86.5 points

# NUMERICAL DIFFERENTIATION AND INTEGRATION

37.5 points

By

Zemenu Mihret

**Bahir Dar University**

February 2007

# Table of Contents

**Table of Contents** iv

**Abstract** v

**1 Introduction** 1

**2 Numerical differentiation** 3
2.1 Taylor expansion method . . . . . 3
2.1.1 Forward difference approximation . . . . . 3
2.1.2 Backward difference approximation . . . . . 4
2.1.3 The central difference approximation . . . . . 4

**3 Numerical integration** 7
3.1 Riemann's sum . . . . . 7
3.2 Trapezoidal rule . . . . . 11
3.3 Simpson's rule . . . . . 14
3.3.1 Simpson's 1/3 rule . . . . . 14
3.3.2 Simpson's 3/8 rule . . . . . 17
3.4 Gauss quadrature . . . . . 19

**4 Initial value problems of ordinary differential equations** 20
4.1 Introduction . . . . . 20
4.2 First order ordinary differential equations . . . . . 20
4.3 Euler methods . . . . . 21
4.3.1 Euler forward method . . . . . 21
4.3.2 Euler's backward method . . . . . 30
4.3.3 Modified Euler's method . . . . . 31
4.4 Runge-Kutta methods . . . . . 31
4.4.1 second-order Runge-Kutta methods . . . . . 31
4.4.2 Third-order Runge-Kutta method . . . . . 32
4.4.3 Fourth-order Runge-Kutta method . . . . . 33

# Abstract

This project,submitted for the course computational physics one, is concerned with numerical techniques of solving equations. Numerical methods of differentiation, integration and solving solutions of initial-value problems are discussed. Under numerical differentiation, forward and backward methods are dealt with and under numerical integration, trapezoidal, Simpson rules, and Gaussian quadrature are discussed. Again, under numerical methods of solving initial-value problems, Euler's methods and Runge-Kutta methods are included. To each part of the methods, practical examples of physical problems with R programs are included.

# Chapter 1

# Introduction

Numerical differentiation, or difference approximation, is a method used to evaluate derivatives of a function using the functional values at discrete data points. If the functional values are known at discrete points, the function can be expressed approximately by an integration polynomial. Then, by differentiating the interpolation polynomial, the derivatives of the function can be evaluated.

In approximating the first and higher derivatives of a function, errors arise. The level of accuracy of the result obtained depends on the number of terms of the Taylor series that are used in solving numerical differentiation formulas. These errors that result from using an approximation in place of an exact mathematical procedure (analytical methods) are called truncation errors.

Numerical integration is the approximate computation of an integral using numerical techniques. The numerical computation of an integral is sometimes called quadrature. The basic principle of numerical integration schemes is to fit a polynomial to functional data points and then integrate it.

There are a wide range of methods available for numerical integration. The most straightforward numerical integration technique uses the Newton-Cotes formulas (also called quadrature formulas), which approximate a function tabulated at a sequence of regularly spaced intervals by various degree polynomials. If the endpoints are tabulated, then the 2- and 3-point formulas are called the trapezoidal rule and Simpson's rule, respectively.

The fundamental theorem of calculus tells us that if we know the rate of change of some quantity, then adding up (or integrating) the rate of change over some interval will give the total change in that quantity over the same interval. For example, if a car is

moving along a straight line and we know the speed of the car as a function of time, it is possible to determine the total change in the car's position over some time interval.

However, if we don't know a formula for the car's velocity, but we only have measured its velocity at certain instants of time, it is difficult to solve the problem analytically. Numerical integration reduces this burden and helps to estimate the change in the car's position the measured discrete data.

# Chapter 2

# Numerical differentiation

6.5 points out of 18

## 2.1 Taylor expansion method

The Taylor expansion method is an alternative way of deriving numerical differentiation formulas, or difference approximations. It not only derives the difference formulas systematically, but also derives the error terms.
For a derivative of order p, the minimum number of data points necessary to derive a difference approximation for the first derivative of a function needs at least two points.
The Taylor expansion of a function f(x) at x+h is given by:

$$f(x+h) = f(x) + h * f'(x) + h^2 * f''(x)/2! + h^3 * f'''(x)/3! + \dots \quad (2.1)$$

where h is called the step size, that is, the length of the interval over which the approximation is made.

### 2.1.1 Forward difference approximation

Here the numerical differentiation technique is c [at where] ifference because it utilizes data at i and i+1 to estimate the derivative o f we ignore all the terms except the first term on the right side of equation 4.1, the forward difference approximation is obtained; the terms ignored constitute the truncation error, which is represented by the leading term,

$$-(h/2)f_i''$$

in appropriate

Other terms vanish more rapidly than the leading term when h is decreased. The Taylor expansion of

$$f_{i+1}$$

is expressed as:

$$f_{i+1} = f_i + h * f_i' + h^2/2! * \quad (2.2)$$

this is wrong

The forward difference approximation is expres[...] error, as

$$f_i' = (f_{i+1} - f_i)/h + E \quad (2.3)$$

where E

$$\cong -h/2 * f_i''$$

The term E indicates that error is approximately proportional to the grid interval h. The error is also proportional to the second derivative f".

### 2.1.2 Backward difference approximation

It is called backward difference because it [...] say to calculate the values at i of

$$f_{i-1}$$

is

$$f_{i-1} = f_i - h * f_i' + h^2/2 * f_i'' - h^3/3! * f_i''' + \ldots \quad (2.4)$$

Solving for

$$f_i$$

the backward difference approximation is obtained as

$$f_i' = (f_i - f_{i-1})/h + E \quad (2.5)$$

### 2.1.3 The central difference approximation

Here the derivative of a function is calculated using data on both sides of a chosen point

this is very rough

using forward and backward two-step for[...]erward, that is, using

and

Then, the three-step formula, or central difference approximation, is derived as shown below.

$$f_{i+1} - f_{i-1} = 2 * h * f_i' + h^3 * f_i''' + \ldots \tag{2.6}$$

where the

$$f_i$$

has been automatically eliminated. by solving for

$$f_i'$$

remove this kind of separated symbols

we get

$$f_i' = (f_{i+1} - f_{i-1})/2 * h - h^2 f_i'''/3! + \ldots \tag{2.7}$$

where

$$-h^2 f_i'''/3! = E$$

is the error term. Here, because of cancellation of the f" term, the error of the central difference approximation is proportional to

$$h^2$$

rather than h. When h is decreased, the error decreases more quickly than with the other two approximations.

***Practice 1***

```
#This is a program that calculates the first and second
derivatives of cos(x) x=c(0,1,2,3,4,5,6)
h=c(.1,.01,.001,.0001,.00001)
first_derivative=matrix(0,length(x),length(h))
second_derivative=matrix(0,length(x),length(h)) for(i in
1:length(x)){ for(j in 1:length(h)){
first_derivative[i,j]=(cos(x[i]+h[j])-cos(x[i]-h[j]))/(2*h[j])
second_derivative[i,j]=(cos(x -sin(x) x[i])-cos(x[i]-h[j]))/h[j]^2}}
first_exact=-cos(x)
plot(x,abs(first_derivative[,1]-first_exact),main=("Deviation of
first numerical derivation of cos(x) from the analitical one "))
lines(x,abs(first_derivative[,1]-first_exact),type="l",col="black",
pch=21,bg=3,lwd=1,cex=2)
> source("C:\\Documents and Settings\\Administrator\\Desktop\\der.R")
```

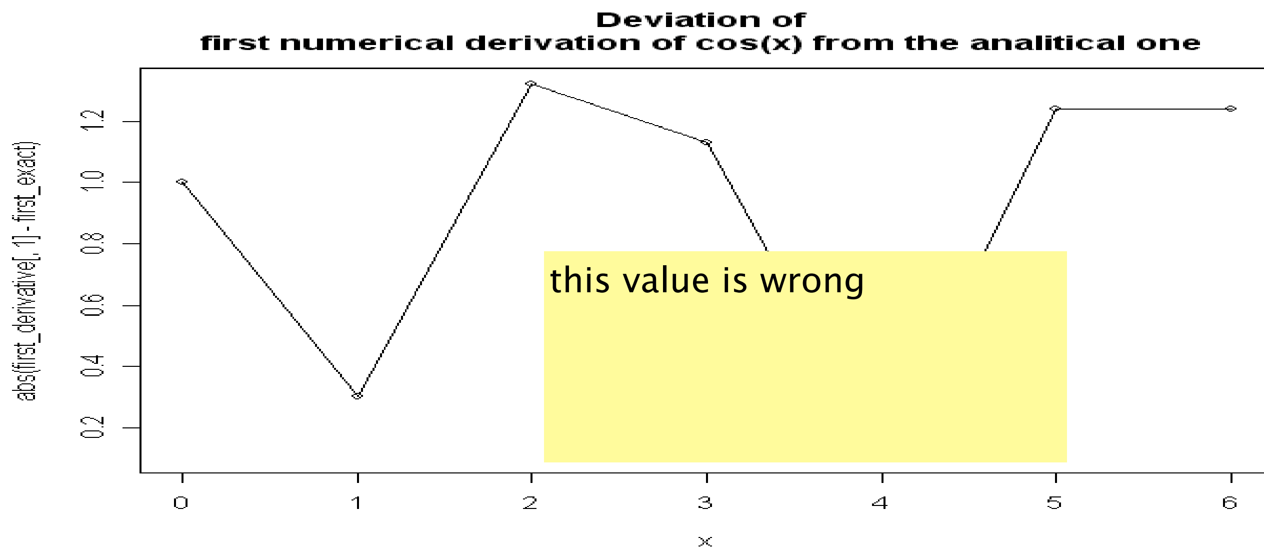


Figure 2.1: comparison of numerical and analytical derivative of cos (x)

```
> first_derivative
           [,1]       [,2]       [,3]       [,4]       [,5]
[1,]  0.0000000  0.0000000  0.0000000  0.0000000  0.0000000 [2,]
-0.8400692 -0.8414570 -0.8414708 -0.8414710 -0.8414710 [3,]
-0.9077827 -0.9092823 -0.9092973 -0.9092974 -0.9092974 [4,]
-0.1408849 -0.1411177 -0.1411200 -0.1411200 -0.1411200 [5,]
0.7555418  0.7567899  0.7568024  0.7568025  0.7568025 [6,]
0.9573269  0.9589083  0.9589241  0.9589243  0.9589243 [7,]
0.2789500  0.2794108  0.2794155  0.2794155  0.2794155
> second_derivative
           [,1]       [,2]       [,3]       [,4]         [,5]
[1,] -200.00000 -20000.000 -2000000.0 -200000000 -20000000000 [2,]
-124.86185 -10974.338 -1082287.6 -108077291 -10806214412 [3,]
65.07371   8141.080   830475.1   83211181   8322754871 [4,]
195.18080  19771.626  1979702.8  197995677  19799821708 [5,]
145.83956  13224.230  1308800.8  130743860  13073023778 [6,]
-37.58590  -5481.462  -565406.5  -56713259  -5673051924 [7,]
-186.45506 -19147.524 -1919781.7 -192028469 -19203349850
```

# Chapter 3

# Numerical integration

## 3.1 Riemann's sum

3.5 points

Let f be a continuous function whose domain includes the closed interval [a, b]. Investigating ways of approximating the definite integral

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

will be useful when we cannot find an elementary antiderivative for f or if the function is defined using data obtained from some experiment.

First, we will divide the interval [a,b] into n subintervals

$$[x_0, x_1], [x_1, x_2], \ldots, [x_{n-1}, xn]$$

where

$$a = x_0 < x_1 < \ldots < x = b.$$

(This is called a partition of the interval.) If the partition is equal,

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

is approximated by

$$I = \sum_0^N f(a + i * h)$$

*h is missing

where N is the number of divisions and h is the length of the interval.

The intervals need not all be the same length, so we can call the lengths of the intervals

$$h_1, h_2, ..$$

put them in the text

respectively. This partition divides the region R into n strips. Next, let's approximate

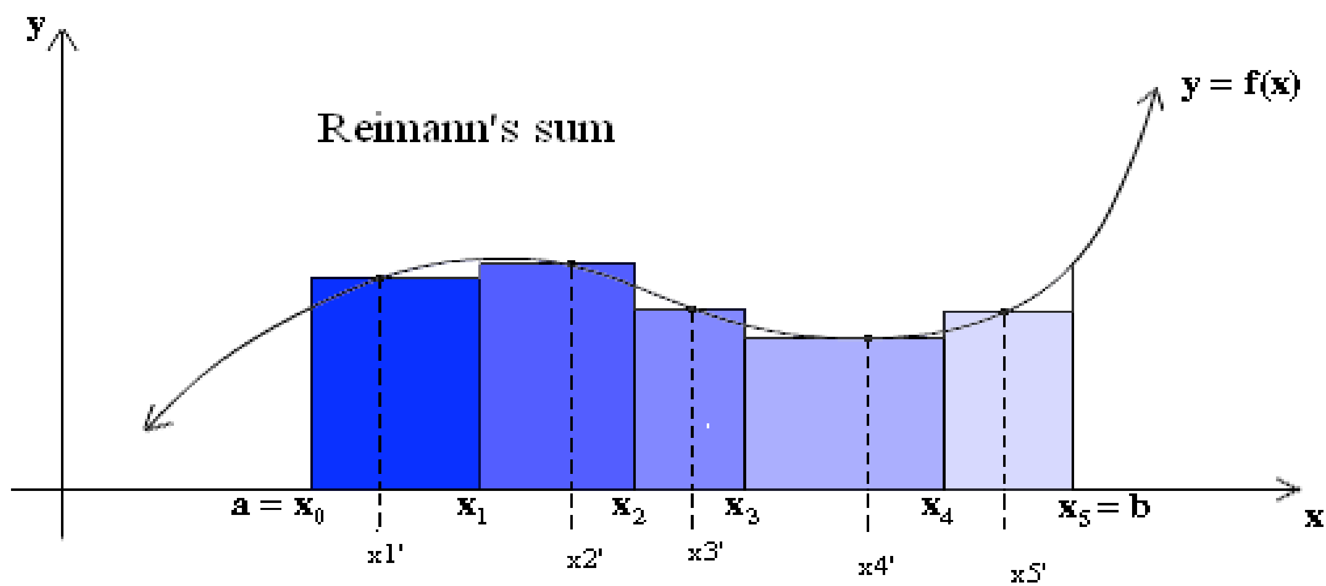


Figure 3.1: Reimann' sum

each strip by a rectangle with height equal to the height of the curve y = f(x) at some arbitrary point in the subinterval.

***Practice 2***

```
#This is a program that integrates the function x^3+2*x
numerically. functionexperssion2=function(x){ x^3+2*x}
start_integrate=0 # initial value of integration
end_integrate=2#a
value where the integration terminates
h=20 # here we set the
integration interval (resolution)
sumdiffh=0 # here we initialize
the variables we will get results for different h
N=(end_integrate-start_integrate)/h # here the number of
subintervals are calculated
H=c() # here we initialize the
variable we will use to collect h
sum=0 # here we initialize the
variable we will use to collect the result
for (j in 0:2){ h=h/10
# Here we improve h by 10 times in each step
 H[j]=h
N=(end_integrate-start_integrate)/h
for (i in 0:N-1){
sum=sum+functionexperssion2(start_integrate+i*h)*h }
sum=sum+h/2*(functionexperssion2(start_integrate)+functionexperssion2(end_integ
sumdiffh[j]=sum sum=0 }
Exact_result=rep(8,2)
pdf("integration_Trapezium.pdf")
plot(H,sumdiffh,xlab="Integration
step size (h)", ylab="Integration result") lines(H,
sumdiffh,type="l",  col="black", pch=1, bg="2", lwd=1,cex=2)
lines(H, Exact_result, type="p",  col="black", pch=24, bg="2",
lwd=1,cex=2) lines(H, Exact_result, lty=3,  col="black", pch=24,
bg="2", lwd=4,cex=2) title( "Performance of numerical
integration") legend(0.2, 66, c("Numerical", "Exact"), pch= c(1,
24) ) dev.off()
```

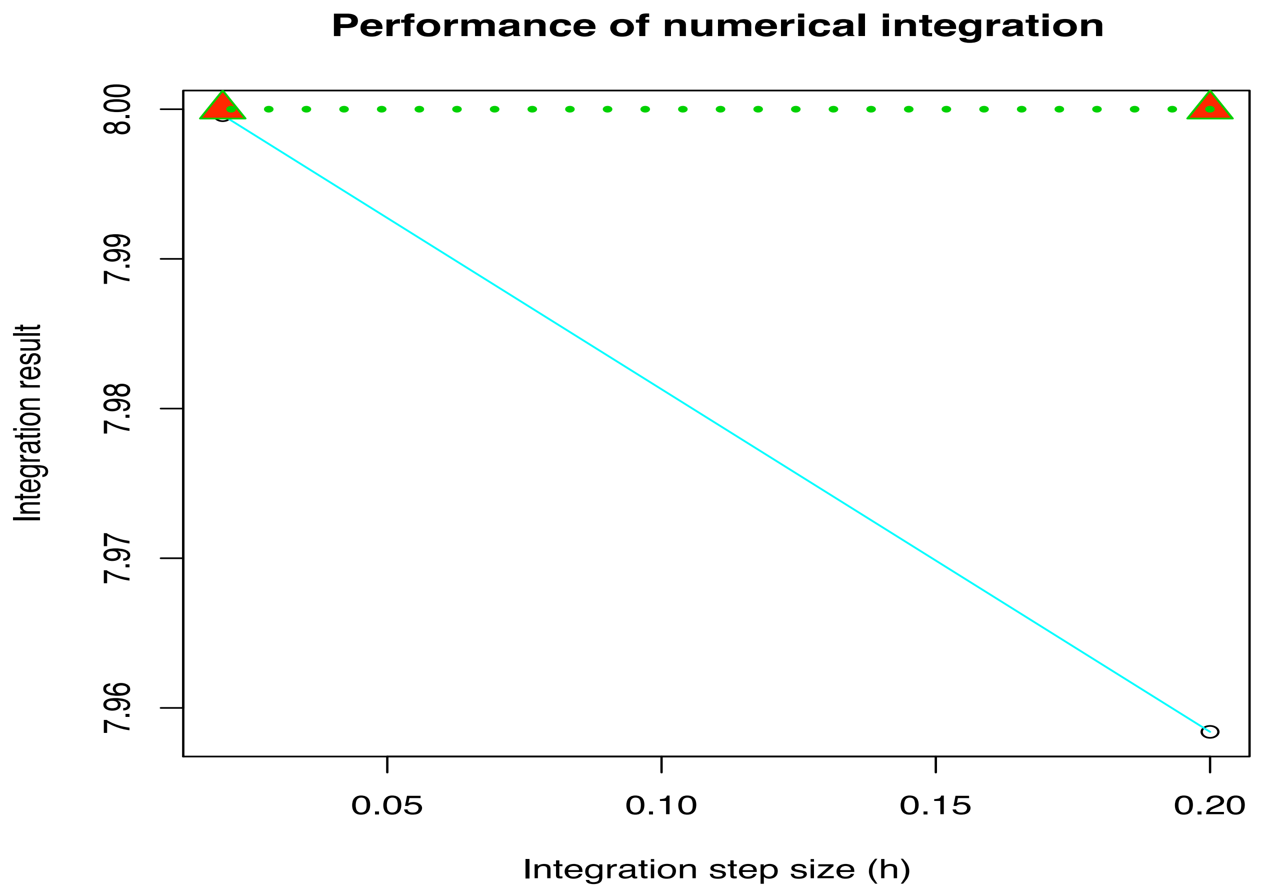


Figure 3.2: Riemann integration

3.5 points

## 3.2 Trapezoidal rule

The trapezoidal rule is a nume[illegible] interpolation polynomial formula. For a single segment, or interval, the trapezoidal rule is given by:

$$I = \int_a^b f_1(x)dx \cong h/2(f_1 + f_2) \tag{3.1}$$

For multiple intervals, if the [illegible]ted by n+1 data points with equally spaced abscissa points, then [illegible]ed repeatedly to each interval. The equation thus obtained for ext[illegible]le and is written as:

where is 4.1

$$I = \int_a^b f(x)dx = h/2(f_1 + 2 * f_2 + \ldots + 2 * f_n + f_{n+1}) + E \tag{3.2}$$

$$h = (b - a)/n$$

$$x_i = a + (i - 1) * h$$

$$f_i = f(x_i)$$

$$i = 1 : n + 1$$

***Practice 3***

```
# A program that integrates the given function using trapezoidal
rule. trapez_two=function(f,h){ n=length(f)-1 s1=f[1]+f[length(f)]
s2=sum(2*f[seq(2,n-1,a + n*h)]) s3=s1+s2 s=(s3)*h/2 list(s=s) }
> source("F:\\func
> x=seq(0,1,0.1)
> f=x
> a=0
> h=0.1
> n=length(f)-1
> n
[1] 10
> trapez_two(f,h)
$s [1] 0.41
```

This program is wrong

Since

$$f''(x_o)$$

vanishes for a linear (first-order polynomial), the trapezoidal rule will be exact if the function being integrated is linear.

***Example*** (1) f(x)=2*x+1 in [0,2]

Numerically

$$f_1(x) = (2-0)*(2*f[0]+12*f[2]+1) = 6$$

Analitically

$$f_1(x) = \int_0^1 (2*x+1)dx = x^2 + x from 0 to 1 = 4+2 = 6$$

***Example*** (2)

$$f(x) = x^2 in [0,2]$$

Numerically

$$f(x) = (2-0)[x^2[0] + x^2[2]]) = 4$$

Analitically

$$f_1(x) = \int_0^2 (x^2)dx = x^3/3 from 0 to 2 = 8/3$$

$$\Longrightarrow$$

trapezoidal rule is effective for first order polynomial.

***Practice 4***

```
# A program that integrates a given function by an extended rule
of trapezoidal trapez_extended=function(f,h){#h is the interval
between the consecutive abssisa point
I=h*(sum(f)-0.5*(f[1]+f[length(f)]))# f is the value of the
function at each point list(I=I) }

> source("F:\\functions\\trapez_extended.R")
> x=seq(0,3,0.1)
> f=x
> h=0.1
> trapez_extended(f,h)
$I [1] 4.5

> x=seq(0,3,0.1)
> f=x
> x=seq(0,3,0.1)
> f=^2
Error: syntax error in "f=^"
> x=seq(0,3,0.1)
```

```
> f=x^2
> h=0.1
> trapez_extended(f,h)
$I [1] 9.005

> f=x^3
> trapez_extended(f,h)
$I [1] 20.2725

> # As we can see from the results of numerical integration obtained
> # trapezoidal rule is effective for a function of degree one and less.  This
> # is because the error term vanishes if first-order polynomial
is used to fit the data.
```

***Practice 5*** In the following program and the figure below it, the forward, backward, and central differentiation of log(x)is compared with its exact or anaclitic derivation.

```
# A program that calculates the first derivative of log(x)
x=1:100
h=c(0.1,0.01,0.001,0.0001,0.00001)
first_forward=matrix(0,length(x),length(h))
for(i in 1:length(x)){
for(j in 1:length(h)){
first_forward[i,j]=(log(x[i]+h[j])-log(x[i]))/(h[j])
}
}


x=1:100 h=c(0.1,0.01,0.001,0.0001,0.00001)
first_backward=matrix(0,length(x),length(h))
for(i in1:length(x)){
for(j in 1:length(h)){
first_backward[i,j]=(log(x[i])-log(x[i]-h[j]))/(h[j])}}


x=1:100 h=c(0.1,0.01,0.001,0.0001,0.00001)
first_central=matrix(0,length(x),length(h))
for(i in 1:length(x)){
for(j in 1:length(h)){
```

```
first_central[i,j]=(log(x[i]+h[j])-log(x[i]-h[j]))/(2*h[j])
}
}


first_exact=1/x pdf("first_derivative.pdf")
plot(x,first_forward[,1],xlab="x
value",ylab="first_derivative",type="l")
lines(x,first_forward[,1],type="l",col=5,pch=21,
bg="3",lwd=3,cex=2,lty="dotted")
lines(x,first_exact,type="l",col=6,pch=21,bg="3",lwd=3,cex=2,lty="dotted")
lines(x,first_backward[,1],type="l",col=3,pch=21,
bg="3",lwd=3,cex=2,lty="dotted")
lines(x,first_central[,1],type="l",col=2,pch=21,
bg="3",lwd=3,cex=2,lty="dotted") title("Comparison of forward,
backward, and central difference derivatives of log(x)")
legend(0.1,0.75, c("first_derivative forward", "first_derivative
Exact","first_derivative backward", "first_derivative
central"),col = c(5,6,3, 2),text.col = "red", lty = c(4, 1, 3, 2))
dev.off()
```

## 3.3 Simpson's rule

4 points

To increase the accuracy of trapezoid [illegible] n is required. Aside from applying the trapezoidal rule with finer segmentation, another way to obtain a more accurate estimate of an integral is to use higher-order polynomial to connect the points. For example, if there is an extra point mid way between f(a) and f(b), the three points can be connected with a parabola. If there are two points equally spaced between f(a) and f(b), the four points can be connected with a third order polynomial. The formula that result from taking the integrals under these polynomials are called Simpson's rules.

### 3.3.1 Simpson's 1/3 rule

$$I = \int_a^b f(x) \cong \int_a^b f_2(x) \tag{3.3}$$

the subscript 2 is to indicate that the function is second order polynomial.

$$I \cong h/3(f(x_0) + 4 * f(x_1) + f(x_2)) + E \tag{3.4}$$

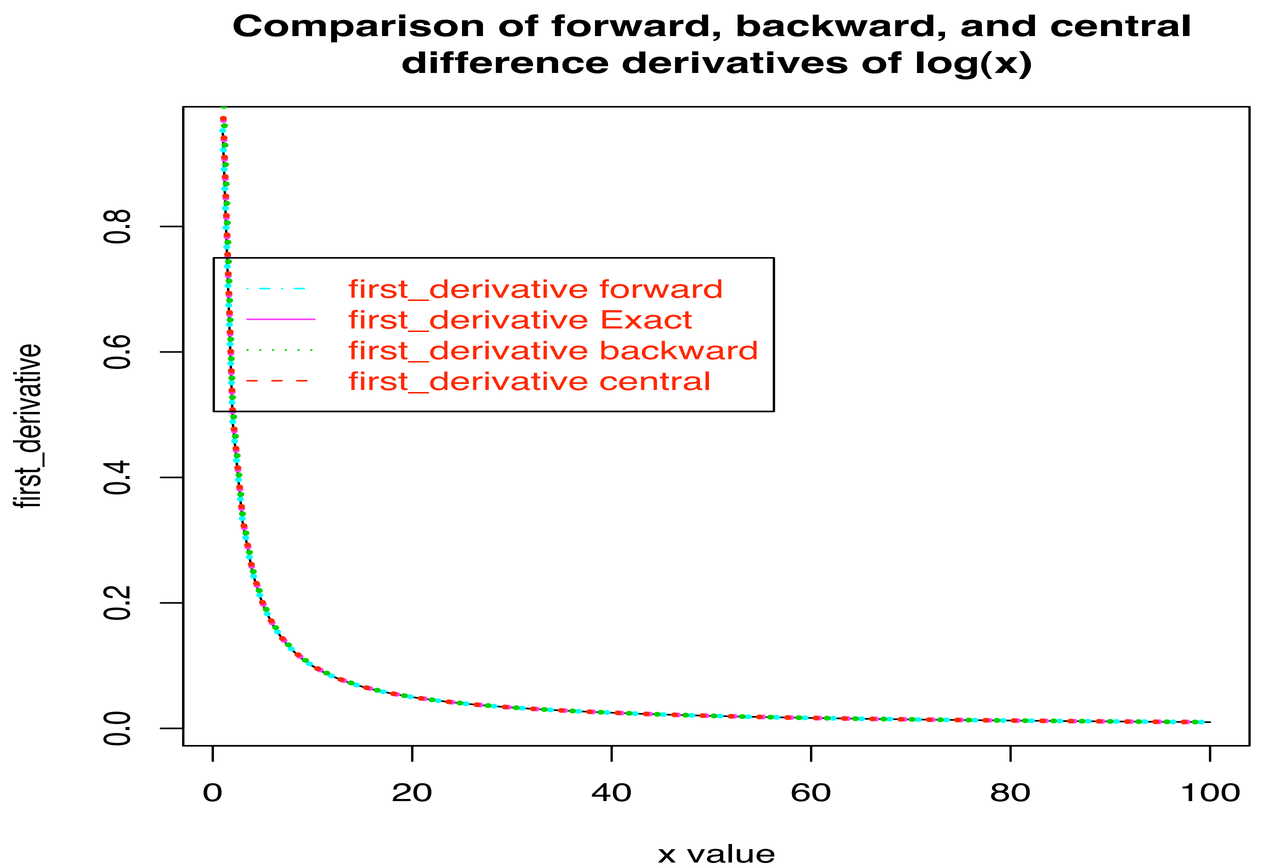


Figure 3.3: Comparison of the first derivative of log(x) using forward, backward, and central differentiation with the exact one

Simpson rule can be improved by dividing the integration interval in to a number of segments of equal width: h=(b-a)/n. The total integration can be represented as:

$$I = \int_a^{x_2} f(x)dx + \int_{x_2}^{x_4} f(x)dx + \ldots + \int_{x_{n-2}}^{b} f(x)dx \tag{3.5}$$

Substituting Simpson's 1/3 rule for the individual integral yields

$$I \cong h/3(f_1 + 4 * f_2 + 2 * f_3 + 4 * f_4 + \ldots + 2 * f_{n-1} + 4 * f_n) + E \tag{3.6}$$

where

$$f_i = f(a) + (i-1) * h$$

$$h = (b-a)/n$$

***Practice 6***

```
# this is a function that integrates the given function by
simpson(1/3) rule # n is the number of data intervals, h the
interval between the two data points Simpson_Three=function(f,h){
n=length(f)-1
# n should be even
I1=f[1]+f[length(f)]
I2=sum(4*f[seq(2,n,2)])
I3=sum(2*f[seq(3,(n-1),2)])
I4=I1+I2+I3
I=(I4/3)*h
list(I=I) }
> source("F:\\functions\\simpson.R")
> x=seq(0,2*pi,pi/8)
> f=x^2+4*x
> n=length(x)-1
> n
[1] 16
> h=pi/8
> Simpson_Three(f,h)
$I [1] 161.6402

> #Here, the analytical result for the integration of x^2+4x is about
> #161.43452 with a difference of 0.205608.  Good approximation.
```

```
#The program works by varying the function and the integration
intervals.
> x=seq(-1,1,.5)
> f=(x+1)^3
> n=length(f)-1
> n
[1] 4
> h=0.5
> Simpson_Three(f,h)
$I [1] 4


> f=1/(x+1)^3
> x=seq(-1,1,.5)
> f=1/(x+1)^3
> n=length(f)-1
> n
[1] 4
> h=0.5
> Simpson_Three(f,h)
$I [1] Inf
```

### 3.3.2 Simpson's 3/8 rule

When integration of data on equi-spaced points is attempted, the extended Simpson 1/3 rule is not applicable if the number of intervals is odd. In this case the 3/8 rule is applied to the first or last three intervals, and then the extended 1/3 rule is applied to the remainder of intervals. Then, the integration is approximated as:

$$I = \int_a^b f(x)dx \cong \int_a^b f_3(x)dx \cong 3/8 * h * [f(x_0) + 3 * f(x_1) + 3 * f(x_2) + f(x_3)] \tag{3.7}$$

***Practice 7***

```
 # this is a function that integrates the given function by
simpson(3/8) rule # n is the number of data intervals, h the
interval between the two data points.
Simpson_Eight=function(f,h){
n=length(f)-1# n should be odd
```

```
I1=f[1]+f[length(f)]
I2=sum(2*f[seq(4,n-2,3)])# n-2 should be greater than or equal to4
I3=sum(3*f[seq(4,n-2,3)])
I4=sum(3*f[seq(2,n,1)])
I5=I4-I3
I6=I1+I2+I5
I7=(3/8)*h
I=I6*I7 list(I=I) }

> source("F:\\functions\\simpson eight.R")
> x=seq(0,1.5,.1)
> f=x^3+2*x
> n=length(x)-1
> n
[1] 15
> h=.1
> Simpson_Eight(f,h)
$I [1] 3.515625
# The result calculated numerically is equal to
the analitical result for
# equations of the third order and less.
But, if the order of the polynomial
# is greater than three, the
derivative that contains the error # does not vanish.
# Example

> source("F:\\functions\\simpson eight.R")
> x=seq(0,1.5,.1)
> f=x^4+4*x
> n=length(x)-1
> n
[1] 15
> h=0.1
> Simpson_Eight(f,h)
$I [1] 6.018795
```

The analitical result obtained is 2.64375. So, it is not good for equations of a polynomial of degree four and above.

## 3.4 Gauss quadrature

Trapezoidal and Simpson's rules are based on evenly spaced function values. Consequently, the location of the base points used in these equations was predetermined or fixed. For example, the trapezoidal rule is based on taking the area under the straight line connecting the function values at the ends of the integration interval.

Because the trapezoidal rule must pass through the end points, there are cases where the formula results in a larger error. Improved estimate of the interval would be achieved by positioning these points wisely. Gauss quadrature is the name for one class of techniques to implement such a strategy.

An improved integral estimate is obtained by taking the area under the straight line passing through two intermediate points.

# Chapter 4

# Initial value problems of ordinary differential equations

## 4.1 Introduction

An equation that involves one or more ordinary derivatives of the unknown function is called an ordinary differential equation.

Problems of solving an ordinary differential equation are classified in to initial-value problems and boundary-value problems, depending on how the conditions at the end points of the domain are specified. These conditions are used to evaluate the constants of integration that result during the solution of the equation. For an

$$n^{th}$$

-order equation, n conditions are required.

If all the conditions are specified at the same value of the independent variable, these class of equations are called initial-value problems. On the other hand, if conditions are spread between both initial and final points,i.e. at extreme points, these equations are called boundary-value problems.

## 4.2 First order ordinary differential equations

The initial-value problems of a first-order ordinary differential equation may be written in the form shown below.

$$y'(t) = f(y, t) \tag{4.1}$$

$$y(0) = y_o \tag{4.2}$$

where

$$f(y,t)$$

a function of y and t, and the second equation is an initial condition without which the solution can not be evaluated. The unknown function y can be computed by numerically integrating f(y,t).

## 4.3 Euler methods

6 points out of 18

These methods are simple methods o

Let's consider the figure below**0**

The first derivative provides a direct estimate of the slope at

$$x_0$$

. Then

no body knews what is y1 and y1+i stands for

$$dy/dx = f(x,y), y \tag{4.3}$$

$$\triangle y/\triangle x = (y_{i+1} - y_i)/x_{i+1} - x_i = f(x_i, y_i) \tag{4.4}$$

Thus,

$$y_{i+1} - y_i = (x_{i+1} - x_i) * f(x_i, y_i) \tag{4.5}$$

$$\Longrightarrow y_{i+1} = y_i + h * f(x_i, y_i) \Longrightarrow y_{i+1} = y_i + h * f(x_i, y_i) \tag{4.6}$$

### 4.3.1 Euler forward method

This method uses values at i and i+1.

$$y' = f(y,t)$$

$$(y_{n+1}, y_n) * h \cong y'_n \tag{4.7}$$

$$\Longrightarrow y_{n+1} = y_n + h * f(y_n, t_n) \tag{4.8}$$

$$y_1 = y_o + h * y'_o = y_o + h * f(y_0, t_0) \tag{4.9}$$

$$y_2 = y_1 + h * y' = y_1 + h * f(y_1, t_1) \tag{4.10}$$

$$y_3 = y_2 + h * f(y_2, t_2) \tag{4.11}$$

...

$$y_n = y_{n-1} + h * f(y_{n-1}, t_{n-1}) \tag{4.12}$$

***Practice 8***

```
# This a program that solves the first order ordinary
differential(Dr.B) #equations R'=f(t), y(0)=0, where f(y,t)
#=40-5*t position_derivative=function(t){ 400-10*t } h=0.01# Here
we decide the interval
 t=seq(0,100,h) # Here we determine the time
interval
R0=0 # Here we specify the initial conditions R=c() #
Here we initialize the variable
R[1]=R0 # Here we set the value we
are looking for at initial points
for (i in 1:(length(t)-1)) {
R[i+1]=R[i]+h*position_derivative(t[i]) }
pdf("ordinaryDE_Euler.pdf") plot(t,R,
type="l",col=3,ylim=c(0,8650), pch=0, lty=2,ylab="Range [m]",
xlab="Time (t)") Rexact=R0+400*t-5*t^2 lines(t,Rexact,pch=0,
col=1, lty=1,lwd=3)

h=1 # Here we decide the interval t=seq(0,100,h) # Here we
determine the time interval R0=0 # Here we specify the initial
conditions R=c() # Here we initialize the variable R[1]=R0 # Here
we set the value we are looking for at initial points for (i in
1:(length(t)-1)) { R[i+1]=R[i]+h*position_derivative(t[i]) }
lines(t,R,pch=0, col=2, lty=3,lwd=3)

h=3 # Here we decide the interval t=seq(0,100,h) # Here we
determine the time interval R0=0 # Here we specify the initial
conditions R=c() # Here we initialize the variable R[1]=R0 # Here
we set the value we are looking for at initial points for (i in
1:(length(t)-1)) { R[i+1]=R[i]+h*position_derivative(t[i]) }
lines(t,R,pch=0, col=5, lty=3,lwd=3)


h=5 # Here we decide the interval t=seq(0,100,h) # Here we
```

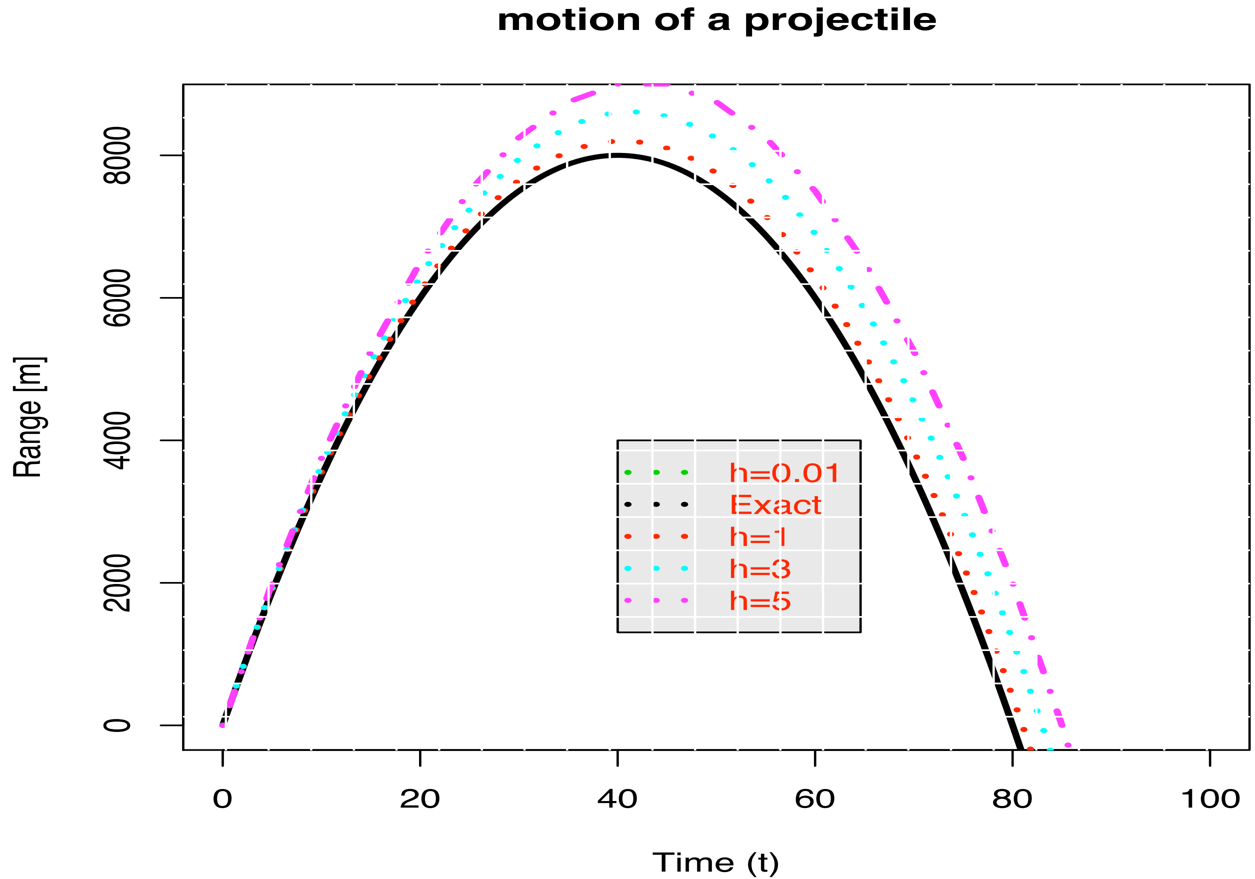


Figure 4.1: Motion of a projectile analyzed at different step size using Euler method

```
determine the time interval R0=0 # Here we specify the initial
conditions R=c() # Here we initialize the variable R[1]=R0 # Here
we set the value we are looking for at initial points for (i in
1:(length(t)-1)) { R[i+1]=R[i]+h*position_derivative(t[i]) }
lines(t,R,pch=0, col=6, lty=4,lwd=3) legend(40, 4000, c("h=0.01",
"Exact","h=1", "h=3","h=5"),col = c(3,1,2,5, 6),
       text.col = "red", lty = "dotted",lwd=3,
       merge = TRUE, bg = 'gray90')
     grid(nx = 25, ny = 20, col = "white", lty = 1,
     lwd =1, equilogs = TRUE)
     title("motion of a projectile")
      dev.off()
```

***Practice 9*** *Example 1* Carbon-11 is a radioisotope that disintegrates at the rate of 3.46 percent, which is equivalent to a half life of 20 minutes. The rate of decay is

expressed by

$$\lambda = 0.0346/minor\lambda = 1.37 * 10^(-11)/s$$

called decay constant. The initial atomic number density at t=0 is given by N0 atoms/cc. Derive a differential equation for the atomic number density and find the solution using Euler's forward method.

$$dN/dt = -\lambda * N(t)$$

;

$$N(t + \Delta t) = N(t) - \lambda * N(t) * \Delta t \tag{4.13}$$

```
# This is a program that calculates the number density of a
radioisotope disintegrates in time using
 Euler's forward method.
 Number_density=function(t,N){
 -l*N } # l represents the decay constant=lambda
l=1.37*10^(-11)
h=1
t=seq(0,100,h)
N0=2.66*10^(21)
N=c()
 N[1]=N0
for(i in
1:(length(t)-1)){
N[i+1]=N[i]+ h*Number_density(t[i],N[i])
 }
pdf("Number_disintegrated.pdf")
Nexact=N0*exp(-l*t)
plot(t,Nexact,ylab="Number density", xlab="Time (t)",type="l")
lines(t,Nexact,pch=0, col=4, lty=1,lwd=2) lines(t,N,
pch=0,col=6,lty=2,lwd=2) legend(0.1,0.75,c("Exact
solution","Forward Euler's method"),
lwd=2,col=c(4,6),text.col="green",pch=c(2,NA),lty=c(3,4)) #grid(nx
= 24, ny = 24, col = "blue", lty =6,lwd =3, equilogs = TRUE)
title("Atomic number density of a radioisotope that integrates in
time") dev.off()
```

***Practice 10***

```
# This is as program that calculates the velocity of a falling
#object under the action of air resistance
```

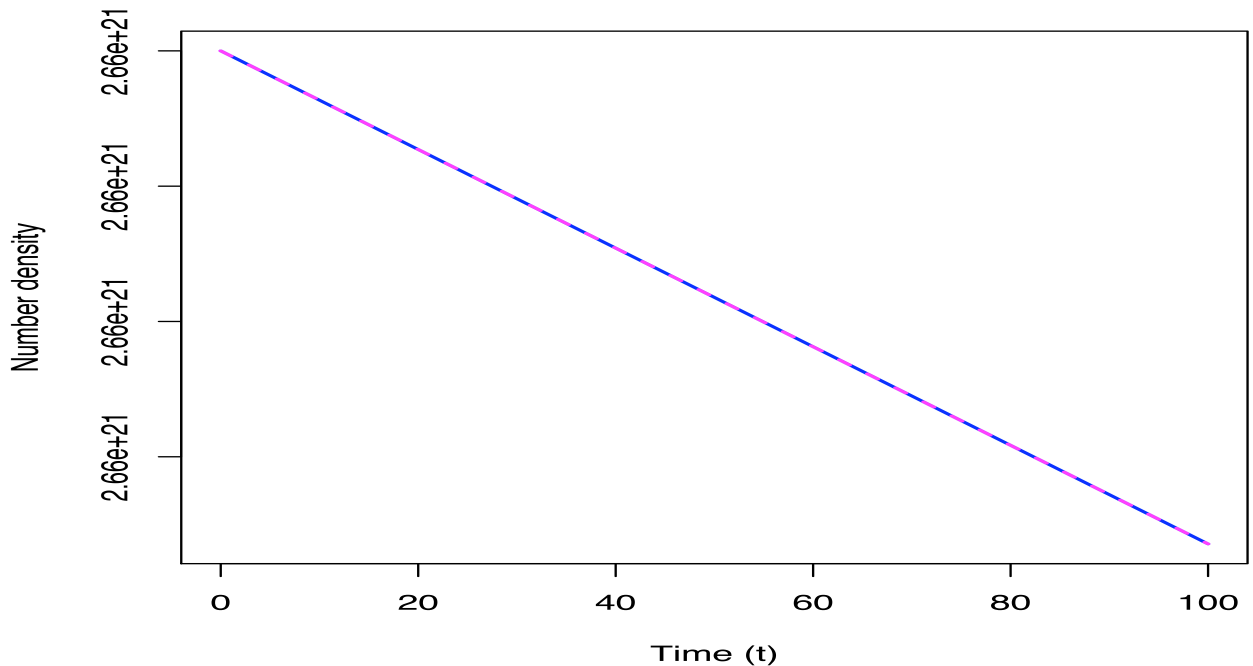


Figure 4.2: Comparison of anaclitic and numeric solution of a radioisotope disintegration

```
terminal_velocity=function(t,v){
g-c/m*v }
h=2
g=9.8
m=68.1
 c=12.5
t=seq(0,100,2)
v0=0
v=c()
 v[1]=v0
 for(i in 1:(length(t)-1)){
v[i+1]=v[i]+(terminal_velocity(t,v))*h}
vexact=g/c*m*(1-exp(-c/m*t))
pdf("terminal_velocity.pdf")
plot(t,vexact)
lines(t,vexact,type="l",col=3,pch=0,lty=2,ylab="velocity",xlab="time")
title("Numerical and analytical solutions for the velocity of a
falling object against air resistance")
lines(t,v,type="l",col=6,pch=0,lty=1) legend(1,5,c("Analytical
solution","Numerical solution"),col=c(3,6),
```

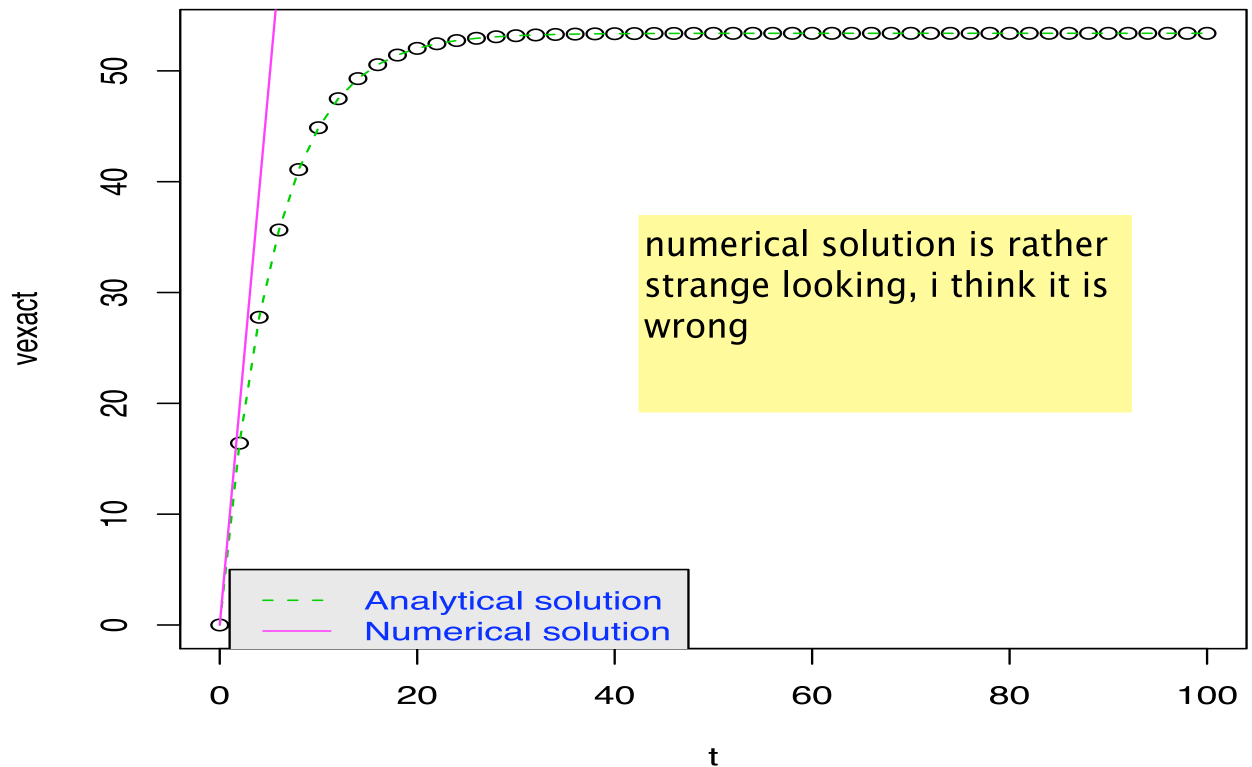


Figure 4.3: Comparison of the velocity of a falling object under the action of gravity by using Euler's method

```
text.col="blue",lty=c(2,1),bg="gray90") dev.off()
```

***Practice 11*** The amount of a uniformly distributed radioactive contaminant in a closed reactor is measured by its concentration c. The contaminant decreases at a decay rate proportional to its concentration; that is, decay rate =-k*c where k=a constant with units of decay$^{-1}$. Therefore, a mass balance for the reactor can be written as dc/dt=-k*c. The right hand side represents change in mass whereas the left hand side represents decrease in decay.

```
# A program that calculates radioactive contaminant in a reactor
radioactive=function(t,c){
         -k*c}
h=0.1 #step size
t=seq(0,1,0.1) # time interval in days
k=0.1 # a
constat with units of day
```

```
c0=10
A=10
c=c()
c[1]=c0
for(i in
1:(length(t)-1)){
c[i+1]=c[i]+h*radioactive(t,c)}
cexact=c0*exp(-k*t)
pdf("radioactivity.pdf")
plot(t,c,type="l")
lines(t,c,ylab="concentration(liter)",xlab="time(s)",
type="l",col=6,pch=0,lty=2)
lines(t,cexact,pch=0,col=4,lty=3,lwd=3,type="l")
legend(2,3,c("Numerical solution","Exact solution"),col=c(2,3),
text.col="red",lty=c(2,1),bg="gray90") title("Radioactive decay")
dev.off()
```

***Practice 12***

Consider the electrical circuit shown below where the swith is closed at t=0.It satisfies

$$I(t) = L * dI(t)/dt + r * I(t) = E$$

```
# This is a program that calculates the current i at time t #
using modified Euler method. current_instant=function(t,I){
-R/L*I+E/L
                     }
t=seq(0,0.02,0.001)#time interval
R=20#resistance in Ohms
L=50*10^(-3)# inductance in Hertz
E=10#voltage in volts
h=0.001#step size
t0=0#initial time
I0=0#initial condition of the
current
I=c()#this is to initialize the variable I
I[1]=0#the
```

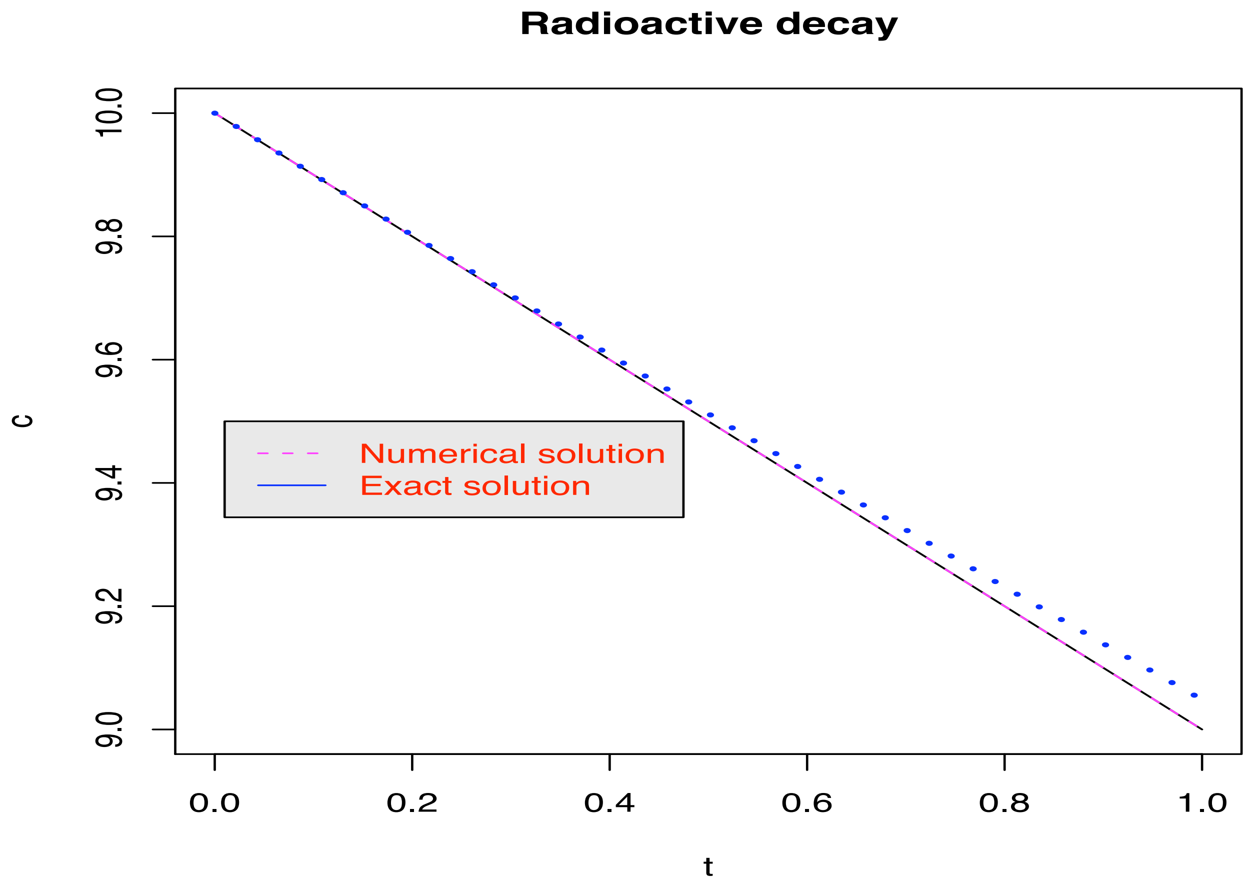


Figure 4.4: Radioactive contaminant in a closed rector

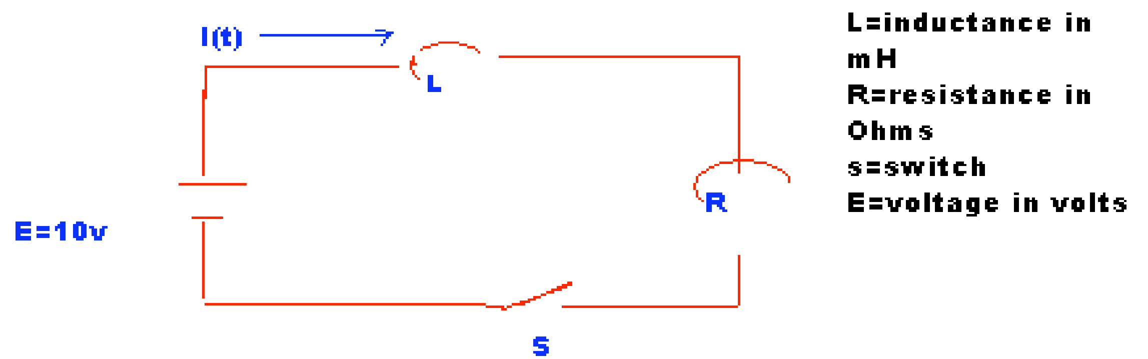

Fig 3. Electric current

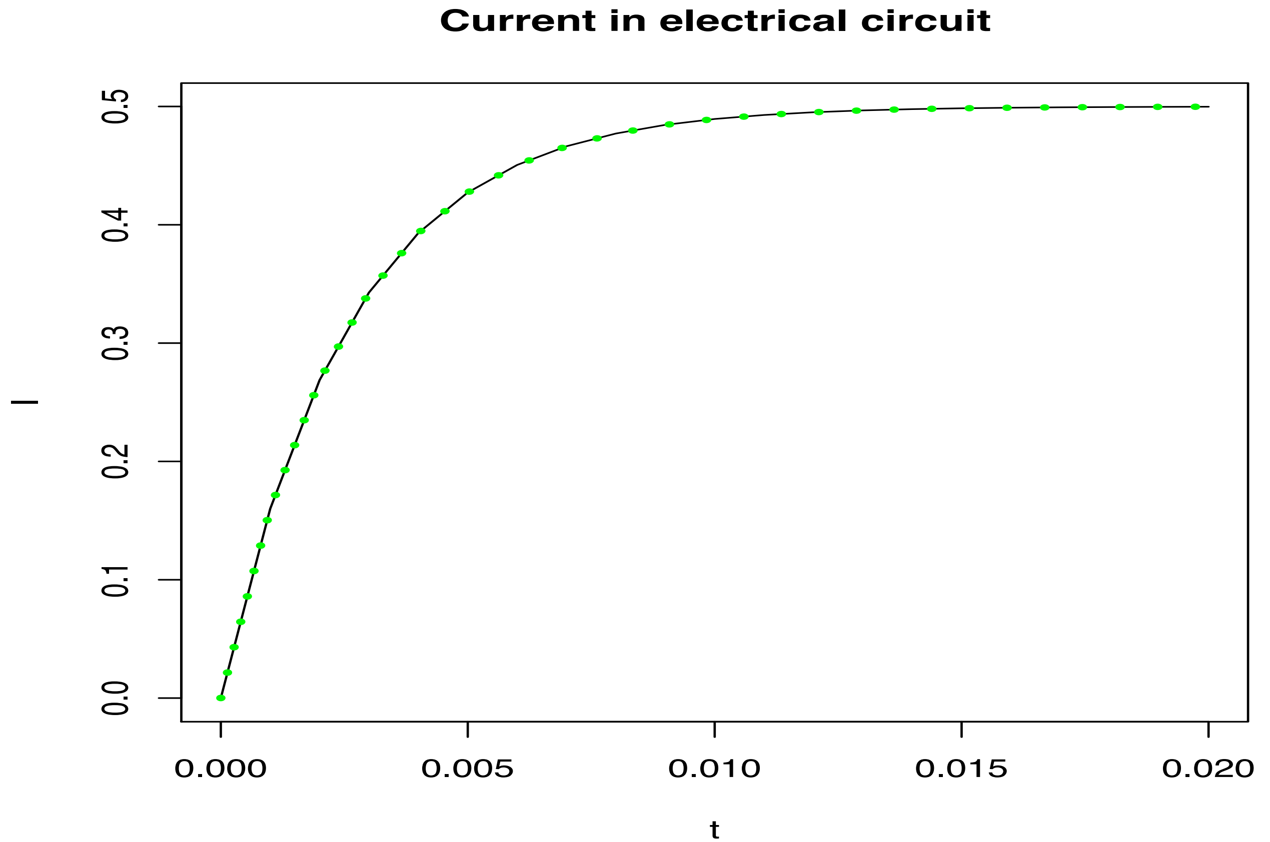


Figure 4.5: Time evolution of current in electrical circuit

```
value of I in the first count
for(i in 1:(length(t)-1)){
k1=h*(-R/L*I[i]+E/L)
k2=h*(-R/L*(I[i]+k1)+E/L)
I[i+1]=I[i]+1/2*(k1+k2)}
pdf("current_instant.pdf")
plot(t,I,type="l")
lines(t,I,xlab="time(s)",ylab="current(A)",col="green",lwd=4,lty=3)
title("Current in electrical circuit") dev.off()
```

### 4.3.2 Euler's backward method

$$y(t + \Delta t) = y(t) + \Delta t * f(y(t + \Delta t), t + \Delta t)$$

### 4.3.3 Modified Euler's method

$$y(t + \Delta t) = y(t) + (\Delta t)/2 * [f(y(t), t) + f(y(t + \Delta t), t + \Delta t)]$$

14 points out of 18

## 4.4 Runge-Kutta methods

In Runge-Kutta methods, the order of accuracy is increased using a higher order numerical integration method. Consider an ordinary differential equation given in 4.1. In order to calculate

$$y_{n+1}$$

with a known value of

$$y_n$$

, we integrate equation4.1 in the interval

$$t_n >= t_{n+1} toyield$$

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(y, t)dt \tag{4.14}$$

Then, Runge-Kutta methods(second, third,and fourth-order) are derived by applying a numerical integration method to the right side of the foregoing equation.

### 4.4.1 second-order Runge-Kutta methods

It applies trapezoidal rule to the right side of the equation

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(y, t)dt$$

$$y_{n+1} - y_n = \int_{t_n}^{t_{n+1}} f(y, t)dt \cong h/2 * [f(y_n, t_n) + f(y_{n+1}, t_{n+1})] \tag{4.15}$$

In the foregoing equation

$$y_{n+1}$$

is not known, so the second term is approximated by

$$f(y_{n+1}^{-}, t_{n+1})$$

where

$$y_{n+1}^{-}$$

is an estimate calculated by the forward Euler method.

$$y^{-}_{n+1} = y_n + h * f(y_n, t_n)$$

$$y_{n+1} = y_n + h/2 * [f(_n, t_n) + f(y^{-}_{n+1}, t_{n+1})]$$

$$k_1 = h * f(y_n, t_n)$$

$$k_2 = h * f(y_{n+k_1}, t_{n+1})$$

$$\Longrightarrow y_{n+1} = y_n + 1/2 * (k_1 + k_2)$$

This is equivalent to the modified Runge-Kutta method.

### 4.4.2 Third-order Runge-Kutta method

This method is derived using Simpson's 1/3 rule.

$$y_{n+1} = y_n + h/6 * [f(y_n, t_n) + 4 * f(y^{-}_{n+1/2}, t_{n+1/2}) + f(y^{-}_{n+1}, t_{n+1})] \tag{4.16}$$

where

and

are estimates because

and

are not known. The estimate

may be obtained by:

This is rather disorganized. I expect to get the meaning of each terms but in fact you left it untouched. Please pay attention to this kind of things

$$y^{-}_{n+1} = y_n + h * f(y_n, t_n); y^{-}_{n+1} = y_n + h * f(y^{-}_{n+1/2}, t_{n+1/2}) \tag{4.17}$$

Here,

$$k_1 = h * f(y_n, t_n)$$

$$k_2 = h * f(y_n + 1/2 * k_1, t_{n+1/2})$$

$$k_3 = h * f(y_n - k_1 + 2 * k_2, t_{n+1})$$

$$y_{n+1} = y_n + 1/6 * (k_1 + 4 * k_2 + k_3)$$

***Practice 13***

```
#Below is a program that calculates a given function numerically
by using third-order Runge-Kutta  method rk_fourth=function(x,y){
     x/y
     }
    h=0.02 # the step size
    A=1# value of y at x=0
    y=c()# variable to collect the values
    y[1]=1
    x=seq(0,0.1,h)# x coordinate in which we are
    # interested to calculate value of y
    for(i in 1:(length(x)-1)){
     k1=h*rk_fourth(x[i],y[i])
     k2=h*rk_fourth(x[i+h/2],y[i+k1/2])
     k3=h*rk_fourth(x[i+h],y[i-k1+2*k2])
     k=1/6*(k1+4*k2+k3)
    y[i+1]=y[i]+k
      }
pdf("Runge_Kutta3.pdf") y_exact=(x^2+A)^(1/2)# A is a constant
determined by initial condition. plot(x,y_exact,type="l")
lines(x,y_exact,col=2,pch=0,lty=1,lwd=3,ylab="Values of y",
xlab="Values of x",) lines(x,y,type="l", col="blue",
pch=0,lty=2,lwd=2) title("Third-order Runge_kutta method")
legend(.5,.75,c("Analytical solution","Numerical
solution"),col=c(5,6), text.col="blue",lty=c(2,1),bg="gray90")
dev.off()
```

### 4.4.3 Fourth-order Runge-Kutta method

This is similar to the third order except one intermidate step of evaluating the derivative is used. There are two versions.The first is:

$$k_1 = h * f(y_n, t_n)$$

$$k_2 = h * f(y_n + k_1/2, t_{n+1/2})$$

$$k_3 = h * f(y_n + k_2/2, t_{n+1/2}$$

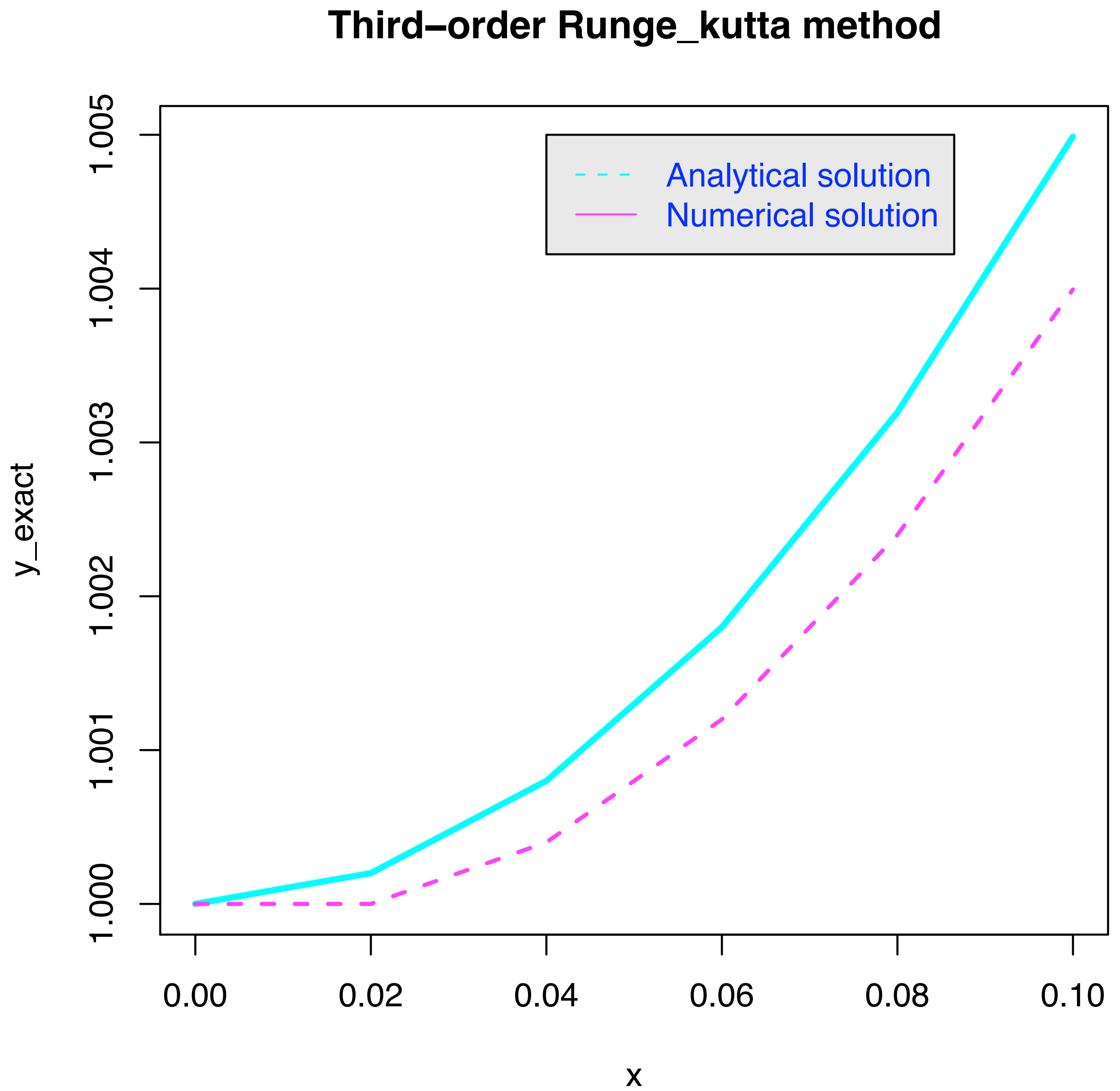


Figure 4.6: Solutions of ordinary differential equations using third-order Runge-Kutta method

$$k_4 = h * f(y_n + k_3, t_n + 1$$

$$y_{n+1} = y_n + 1/6 * (k_1 + 2 * k_2 + k_4) \tag{4.18}$$

The second is:

$$k_1 = h * f(y_n, t_n)$$

$$k_2 = h * f(y_n + k_1/3, t_n + 1/3)$$

$$k_3 = h * f(y_n + k_1/3 + k_2/3, t_n + 2/3)$$

$$k_4 = h * f(y_n + k_1 - k_2 + k_3, t_{n+1})$$

$$y_{n+1} = y_n + 1/8 * (k_1 + 3 * k_2 + 3 * k_3 + k_4) \tag{4.19}$$

***Practice 14***

```
# This is a program that calculates the given ordinary
differential equation using forth-order Runge_Kutta
method\\
rk_fourth=function(x,y){
     x/y
     }
    h=0.02 # the step size
    A=1# value of y at x=0
    y=c()# variable to collect the values
    y[1]=1
    x=seq(0,0.1,h)# x coordinate in which we are
    # interested to calculate value of y
    for(i in 1:(length(x)-1)){
     k1=h*rk_fourth(x[i],y[i])
     k2=h*rk_fourth(x[i+h/2],y[i+k1/2])
     k3=h*rk_fourth(x[i+h/2],y[i+k2/2])
    k4=h*rk_fourth(x[i+h],y[i+k3])
     k=1/6*(k1+2*k2+2*k3+k4)
    y[i+1]=y[i]+k
      }
pdf("Runge_Kutta.pdf") y_exact=(x^2+A)^(1/2)# A is a constant
determined by initial condition. plot(x,y_exact,pch=2,
col=4,lwd=5, lty=1,ylab="Values of y", xlab="Values of
x",type="l") lines(x,y, col=6, pch=0, lty=2,lwd=3,type="l")
title("Forth-order Runge_kutta method") legend(0.1,0.75, c("Exact
```

```
solution", "Numerical solution"),col = c(4,6),text.col = "red",
lty = c(4, 2)) #legend(0.1, 0.75, c( "Exact solution","Numerical
solution"),col = c(4,6), text.col = "red", pch=c(2,2 #),lty = c(1,
2),bg = "grey90", lwd=2) dev.off()
```

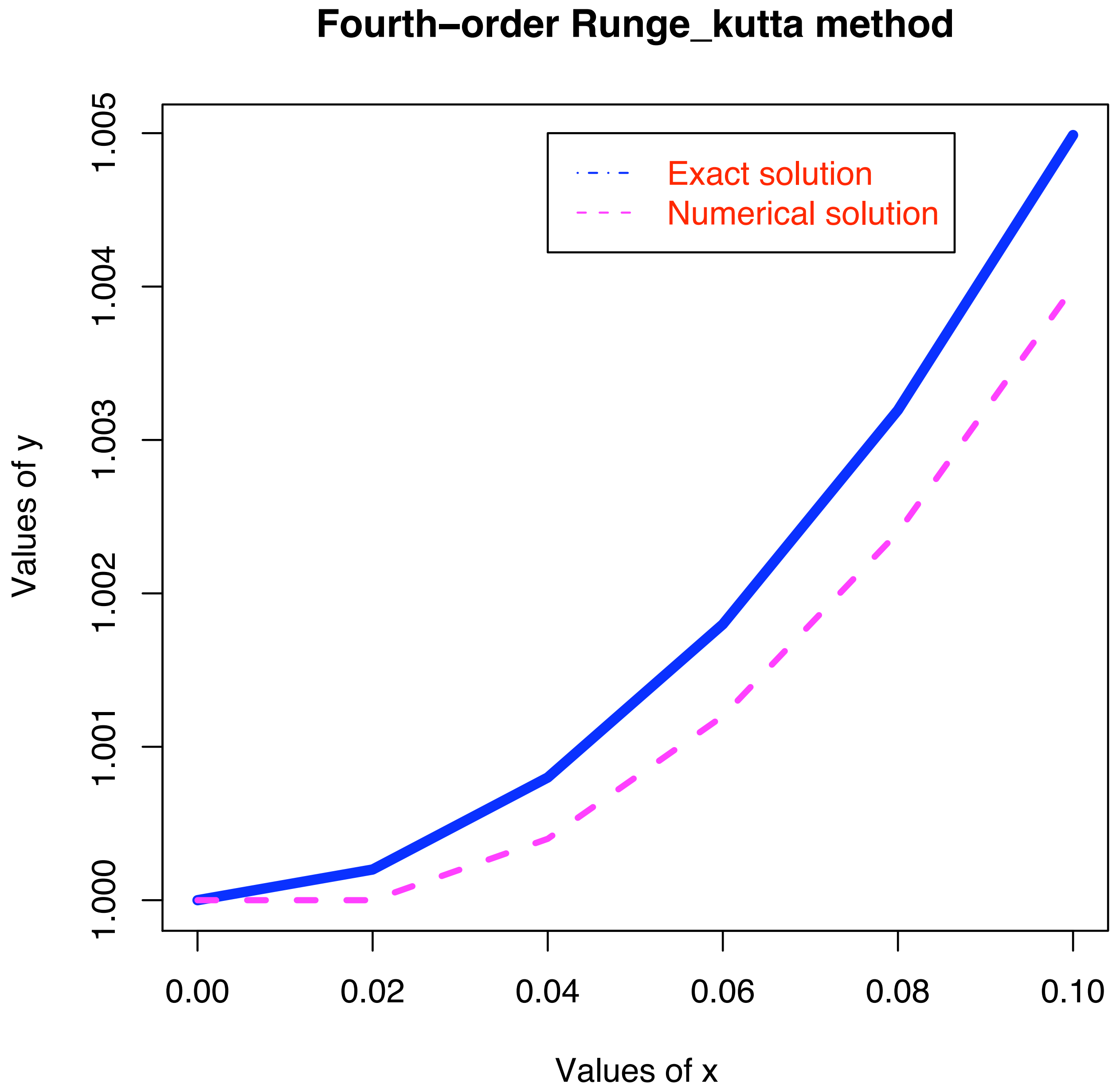


Figure 4.7: Comparison of exact and numerical solution of a function using forth-order Runge-Kutta method

# Bibliography

[1] Nakamura,S., Numerical analysis and graphic visualization with matlab, *Prentice Hall*. New Jersey, 1996.

[2] Chapra,S.C.,Numerical methods for engineers, *Mc Graw-Hill*. New York, 1998.