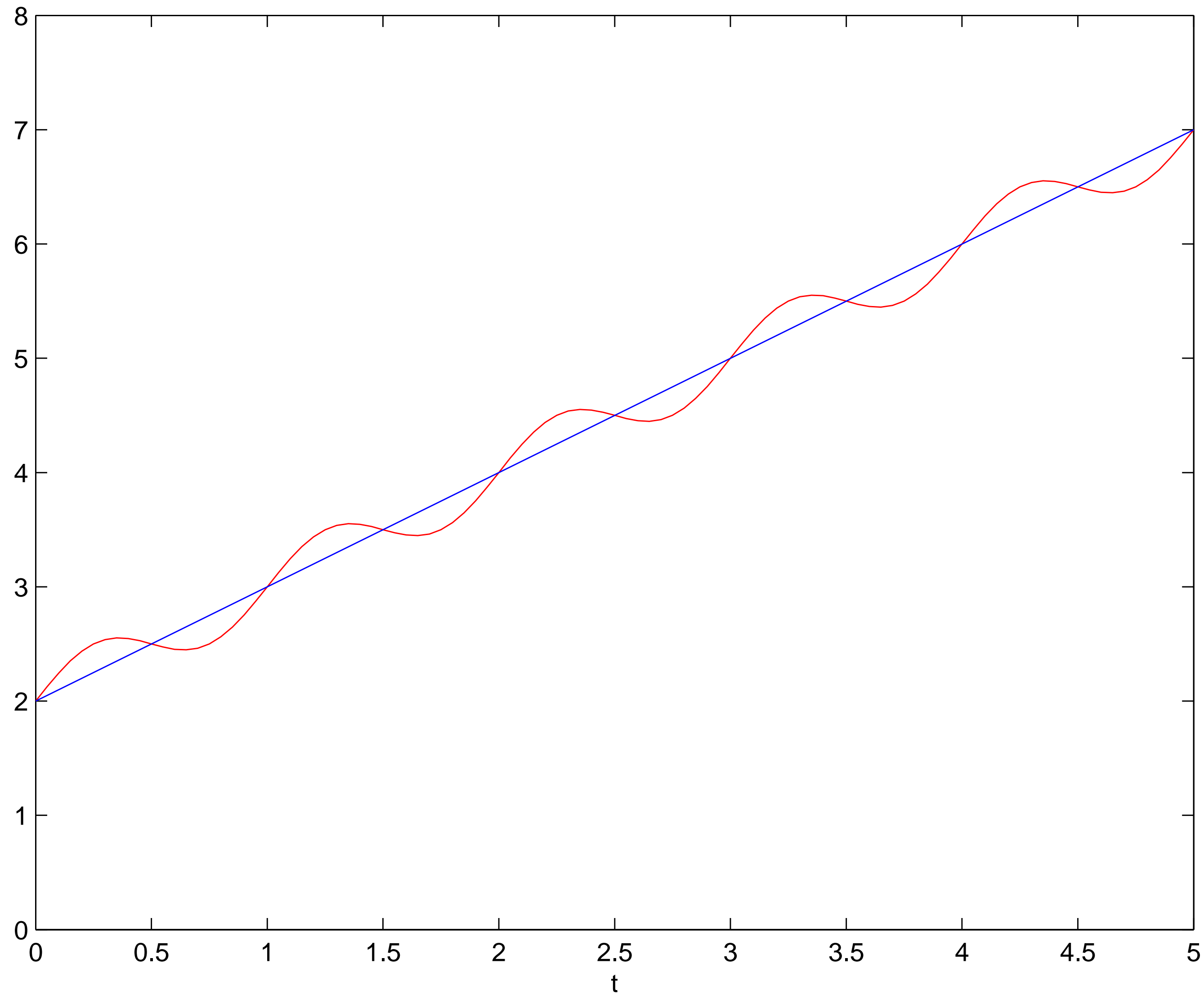

Figure 21: A function $y = y(t)$ and a linear approximation $p = p(t)$.

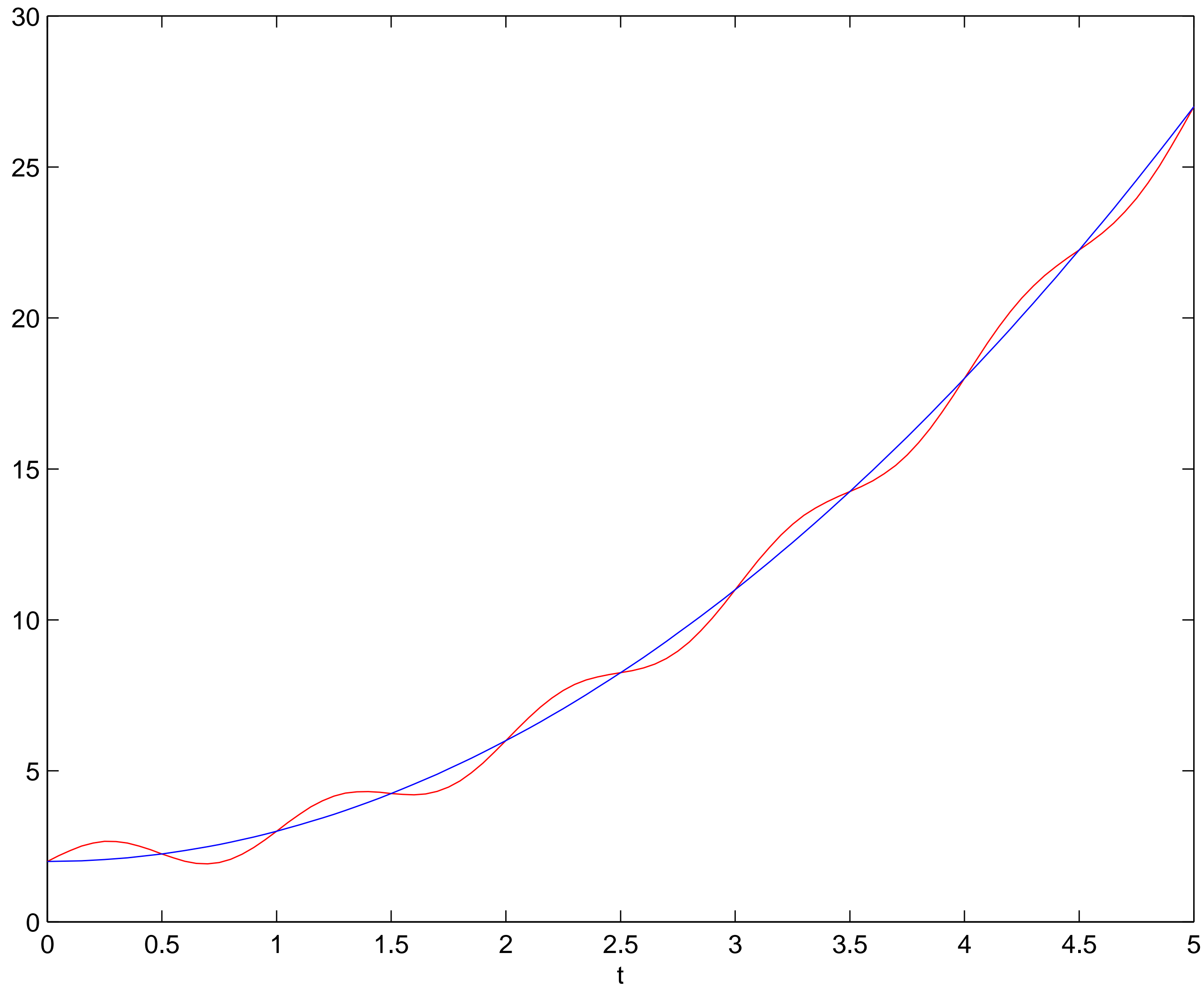

Figure 22: A function $y = y(t)$ and a quadratic approximation $p = p(t)$.

World mean temperature deviations

Calendar year	Computational year t_i	Temperature deviation y_i
1991	1	0.29
1992	2	0.14
1993	3	0.19
1994	4	0.26
1995	5	0.28
1996	6	0.22
1997	7	0.43
1998	8	0.59
1999	9	0.33
2000	10	0.29

Table 7: The global annual mean temperature deviation measured in $^{\circ}C$ for years 1991-2000.

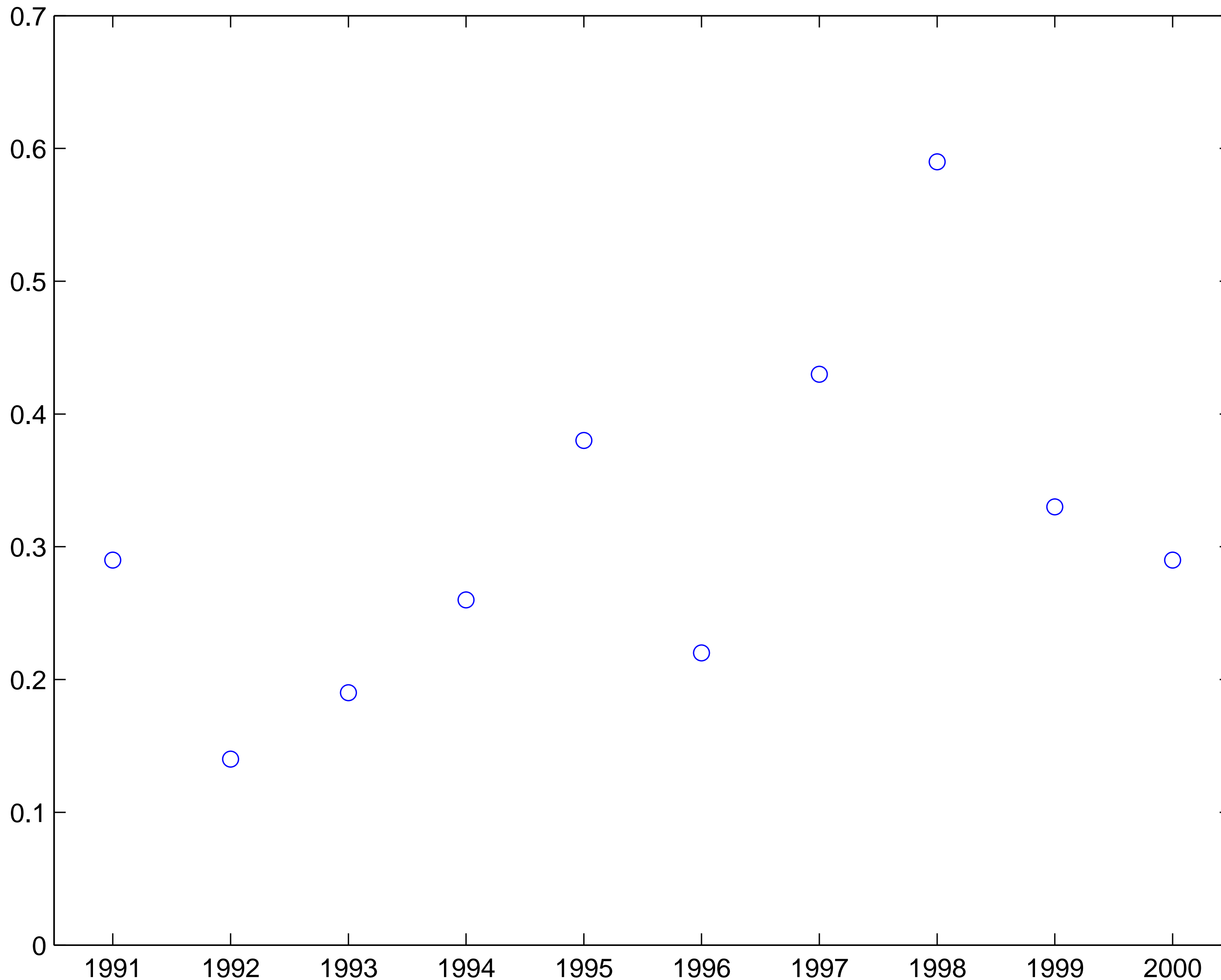

Figure 23: The global annual mean temperature deviation measurements for the period 1991-2000.

Approximating by a constant

- We will study how this set of data can be approximated by simple functions
- First, how can this data set be approximated by a constant function

$$p(t) = \alpha?$$

- The most obvious guess would be to choose α as the arithmetic average

$$\alpha = \frac{1}{10}\sum_{i=1}^{10} y_i = 0.312 \qquad (102)$$

- We will study this guess in more detail

Approximating by a constant

- Assume that we want the solution to minimize the function

$$F(\alpha) = \sum_{i=1}^{10} (\alpha - y_i)^2 \qquad (103)$$

- The function F measures a sort of deviation from α to the set of data $(t_i, y_i)_{i=1}^{10}$
- We want to find the α that minimizes $F(\alpha)$, i.e. we want to find α such that $F'(\alpha) = 0$
- We have

$$F'(\alpha) = 2\sum_{i=1}^{10} (\alpha - y_i) \qquad (104)$$

Approximating by a constant

- This leads to

$$2\sum_{i=1}^{10}\alpha^* = 2\sum_{i=1}^{10} y_i, \tag{105}$$

or

$$\alpha^* = \frac{1}{10}\sum_{i=1}^{10} y_i, \tag{106}$$

which is the arithmetic average

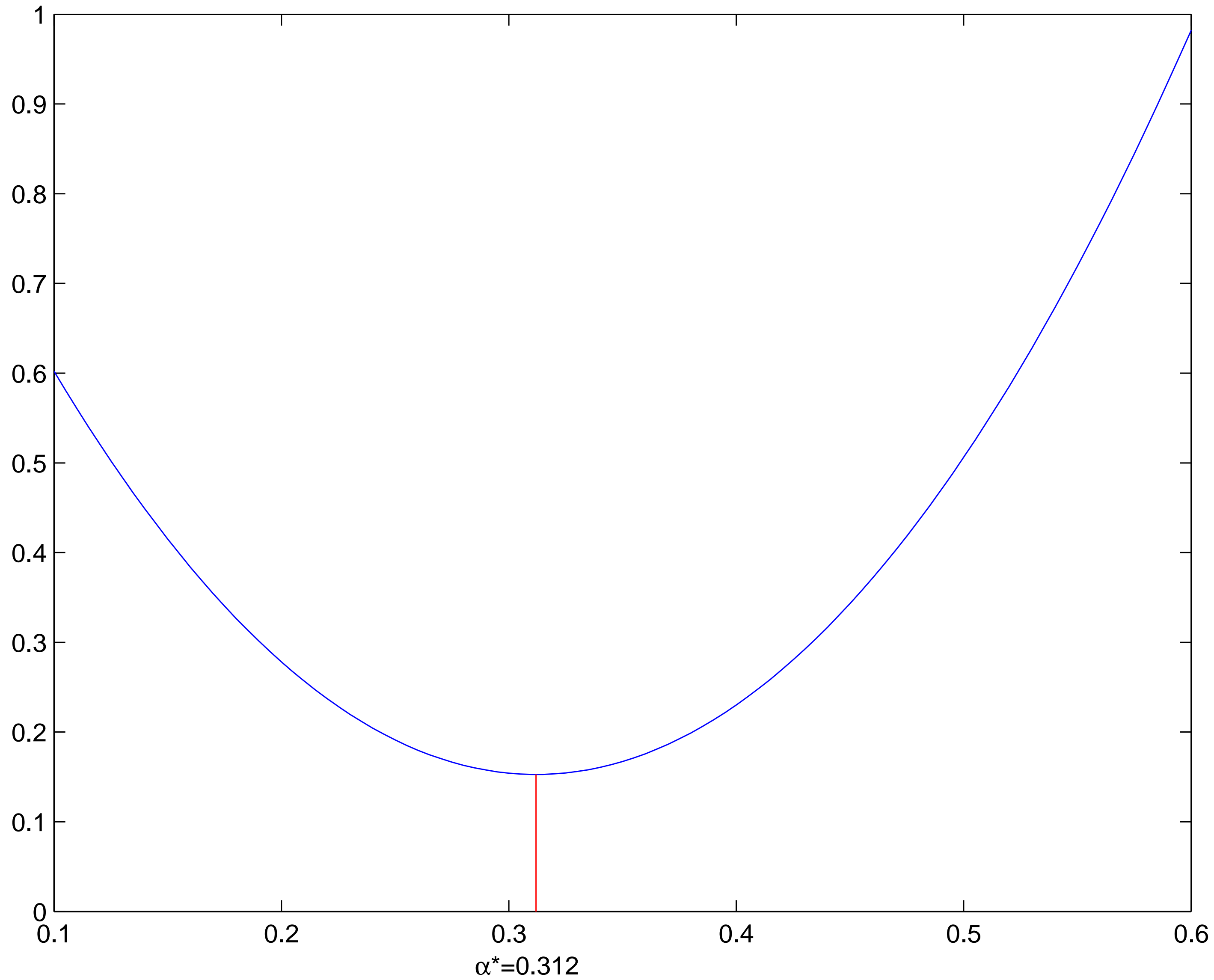

Figure 24: A graph of $F = F(\alpha)$ given by (103).

Approximating by a constant

- Since

$$F''(\alpha) \;=\; 2\sum_{i=1}^{10} 1 \;=\; 20 \;>\; 0, \qquad (107)$$

  it follows that the arithmetic average is the minimizer for F
- We can say that the average value is the optimal constant approximating the global temperature
- This way of defining an optimal constant, where we minimize the sum of the square of the distances between the approximation and the data, is referred to as *the method of least squares*
- There are other ways to define an optimal constant

Approximating by a constant

- Define

$$G(\alpha) = \sum_{i=1}^{10} (\alpha - y_i)^4 \quad (108)$$

- $G(\alpha)$ also measures a sort of deviation from α to the data
- We have that

$$G'(\alpha) = 4\sum_{i=1}^{10} (\alpha - y_i)^3 \quad (109)$$

- And in order to minimize G we need to solve $G'(\alpha) = 0$, (and check that $G''(\alpha) > 0$)

Approximating by a constant

- Solving $G'(\alpha) = 0$ leads to a nonlinear equation that can be solved with the Newton iteration from the previous lecture
- We use Newton's method with
  - initial approximation: $\alpha_0 = 0.312$
  - tolerance specified by: $\varepsilon = 10^{-8}$

  This gives $\alpha^* \approx 0.345$, in three iterations
- α^* is a minimum of G since

$$G''(\alpha^*) \;=\; 12\sum_{i=1}^{10}(\alpha^* - y_i)^2 > 0$$

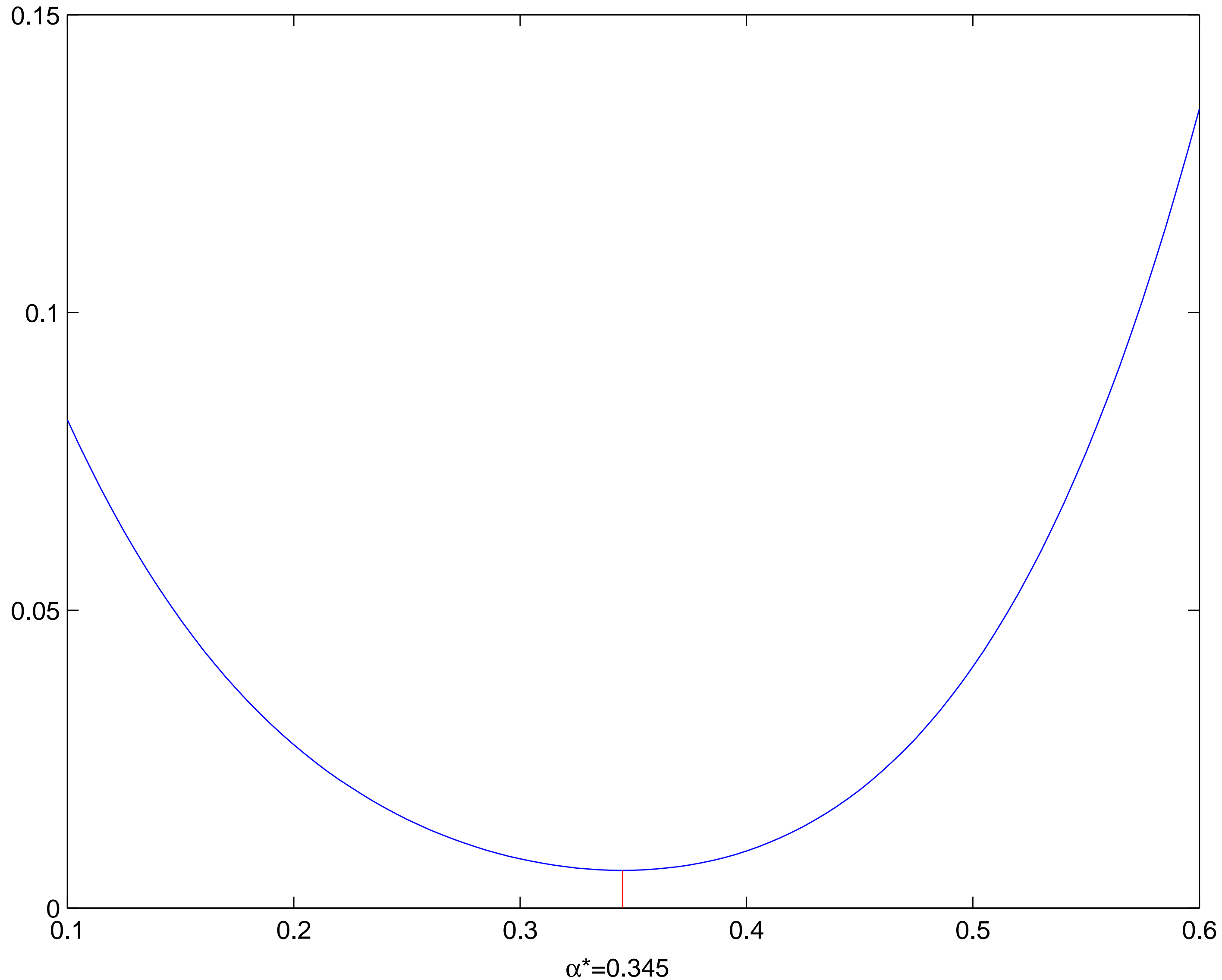

Figure 25: A graph of $G = G(\alpha)$ given by (108).

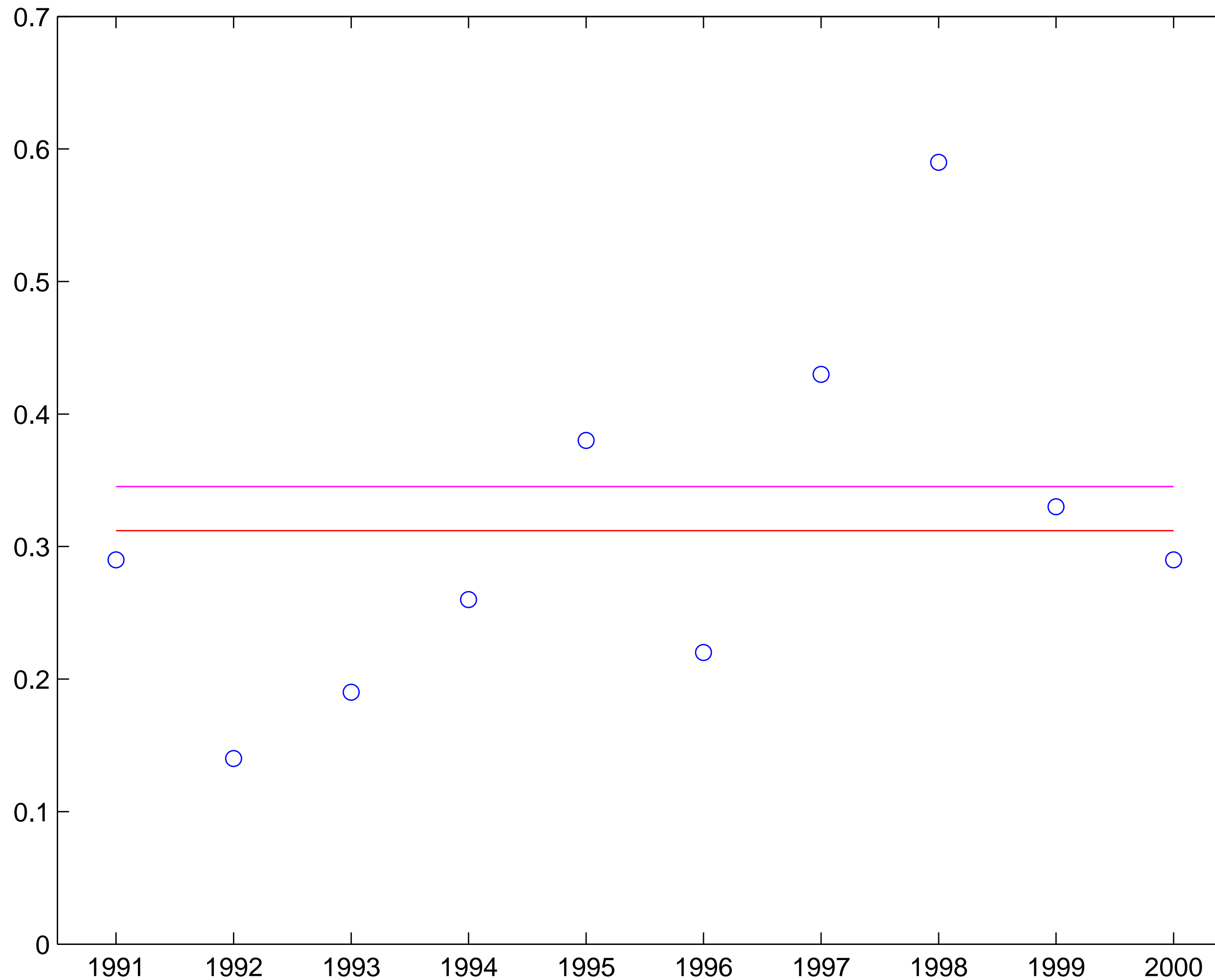

Figure 26: Two constant approximations of the global annual mean temperature deviation measurements from year 1991 to 2000.

Approximating by a linear function

- Now we will study how we can approximate the world mean temperature deviation with a linear function
- We want to determine two constants α and β such that

$$p(t) = \alpha + \beta t \qquad (110)$$

fits the data as good as possible in the sense of least squares

Approximating by a linear function

- Define

$$F(\alpha, \beta) = \sum_{i=1}^{10} (\alpha + \beta t_i - y_i)^2 \tag{111}$$

- In order to minimize F with respect to α and β, we can solve

$$\frac{\partial F}{\partial \alpha} = \frac{\partial F}{\partial \beta} = 0 \tag{112}$$

Approximating by a linear function

We have that

$$\frac{\partial F}{\partial \alpha} = 2\sum_{i=1}^{10}(\alpha + \beta t_i - y_i), \tag{113}$$

and therefore the condition $\frac{\partial F}{\partial \alpha} = 0$ leads to

$$10\alpha + \left(\sum_{i=1}^{10} t_i\right)\beta = \sum_{i=1}^{10} y_i. \tag{114}$$

Approximating by a linear function

Here

$$\sum_{i=1}^{10} t_i \;=\; 1+2+3+\cdots+10 \;=\; 55,$$

and

$$\sum_{i=1}^{10} y_i \;=\; 0.29+0.14+0.19+\cdots+0.29 \;=\; 3.12,$$

so we have

$$10\alpha+55\beta \;=3.12. \tag{115}$$

Approximating by a linear function

Further, we have that

$$\frac{\partial F}{\partial \beta} = 2\sum_{i=1}^{10}(\alpha + \beta t_i - y_i)t_i,$$

and therefore the condition $\frac{\partial F}{\partial \beta} = 0$ gives

$$\left(\sum_{i=1}^{10} t_i\right)\alpha + \left(\sum_{i=1}^{10} t_i^2\right)\beta = \sum_{i=1}^{10} y_i t_i.$$

Approximating by a linear function

We can calculate

$$\sum_{i=1}^{10} t_i^2 \;=\; 1 + 2^2 + 3^2 + \cdots + 10^2 \;=\; 385,$$

and

$$\sum_{i=1}^{10} t_i y_i \;=\; 1 \cdot 0.29 + 2 \cdot 0.14 + 3 \cdot 0.19 + \cdots + 10 \cdot 0.29 \;=\; 20,$$

so we arrive at the equation

$$55\alpha + 385\beta \;= 20. \qquad (116)$$

Approximating by a linear function

We now have a 2×2 system of linear equations which determines α and β:

$$\begin{pmatrix} 10 & 55 \\ 55 & 385 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} 3.12 \\ 20 \end{pmatrix}.$$

With our knowledge of linear algebra, we see that

$$\begin{aligned} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} &= \begin{pmatrix} 10 & 55 \\ 55 & 385 \end{pmatrix}^{-1} \begin{pmatrix} 3.12 \\ 20 \end{pmatrix} \\ &= \frac{1}{825} \begin{pmatrix} 385 & -55 \\ -55 & 10 \end{pmatrix} \begin{pmatrix} 3.12 \\ 20 \end{pmatrix} \approx \begin{pmatrix} 0.123 \\ 0.034 \end{pmatrix}. \end{aligned}$$

Approximating by a linear function

We conclude that the linear model

$$p(t) = 0.123 + 0.034t \tag{117}$$

approximates the data optimally in the sense of least squares.

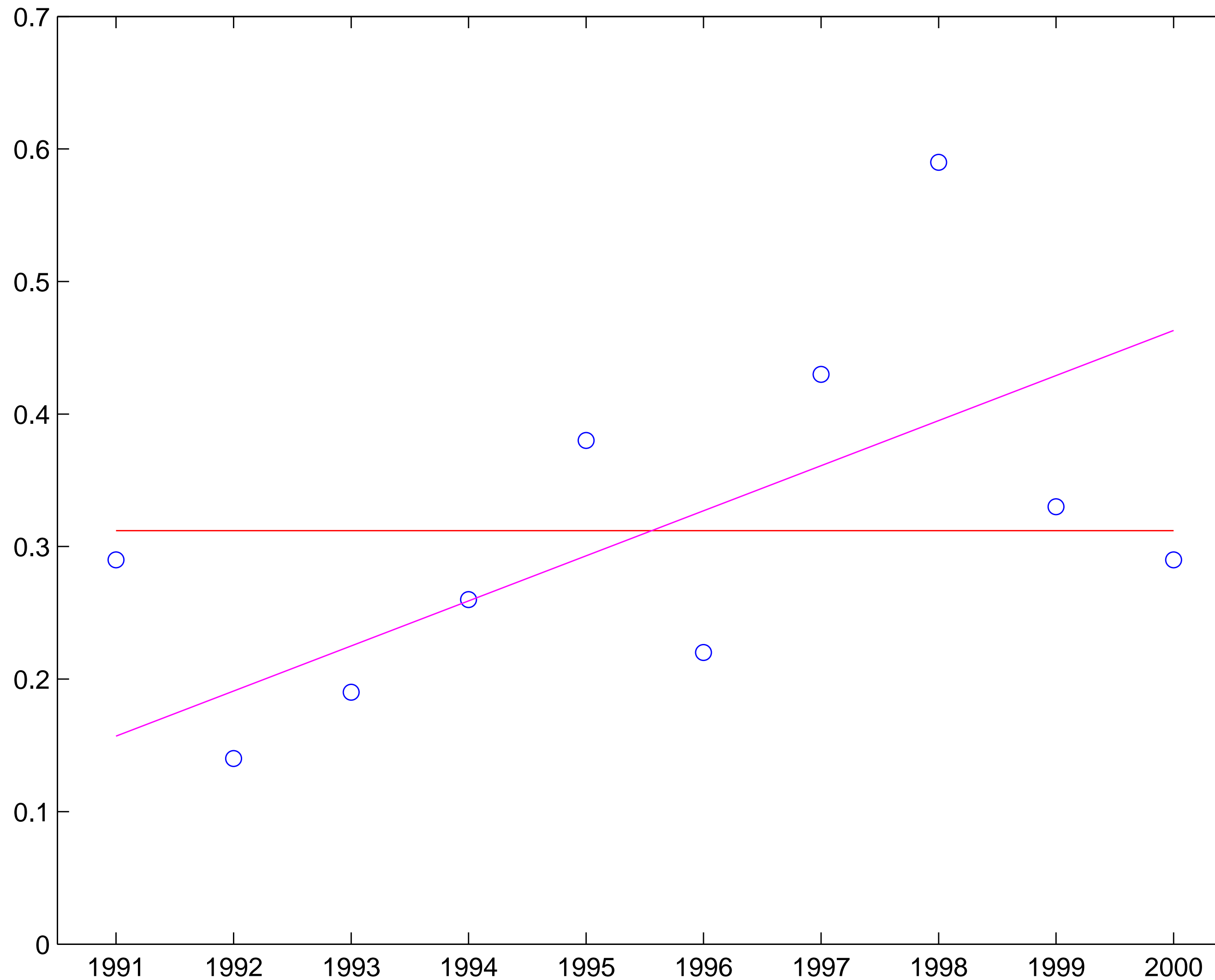

Figure 27: Constant and linear least squares approximations of the global annual mean temperature deviation measurements from year 1991 to 2000.

Approx. by a quadratic function

- We now want to determine constants α, β and γ, such that the quadratic polynomial

$$p(t) = \alpha + \beta t + \gamma t^2 \tag{118}$$

fits the data optimally in the sense of least squares

- Minimizing

$$F(\alpha, \beta, \gamma) = \sum_{i=1}^{10} (\alpha + \beta t_i + \gamma t_i^2 - y_i)^2 \tag{119}$$

requires

$$\frac{\partial F}{\partial \alpha} = \frac{\partial F}{\partial \beta} = \frac{\partial F}{\partial \gamma} = 0 \tag{120}$$

Approx. by a quadratic function

- $\frac{\partial F}{\partial \alpha} = 2\sum_{i=1}^{10}\left(\alpha + \beta t_i + \gamma t_i^2 - y_i\right) \; = \; 0$ leads to

$$10\alpha + \left(\sum_{i=1}^{10} t_i\right)\beta + \left(\sum_{i=1}^{10} t_i^2\right)\gamma \; = \; \sum_{i=1}^{10} y_i$$

- $\frac{\partial F}{\partial \beta} = 2\sum_{i=1}^{10}\left(\alpha + \beta t_i + \gamma t_i^2 - y_i\right) t_i \; = \; 0$ leads to

$$\left(\sum_{i=1}^{10} t_i\right)\alpha + \left(\sum_{i=1}^{10} t_i^2\right)\beta + \left(\sum_{i=1}^{10} t_i^3\right)\gamma \; = \; \sum_{i=1}^{10} y_i t_i$$

- $\frac{\partial F}{\partial \gamma} = 2\sum_{i=1}^{10}\left(\alpha + \beta t_i + \gamma t_i^2 - y_i\right) t_i^2 \; = \; 0$ leads to

$$\left(\sum_{i=1}^{10} t_i^2\right)\alpha + \left(\sum_{i=1}^{10} t_i^3\right)\beta + \left(\sum_{i=1}^{10} t_i^4\right)\gamma \; = \; \sum_{i=1}^{10} y_i t_i^2$$

Approx. by a quadratic function

Here

$$\sum_{i=1}^{10} t_i = 55, \quad \sum_{i=1}^{10} t_i^2 = 385, \quad \sum_{i=1}^{10} t_i^3 = 3025,$$
$$\sum_{i=1}^{10} t_i^4 = 25330, \quad \sum_{i=1}^{10} y_i = 3.12, \quad \sum_{i=1}^{10} t_i y_i = 20,$$
$$\sum_{i=1}^{10} t_i^2 y_i = 138.7,$$

which leads to the linear system

$$\begin{pmatrix} 10 & 55 & 385 \\ 55 & 385 & 3025 \\ 385 & 3025 & 25330 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} 3.12 \\ 20 \\ 138.7 \end{pmatrix}. \quad (121)$$

Solving the linear system (121) with, e.g., matlab we get

$$\begin{aligned} \alpha &\approx -0.4078, \\ \beta &\approx 0.2997, \\ \gamma &\approx -0.0241. \end{aligned} \tag{122}$$

We have now obtained three approximations of the data

- The constant

$$p_0(t) = 0.312$$

- The linear

$$p_1(t) = 0.123 + 0.034t$$

- The quadratic

$$p_2(t) = -0.4078 + 0.2997t - 0.0241t^2$$

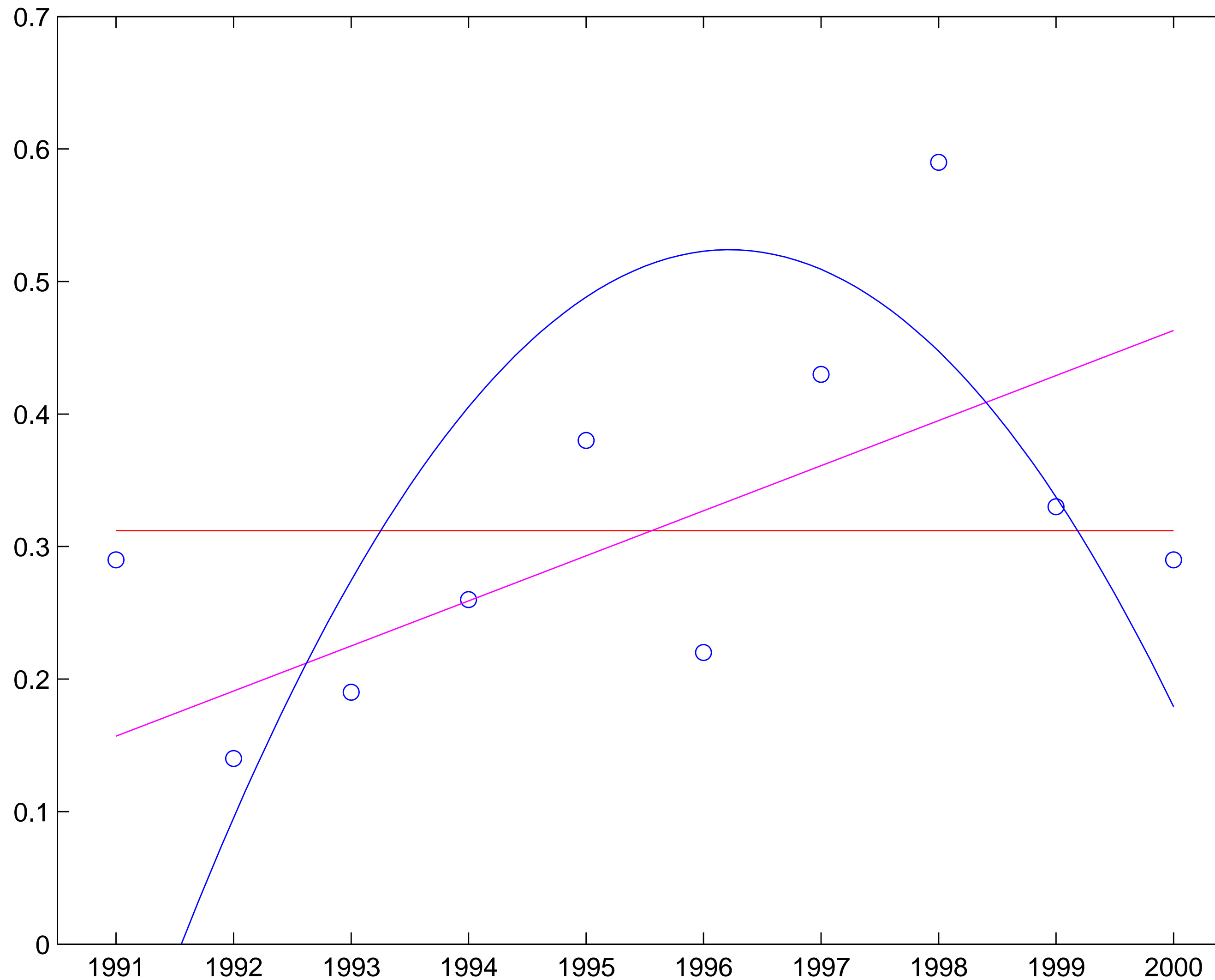

Figure 28: Constant, linear and quadratic approximations of the global annual mean temperature deviation measurements from the year 1991 to 2000.

Summary

Approximating a data set

$$(t_i, y_i) \quad i = 1, \ldots, n,$$

with a constant function

$$p_0(t) \;=\; \alpha.$$

Using the method of least squares gives

$$\alpha \;=\; \frac{1}{n}\sum_{i=1}^{n} y_i, \tag{123}$$

which is recognized as the arithmetic average.

Summary

Approximating the data set with a linear function

$$p_1(t) \quad = \quad \alpha + \beta t$$

can be done by minimizing

$$\min_{\alpha,\beta} F(\alpha,\beta) = \min_{\alpha,\beta} \sum_{i=1}^{n} (p_1(t_i) - y_i)^2,$$

which leads to the following 2×2 linear system

$$\begin{pmatrix} n & \sum_{i=1}^{n} t_i \\ \sum_{i=1}^{n} t_i & \sum_{i=1}^{n} t_i^2 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} t_i y_i \end{pmatrix}. \qquad (124)$$

Summary

A quadratic approximation on the form

$$p_2(t) \quad = \quad \alpha + \beta t + \gamma t^2$$

can be done by minimizing $\min_{\alpha,\beta,\gamma} F(\alpha,\beta,\gamma) = \min_{\alpha,\beta,\gamma} \sum_{i=1}^{n} (p_2(t_i) - y_i)^2$, which leads to the following 3×3 linear system

$$\begin{pmatrix} n & \sum_{i=1}^{n} t_i & \sum_{i=1}^{n} t_i^2 \\ \sum_{i=1}^{n} t_i & \sum_{i=1}^{n} t_i^2 & \sum_{i=1}^{n} t_i^3 \\ \sum_{i=1}^{n} t_i^2 & \sum_{i=1}^{n} t_i^3 & \sum_{i=1}^{n} t_i^4 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} y_i t_i \\ \sum_{i=1}^{n} y_i t_i^2 \end{pmatrix}. \qquad (125)$$

```
import numpy as np
import pylab
y = np.array([1.1, 2.1, 3.2, 4.1, 6.4])
t = np.linspace(0,1,5)
pylab.plot(t,y)

n = len(t)
t1 = sum(t)
t2 = sum(t**2)
t3 = sum(t**3)
t4 = sum(t**4)
A = np.array([[n,t1,t2],[t1,t2,t3],[t2,t3,t4]])

y1 = sum(y)
yt = sum(y*t)
yt2 = sum(y*t**2)
b = np.array([y1,yt,yt2])

p = np.linalg.solve(A, b)
x = np.linspace(0,1,101);
f = p[2]*x**2 + p[1]*x + p[0]
pylab.plot(x,f)

pylab.show()
```

Approximations of Functions

- Above we have studied continuous representation of discrete data
- Next we will consider continuous approximation of continuous functions
- Consider the function

$$y(t) \quad = \quad \ln\left(\frac{1}{10}\sin(t)+e^{t}\right) \tag{126}$$

- In Figure 29 we see that $y(x)$ seems to be close to the linear function $p(t)=t$ on the interval $[0,1]$
- In Figure 30 we see that $y(x)$ seems to be even closer to the linear function plotted on $t \in [0,10]$

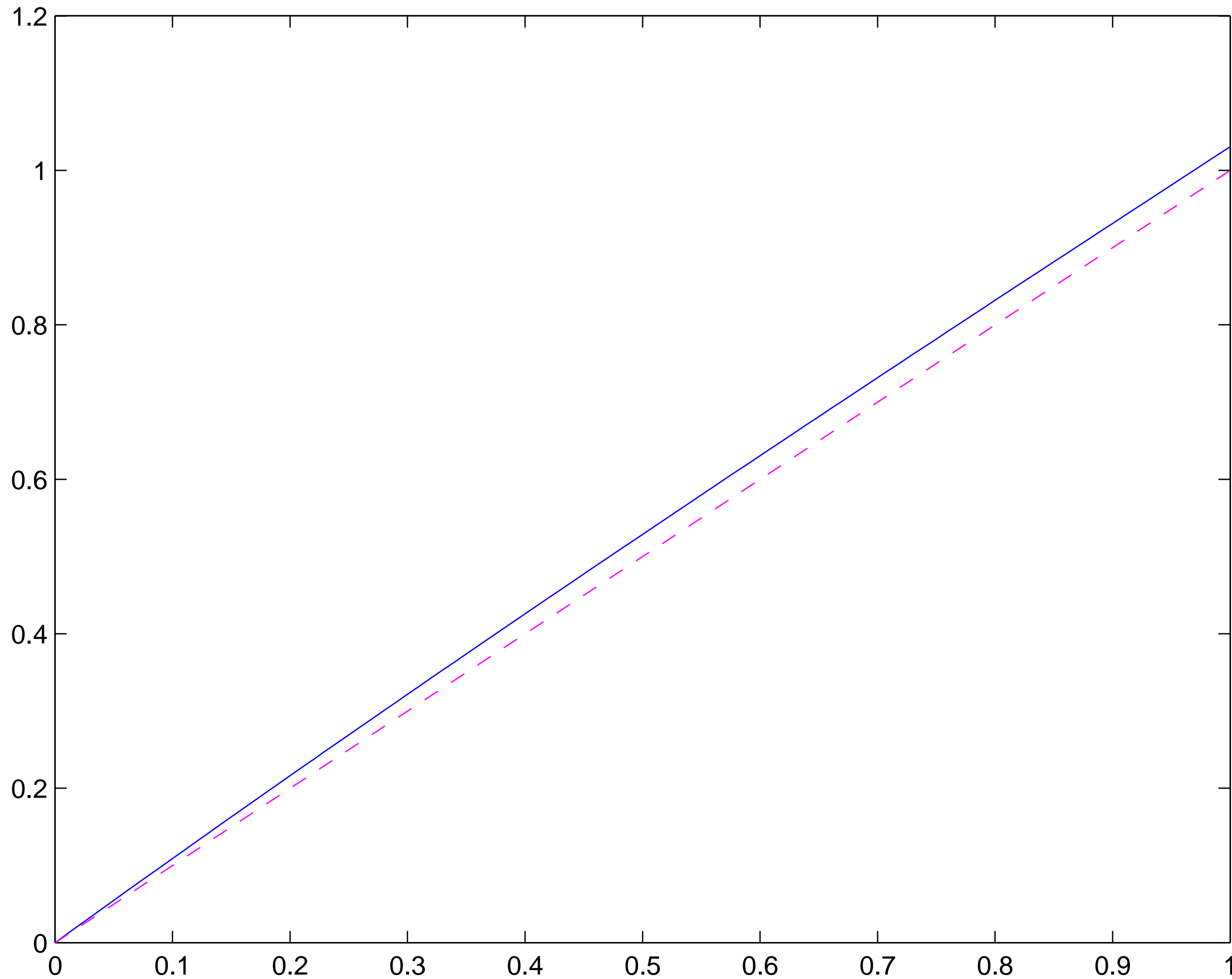

Figure 29: The function $y(t) = \ln\left(\frac{1}{10}\sin(t) + e^t\right)$ (solid curve) and a linear approximation (dashed line) on the interval $t \in [0,1]$.

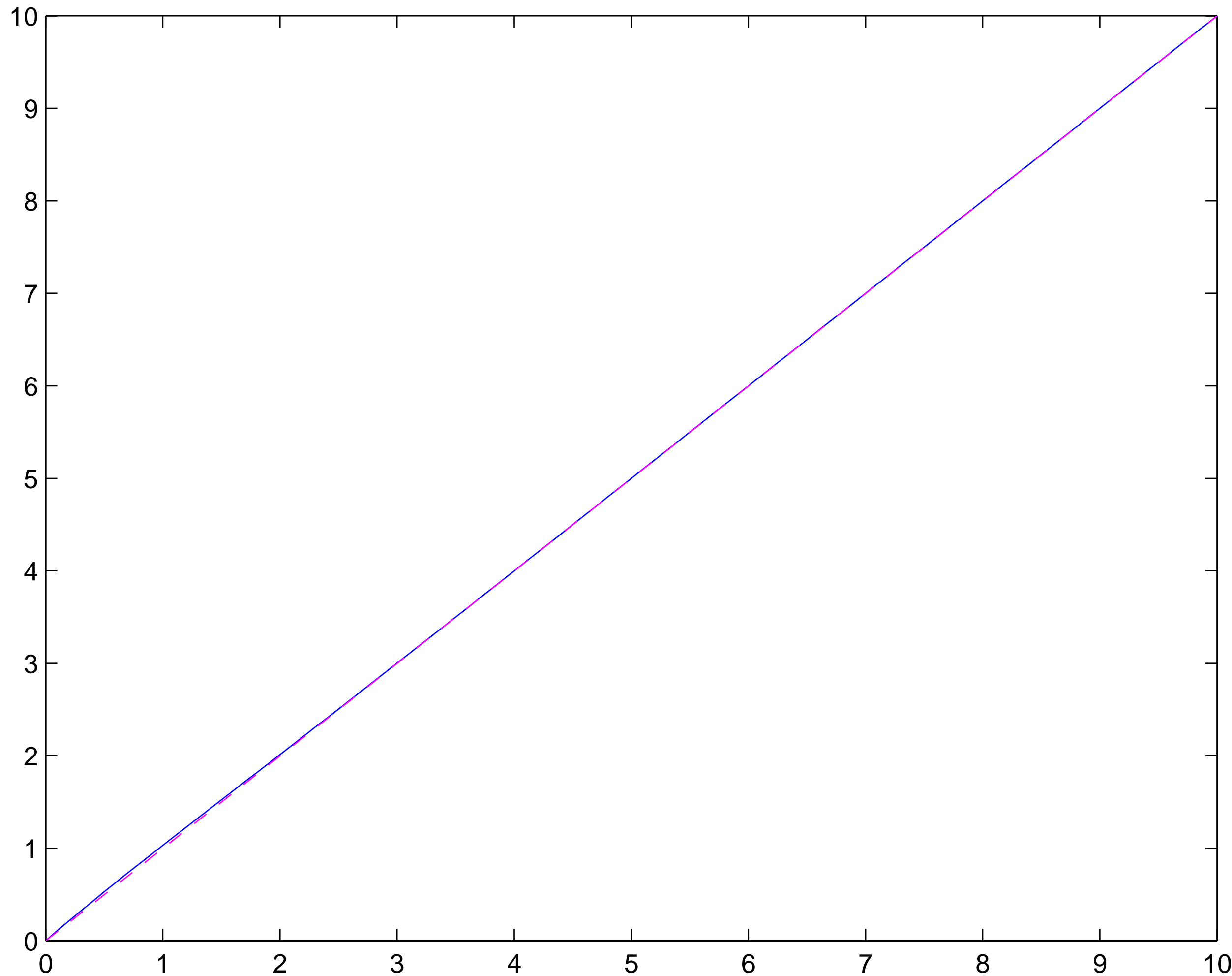

Figure 30: The function $y(t) = \ln\left(\frac{1}{10}\sin(t) + e^t\right)$ (solid curve) and a linear approximation (dashed line) on the interval $t \in [0, 10]$.

Approximations by constants

- For a given function $y(t)$, $t \in [a,b]$, we want to compute a constant approximation of it

$$p(t) = \alpha \tag{127}$$

for $t \in [a,b]$, in the sense of least squares

- That means that we want to minimize the integral

$$\int_a^b (p(t) - y(t))^2 \, dt = \int_a^b (\alpha - y(t))^2 \, dt$$

Approximations by constants

- Define the function

$$F(\alpha) \;=\; \int_a^b (\alpha - y(t))^2 \, dt \tag{128}$$

- The derivative with respect to α is

$$F'(\alpha) \;=\; 2\int_a^b (\alpha - y(t)) \, dt$$

- And solving $F'(\alpha) = 0$ gives

$$\alpha \;=\; \frac{1}{b-a}\int_a^b y(t) dt \tag{129}$$

Note that

- The formula for α is the integral version of the average of y on $[a,b]$. In the discrete case we would have written

$$\alpha = \frac{1}{n}\sum_{i=1}^{n} y_i, \tag{130}$$

If y_i in (130) is $y(t_i)$, where $t_i = a + i\Delta t$ and $\Delta t = \frac{b-a}{n}$, then

$$\frac{1}{n}\sum_{i=1}^{n} y_i = \frac{1}{b-a}\Delta t \sum_{i=1}^{n} y(t_i) \approx \frac{1}{b-a}\int_a^b y(t)\,dt.$$

We therefore conclude that (129) is a natural continuous version of (130).

Note that

- We used

$$\frac{d}{d\alpha}\int_a^b (\alpha - y(t))^2 dt = \int_a^b \frac{\partial}{\partial\alpha}(\alpha - y(t))^2 dt$$

  Is that a legal operation? This is discussed in Exercise 5.

- The α given by (129) is a minimum, since

$$F''(\alpha) = 2(b-a) > 0$$

Example 15; const. approx.

Consider

$$y(t) = \sin(t)$$

defined on $0 \le t \le \pi/2$. A constant approximation of y is given by

$$\begin{aligned} p(t) = \alpha \quad &\stackrel{(129)}{=} \quad \frac{2}{\pi}\int_0^{\pi/2} \sin(t)\,dt = \frac{-2}{\pi}\left[\cos(t)\right]_0^{\pi/2} \\ &= \quad \frac{-2}{\pi}(0-1) = \frac{2}{\pi}. \end{aligned}$$

Example 16; const. approx.

Consider

$$y(t) = t^2 + \frac{1}{10}\cos(t)$$

defined on $0 \leq t \leq 1$. A constant approximation of y is given by

$$p(t) = \alpha \overset{(129)}{=} \int_0^1 \left(t^2 + \frac{1}{10}\cos(t) \right) dt = \left[\frac{1}{3}t^3 + \frac{1}{10}\sin(t) \right]_0^1$$

$$= \frac{1}{3} + \frac{1}{10}\sin(1) \approx 0.417.$$

Approximations by Linear Functions

- Now, we search for a linear approximation of a function $y(t)$, $t \in [a,b]$, i.e.

$$p(t) = \alpha + \beta t \quad (131)$$

in the sense of least squares

- Define

$$F(\alpha,\beta) = \int_a^b (\alpha + \beta t - y(t))^2 dt \quad (132)$$

- A minimum of F is obtained by finding α and β such that

$$\frac{\partial F}{\partial \alpha} = \frac{\partial F}{\partial \beta} = 0$$

Approximations by Linear Functions

- We have

$$\frac{\partial F}{\partial \alpha} = 2\int_a^b (\alpha + \beta t - y(t))\, dt$$
$$\frac{\partial F}{\partial \beta} = 2\int_a^b (\alpha + \beta t - y(t)) t\, dt$$

- Therefore α and β can be determined by solving the following linear system

$$\begin{aligned} (b-a)\alpha + \frac{1}{2}(b^2 - a^2)\beta &= \int_a^b y(t)\, dt \\ \frac{1}{2}(b^2 - a^2)\alpha + \frac{1}{3}(b^3 - a^3)\beta &= \int_a^b t\, y(t)\, dt \end{aligned} \qquad (133)$$

Example 15; linear approx.

Consider

$$y(t) \;=\; \sin(t)$$

defined on $0 \leq t \leq \pi/2$.
We have

$$\int_0^{\pi/2} \sin(t)\, dt \;=\; 1$$

and

$$\int_0^{\pi/2} t \sin(t)\, dt \;=\; 1.$$

Example 15; linear approx.

The linear system now reads

$$\begin{pmatrix} \pi/2 & \pi^2/8 \\ \pi^2/8 & \pi^3/24 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

The solution is

$$\begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \frac{1}{\pi^2} \begin{pmatrix} 8\pi - 24 \\ \frac{96}{\pi} - 24 \end{pmatrix} \approx \begin{pmatrix} 0.115 \\ 0.664 \end{pmatrix}.$$

Therefore the linear approximation is given by

$$p(t) \approx 0.115 + 0.664\,t.$$

Example 16; linear approx.

Consider

$$y(t) \quad = \quad t^2 + \frac{1}{10}\cos(t)$$

defined on $0 \leq t \leq 1$. The linear system (133) then reads

$$\begin{pmatrix} 1 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{3} \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \quad = \quad \begin{pmatrix} \frac{1}{3} + \frac{1}{10}\sin(1) \\ \frac{3}{20} + \frac{1}{10}\cos(1) + \frac{1}{10}\sin(1) \end{pmatrix},$$

with solution $\alpha \approx -0.059$ and $\beta \approx 0.953$.
We conclude that the linear least squares approximation is given by

$$p(t) \quad \approx \quad -0.059 + 0.953\,t.$$

Approx. by Quadratic Functions

- We seek a quadratic function

$$p(t) = \alpha + \beta t + \gamma t^2 \tag{134}$$

  that approximates a given function $y = y(t)$, $a \leq t \leq b$, in the sense of least squares

- Let

$$F(\alpha, \beta, \gamma) = \int_a^b (\alpha + \beta t + \gamma t^2 - y(t))^2 dt \tag{135}$$

- Define α, β and γ to be the solution of the three equations:

$$\frac{\partial F}{\partial \alpha} = \frac{\partial F}{\partial \beta} = \frac{\partial F}{\partial \gamma} = 0$$

Approx. by Quadratic Functions

- By taking the derivatives, we have

$$\frac{\partial F}{\partial \alpha} = 2\int_a^b (\alpha + \beta t + \gamma t^2 - y(t))\, dt$$

$$\frac{\partial F}{\partial \beta} = 2\int_a^b (\alpha + \beta t + \gamma t^2 - y(t))\, t\, dt$$

$$\frac{\partial F}{\partial \gamma} = 2\int_a^b (\alpha + \beta t + \gamma t^2 - y(t))\, t^2\, dt$$

- The coefficients α, β and γ can now be determined from the linear system

$$\begin{aligned}
(b-a)\alpha + \frac{1}{2}(b^2-a^2)\beta + \frac{1}{3}(b^3-a^3)\gamma &= \int_a^b y(t)\,dt \\
\frac{1}{2}(b^2-a^2)\alpha + \frac{1}{3}(b^3-a^3)\beta + \frac{1}{4}(b^4-a^4)\gamma &= \int_a^b t\,y(t)\,dt \\
\frac{1}{3}(b^3-a^3)\alpha + \frac{1}{4}(b^4-a^4)\beta + \frac{1}{5}(b^5-a^5)\gamma &= \int_a^b t^2\,y(t)\,dt
\end{aligned}$$

Example 15; quad. approx.

For the function

$$y(t) = \sin(t), \quad 0 \le t \le \pi/2,$$

the linear system reads

$$\begin{pmatrix} \pi/2 & \pi^2/8 & \pi^3/24 \\ \pi^2/8 & \pi^3/24 & \pi^4/64 \\ \pi^3/24 & \pi^4/64 & \pi^5/160 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ \pi - 2 \end{pmatrix},$$

and the solution is given by $\alpha \approx -0.024$, $\beta \approx 1.196$ and $\gamma \approx -0.338$, which gives the quadratic approximation

$$p(t) = -0.024 + 1.196t - 0.338t^2.$$

Example 16; quad. approx.

Let us consider

$$y(t) = t^2 + \frac{1}{10}\cos(t)$$

for $0 \le t \le 1$. The linear system takes the form

$$\begin{pmatrix} 1 & 1/2 & 1/3 \\ 1/2 & 1/3 & 1/4 \\ 1/3 & 1/4 & 1/5 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} \frac{1}{3} + \frac{1}{10}\sin(1) \\ \frac{3}{20} + \frac{1}{10}\cos(1) + \frac{1}{10}\sin(1) \\ \frac{1}{5} + \frac{1}{5}\cos(1) - \frac{1}{10}\sin(1) \end{pmatrix}$$

and the solution is given by $\alpha \approx 0.100$, $\beta \approx -0.004$ and $\gamma \approx 0.957$, and the quadratic approximation is

$$p(t) = 0.100 - 0.004\,t + 0.957\,t^2.$$

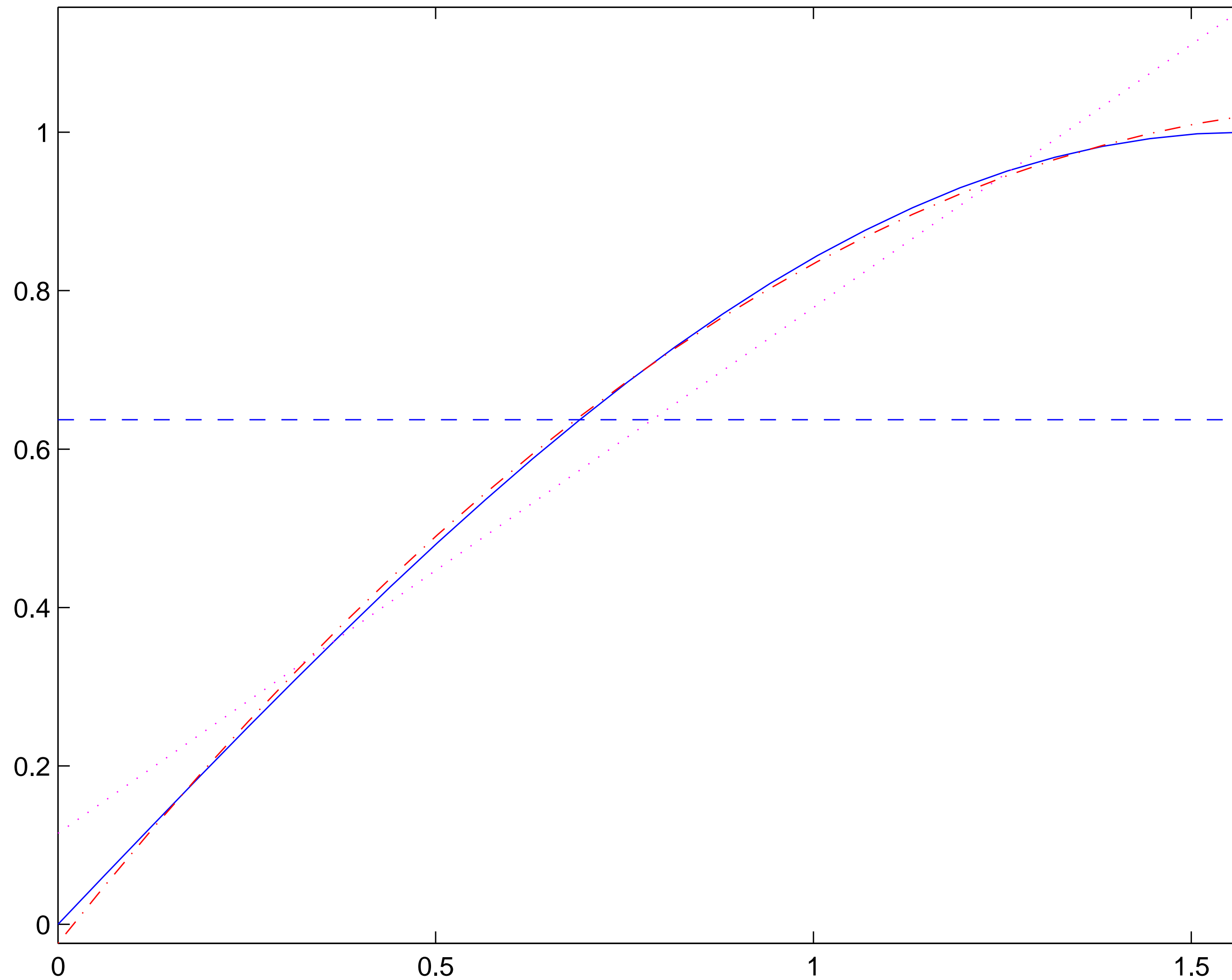

Figure 31: The function $y(t) = \sin(t)$ (solid curve) and its least squares approximations: constant (dashed line), linear (dotted line) and quadratic (dashed-dotted curve).

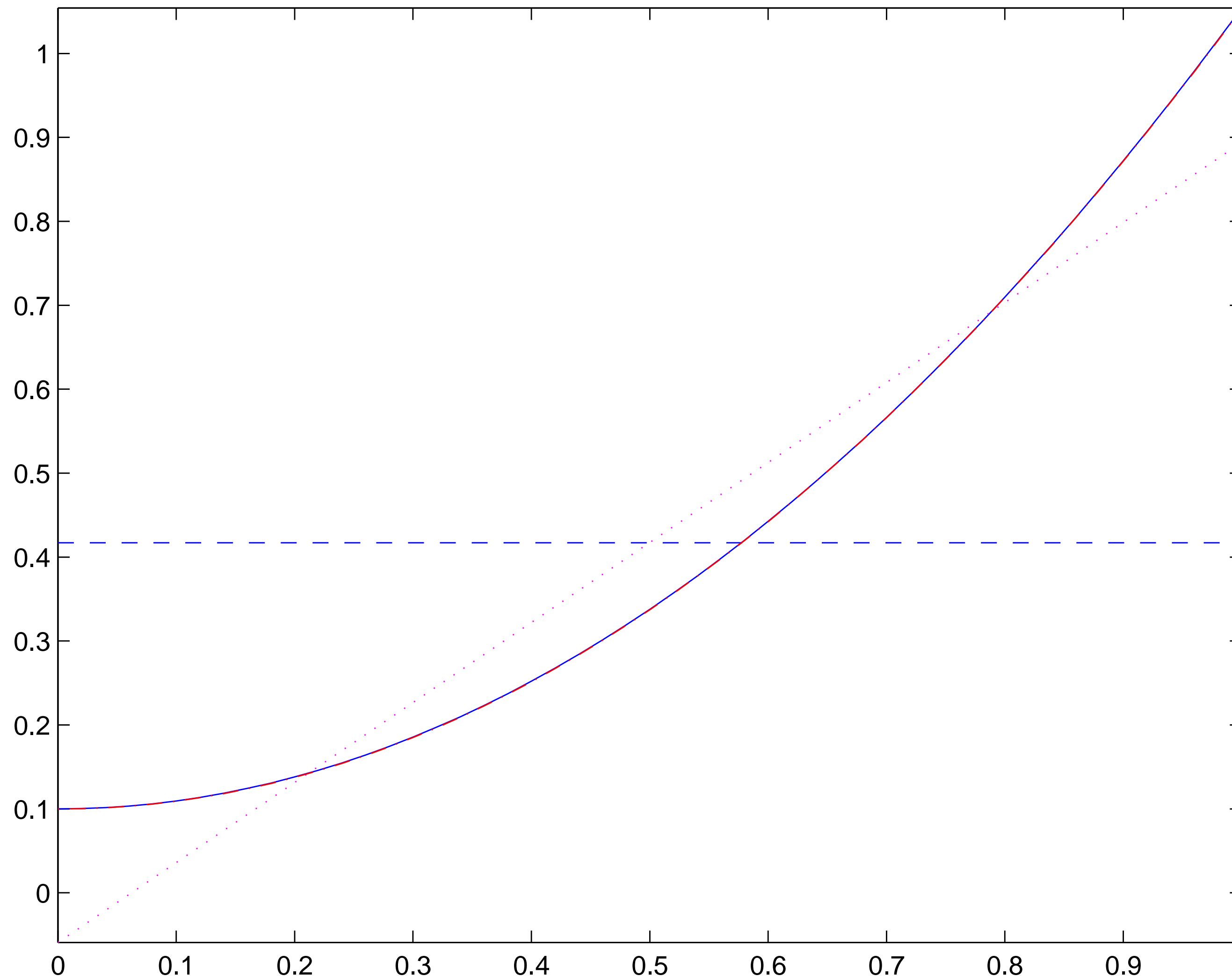

Figure 32: The function $y(t) = t^2 + \frac{1}{10}\cos(t)$ (solid curve) and its least squares approximations: constant (dashed line), linear (dotted line) and quadratic (dashed-dotted curve).

```
import numpy as np
from scipy.integrate import quad as integrator

def approximate(f, n, x0, x1):

    A = np.zeros((n,n))
    b = np.zeros((n,1))

    for i in range(n):
        b[i] = integrator(lambda x: f(x)*x**i, x0, x1)[0]
        for j in range(n):
            A[i,j] = integrator(lambda x: (x**i)*(x**j), x0, x1)[0]

    p = np.linalg.solve(A, b)

    return p

def f(x):
    return np.sin(x)

x0 = 0.
x1 = 10.
p = approximate(f, 7, x0, x1)
```

From Mathematical Formula to Scientific Software

Scientific software

- Desired properties
  - Correct
  - Efficient (speed, memory, storage)
  - Easily maintainable
  - Easily extendible
- Important skills
  - Understanding numerics
  - Designing data structures
  - Using libraries and programming tools
  - (Quick learning of new programming languages)

A typical scientific computing code

- Starting point
  - Numerical problem
- Pre-processing
  - Data input and preparation
  - Build-up of internal data structure
- Main computation
- Post-processing
  - Result analysis
  - Display, output and visualization

A two-step strategy

- Correct implementation of a complicated numerical problem is a challenging task
- Divide the task into two steps:
  - Express the numerical problem as a complete algorithm
  - Translate the algorithm into a computer code using a specific programming language

Advantages

- Small gap between the numerical method and the complete algorithm (few software issues to consider)
- Easy translation from the complete algorithm to a computer code (no numerical issues)
- An effective approach
- Easy to debug
- Easy to switch to another programming language

Writing complete algorithms

- Complete algorithm = mathematical pseudo code: programming language independent!
- Rewrite a compact mathematical formula as a set of simple operations (e.g., replace $\sum$ with a for-loop or do-loop in Fortran)
- Identify input and output
- Give names to mathematical entities and make them variables/arrays
- Introduce intermediate variables (if necessary)

Optimization; rule of thumb

- Adopt good programming habits
- Maintain the clear structure of the numerical method
- Avoid “premature optimization”
- Leave part of the optimization work to a compiler

Example 20: Simpson's rule

- Want to approximate $\int_a^b f(x)dx$
- Similar idea as Trapezoidal rule, better accuracy

$$\int_a^b f(x)dx \approx \frac{h}{6}\sum_{i=1}^{n}\left\{f(x_{i-1}) + 4f(x_{i-\frac{1}{2}}) + f(x_i)\right\}$$

- $h = \dfrac{b-a}{n}$, $x_i = a + ih$, $x_{i-\frac{1}{2}} = \frac{1}{2}(x_{i-1} + x_i)$

Complete algorithm (I)

```
simpson (a, b, f, n)
    h = (b−a)/n
    s = 0
    for i = 1, ..., n
        x⁻ = a + (i − 1)h
        x⁺ = a + ih
        x = ½(x⁻ + x⁺)
        s ← s + f(x⁻) + 4f(x) + f(x⁺)
    end for
    s ← (h/6)s
return s
```

- Input: a, b, f, n
- Output: s
- Intermediate variables: x^-, x, x^+

Efficiency consideration

- $f(x^+)$ in iteration i is the same as $f(x^-)$ in iteration $i+1$

$$
\begin{aligned}
f(x_0)+4f(x_{\frac{1}{2}})+f(x_1)+ \\
f(x_1)+4f(x_{1+\frac{1}{2}})+f(x_2)+ \\
\cdots \\
f(x_{n-1})+4f(x_{n-\frac{1}{2}})+f(x_n)
\end{aligned}
$$

- Unnecessary function evaluations should be avoided for efficiency!
- Rewrite Simpson's rule

$$\int_a^b f(x)dx \approx \frac{h}{6}\left[f(a)+f(b)+2\sum_{i=1}^{n-1} f(x_i)+4\sum_{i=1}^{n} f(x_{i-\frac{1}{2}})\right]$$

Complete algorithm (II)

```
simpson (a, b, f, n)
    h = (b−a)/n
    s1 = 0 x = a
    for i = 1, ..., n−1
        x ← x + h
        s1 ← s1 + f(x)
    end for
    s2 = 0 x = a + 0.5·h
    for i = 1, ..., n
        s2 ← s2 + f(x)
        x ← x + h
    end for
    s = h/6 (f(a) + f(b) + 2s1 + 4s2)
return s
```

- New intermediate variables s_1 and s_2
- Two for-loops (can we combine them into one loop?)

Choosing a programming language

- Many programming languages exist
- We examine 7 languages: Fortran 77, C, C++, Java, Maple, Matlab & Python
- Issues that influence the choice of a programming language
  - Static typing vs. dynamic typing
  - Computational efficiency
  - Built-in high-performance utilities
  - Support for user-defined data types

Static typing vs. dynamic typing

- Statically typed programming languages
  - Each variable must be given a specific type (int, char, float, double etc.)
  - Compiler is able to detect obvious syntax errors
  - Special rules for transformation between different types
- Dynamically typed programming language
  - No need to give a specific type to a variable
  - Typing is dynamic and adjusts to the context
  - Great flexibility and more “elegant” syntax
  - Difficult to detect certain “typos”

Computational efficiency

- Compiled languages run normally fast
  - Program code $\stackrel{\text{compilation \& linking}}{\longrightarrow}$ executable (machine code)
- Interpreted languages run normally slow
  - Statements are interpreted directly as function calls in a library
  - Translation takes place “on the fly”
- Different compiled languages may have different efficiency

Built-in utilities

- Compiled languages have very fast loop-instructions
- Plain loops in interpreted languages (Maple, Matlab & Python) are very slow
- Important for interpreted languages to have built-in numerical libraries
- Need to “break” a complicated numerical method into a series of simple steps when using an interpreted language

User-defined data types

- Built-in primitive data types may not be enough for complicated numerical programming
- Need to “group” primitive variables into a new data type
  - `struct` in C (only data, no function)
  - `class` in C++, Java & Python
  - Class hierarchies $\Rightarrow$ powerful tool $\Rightarrow$ object-oriented programming

Different programming languages

- Different syntax
- Similar structure for main computation
- Different ways for function transfer
- Different I/O
- Different ways for writing comments
- No need to learn all the details at once!
- Learn from the examples!

Vectorization

- Loops are very slow in interpreted languages
- Should use built-in vector functionality when possible

```
trapezoidal_vec (a, b, f, n)
    h = (b−a)/n
    x = (a, a+h, ..., b)
    v = f(x)
    s = h · (sum(v) − 0.5 · (v_1 + v_{n+1}))
return s
```

trapezoidal_vec (a, b, f, n)

$h = \frac{b-a}{n}$

$\mathbf{x} = (a, a+h, \ldots, b)$

$\mathbf{v} = f(\mathbf{x})$

$s = h \cdot (\text{sum}(\mathbf{v}) - 0.5 \cdot (v_1 + v_{n+1}))$

return s

Guidelines on implementation

- Understand the numerics (make use of literature)
- Close resemblance between mathematical pseudo code and numerical method
- Test the implementation on first problems with known solutions
- No premature optimization before code verification
- During later optimization, refer to the “non-optimized” code as reference for checking

Diffusion Processes

Diffusion processes

Examples of diffusion processes

- Heat conduction
    - Heat moves from hot to cold places
- Diffusive (molecular) transport of a substance
    - Ink in water
    - Sugar/Cream in coffee
    - Perfume/Gas in air
- Thin-film fluid flow

Diffusion processes

- Diffusion processes smoothes out differences
- A physical property (heat/concentration) moves from high concentration to low concentration

One dimension

- For simplicity, we will in the following focus on one dimensional examples
- This simplifies the complexity of the numerics and codes, but it would still be realistic in examples with
  - Long thin geometries
  - One dimensional variation only
  - Cylindrical or spherical symmetry
  - Mathematical splitting of dimension

$$u(x,y,z,t) = F(x,t) + G(y,z,t)$$

or

$$u(x,y,z,t) = F(x,t)G(y,z,t)$$

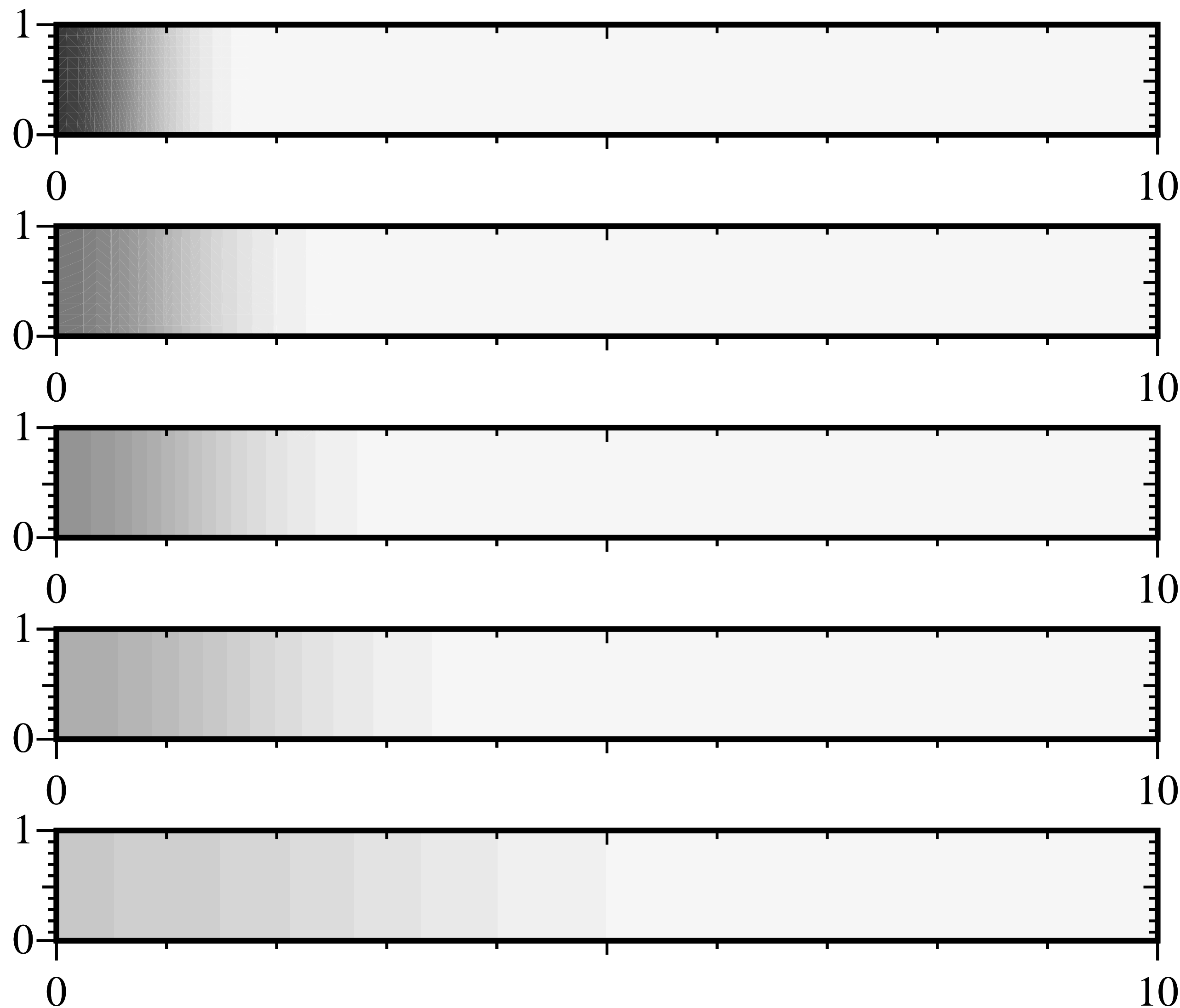

Figure 33: Diffusion of ink in a long and thin tube. The top figure shows the initial concentration (dark is ink, white is water). The three figures below show the concentration of ink at (scaled) times $t = 0.25$, $t = 0.5$, $t = 1$, and $t = 3$, respectively. The evolution is clearly one-dimensional.

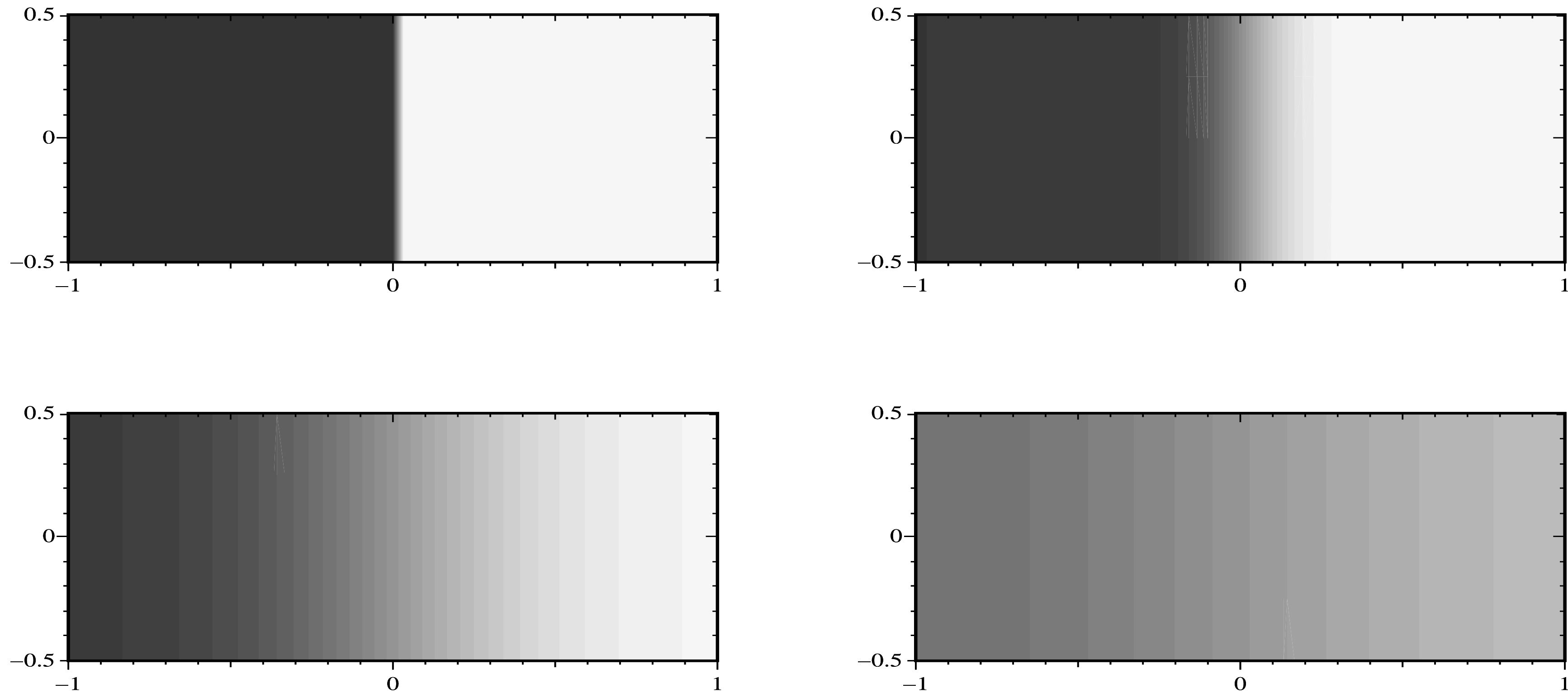

Figure 34: The evolution of the temperature in a medium composed of two pieces of metal, at different initial temperatures. In the gray scale plots, dark is hot and white is cool. The plots correspond to $t = 0$, $t = 0.01$, $t = 0.1$, and $t = 0.5$. All boundaries are insulated, and the temperature approaches a constant value, equal to the average $(T_1 + T_2)/2$ of the initial temperature values.

The Basics of the Mathematical Model

The diffusion equation reads

$$\frac{\partial u}{\partial t} = k\frac{\partial^2 u}{\partial x^2} + f(x,t), \quad x \in (a,b),\ t > 0 \tag{136}$$

- k is a physical parameter
- Large k implies that u spreads quickly

Initial and Boundary conditions

- Let u be a solution of (136), then for any constant C, $u+C$ will also be a solution (136)
- Thus, there are infinitely many solutions of (136)
- In order to make a problem with unique solution we need some initial and boundary conditions
- Initial conditions is that we now the solution initially $u(x,0)$ for $x \in [a,b]$
- Boundary conditions is that we have some information about the solution at the endpoints $u(a,t)$ and $u(b,t)$

Diffusion equation

- In 3 dimensions the diffusion equation reads

$$\frac{\partial u}{\partial t} = k\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) + f(x,y,z,t) \qquad (137)$$

- This equation is sometimes written on a more compact form

$$\frac{\partial u}{\partial t} = k\nabla^2 u + f, \qquad (138)$$

  where the operator ∇^2 is defined by $\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}$

- ∇^2 is called the Laplace operator

Initial conditions

In order to solve the diffusion equation we need some initial condition and boundary conditions.

- The initial condition gives the concentration in the tube at t=0

$$c(x,0) = I(x), \quad x \in (0,1) \tag{139}$$

- Physically this means that we need to know the concentration distribution in the tube at a moment to be able to predict the future distribution

Boundary conditions

Some common boundary conditions are

- Prescribed concentrations, S_0 and S_1, at the endpoints

$$c(0,t) = S_0 \quad \text{and} \quad c(1,t) = S_1$$

- Impermeable endpoints, i.e. no out flow at the endpoints

$$q(0,t) = 0 \quad \text{and} \quad q(1,t) = 0$$

- By Fick's law we get

$$\frac{\partial c(0,t)}{\partial x} = 0 \quad \text{and} \quad \frac{\partial c(1,t)}{\partial x} = 0$$

Boundary conditions

- Prescribed outflows Q_0 and Q_1 at the endpoints

$$-q(0,t) = Q_0 \quad \text{and} \quad q(1,t) = Q_1$$

  - Here the minus sign in the first expression, $-q(0,t) = Q_0$, comes since Q_0 measures the flow out of the tube, and that is the negative direction (from right to left)
  - By Fick's law we get

$$k\frac{\partial c(0,t)}{\partial x} = Q_0 \quad \text{and} \quad -k\frac{\partial c(1,t)}{\partial x} = Q_1$$

Numerical methods

First we consider a version of the heat equation where any varying parameters are scaled away:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + f(x,t), \quad x \in (0,1),\ t > 0. \tag{140}$$

- The solution of this equation is a continuous function of time and space
- We approximate the solution at a finite number of space points and at a finite number of time levels
- This approximation is referred to as discretization
- There are several ways of discretizing (140) - in the following we will consider a technique which is called the finite difference method

Numerical methods

Applying the finite difference method to the problem (140) implies

1. constructing a *grid*, with a finite number of points in (x, t) space, see Figure 35
2. requiring the PDE (140) to be satisfied at each point in the grid
3. replacing derivatives by finite difference approximations
4. calculating u at the grid points only

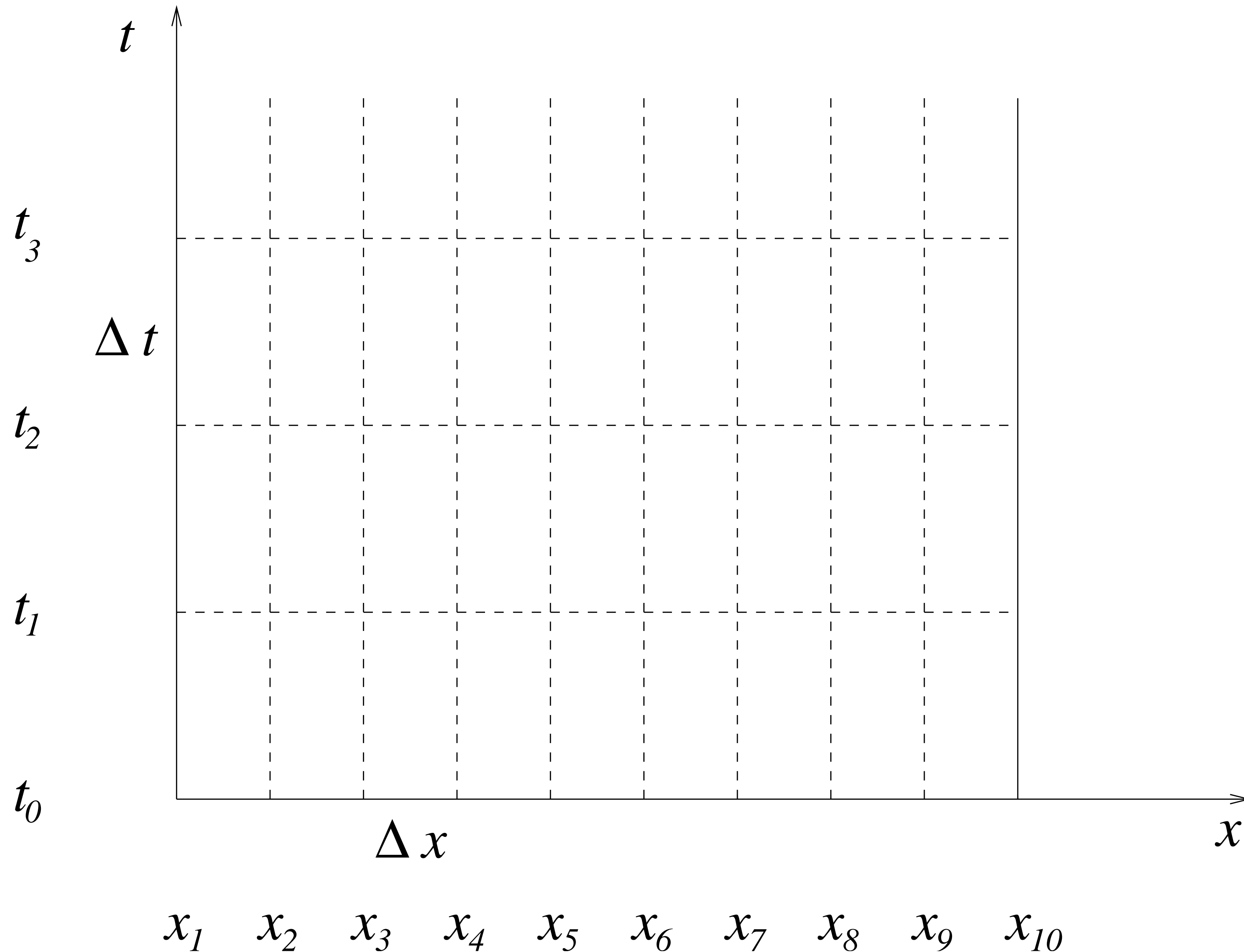

Figure 35: Computational grid in the x,t-plane. The grid points are located at the points of intersection of the dashed lines.

Discrete functions on a grid

- Chose a spatial discretization size Δx and a temporal discretization size Δt
- Functions are only defined in the grid points

$$(x_i, t_\ell),$$

for $i = 1, \ldots, n$ and $\ell = 0, \ldots, m$ where

  - n is the number of approximation points in space ($\Delta x = \frac{1}{n-1}$)
  - $m+1$ is the number of time levels
- The value of an arbitrary function $Q(x,t)$ at a grid point (x_i, t_ℓ) is denoted

$$Q_i^\ell = Q(x_i, t_\ell), \quad i = 1, \ldots, n, \ \ell = 0, \ldots, m$$

Discrete functions on a grid

- The purpose of a finite difference method is to compute the values u_i^ℓ for $i = 1, \ldots, n$ and $\ell = 0, \ldots, m$
- We can now write the PDE (140) as

$$\frac{\partial}{\partial t} u(x_i, t_\ell) = \frac{\partial^2}{\partial x^2} u(x_i, t_\ell) + f(x_i, t_\ell), \qquad (141)$$
$$i = 1, \ldots, n, \; \ell = 1, \ldots, m$$

Finite difference approximation

Now we approximate the terms in (141) that contains derivatives. The approximation is done as follows

- The right hand side is approximated

$$\frac{\partial}{\partial t}u(x_i,t_\ell) \approx \frac{u_i^{\ell+1}-u_i^\ell}{\Delta t} \tag{142}$$

- The first term on left hand side is approximated

$$\frac{\partial^2}{\partial x^2}u(x_i,t_\ell) \approx \frac{u_{i-1}^\ell - 2u_i^\ell + u_{i+1}^\ell}{\Delta x^2} \tag{143}$$

- The first approximation (142) can be motivated directly from the definition of derivatives, since Δt is small, and it is called a finite difference approximation

Finite difference approximation

The motivation for (143) is done in two steps and the finite difference approximation is based on centered difference approximations.

- We first approximate the"outer" derivative at $x = x_i$ (and $t = t_\ell$), using a fictitious point $x_{i+\frac{1}{2}} = x_i + \frac{1}{2}\Delta x$ to the right and a fictitious point $x_{i-\frac{1}{2}} = x_i - \frac{1}{2}\Delta x$ to the left

$$\frac{\partial}{\partial x}\left[\left(\frac{\partial u}{\partial x}\right)\right]_i^\ell \approx \frac{1}{\Delta x}\left[\left[\frac{\partial u}{\partial x}\right]_{i+\frac{1}{2}}^\ell - \left[\frac{\partial u}{\partial x}\right]_{i-\frac{1}{2}}^\ell\right]$$

Finite difference approximation

- The first-order derivative at $x_{i+\frac{1}{2}}$ can be approximated by a centered difference using the point x_{i+1} to the right and the point x_i to the left:

$$\left[\frac{\partial u}{\partial x}\right]_{i+\frac{1}{2}}^{\ell} \approx \frac{u_{i+1}^{\ell} - u_i^{\ell}}{\Delta x}$$

- Similarly, the first-order derivative at $x_{i-\frac{1}{2}}$ can be approximated by a centered difference using the point x_i to the right and the point x_{i-1} to the left

$$\left[\frac{\partial u}{\partial x}\right]_{i-\frac{1}{2}}^{\ell} \approx \frac{u_i^{\ell} - u_{i-1}^{\ell}}{\Delta x}$$

- Combining these finite differences gives (143)

The Finite Difference Scheme

- Inserting the difference approximations (142) and (143) in (141) results in the following finite difference scheme

$$\frac{u_i^{\ell+1} - u_i^{\ell}}{\Delta t} = \frac{u_{i-1}^{\ell} - 2u_i^{\ell} + u_{i+1}^{\ell}}{\Delta x^2} + f_i^{\ell} \tag{144}$$

- We solve (144) with respect to $u_i^{\ell+1}$, yielding a simple formula for the solution at the new time level

$$u_i^{\ell+1} = u_i^{\ell} + \frac{\Delta t}{\Delta x^2}\left(u_{i-1}^{\ell} - 2u_i^{\ell} + u_{i+1}^{\ell}\right) + \Delta t\, f_i^{\ell} \tag{145}$$

- This is referred to as a numerical scheme for the diffusion equation

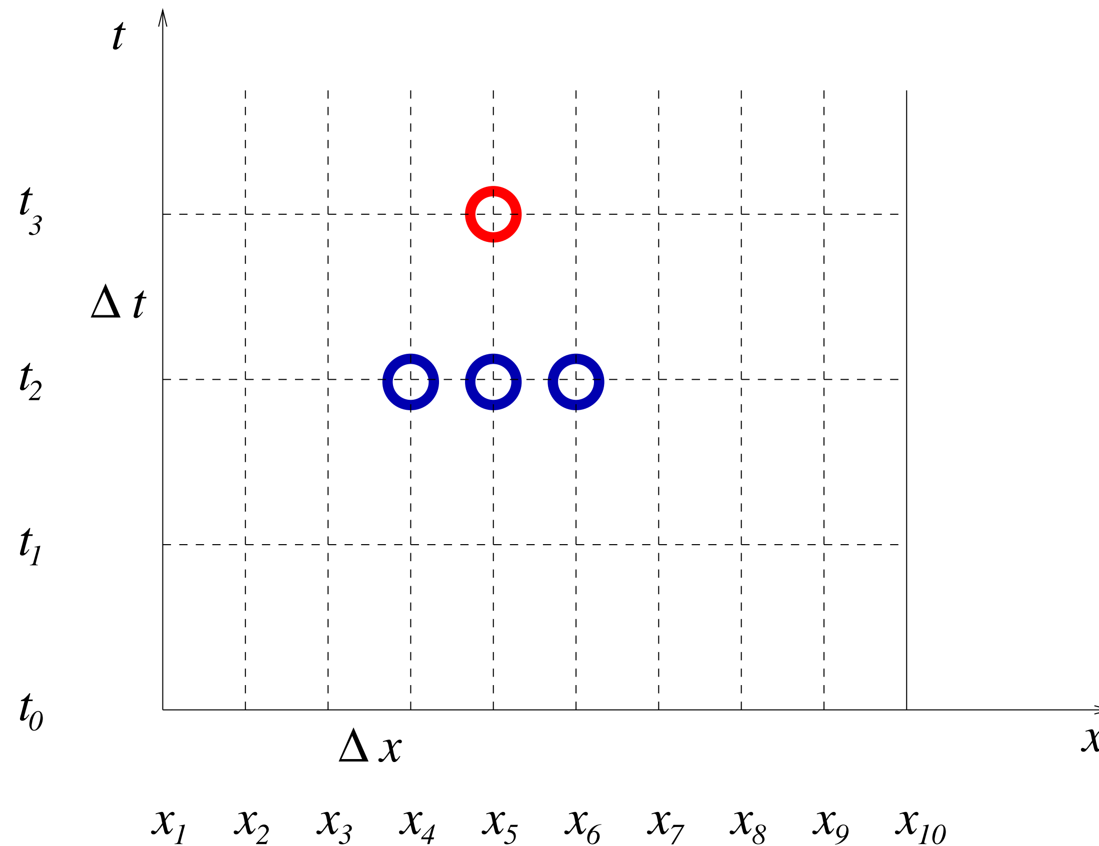

Figure 36: Illustration of the updating formula (145); u_5^3 is computed from u_4^2, u_5^2, and u_6^2.

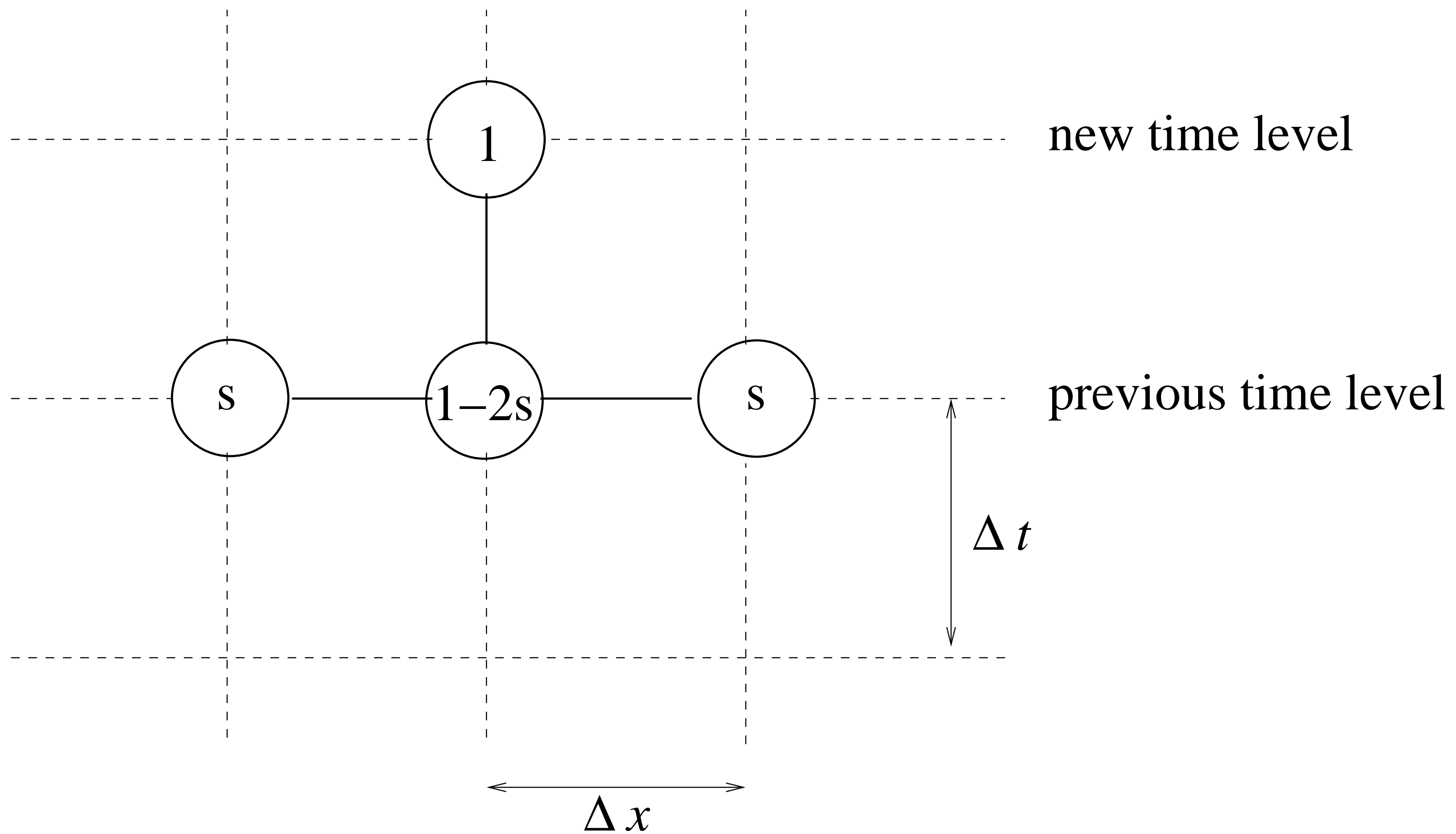

Figure 37: Illustration of the computational molecule corresponding to the finite difference scheme (145). The weight s is $\Delta t/\Delta x^2$.

Incorporating Boundary Conditions

- (145) can not be used for computing new values at the boundary $u_1^{\ell+1}$ and $u_n^{\ell+1}$, because (145) for $i = 1$ and $i = n$ involves values u_{-1}^{ℓ} and u_{n+1}^{ℓ} *outside* the grid.
- Therefore we need to use the boundary conditions to update on the boundary $u_1^{\ell+1}$ and $u_n^{\ell+1}$

Dirichlet Boundary Condition

- Suppose we have the following Dirichlet boundary conditions

$$u(0,t) = g_0(t), \quad u(1,t) = g_1(t),$$

where $g_0(t)$ and $g_1(t)$ are prescribed functions

- The new values on the boundary can then be updated by

$$u_1^{\ell+1} = g_0(t_{\ell+1}), \quad u_n^{\ell+1} = g_1(t_{\ell+1})$$

- The numerical scheme (145) update all inner points

Algorithm 1. Diffusion equation with Dirichlet boundary conditions.
Set initial conditions:

$$u_i^0 = I(x_i), \quad \text{for } i = 1, \ldots, n$$

for $\ell = 0, 1, \ldots, m$:

- Update all inner points:

$$u_i^{\ell+1} = u_i^\ell + \frac{\Delta t}{\Delta x^2}\left(u_{i-1}^\ell - 2u_i^\ell + u_{i+1}^\ell\right) + \Delta t f_i^\ell$$
$$\text{for } i = 2, \ldots, n-1$$

- Insert boundary conditions:

$$u_1^{\ell+1} = g_0(t_{\ell+1}), \quad u_n^{\ell+1} = g_1(t_{\ell+1})$$

Neumann Boundary Conditions

Assume that we have Neumann conditions on the problem

$$\frac{\partial}{\partial x}u(0,t) = h_0 \quad \text{and} \quad \frac{\partial}{\partial x}u(1,t) = h_1$$

Implementing the first condition, $\frac{\partial}{\partial x}u(0,t) = h_0$, can be done as follows

- We introducing a fictisous value u_0^ℓ
- The property $\frac{\partial}{\partial x}u(0,t)$ can then be approximated with a centered difference

$$\frac{u_2^\ell - u_0^\ell}{2\Delta x} = h_0$$

Neumann Boundary Conditions

- The discrete version of the boundary condition then reads

$$\frac{u_2^\ell - u_0^\ell}{2\Delta x} = h_0 \tag{146}$$

or

$$u_0^\ell = u_2^\ell - 2h_0\Delta x$$

- Setting $i = 1$ in (145), gives

$$\begin{aligned} u_1^{\ell+1} &= u_1^\ell + \frac{\Delta t}{\Delta x^2}\left(u_0^\ell - 2u_1^\ell + u_2^\ell\right) + f_1^\ell \\ &= u_1^\ell + \frac{\Delta t}{\Delta x^2}\left(u_2^\ell - 2h_0\Delta x - 2u_1^\ell + u_2^\ell\right) + f_1^\ell \end{aligned}$$

Neumann Boundary Conditions

- We now have a formula for updating the boundary point

$$u_1^{\ell+1} = u_1^\ell + 2\frac{\Delta t}{\Delta x^2}\left(u_2^\ell - u_1^\ell - h_0 \Delta x\right) + f_1^\ell$$

- For the condition $\frac{\partial}{\partial x}u(1,t) = h_1$, we can define a fictitious point u_{n+1}^ℓ
- Similar to above we can use a centered difference approximation of the condition, use (145) with $i = n$ and get

$$u_n^{\ell+1} = u_n^\ell + 2\frac{\Delta t}{\Delta x^2}\left(u_{n-1}^\ell - u_n^\ell + h_1 \Delta x\right) + f_n^\ell \qquad (147)$$

Algorithm 2. Diffusion equation with Neumann boundary conditions.
Set initial conditions:

$$u_i^0 = I(x_i), \quad \text{for } i = 1, \ldots, n$$

for $\ell = 0, 1, \ldots, m$:

- Update all inner points:

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

$$\text{for } i = 2, \ldots, n-1$$

- Insert boundary conditions:

$$u_1^{\ell+1} = u_1^\ell + 2\frac{\Delta t}{\Delta x^2}\left(u_2^\ell - u_1^\ell - h_0 \Delta x\right) + f_1^\ell$$

$$u_n^{\ell+1} = u_n^\ell + 2\frac{\Delta t}{\Delta x^2}\left(u_{n-1}^\ell - u_n^\ell + h_1 \Delta x\right) + f_n^\ell$$

Implementation

We study how Algorithm 1 can be implemented in Python

- Arrays in Python has zero as the first index
- We rewrite Algorithm 1 so that the index i goes from 0 to $n-1$
- That is, we change i with $i-1$

Implementation

- In Algorithm 1, we see that we need to store n numbers for $m+1$ time levels, i.e. $n(m+1)$ numbers in a two-dimensional array
- But, when computing the solution at one time level, we only need to have stored the solution at the previous time level - older levels are not used
- So, if we do not need to store all time levels, we can reduce the storage requirements to $2n$ in two one-dimensional arrays
- Introducing u_i for $u_i^{\ell+1}$ and u_i^- for u_i^ℓ, we arrive at the mathematical pseudo code presented as Algorithm 3

Algorithm 3. Pseudo code for diffusion equation with general Dirichlet conditions.

Set initial conditions:

$$u_i^- = I(x_i), \quad \text{for } i = 0, \ldots, n-1$$

for $\ell = 0, 1, \ldots, m$:

- Set $h = \frac{\Delta t}{\Delta x^2}$ and $t = \ell \Delta t$
- Update all inner points:
$$u_i = u_i^- + h\left(u^- - 2u_i^- + u_{i+1}^-\right) + \Delta t f(x_i, t)$$
$$\text{for } i = 1, \ldots, n-2$$
- Insert boundary conditions:
$$u_0 = g_0(t), \;\; u_{n-1} = g_1(t)$$
- Update data structures for next step:
$$u_i^- = u_i, \; i = 0, \ldots, n-1$$

```
def diffeq(I, f, g0, g1, dx, dt, m, action=None):
    n = int(1/dx + 1)    h = dt/(dx*dx)  # help variable in the scheme
    x = arrayrange(0, 1+dx/2, dx, Float)  # grid points in x dir
    user_data = []  # return values from action function
    # set initial condition:
    um = I(x)
    u = zeros(n, Float)  # solution array

    for l in range(m+1):  # l=0,...,m
        t = l*dt
        # update all inner points:
        for i in range(1,n-1,1):  # i=1,...,n-2
            u[i] = um[i] + h*(um[i-1] - 2*um[i] + um[i+1]) +  dt*f(x[i],
        # insert boundary conditions:
        u[0] = g0(t);  u[n-1] = g1(t)

        # update data structures for next step:
        for i in range(len(u)): um[i] = u[i]
        if action is not None:
            r = action(u, x, t)  # some user-defined action
            if r is not None:
                user_data.append(r)  # r can be arbitrary data...
    return user_data
```

Comments

- The functions f, g_0, and g_1 are given as function arguments for convenience
- We need to specify each array element in the solution `u` to be a floating-point number, otherwise the array would consist of integers. The values of `u` are of no importance before the time loop.
- The `action` parameter may be used to invoke a function for computing the error in the solution, if the exact solution of the problem is known, or we may use it to visualize the graph of $u(x,t)$. The `action` function can return any type of data, and if the data differ from `None`, the data are stored in an array `user_data` and returned to the user.

Verifications

- A well known solution to the diffusion equation is

$$u(x,t) = e^{-\pi^2 t} \sin \pi x, \tag{148}$$

  which is the solution when $f = 0$ and $I(x) = \sin \pi x$ and the Dirichlet boundary conditions are $g_0(t) = 0$ and $g_1(t) = 0$
- We shall see how this exact solution can be used to test the code
- In Python the initial and boundary conditions can specified by

```
def IC_1(x):   return sin(pi*x)
def g0_1(t):   return 0.0
def g1_1(t):   return 0.0
```

Verifications

- We can now construct a function `compare_1` as `action` parameter, where we compute and return the error:

```
def error_1(u, x, t):
    e = u - exactsol_1(x, t)
    e_norm = sqrt(innerproduct(e,e)/len(e))
    return e_norm

def exactsol_1(x, t): return exp(-pi*pi*t)*sin(pi*x)
```

- The `e_norm` variable computes an approximation to the a scalar error measure

$$E = \sqrt{\int_0^1 (\hat{u} - u)^2 dx},$$

where $\hat{u}$ denotes the numerical solution and u is the exact solution

Verifications

- We actually computes a Riemann approximation of this integral since

$$E^2 = \int_0^1 (\hat{u} - u)^2 dx \approx \sum_{i=0}^{n-1} e_i^2 \Delta x = \frac{1}{n-1} \sum_{i=0}^{n-1} e_i^2,$$

where

$$e_i = u_i^\ell - \exp\left(-\pi^2 \ell \Delta t\right) \sin(\pi i \Delta x)$$

(the code divide by n instead of $n-1$, for convenience)

- The final call to `diffeq` reads

```
e = diffeq(IC_1, f0, g0_1, g1_1, dx, dt, m, action=error_1)
print "error at last time level:", e[-1]
```

Verifications

- Theoretically, it is known that

$$E = C_1 \Delta x^2 + C_2 \Delta t$$

- Choosing $\Delta t = D\Delta x^2$ for a positive constant D, we get

$$E = C_3 \Delta x^2, \quad C_3 = C_1 + C_2 D$$

- Hence, $E/\Delta x^2$ should be constant
- A few lines of Python code conduct the test

```
dx = 0.2
for counter in range(4):  # try 4 refinements of dx
    dx = dx/2.0; dt = dx*dx/2.0; m = int(0.5/dt)
    e = diffeq(IC_1, f0, g0_1, g1_1, dx, dt, m, action=error_1)
    print "dx=%12g error=%12g  ratio=%g" % (dx, e[-1], e[-1]/(dx*dx))
```

Verifications

- The output becomes

```
dx=          0.1 error= 0.000633159  ratio=0.0633159
dx=         0.05 error=  0.00016196  ratio=0.0647839
dx=        0.025 error= 4.09772e-05  ratio=0.0655636
dx=       0.0125 error= 1.03071e-05  ratio=0.0659656
```

- This confirms that $E \sim \Delta x^2$

```
from numpy import linspace, zeros, exp, sin, pi
import pylab

def solve(I, f, g0, g1, T, m, L, n):
    dx = L/(n-1.) # n unknowns, n-1 intervals of length dx.
    dt = 1.*T/m;
    alpha = dt/dx**2;

    x = linspace(0, L, n);
    u_new=zeros(n)
    u=I(x)

    im = range(0,n-2);
    i = range(1,n-1);
    ip = range(2,n);

    for l in range(m):
        t = (l+1)*dt
        # inner nodes
        u_new[i] = u[i] + alpha*(u[ip]-2*u[i]+u[im]) + dt*f(t,x[i])
        # boundary conditions
        u_new[0] = g0(t)
        u_new[n-1] = g1(t)
        # copy solution
```

The Heat Equation

The Heat Equation

We study the heat equation:

$$u_t = u_{xx} \quad \text{for } x \in (0,1),\, t > 0, \tag{149}$$
$$u(0,t) = u(1,t) = 0 \quad \text{for } t > 0, \tag{150}$$
$$u(x,0) = f(x) \quad \text{for } x \in (0,1), \tag{151}$$

where f is a given initial condition defined on the unit interval $(0,1)$. We shall in the following study

- physical properties of heat conduction versus the mathematical model (149)-(151)
- analyze the stability properties of the explicit numerical method

Energy arguments

- We define the "energy" of the solution u at a time t by

$$E_1(t) = \int_0^1 u^2(x,t)\,dx \quad \text{for } t \geq 0. \tag{152}$$

- Note that this is not the physical energy
- This "energy" is a mathematical tool, used to study the behavior of the solution
- We shall see that $E_1(t)$ is a non-increasing function of time

Energy arguments

- If we multiply the left and right hand sides of the heat equation (149) by u it follows that

$$u_t u = u_{xx} u \quad \text{for } x \in (0,1),\, t > 0$$

- By the chain rule for differentiation we observe that

$$\frac{\partial}{\partial t} u^2 = 2 u u_t$$

- Hence

$$\frac{1}{2} \frac{\partial}{\partial t} u^2 = u_{xx} u \quad \text{for } x \in (0,1),\, t > 0$$

Energy arguments

- By integrating both sides with respect to x, and applying the rule of integration by parts, we get

$$\begin{aligned}
\frac{1}{2}\int_0^1 \frac{\partial}{\partial t} u^2(x,t)\,dx &= \int_0^1 u_{xx}(x,t)u(x,t)\,dx \qquad (153)\\
&= u_x(1,t)u(1,t) - u_x(0,t)u(0,t)\\
&\quad - \int_0^1 u_x(x,t)u_x(x,t)\,dx\\
&= -\int_0^1 u_x^2(x,t)\,dx \quad \text{for } t > 0,
\end{aligned}$$

where the last equality is a consequence of the boundary condition (150)

Energy arguments

- We assume that u is a smooth solution of the heat equation, which implies that we can interchange the order of integration and derivation in (153), that is

$$\frac{\partial}{\partial t}\int_0^1 u^2(x,t)\,dx = -2\int_0^1 u_x^2(x,t)\,dx \quad \text{for } t>0 \qquad (154)$$

- Therefore

$$E_1'(t) = -2\int_0^1 u_x^2(x,t)\,dx \quad \text{for } t>0$$

- This implies that

$$E_1'(t) \leq 0$$

Energy arguments

- Thus E_1 is a non-increasing function of time t, i.e.,

$$E_1(t_2) \leq E_1(t_1) \quad \text{for all } t_2 \geq t_1 \geq 0$$

- In particular

$$\int_0^1 u^2(x,t)\,dx \leq \int_0^1 u^2(x,0)\,dx \;=\; \int_0^1 f^2(x)\,dx \quad \text{for } t > 0 \qquad (155)$$

Energy arguments

- This means that the energy, in the sense of $E_1(t)$, is a non-increasing function of time
- The integral of u_x^2 with respect to x, tells us how fast the energy decreases

Maximum principles

A smooth solution of the problem (149)-(151) must satisfy the bound

$$m \le u(x,t) \le M \quad \text{for all } x \in [0,1],\, t > 0, \qquad (156)$$

where

$$m = \min\left(\min_{t\ge 0} g_1(t),\, \min_{t\ge 0} g_2(t),\, \min_{x\in(0,1)} f(x)\right), \qquad (157)$$

$$M = \max\left(\max_{t\ge 0} g_1(t),\, \max_{t\ge 0} g_2(t),\, \max_{x\in(0,1)} f(x)\right). \qquad (158)$$

Stability analysis of the num. sol.

- We shall now study the stability properties of the explicit finite difference scheme for heat equation presented earlier
- As above, the discretization parameters are defined by

$$\Delta t = \frac{T}{m} \quad \text{and} \quad \Delta x = \frac{1}{n-1},$$

and functions are only defined in the gridpoints

$$u_i^\ell = u(x_i, t_\ell) = u((i-1)\Delta x, \ell \Delta t)$$

for $i = 1, \ldots, n$ and $\ell = 0, \ldots, m$

Stability analysis of the num. sol.

- The numerical scheme is written

$$u_i^{\ell+1} = u_i^\ell + \frac{\Delta t}{\Delta x^2}(u_{i-1}^\ell - 2u_i^\ell + u_{i+1}^\ell) = \alpha u_{i-1}^\ell + (1-2\alpha)u_i^\ell + \alpha u_{i+1}^\ell \tag{159}$$

for $i = 2, \ldots, n-1$ and $\ell = 0, \ldots, m-1$, where

$$\alpha = \frac{\Delta t}{\Delta x^2} \tag{160}$$

- Boundary conditions are $u_0^\ell = u_1^\ell = 0$ for $\ell = 1, \ldots, m$
- We shall see that this numerical scheme is only conditionable stable, and the stability depends on the parameter α

Example 29

Consider the following problem

$$
\begin{aligned}
u_t = u_{xx} \quad & \text{for } x \in (0,1),\, t > 0, \\
u(0,t) = u(1,t) = 0 \quad & \text{for } t > 0, \\
u(x,0) = \sin(3\pi x) \quad & \text{for } x \in (0,1),
\end{aligned}
$$

with the analytical solution

$$
u(x,t) = e^{-9\pi^2 t} \sin(3\pi x).
$$

In Figures 38-40 we have graphed this function and the numerical results generated by the scheme (159) for various values of the discretization parameters in space and time. Notice how the solution depends on α.

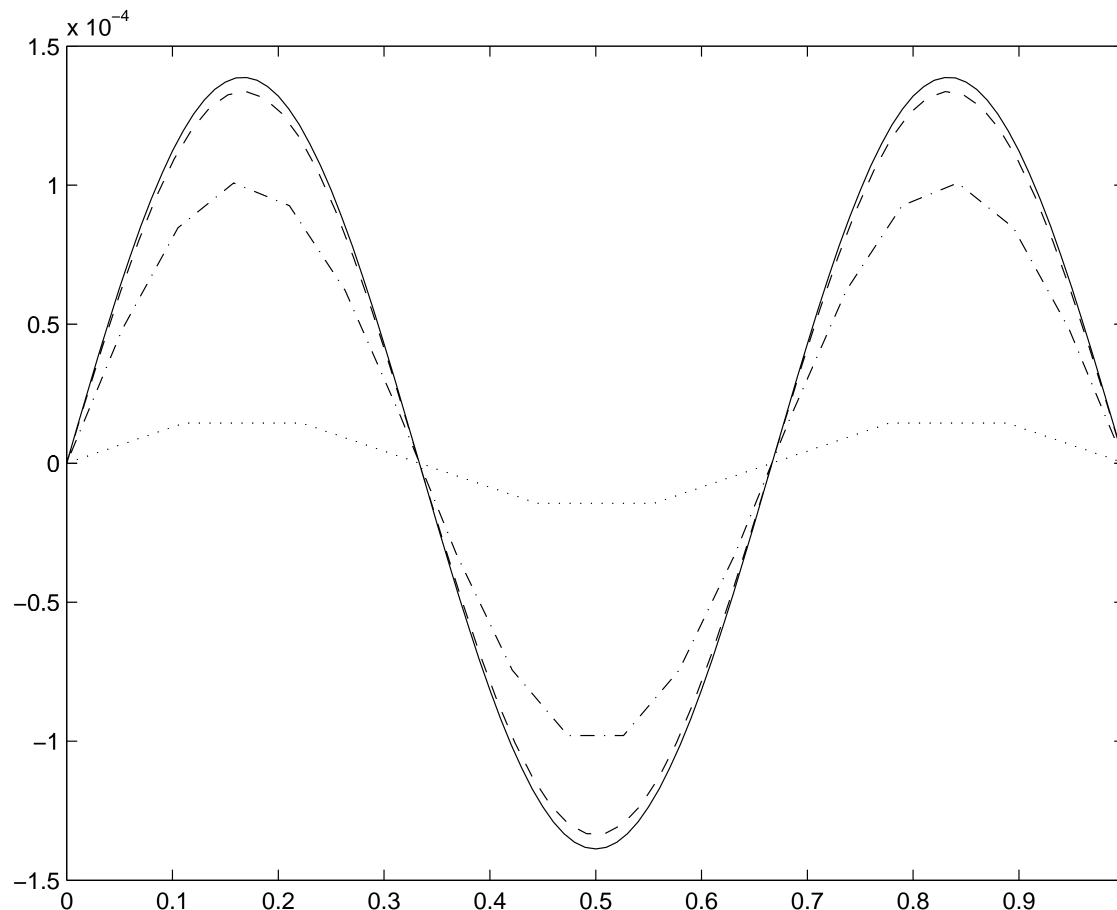

Figure 38: The solid line represents the solution of the problem studied in Example 29. The dotted, dash-dotted and dashed lines are the numerical results generated in the cases of $n = 10$ and $m = 17$ ($\alpha = 0.4765$), $n = 20$ and $m = 82$ ($\alpha = 0.4402$), $n = 60$ and $m = 706$ ($\alpha = 0.4931$), respectively.

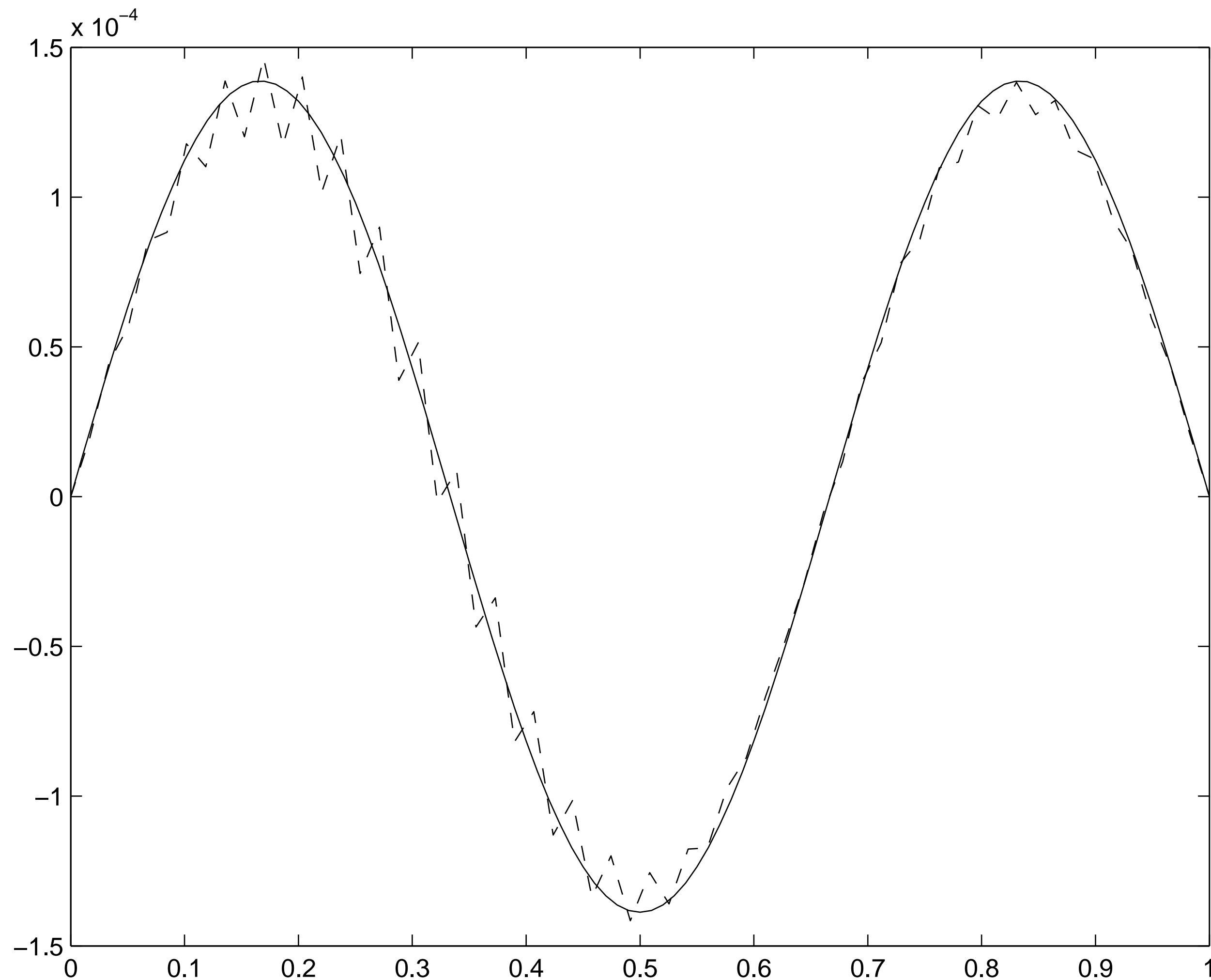

Figure 39: The dashed line represents the results generated by the explicit scheme (159) in the case of $n = 60$ and $m = 681$, corresponding to $\alpha = 0.5112$, in Example 29. The solid line is the graph of the exact solution of the problem studied in this example.

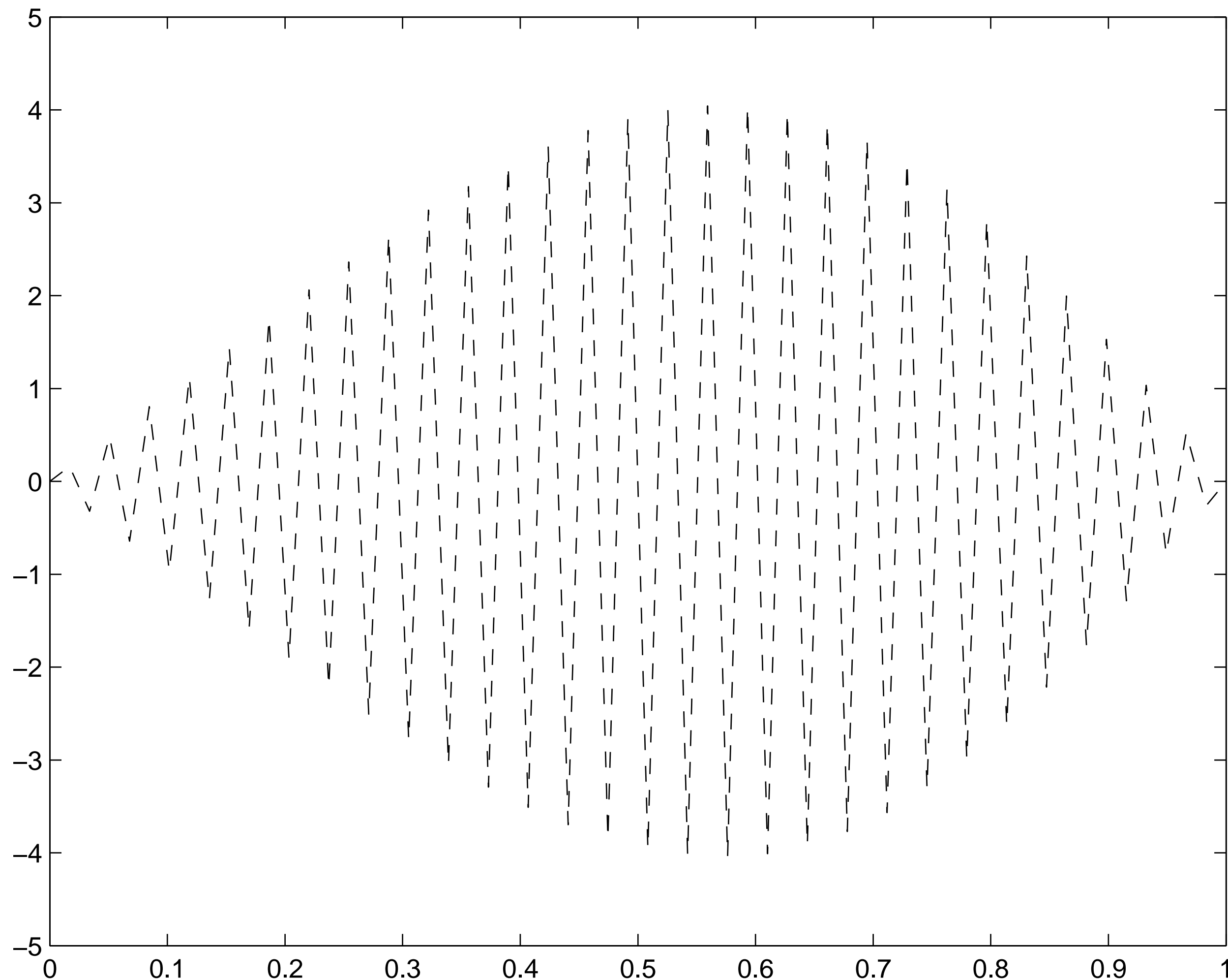

Figure 40: A plot of the numbers generated by the explicit scheme (159), with $n = 60$ and $m = 675$, in Example 29. Observe that $\alpha = 0.5157 > 0.5$ and that, for these discretization parameters, the method fails to solve the problem under consideration!

Analysis

- We have observed that the explicit scheme (159) works fine, provided that $\alpha \leq 1/2$
- For small discretization parameters Δt and Δx, it seems to produce accurate approximations of the solution of the heat equation
- However, for $\alpha > 1/2$ the scheme tends to "break down", i.e., the numbers produced are not useful. Our goal now is to investigate this property from a theoretical point of view
- We will derive, provided that $\alpha \leq 1/2$, a discrete analogue to the maximum principle
- Note that, for (149)-(151), the maximums principle implies

$$|u(x,t)| \leq \max_x |f(x)| \quad \text{for all } x \in (0,1) \text{ and } t \geq 0$$

Analysis

- Assume that Δt and Δx satisfy

$$\alpha = \frac{\Delta t}{\Delta x^2} \leq \frac{1}{2}$$

- Then

$$1 - 2\alpha \geq 0 \tag{161}$$

- We introduce

$$\bar{u}^{\ell} = \max_i |u_i^{\ell}| \quad \text{for } \ell = 0, \ldots, m$$

- Note that

$$\bar{u}^0 = \max_i |f(x_i)|$$

Analysis

- Recall that $u_i^{\ell+1} = \alpha u_{i-1}^{\ell} + (1-2\alpha)u_i^{\ell} + \alpha u_{i+1}^{\ell}$
- It now follows from the triangle inequality that

$$
\begin{aligned}
|u_i^{\ell+1}| &= |\alpha u_{i-1}^{\ell} + (1-2\alpha)u_i^{\ell} + \alpha u_{i+1}^{\ell}| \\
&\leq |\alpha u_{i-1}^{\ell}| + |(1-2\alpha)u_i^{\ell}| + |\alpha u_{i+1}^{\ell}| \\
&= \alpha|u_{i-1}^{\ell}| + (1-2\alpha)|u_i^{\ell}| + \alpha|u_{i+1}^{\ell}| \\
&\leq \alpha\bar{u}^{\ell} + (1-2\alpha)\bar{u}^{\ell} + \alpha\bar{u}^{\ell} \\
&= \bar{u}^{\ell} \qquad (162)
\end{aligned}
$$

for $i = 2, \ldots, n-1$

- Note that

$$u_1^{\ell+1} = u_n^{\ell+1} = 0$$

Analysis

- Since (162) is valid for $i = 2, \ldots, n-1$, we get

$$\max_i |u_i^{\ell+1}| \leq \bar{u}^\ell$$

- or

$$\bar{u}^{\ell+1} \leq \bar{u}^\ell$$

- Finally, by a straightforward induction argument we conclude that

$$\bar{u}^{\ell+1} \leq \bar{u}^0 = \max_i |f(x_i)|$$

Analysis

Assume that the discretization parameters Δt and Δx satisfy

$$\alpha = \frac{\Delta t}{\Delta x^2} \leq \frac{1}{2}. \tag{163}$$

Then the approximations generated by the explicit scheme (159) satisfy the bound

$$\max_i |u_i^\ell| \leq \max_i |f(x_i)| \quad \text{for } \ell = 0, \ldots, m, \tag{164}$$

where f is the initial condition in the model problem (149)-(151).

Consequences

- For a given n, m must satisfy

$$m \geq 2T(n-1)^2$$

- Hence, the number of time steps, m, needed increases rapidly with the number of grid points, n, used in the space dimension
- If $T = 1$ and $n = 101$, then m must satisfy $m \geq 20000$, and in the case of $n = 1001$ at least $2 \cdot 10^6$ time steps must be taken!
- This is no big problem in 1D, but in 2D and 3D this problem may become dramatic