Numerical Analysis Lecture Notes

Juan C. Meza

May 13, 2024

Table of contents

Preface **3**

I Introduction **5**

1 Foundations **7**
1.1 Frequently Used Theorems from Calculus 7
1.2 Computer Programming 8
1.3 Other Useful References 8

2 Taylor's Theorem **10**
2.1 Taylor Polynomials 10
2.2 Taylor's Theorem 12
2.3 Examples/Exercises 13
2.4 Other forms for the Remainder Term 15
2.5 Choosing $P_n(x)$ to achieve a desired accuracy 15
2.6 Summary 17
2.7 References 17

3 Errors **18**
3.1 Sources of Errors 18
3.1.1 Precision and Accuracy 19
3.2 Sources of Errors 19
3.3 Summary 22

4 Floating Point Systems **23**
4.1 Floating Point Systems 23
4.1.1 Types of Rounding 27
4.2 Summary 28
4.3 Advanced 28
4.4 References 28

5 IEEE Floating Point Standard **29**
5.1 IEEE Floating Point Standard 29
5.2 Important Parameters for IEEE Floating Point System 31
5.2.1 Unit roundoff and machine epsilon 32

5.2.2 Range of Floating Point Numbers . . . 32
5.2.3 Relative error of floating point representation . . . 33
5.2.4 Floating Point Arithmetic Operations . . . 34
5.3 Summary . . . 34
5.4 Codes . . . 35
5.5 References . . . 35

6 Stability and Conditioning **37**
6.1 Algorithms . . . 37
6.2 Desired Properties of Algorithms . . . 37
6.3 Stability and Conditioning . . . 39
6.4 Tips on designing stable algorithms . . . 42
6.5 Summary . . . 43
6.6 Advanced . . . 43
6.7 References . . . 43

II Nonlinear Equations **44**
Nonlinear equations . . . 45

7 Bisection Method **47**
7.1 Bisection . . . 47
7.2 Stopping (Convergence) Criteria . . . 50
7.3 Convergence of Bisection method . . . 52
7.4 Summary . . . 53

8 Newton's Method **54**
8.1 Newton's Method . . . 54
8.2 Error Analysis for Newton's Method . . . 57
8.2.1 Summary for Newton's Method . . . 58

9 Secant Method **59**
9.1 Secant Method . . . 59
9.1.1 Summary for Secant Method . . . 60
9.2 Advanced: Root Finding in Higher Dimensions . . . 61

10 Fixed Point Method **63**
10.1 Fixed Point . . . 63

11 Fixed Point Existence and Uniqueness **66**
11.1 Existence and Uniqueness of Fixed Points . . . 66
11.2 More on convergence of sequences . . . 67
11.2.1 Examples. . . . 68
11.2.2 Solutions: . . . 68

12 Fixed Point Iteration Convergence (Part 1) **69**
12.1 Fixed Point Iteration Convergence (Part 1) . . . 69
12.1.1 Example: . . . 70
12.1.2 Solution: . . . 71

13 Fixed Point Iteration Convergence (Part 2) **73**
13.1 Fixed Point Iteration Convergence (Part 2) . . . 73
13.2 Final note on convergence . . . 75
13.3 Comparison of nonlinear equation methods . . . 76

III Interpolation and Approximation **78**
Introduction . . . 79
Concepts Covered . . . 79

14 Polynomial Interpolation **80**
14.1 Interpolation and Extrapolation . . . 80
14.1.1 Motivation . . . 80
14.1.2 Data fitting . . . 80
14.2 General Representation . . . 81
14.3 Polynomial (Monomial) Interpolation . . . 82
14.3.1 Vandermonde Matrices . . . 84
14.3.2 Summary . . . 85
14.4 Supplemental Materials . . . 85
14.4.1 Weierstrass Approximation . . . 85
14.4.2 Alternate derivation of linear interpolation using monomial basis functions 86
14.4.3 Computational Tip . . . 86
14.5 References . . . 87

15 Lagrange Polynomial Interpolation **88**
15.1 Lagrange Polynomials . . . 88
15.2 General Form of Lagrange Polynomials . . . 92
15.2.1 Summary . . . 95
15.3 References . . . 95

16 Divided Differences and Newton Interpolating Polynomials **96**
16.1 Divided Differences . . . 96
16.2 Newton Interpolating Polynomials . . . 99
16.3 Summary . . . 100

17 Interpolation Error **101**
17.1 Error in Polynomial Interpolation . . . 101
17.2 Practical Tips . . . 103
17.3 Chebyshev Points . . . 104

18 Piecewise Polynomials and Cubic Splines **106**
18.1 Piecewise Polynomials . . . 106
18.1.1 Motivation . . . 106
18.2 Piecewise Linears . . . 107
18.3 Error Bounds . . . 108
18.4 Piecewise Constants . . . 109
18.5 Cubic Splines . . . 109
18.6 Summary . . . 112

IV Numerical Differentiation **113**

19 Finite Differences **115**
19.1 Motivation . . . 115
19.2 Taylor Series Approach . . . 115
19.3 Central Differences . . . 118
19.4 Second Derivative Formulas . . . 119

20 Finite Differences via Lagrange Polynomials (Optional) **122**
20.1 ($n+1$)-point Formulas . . . 122
20.2 Three-Point Formulas . . . 124

21 Richardson Extrapolation and Finite Difference Stability **127**
21.1 Richardson Extrapolation . . . 127
21.2 Stability . . . 129
21.3 Summary . . . 131

V Numerical Integration (Quadrature) **132**
Introduction . . . 133
Practical Applications . . . 134

22 Numerical Integration Basics **135**
22.1 Introduction . . . 135
22.2 Interpolating Polynomials . . . 136
22.3 Simpson's Rule . . . 140
22.3.1 Setting up the integral of the interpolating polynomial. . . . 141
22.3.2 Evaluating the integral of the interpolating polynomial. . . . 142

23 Newton-Cotes Methods **145**
23.1 Newton-Cotes . . . 145
23.1.1 Closed Newton-Cotes formulas: . . . 146
23.1.2 Open Newton-Cotes formulas: . . . 147
23.2 Error Estimates . . . 147

23.3 Summary . . . 150

24 Composite Quadrature Rules **151**
24.1 Composite Integration . . . 151
24.2 Error Analysis and Stability . . . 155
24.3 Summary . . . 158

25 Adaptive Quadrature **159**
25.1 Adaptive Quadrature . . . 159
25.2 Summary . . . 162

VI Initial-Value Problems for ODEs **164**

VII Initial-Value Problems for ODEs **165**

26 Euler's Method **167**
26.1 Introduction . . . 167
26.2 Euler's Method . . . 168
26.3 Backward Euler . . . 174
26.4 Key Points . . . 175

27 Error Analysis for Euler's Method **176**
27.1 Error Analysis . . . 176
27.2 Convergence and Global Error Estimates . . . 177
27.3 **Roundoff Error Analysis** . . . 178
27.4 Proof of Convergence for Euler's Method . . . 179

28 Higher-order Methods **182**
28.1 Higher-order Taylor Methods . . . 182
28.2 Runge-Kutta Order 2 Methods . . . 183
28.3 Runge Kutta Order 4 . . . 185
28.4 Summary . . . 186
28.5 General form for Runge Kutta methods . . . 186

29 Multi-Step Methods for IVP **189**
29.1 Motivation . . . 189
29.2 Demo: Basic SIR Model . . . 191
29.3 Summary of Methods Studied . . . 193
29.4 Derivation of multi-step methods . . . 194

30 Summary **197**

References **198**

Appendices **199**

A Big O Notation **199**
A.1 Introduction 199
A.2 Big O for approximations 199
A.3 Big O for computational workload 201
A.4 Summary: Why is this important? 204
A.5 References 204

B Existence and Uniqueness of IVP **205**
B.1 The Initial Value Problem (IVP) 205
B.2 Lipschitz Condition and Convex Sets 206
B.3 Fundamental Existence and Uniqueness of IVP 208
B.4 Well-Posed Problems 210

C DataSets **212**

Preface

The first time I heard about numerical analysis was a course at Rice University taught by Prof. Richard Tapia.

After I graduated and as a freshly minted PhD, I had the opportunity to teach the same course as a postdoc. It was both a pleasure to teach the same material and utterly terrifying as it challenged me to go over every single concept I thought I knew so well and realize that in fact there were subtleties that I had never known existed.

Over the years I've practiced computational mathematics at various places including Exxon Production Research, Sandia National Laboratories, and Lawrence Berkeley Laboratory. The one common thread in all of these places was the sheer breadth of the applications that one can apply the fundamental concepts of numerical analysis to real-world problems. However, it also became clear that while the concepts were widely applicable, many of the techniques didn't work, either because of the scale of the problem or because of the underlying assumptions. As we used to joke, "In theory, there is no difference between theory and practice, but in practice there is.

Ever since I have always wanted to document the types of problems that a practicing numerical analyst might encounter in everyday situations. Without taking anything away from standard text books, what I wanted to do was to provide the underlying concepts, but also point out where the standard assumptions might fail and what one can do to overcome those barriers.

This book then is intended as an undergraduate level numerical analysis course, with all of the Important concepts described. In addition, however, the reader will frequently be pointed to a real-world problem for which those same standard techniques will not work - at least not out of the box. Instead, I'll try to point to the fundamental concepts as a starting point and how one can use those techniques to adapt, modify, and extend them to work on real-world problems.

This is a compilation of various numerical analysis notes from over the years. The material is at the level of a junior or senior level applied mathematics student. The main reason for this book is to include code snippets that demonstrate some of the fundamental concepts in numerical analysis.

Here, our goal is to provide both an introduction to some of the major numerical methods techniques as well as some practical advice as to when one would use one method versus another.

Another goal is to give the reader some of the elementary theoretical foundations needed to be able to analyze the methods and their performance. One consequence of this is that when one encounters real-world problems, the assumptions that mathematicians make when analyzing the methods will generally not hold. Nonetheless it is important to be able to recognize the assumptions being made so that one can recognize when a method isn't working because the assumptions have been violated rather than because of some other issue.

Finally, my own experience has taught me that the history behind many of the methods can provide a rich context as to why a numerical method was initially proposed and how others extended the methods to other problems. This continual evolution of numerical analysis can sometimes point the student towards how they could in turn extend the methods to their own problems.

Acknowledgments

I am grateful for all the insightful comments offered by the many people who have looked over the draft versions of these notes. The generosity and expertise of one and all have improved these notes in innumerable ways and saved me from many errors; those that inevitably remain are entirely my own responsibility.

This is a Quarto book. To learn more about Quarto books visit https://quarto.org/docs/books.

Part I

Introduction

Numerical analysis has a long history that goes back beyond the modern electronic computing era. With the advent of computers the field has grown even more rapidly and is now used in virtually all scientific areas. The history is one of computational advances going hand in hand with advances in numerical methods.

But what is numerical analysis? One popular definition of numerical analysis "is the study of algorithms for the problems of continuous mathematics" (Trefethen 2022). This is a great definition and captures the essence of numerical analysis succinctly. I also adhere to this definition, although I would expand or change the last part to include some areas of discrete mathematics as well.

In this book, I would also like to advance another possible way of viewing numerical analysis. In practice, numerical analysis is a balancing act of several goals when solving mathematical problems on a computer. From this viewpoint, numerical analysis is the fine art of balancing accuracy, efficiency, and robustness of numerical algorithms for mathematical problems.

This book is intended as an introduction to the fundamental concepts and methods used in numerical analysis. As such, we need to assume that the reader has a foundation in some of the basic concepts of calculus. In particular, we will have need to use ideas such as the Mean Value Theorem, the Intermediate Value Theorem, the Weighted Mean Value Theorem, and Taylor's Theorem. You will find a brief review of the first 3 of these in the Appendix. Here we will provide a quick refresher on Taylor's Theorem - if you are familiar with this idea and how to use it, you may want to skip to the next section.

Any numerical method today will also entail some form of implementation on a computer. As such, we also review how arithmetic is done on a computer and the consequences of only being able to do computations with finite precision. If you're familiar with these concepts, you may also want to skip those sections.

Whichever view you decide to take, I hope you enjoy the concepts and the immense practical value and impact that numerical analysis has had on solving some of today's most pressing scientific and societal problems.

1 Foundations

> *If people do not believe that mathematics is simple, it is only because they do not realize how complicated life is.*
>
> *John von Neumann*

The material we will be studying in numerical analysis assumes a good foundation in calculus. In particular, we will have a need to use basic facts of continuity, differentiability, integration, and power series. In addition, we will present many of the ideas in the form of numerical algorithms, so a good working knowledge of some programming language will be needed. Our suggestions include Matlab, python, or R.

1.1 Frequently Used Theorems from Calculus

Numerical analysis relies on several fundamental theorems or analysis. We will refer to several of these repeatedly and have use of the following 4 in particular.

Theorem 1.1 (Mean Value Theorem)**.** *Suppose that* (1) f *is continuous on the closed finite interval* $[a, b]$ *and* (2) $f'(x)$ *exists for every* x *in the open interval* (a, b)*. Then there exists a point* c *such that*

$$a < c < b$$

and

$$f'(c) = \frac{f(b) - f(a)}{b - a}$$

We will be looking at numerous cases of continuous functions over closed and bounded intervals. The following theorem will prove useful in our analyses.

Theorem 1.2 (Intermediate Value Theorem)**.** *Suppose that* (1) f *is continuous on the closed finite interval* $[a, b]$ *and* (2) $f(a) < c < f(b)$*. Then there exists some point* $x \in [a, b]$ *such that* $f(x) = c$*.*

Remark: One way to interpret the IVT says is that if a continuous function on an interval takes on any 2 values, it takes on every value in between. This will be particularly useful in our analysis of root finding methods.

Similar to the MVT above, there is a variation that applies to integrals. It is well worth noting that in this case, there is an important assumption without which the theorem does not apply, so care must be taken when applying it to certain problems.

Theorem 1.3 (Weighted Mean Value Theorem for Integrals)**.** *Suppose that $f \in C[a,b]$, the Riemann integral of g exists on $[a,b]$, and $g(x)$ does not change sign of $[a,b]$. Then there exists a number $c \in (a,b)$ such that*

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

1.2 Computer Programming

In terms of programming, we suggest one of 3 possible languages: Matlab, python, or R. Python and R have the advantage of being open-source and most of our examples will be in R and sometimes in python. In addition, both python and R can be easily installed on most computer platforma and both have powerful programming environments similar to Matlab. For python, one can use JupyterLab (jupyter); for R, one can use Rstudio/Posit (Posit).

We will also note that in real-world applications, most scientific codes will use other languages such as Fortran or C++. For the purposes of this introductory course, any high-level language will suffice.

1.3 Other Useful References

You should be able to find references to all of the material here in standard introductory courses on calculus. A good online reference for some of the material above can be found at openstax.org.

- Mean Value Theorem
- Intermediate Value Theorem
- Mean Value Theorem for Integrals
- Taylor polynomials and Taylor's Theorem

Another good set of resources are the one-pagers provided by the UCM Math Center. You can check them all out at: UCM The Math Center and in particular the Math 23 refresher one-pager might prove useful: Math 23 refresher.

2 Taylor's Theorem

All models are wrong but some models are useful

George E.P. Box

Topics Covered:

- How can we approximate mathematical functions that cannot be evaluated exactly?
- How does the approximation behave near the approximation point?
- Can we quantify the error in the approximation?

2.1 Taylor Polynomials

A fact that is highly underappreciated is that most functions in mathematics cannot be evaluated exactly. Common examples include functions such as $\exp(x), \cos(x), \ln(x)$.

This leads us to consider ways to approximate a function $f(x)$ by something else that is easier to compute. Let's consider the simple case of the exponential:

$$\exp(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ...$$

The first approximation we might try is $1 + x$, which is clearly easy to compute, but likely not very accurate. A second attempt might be to use the first three terms $1 + x + \frac{x^2}{2}$. We could proceed in the obvious way, adding new terms until we are satisifed or we get tired.

But there is an underlying question - when should we stop? One would hope that as k increases we should get more accuracy, but it's also clear that the more terms we use the harder is to compute the approximation.

Goal: Find functions that *approximate* f at some point, but with some *guarantee of accuracy*. Hopefully, these approximating functions are also *cheaper* to compute than the original function.

> **!** Important
>
> In numerical analysis there is a constant trade off between accuracy and computational work. We seek to balance these two goals for a given problem.

A first natural approach is to use a polynomial. They are 1) easy to understand, 2) easy to compute, and 3) in general easier to analyze. One commonly used approach is that of approximating a function via a Taylor polynomial. In addition, we will see that Taylor's Theorem with remainder can be used to analyze our approximations. You should be familiar with both the concepts and how to apply them in different situations.

Definition 2.1 (Taylor Polynomials)**.** Suppose f is a function with n derivatives at the point $x = x_0$. Then the nth Taylor polynomial for f at x_0 is given by:

$$P_n(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2!}f''(x_0) + \cdots + \frac{(x - x_0)^n}{n!}f^{(n)}(x_0) \tag{2.1}$$

Let's take a quick look at how some of the approximations work on $exp(x)$.

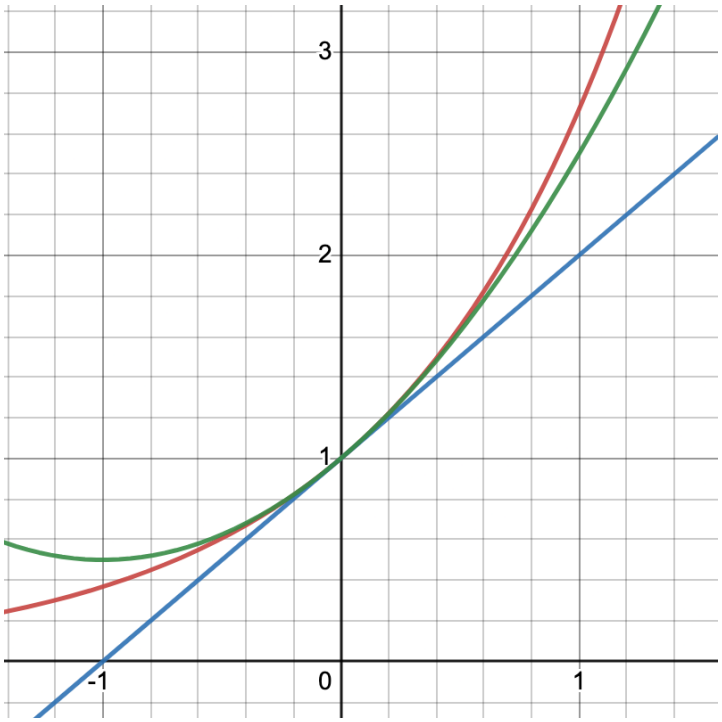


Figure 2.1: First and second order approximations to e^x. Red is exp(x), green is $1 + x + x^2/2$, blue is $1 + x$

Some observations:

- As n increases, we might expect accuracy to increase
- As we move away from the selected point x_0, we might expect the accuracy to decrease.

Using Taylor polynomials to approximate a function $f(x)$ leads to several natural questions:

1. How do we know all the derivatives exist?
2. What is the error when approximating $f(x)$ by $P_n(x)$?
3. Given a desired accuracy, how do we choose n?

The question of the existence of all of the derivatives is a non-trivial one. We will have need of them in order to analyze the approximation, but in practice, one rarely has more than the first derivatives available. For the remainder of the course, we will assume that we have enough derivatives to allow us to proceed with our analysis, but the student should be aware that such assumptions are difficult to establish in real-world problems.

2.2 Taylor's Theorem

In order to answer questions 2-3 above, we will need the following theorem.

Theorem 2.1 (Taylor's Theorem with remainder)**.** *Let $f \in C^{n+1}$ on an interval I containing the real number x_0. Also, let $P_n(x)$ be the nth Taylor polynomial of f at x_0 as defined by Equation 2.1. Then for each x in I, we can write*

$$f(x) = P_n(x) + R_n(x),$$

where $R_n(x)$ is given by:

$$R_n(x) = \frac{(x - x_0)^{n+1}}{(n+1)!} f^{(n+1)}(\xi),$$

for some ξ between x_0 and x.

> **!** Important
>
> It is important to note that the value of ξ will depend on x, (and hence the derivative term $f^{n+1}(\xi)$).

Taylor's theorem will prove to be incredibly useful in our analysis of algorithms. In particular, we will use it to determine the accuracy of various approximations and more importantly it will be essential in the error analysis of algorithms.

> **Remark**
>
> We will have use of other forms of Taylor's Theorem - you should get comfortable recognizing and using them. For example, use the substitution $h = x - x_0$ to rewrite Taylor's Theorem.

2.3 Examples/Exercises

Example 2.1. Consider $f(x) = \exp(x)$. Evaluate the nth Taylor polynomial about the point $x_0 = 0$ and evaluate its remainder at x_0.

Solution.

First we write down the Taylor Polynomial for $f(x) = \exp(x)$ at x_0.

$$P_n(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2!}f''(x_0) + \cdots + \frac{(x - x_0)^n}{n!}f^{(n)}(x_0)$$

Next we note that $f^{(k)}(x) = \exp(x), k = 0, 1, 2, ...$, so all of the derivatives exist. In particular, for the point $x_0 = 0$, we can also evaluate the function and all of its derivatives, $e^{x_0} = e^0 = 1, \forall\ k = 0, 1, \ldots$.

That means we can write the Taylor polynomial and the remainder term as follows:

$$P_n(x) = 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!}$$
$$R_n(x) = \frac{x^{n+1}}{(n+1)!}e^{\xi(x)},$$

Exercise 2.1 (In class exercise)**.** Consider $f(x) = \sqrt[3]{x}$. a) Find the first and second Taylor Polynomials for $f(x)$ at $x_0 = 8$. b) Evaluate both Taylor polynomials at the point $x = 11$.

Solution.

a) Let's break this down into steps:

Step 1. Writing out the first two polynomials using Taylor's theorem we see that:

$$P_1(x) = f(x_0) + (x - x_0)f'(x_0)$$
$$P_2(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2!}f''(x_0)$$

Step 2. Now let's substitue for the point $x_0 = 8$:

$$\begin{aligned} P_1(x) &= f(8) + (x - 8)f'(8) \\ P_2(x) &= f(8) + (x - 8)f'(8) + \frac{(x - 8)^2}{2!}f''(8) \end{aligned} \tag{2.2}$$

Step 3. The next step is to calculate the various derivative terms in the polynomial and evaluate them at the point $x_0 = 8$. It is helpful to write out a table that includes all of the necessary terms as shown below.

Table 2.1: Taylor Polynomial terms for $\sqrt[3]{x}$

Term	Evaluation at $x_0 = 8$
$f(x) = \sqrt[3]{x}$	$f(8) = \sqrt[3]{8} = 2$
$f'(x) = \frac{1}{3x^{2/3}}$	$f'(8) = \frac{1}{3 \cdot 8^{2/3}} = \frac{1}{12}$
$f''(x) = \frac{-2}{9x^{5/3}}$	$f''(8) = -\frac{2}{9 \cdot 8^{5/3}} = -\frac{1}{144}$

Step 4. Finally, inserting these values into Equation 2.2 yields:

$$P_1(x) = 2 + \frac{(x - 8)}{12}$$
$$P_2(x) = 2 + \frac{(x - 8)}{12} - \frac{(x - 8)^2}{288}$$

b) Estimate $f(11) = \sqrt[3]{11}$

$$P_1(11) = 2 + \frac{(11 - 8)}{12} = 2.25$$
$$P_2(11) = 2 + \frac{(11 - 8)}{12} - \frac{(11 - 8)^2}{288} = 2.21875$$

For comparison, $\sqrt[3]{11} \approx 2.22398009$

2.4 Other forms for the Remainder Term

There are alternate forms for the remainder term that you may see, the most common of which is the *integral form*, which is given by:

$$R_n(x) = \frac{1}{n!}\int_{x_0}^{x}(x-t)^n f^{(n+1)}(t)dt,$$

It is fairly easy to show that the two forms are equivalent through the use of the Weighted Mean Value Theorem for Integrals (Theorem 1.3). First note that the first term $(x-t)^n$ does not change sign over the interval $[x_0, x]$. Also by assumption $f^{(n+1)}$ is continuous on the same interval. As a result, we can use the WMVTI as follows.

$$\begin{aligned}\int_{x_0}^{x}(x-t)^n f^{(n+1)}(t)dt &= f^{(n+1)}(\xi)\int_{x_0}^{x}(x-t)^n dt,\\ &= f^{(n+1)}(\xi)\cdot\frac{-(x-t)^{n+1}}{n+1}\Bigg|_{x_0}^{x},\\ &= f^{(n+1)}(\xi)\cdot\frac{(x-x_0)^{n+1}}{n+1}.\end{aligned}$$

The last step follows since the integrand vanishes at the upper endpoint. If We now multiply both sides by $1/n!$ we get the desired result.

> **i** Remark
>
> There are other forms of the remainder term as well. The ones presented here are the most common. Which form you decide to use will depend on the particular situation. But it's nice to have options.

2.5 Choosing $P_n(x)$ to achieve a desired accuracy

This now leaves us with our finaly question: can we determine what degree polynomial is required for a given accuracy?

Example 2.2. Once again, let's consider $f(x) = \exp(x)$. Evaluate the *nth* Taylor polynomial about the point $x_0 = 0$. How big should n be to approximate $\exp(1)$ to an accuracy of 10^{-9}.

Solution.

Recall that

$$R_n(x) = \frac{x^{n+1}}{(n+1)!} \cdot e^{\xi}$$

Evaluating at $x = 1$ gives us

$$R_n(x) = \frac{1}{(n+1)!} \cdot e^{\xi}$$

Our overall goal is to bound the remainder (error) such that:

$$l \leq R_n(1) \leq u$$

where l, u are some chosen bounds. In general it will be easier to work with the following form:

$$|R_n(1)| \leq M$$

for some value of M, in this case e.g. we would let $M = 10^{-9}$.

As with many cases in numerical analysis, we will need to make use of any information we may have about the function and the domain we're working with. In particular for this function we know that, $0 < \xi < 1$, by Taylor's Theorem. That means that:

$$1 = e^0 < e^{\xi} < e^1 = e$$

dividing by $(n+1)!$ we see that:

$$\frac{1}{(n+1)!} < \frac{e^{\xi}}{(n+1)!} < \frac{e}{(n+1)!}$$

Note that the middle term is just $R_n(1)$. Since we're not supposed to actually know the value of e, let's just assume that we know a rough upper bound, say 3, (but we could just as easily have used any other number) and substitute it into the last term. Then what we're really saying is that we would like to have:

$$|R_n(1)| < \frac{3}{(n+1)!} < 10^{-9}$$

A quick calculation suggests that $n + 1 \geq 13$ satisfies this condition, so that a $12th$ degree Taylor polynomial should gives us the desired accuracy.

Exercise 2.2 (In class exercise)**.**

a) Write out the nth Taylor Polynomial for $f(x) = \sin(x)$ about $\pi/4$.
b) Find the smallest degree Taylor polynomial such that the remainder

$$|R_n(\pi/4)| < 10^{-6}$$

on $[0, \pi/4]$.

2.6 Summary

- We showed that we can approximate a given function through the use of a Taylor Polynomial.
- Taylor's Theorem allows us to write down exactly what the remainder (error) term is.
- Two (equivalent) forms of the remainder term were introduced: Lagrange form and the Integral Form.
- We can estimate the degree of the Taylor Polynomial needed to guarantee a desired accuracy over a given interval.
- Useful bounds will depend on knowledge of the given function and its derivatives.

2.7 References

A good online reference for some of the material above can be found at openstax.org.

- Taylor polynomials and Taylor's Theorem

Another good set of resources are the one-pagers provided by the UCM Math Center. You can check them all out at: UCM The Math Center and in particular the Math 23 refresher one-pager might prove useful: Math 23 refresher.

And of course, there's always Wikepedia:Taylor's Theorem

Revised: Tuesday, Sept. 5, 2023

3 Errors

There will always be a small but steady demand for error analysts to … expose bad algorithms' big errors and , more importantly, supplant bad algorithms with provably good ones

W. Kahan (1980)

There's no sense in being precise when you don't even know what you're talking about

John von Neumann

Topics Covered:

- What types of errors might one encounter in numerical computing and what are their sources?
- When is a computed solution accurate enough?
- What is the difference between precision and accuracy?

3.1 Sources of Errors

One of the first challenges when working in numerical analysis is the question of how to determine when a computed solution is sufficiently accurate. In order to talk about this we first need to develop some nomenclature and definitions that will allow us to quantify more precisely how good a solution we have.

Definition 3.1. Suppose we have x and an approximation $\hat{x}$. Then the **absolute error** is given by

$$|x - \hat{x}|.$$

Definition 3.2. Similarly the **relative error** is defined by

$$\frac{|x - \hat{x}|}{|x|}.$$

Both absolute and relative errors are used in our analysis, but the relative error is generally preferred in practice as it is scale independent, and as long as you stay away from $x = 0$.

3.1.1 Precision and Accuracy

Precision and accuracy are often confused. In numerical analysis, they will have a specific meaning.

By **accuracy**, we mean the absolute or relative error of an approximation as defined in (Definition 3.1,Definition 3.2). **Precision** will refer to the accuracy with which the basic arithmetic operations (+, -, *, /) are performed. More of this to come.

One should also note that accuracy is not limited by the precision of a computer - there are many software packages that can provide extended precision in numerical calculations.(Bailey 1993)

3.2 Sources of Errors

The next question that naturally arises is what types of errors might we encounter in solving a problem numerically and where might they arise? As it turns out, there are many different ways to classify errors. We will use the following general categories:

- **model errors/uncertainty**:
  - errors in mathematical models used to approximate a real world problem
  - errors in the input data, e.g. physical measurements, observations, etc.
- **approximation errors**:
  - errors in approximating our problem due to truncation/discretization , e.g. Taylor series, interpolation, integration.
- **roundoff errors**:
  - errors due to the finite precision inherent of computer arithmetic. This is a basic fact of computational mathematics and there is not much we can do to limit these. However, wise choices of algorithms can circumvent some of the problems and bad choices of algorithms can exarcebate them.

! Remarks

1. Model errors and their quantification is a subject area all of its own - we will not cover it. The interested student can check out the vast literature on topics such as *uncertainty quantification, verification, and validation.*
2. Approximation errors (truncation/discretization) usually dominate other types of errors and their analysis is a major task of numerical analysis

Example 3.1 (Approximation/Discretization) Error)**.**

Suppose we want to compute an approximation to $f'(x)$ for some function $f(x)$ at the point $x = x_0$.

This situation might arise if you have the function but do not know $f'(x)$, or perhaps it is too expensive to compute.

Let's use Taylor's Theorem. Recall

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2!}f''(x_0) + \cdots + \frac{(x - x_0)^{(n+1)}}{(n+1)!}f^{(n+1)}(c).$$

for some c between x and x_0. Note, that we need to assume f has $n + 1$ derivatives in some interval.

Now let's introduce some new notation. In particular, let

$$x = x_0 + h, \quad h > 0,$$

so that

$$h = x - x_0.$$

Then we can rewrite the Taylor expansion as

$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2!}f''(x_0) + \cdots + \frac{h^{(n+1)}}{(n+1)!}f^{(n+1)}(c).$$

Rearranging the terms to put the derivative on the left hand side of the equation gives us:

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} - \left[\frac{h}{2}f''(x_0) + \cdots + \frac{h^{(n)}}{(n+1)!}f^{(n+1)}(c)\right].$$

> **Idea**
>
> Use the first two terms on the RHS to approximate the derivative at x_0 and "throw away" the rest of the terms.

Taking absolute values, we can see that:

$$\left|f'(x_0) - \frac{f(x_0 + h) - f(x_0)}{h}\right| = \left|\frac{h}{2}f''(x_0) + \cdots + \frac{h^{(n)}}{(n+1)!}f^{(n+1)}(c)\right|, \tag{3.1}$$

where we denote the RHS of Equation 3.1 by the **discretization error**. If $f''(x_0) \neq 0$, then for small enough h, we can estimate the discretization error by:

$$\left| f'(x_0) - \frac{f(x_0+h) - f(x_0)}{h} \right| \approx \left| \frac{h}{2} f''(x_0) \right|.$$

Again, we are discarding all higher order terms, which if h is "small enough" seems like a reasonable idea (of course we will need to prove this rigorously, but that comes later). In the meantime, it will prove helpful to denote quantities like the right hand side by the following notation:

$$\left| \frac{h}{2} f''(x_0) \right| = O(h).$$

Here by the notation, $O(h)$ (read Big O of h), we mean that a quantity is at most proportional to h, for example Ch, for some constant C. One way to think of it is as

$$\lim_{h \to 0} \frac{O(h)}{h} = C < \infty$$

This can be generalized to higher orders of h as well. (We'll talk more about his later as well.)

The important concept is that we talk about the above *approximation for the derivative as being an $O(h)$ approximation to the true derivative.* For more details (or if you need a refresher) on Big O notation see Section A.1.

Example 3.2. Approximate $f'(x)$ for $f(x) = \cos(x)$ and $h = 10^{-3} - 10^{-15}, x = \pi/6$.

> **!** Remark
>
> Notice that for each decrease in the value of h by an order of magnitude, both the absolute and relative error have a corresponding decrease in their values. *This is exactly what the theory predicts.* However, it is important to note that this is true only up to a certain point. We'll discuss this more fully when we get to the sections on computer arithmetic and roundoff error.

Exercise 3.1. Approximate $f'(x)$ for $f(x) = \sin(x)$ at the point $x_0 = 1.2$ and using $h = 0.1, 0.01, 0.001$. Can you estimate the discretization error? Explain.

Solution.

3.3 Summary

This section covered

- the concepts of absolute error and relative error,
- the difference between accuracy and precision,
- presented some of the most common sources of errors when modeling problems through the use of mathematical or computational models,
- the effects of discretization error and,
- provided an example for approximating the first derivative of some known function.

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

4 Floating Point Systems

Topics Covered:

- What is a floating-point system?
- What are the fundamental parameters describing a floating point system
- What are roundoff errors?
- What are cancellation Errors

4.1 Floating Point Systems

Computational mathematics is different than "continuous" mathematics, with the main difference being that instead of having infinite precision, the arithmetic is both *fixed and finite,* due to the fact that we are working on a computer. This means that not all real numbers can be represented on a computer - and most real numbers can only be approximated.

To help us understand some of the complications, let's review how we normally represent numbers in scientific notation. For example to represent the number 51.7538 in scientific notation we could write it as:

$$+0.517538 \cdot 10^2$$

In this notation, the various components are given the following names:

Table 4.1: Parts of Scientific Notation

Name	Value
Sign	+/-
Base	10
Exponent	+2
Significand (Mantissa)	0.517538

The number of digits in the significand is known as the **precision.** This will become important in future discussions. It is also important to note that the scientific notation is not unique. We could just as easily have written:

$$+0.0517538 \cdot 10^3$$

or any number of other ways, by changing the exponent. As a result, we usually restrict our mantissas to have the leading digit (most significant) be nonzero. These numbers are known as ***normalized numbers***.

Remarks.

1. We can only store a finite number of the set of real numbers on any computer. At best we can expect a solution to be as accurate as the precision of that computer.
2. Every computer operation we do will result in intermediate results which are also only as accurate as we can store them.
3. These inaccuracies can accumulate further compromising the accuracy of the final result. These accumulated errors are known as **roundoff errors.** Some typical situations that cause roundoff include:
    - inaccurate additions of large sequences of numbers
    - taking the difference of two nearly equal numbers
    - inaccurate solution of ill-conditioned problems (more on this later).

> Tip
>
> We need to be aware of potential numerical pitfalls and code programs accordingly. **A good numerical analyst is always on the lookout for these problems!**

Let's adapt our scientific notation for arithmetic on a computer. Specifically, instead of the usual base 10 system, we use a system with base 2 (binary). As before, let's represent a number as follows:

$$x = \pm(0.d_1 d_2 d_3 \ldots d_t) \cdot 2^e$$

for some exponent e. Notice that when we write it in this form what we're really saying is that

$$x = \pm\left(\frac{d_1}{2^1} + \frac{d_1}{2^2} + \frac{d_1}{2^3} + \ldots \frac{d_1}{2^t}\right) \cdot 2^e$$

The representation of a real number on a computer is called the floating point representation, and we will denote it by $fl(x)$. Here again, we will assume that $d_1 \neq 0$.

With these changes in mind, let's consider how to represent the finite set of floating points on a computer.

By a **floating-point number system** we mean a finite subset $F = F(\beta, t, e_{min}, e_{max})$, of the real numbers whose elements have the form:

$$x = \pm m \cdot 2^{e-t+1}$$

Here β is the base (2 on almost all computers), t is the precision and $e \in [e_{min}, e_{max}]$, and m is the significand satisfying $m \leq 2^t - 1$.

> **i** Why the +1?
>
> There's a +1 added to the exponent so as to be consistent with IEEE standard, which we will discuss later on. For now, you can just simply take it as part of the definition.

In order to ensure that we have a unique representation for each floating point number, we also make the assumption that $m \geq \beta^{t-1}$, i.e. the leading digit of m is nonzero and hence it is normalized.

Example 4.1. Let's see what this means on a toy floating point system. Suppose we have a system with $\beta = 2, t = 3, e_{min} = -2, e_{max} = 3$. What does this look like?

The figure below depicts all of the positive numbers representable in our system F.

There are some general remarks we can make about our floating-point system:

- There are a total of 49 points: 24 positive, 24 negative, and 1 zero.
- The numbers are equally spaced between each power of 2
- The spacing increases by a factor of 2 at each power of 2.

There are several important numbers in a floating-point system that we will use a lot:

The ***unit roundoff*** (rounding unit) is defined by

$$u = \frac{1}{2}\beta^{1-t}. \tag{4.1}$$

In our example,

$$u = \frac{1}{2}2^{1-t} = \frac{1}{2}2^{-2} = 0.125.$$

The unit roundoff is frequently used in error analysis.

Machine epsilon is defined by

$$\epsilon = \beta^{1-t}. \tag{4.2}$$

Machine epsilon, is frequently used in practice and as a means to denote very small quantities in our computations.

In our example,

$$u = 2^{1-t} = 2^{-2} = 0.25.$$

The **maximum number** x_{max} is defined by

$$x_{max} = \beta^{e_{max}}(\beta - \beta^{1-t}).$$

In our example,

$$x_{max} = 2^3(2 - 2^{-2}) = 8 \cdot (2 - 1/4) = 14.$$

The **minimum number** x_{min} is defined by

$$x_{min} = \beta^{e_{min}}.$$

In our example,

$$x_{min} = 2^{-2} = 1/4 = 0.25.$$

Both the maximum and minimum value of x are important to know so that we have a better understanding of our computational results.

> **Unit Roundoff and Machine Epsilon**
>
> Unit roundoff is just $2 * \epsilon$, i.e. twice machine epsilon (for binary computers). You only need remember one of the two. For numerical analysts unit roundoff is slightly more important.

4.1.1 Types of Rounding

Since F only has a finite number of floating point numbers it can represent, most calculations will produce a result that needs to be mapped onto our field F. There are two possible choices: chopping and rounding. In chopping, the computed result Is simply truncated to whatever precision is available. In rounding, we take as our result whichever floating point is nearer. If there's a tie, we need a rule to break the tie, and normally we will round to the number that has an even last digit - round to even.

In terms of the notation we used above, chopping and rounding can be described as:

- **Chopping:** truncate all digits after digit t:

$$fl(x) = \pm .d_1 d_2 d_3 \cdots d_t \times \beta^e$$

- **Rounding:**

$$fl(x) = \pm .d_1 d_2 d_3 \cdots d_t \times \beta^e \quad d_t < \beta/2$$

  or

$$fl(x) = \pm (.d_1 d_2 d_3 \cdots d_t + \beta^{1-t}) \times \beta^e \quad d_t > \beta/2$$

In the case of a tie, one must also choose a tie-breaking rule.

> **Other forms of rounding**
>
> There are several ways that one can round including, round down, round up, round towards zero, round to nearest. Most often when speaking about rounding we will mean round to nearest. In addition, the IEEE standard requires that the default rounding mode be round to nearest. In this case, we always round towards the nearest number and if there's a tie, we take the number whose least significant bit is equal to zero.

4.2 Summary

The following table summarizes some of the main characteristics of floating point systems.

Interpretation	Parameter	Value
Unit roundoff. Spacing of numbers between 1/2 and 1	u	$\frac{1}{2}\beta^{1-t}$
Machine epsilon. Spacing of numbers between 1 and 2	ϵ	β^{1-t}
Largest floating point number representable by F	x_{max}	$\beta^{e_{max}}(\beta^t - 1)$
Smallest normalized floating point number representable by F	x_{min}	$\beta^{e_{min}}$
Total number of floating point numbers representable by F		$2(\beta - 1)\beta^{t-1}(e_{max} - e_{min} + 1) + 1$

4.3 Advanced

We will note in passing that it is possible to represent even smaller numbers below N_{min} through the use of what are known as subnormal numbers. However, this is beyond the scope of this introductory course. The interested reader can check out one of many references on this topic (e.g. Accuracy and Stability of Numberical Algorithms, 2nd ed., N. Higham (2002)).

4.4 References

1. (Goldberg 1991) *What Every Computer Scientist Should Know About Floating-Point Arithmetic.*
2. (Nicholas J. Higham 2021) *The Mathematics of Floating Point Arithmetic.*
3. (Nicholas J. Higham 2002) Accuracy and Stability of Numerical Algorithms.
4. (Overton 2001) *Numerical Computing with IEEE Floating Point Arithmetic.*


```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

5 IEEE Floating Point Standard

Topics Covered:

- How is floating point arithmetic implemented on modern computers (IEEE Floating Point Standard)?
- What are the fundamental machine parameters describing precision and limits of floating point numbers for IEEE?
- What can we expect in terms of errors when doing computer arithmetic?

5.1 IEEE Floating Point Standard

Computers are all based on binary arithmetic, i.e. they only store 1's and 0's. Instead of decimal digits, computers use **b**inary dig**its**, which have come to be known as **bits**.

A long time ago machines all did floating point representations and arithmetic slightly differently. This led to all sorts of problems. In 1985, IEEE published the IEEE 754 Floating Point Standard (since updated in 2008). Now almost all computers conform to this standard. Codes are much more portable and less reliant on special programming tricks (affectionally known as kludges).

IEEE has both single and double precision representations, the main difference being the number of bits used to represent the floating point number; 32 bits for single precision and 64 bits for double precision. Almost everyone uses double precision and we will assume so for the rest of this course.

The IEEE double precision representation uses 64 bits (binary digits) that are divided up as follows:

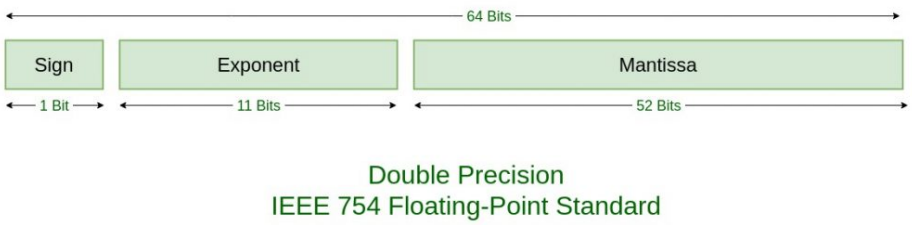


Figure 5.1: IEEE Double Precision

> **Single Precision**
>
> For completeness, the IEEE ***single precision*** standard specifies 8 bits for the exponent and 22 bits for the mantissa. While it was initially used widely, today this is reserved for special applications requiring more speed.

In the IEEE standard, the double precision representation of a floating point number is written as:

$$fl(x) = (-1)^s \ 2^{e-1023} \cdot (1+m) \tag{5.1}$$

Each of the components plays a different role:

1. The sign is denoted by s and is either 0 (for positive) or 1 (for negative).
2. The exponent is known as a biased exponent, in the case of double precision the bias is 1023. Since the exponent stored is interpreted as an unsigned integer, we need some mechanism for allowing both positive and negative exponents. As a result of using the bias, all stored exponents less than 1023 can be interpreted as negative exponents.
3. IEEE assumes that floating point numbers are normalized, meaning that the leading significant digit is 1. This allows us to not store it, thereby giving us one more bit of precision. The leading digit is called a *hidden* bit.
4. The mantissa m corresponds to the digits to the right of the decimal point. In double precision, this amounts to 52 digits, but because of the hidden bit, we actually have 53 bits of precision. This results in approximately 16 decimal digit precision.

Example 5.1. What is the floating point number represented by the following bit string?

Sign	Exp (11 bits)	Mantissa (52 bits)
0	10000000011	1011100100010000000000000000000000000…….. 0

Solution:

To compute the floating point number represented by this bit string, we need to use Equation 5.1 and decipher the various components: s, e, and m. Let's take these one step at a time.

1. The sign, s = 0, which means we're dealing with a positive number.

2. The unbiased exponent can be computed by:

$$\begin{aligned} e &= 10000000011 \\ &= 1 \cdot 2^{10} + 0 + ... + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= 1024 + 0 + ... + 2 + 1 \\ &= 1027 \end{aligned}$$

To compute the biased exponent for the real number we need to subtract the bias, which for double precision is 1023 as noted above. This means that the exponent for the floating point number is $1027 - 1023 = 4$.

3. Following a similar procedure for the mantissa we get:

$$\begin{aligned} mantissa &= 10111001000100000000000000000000000000000000....0 \\ &= 1 \cdot 1/2 + 0 \cdot 1/2^2 + 1 \cdot 1/2^3 + 1 \cdot 1/2^4 + 1 \cdot 1/2^5 + 1 \cdot 1/2^8 + 1 \cdot 1/2^{12} + 0 + ... + 0 \\ &= 1/2 + 1/8 + 1/16 + 1/32 + 1/256 + 1/4096 + 0 + \cdots + 0 \\ &= 0.722290039 \end{aligned}$$

Here we have to remember that the IEEE standard states that floating point numbers are normalized, so the leading (hidden) bit is 1 and the mantissa represents the digits to right of the decimal point.

Putting it all together and using Equation 5.1 we get:

$$\begin{aligned} fl(x) &= + \ 2^4 \cdot (1 + 0.722290039) \\ &= 16 \cdot (1.722290039) \\ &= 27.56640626 \end{aligned}$$

5.2 Important Parameters for IEEE Floating Point System

When doing numerical computations, it is important to be aware of what the range of floating point numbers on a computer is. It is not uncommon to have a long computation where some of the intermediate results may extend outside the range that is allowed on the computer.

5.2.1 Unit roundoff and machine epsilon

Recall from Section 4.1 Equation 4.1 that unit roundoff was defined as:

$$u = 1/2 \cdot \beta^{1-t}.$$

Hence for a double precision IEEE system, this results in a unit roundoff of:

$$u = 1/2 \cdot 2^{-52} \approx 1.11 \cdot 10^{-16},$$

and

$$u = 1/2 \cdot 2^{-23} \approx 5.96 \cdot 10^{-8},$$

for single precision.

Machine epsilon is as previously defined (Equation 4.2) just $2 \cdot u$, i.e. twice the unit roundoff. You only need remember one of the two. For numerical analysts *unit roundoff* is slightly more important.

5.2.2 Range of Floating Point Numbers

The IEEE standard states that the smallest exponent for double precision is $e_{min} = -1022$, and the largest exponent is $e_{max} = 1023$. Given this we can compute the smallest and largest numbers representable in double precision using Equation 5.1.

The smallest number representable:

- $s = 0$
- $e_{min} = -1022$
- $m = 0$

so

$$fl(x) = +2^{-1022} \cdot (1 + 0.0)$$

$$N_{min} \approx 2.225073858507201 \cdot 10^{-308}$$

The largest number representable:

- $s = 0$
- $e_{max} = 1023$
- $m = 1 - 2^{-52}$

so

$$\begin{aligned} fl(x) &= +2^{1023} \cdot (1 + 1 - 2^{-52}) \\ &= 2^{1023} \cdot (2 - \epsilon) \end{aligned}$$

where $\epsilon = 2^{-52}$.

$$N_{max} \approx 1.797693134862316 \cdot 10^{308}$$

One might ask what happens if a computation generates a floating point number outside the range of the allowed numbers within the IEEE floating point standard. If we compute a number that is less than N_{min} we get **underflow** (which is generally set to 0). If we compute a number that is greater than N_{max} we get **overflow** (which could set the value to Infinity or it might generate an exception).

It is also possible to generate two other types of "numbers", specifically an Infinity (for example dividing by 0) and something called "not a number" or a **NaN** (e.g. through a computation such as $0/0, 0 \times \infty$, or the square root of a negative number). The IEEE standard reserves the exponent 2047 for both cases, which is why $e_{max} = 1023$. Ideally your codes should be on the watch for situations when this might occur and take the appropriate precautions.

5.2.3 Relative error of floating point representation

One of the nice things about the IEEE standard is that it can provide certain guarantees about the accuracy of the floating point representation of a real number. In particular, it can be shown that:

$$fl(x) = x(1 + \delta), \quad |\delta| \leq u,$$

where $u = \frac{1}{2} \cdot 2^{1-t}$ is the unit roundoff, and t is the computer precision.

Another way to write this is as:

$$\left| \frac{fl(x) - x}{x} \right| \leq u.$$

> **!** Important
>
> In other words, the relative error in the floating point representation of a real number will be less than or equal to the unit roundoff on the machine.

5.2.4 Floating Point Arithmetic Operations

The IEEE standard specifies that all computer operations must behave as if the requested operation were done in infinite precision and then rounded to the nearest floating point within the system. In particular, it can be shown that:

$$fl(x\ op\ y) = (x\ op\ y)(1+\delta), \quad |\delta| \leq u,$$

where $op = +, -, *, /$.

One way to interpret this is that the computed result from any floating point operation is as good as the exact answer rounded to the nearest floating point.

5.3 Summary

The following table summarizes some of the main characteristics of IEEE floating point formats useful for numerical analysis:

Type	Size	Mantissa	Exponent	Unit Roundoff	emin	emax	Range
Single	32 bits	23 bits + 1 hidden	8 bits	$2^{-24} \approx 5.96 \cdot 10^{-8}$	-126	127	$10^{\pm 38}$
Double	64 bits	52 bits + 1 hidden	11 bits	$2^{-53} \approx 1.11 \cdot 10^{-16}$	-1022	-1023	$10^{\pm 308}$

i Advanced

We will note in passing that it is possible to represent even smaller numbers below N_{min} through the use of what are known as ***subnormal*** numbers. However, this is beyond the scope of this introductory course. The interested reader can check out one of many references on this topic (e.g. Accuracy and Stability of Numerical Algorithms, 2nd ed., N. Higham (2002)).

5.4 Codes

Coding tip

In matlab you can call the function *realmin* to compute N_{min}; likewise the function *realmax* will compute the value for N_{max}.
You can do something similar in python through the use of the numpy **finfo** function. For example:

```
import numpy as np
print(np.finfo(float).eps)
print(np.finfo(float).max)
print(np.finfo(float).tiny)
```

R has a similar capability which prints out the entire set of machine characteristics through the **.Machine** function. For specific values you need to append the name. For example:

```
# Output machine parameters such as eps, Nmax, Nmin, etc.
.Machine$double.eps
```

```
[1] 2.220446e-16
```

```
.Machine$double.xmax
```

```
[1] 1.797693e+308
```

```
.Machine$double.xmin
```

```
[1] 2.225074e-308
```

5.5 References

1. (Goldberg 1991) *What Every Computer Scientist Should Know About Floating-Point Arithmetic.*
2. (Nicholas J. Higham 2021) *The Mathematics of Floating Point Arithmetic.*
3. (Nicholas J. Higham 2002) Accuracy and Stability of Numerical Algorithms.
4. (Overton 2001) *Numerical Computing with IEEE Floating Point Arithmetic.*
5. (Hennessy and Patterson 2011)Computer Architecture A Quantitative Approach, Appendix A Computer Arithmetic


```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

6 Stability and Conditioning

Topics Covered:

- What is an algorithm?
- What is a stable algorithm?
- What types of problems are amenable to accurate solutions?
- What is the condition number of a problem?

6.1 Algorithms

Algorithms have a long history dating back over 4000 years. The name itself is said to be derived from the mathematician al-Khwārizmī (c.780-c.850), who wrote a book on the subject.

In the broadest sense, one can think of an algorithm as a step by step procedure for solving some problem. A common analogy is that of a recipe in which one is given explicit instructions on how to make something. One can also think of an algorithm as a sytematic calculation or process for achieving some desired result.

More recently, the notion of *finiteness* has been added to the concept although it is not essential.

For our purposes we will use the following definition:

Definition 6.1. An ***algorithm*** is a well-defined process that takes an input (or inputs) and produces an output in a finite-number of steps or time.

6.2 Desired Properties of Algorithms

What kind of properties should a good numerical algorithm have? While one can come up with a long list of desirable characteristics for an algorithm, we will concentrate on three that were briefly introduced on the first day of class:

- Accuracy

- Efficiency
- Robustness

Let's take a look at each of these in turn.

Accuracy. Since a numerical algorithm is designed to solve some mathematical problem, one of the key characteristics should be that it provides us with an accurate approximation to the true solution, whatever that might be. We've already seen some examples of these and how we can use error analysis to allow us to predict the behavior of an algorithm.

Efficiency. We also would like to have an algorithm that is computationally as efficient as possible. For example, let's consider the problem of evaluating a polynomial given by:

$$p_n(x) = c_0 + c_1 x + c_2 x^2 + \dots c_n x^n.$$

A naive algorithm would require $O(n^2)$ operations or flops (floating point operations) as they are called. To see this all you need to do is think about each of the terms in the polynomial in turn and add up all of the operations. For example, the last term requires n multiplications: $n-1$ to compute x^n plus one more to multiply by c_n. Each term of lower order will require one less multiplication. In all, there are $n + (n-1) + (n-2) + \dots + 1 = O(n^2)$ multiplications. Finally there are the n additions required to sum up all of the individual terms.

Instead, we should consider rewriting the formula in what is known as the ***nested form***:

$$p_n(x) = c_0 + x(c_1 + x(c_2 + \cdots + x(c_{n-1} + x(c_n))\cdots)),$$

which only requires $O(n)$ operations. (You should work this out for yourself.) This may not make much of a difference for small values of n, but one nonetheless needs to be careful as generally speaking we don't know ahead of time how an algorithm will be used. For example, this polynomial evaluation could be at the heart of an algorithm that is called millions of times.

> **Flops**
>
> Historically, the operation count (flops) has been an important measure of efficiency (or at least workload). On today's computers, and especially supercomputers, there are many more things to consider when considering computational efficiency (memory access, communication, vectorization/parallelization, etc.).

Robustness. Finally, we would like an algorithm that we can "trust", in the sense that it either gives us an answer or reports back that something went wrong. Ideally, an algorithm should also work under most foreseeable circumstances and different values of inputs. Another thing to watch out for is the rate of accumulation of errors within an algorithm.

6.3 Stability and Conditioning

Next we consider what we can say about the problem and the algorithm to help us in making good decisions.

The first concept is that of the problem sensitivity or as it is more commonly referred, ***conditioning.*** By this we mean that a problem is ***well-conditioned*** if small changes in the inputs do not lead to large changes in the outputs. Otherwise the problem is said to be ***ill-conditioned.***

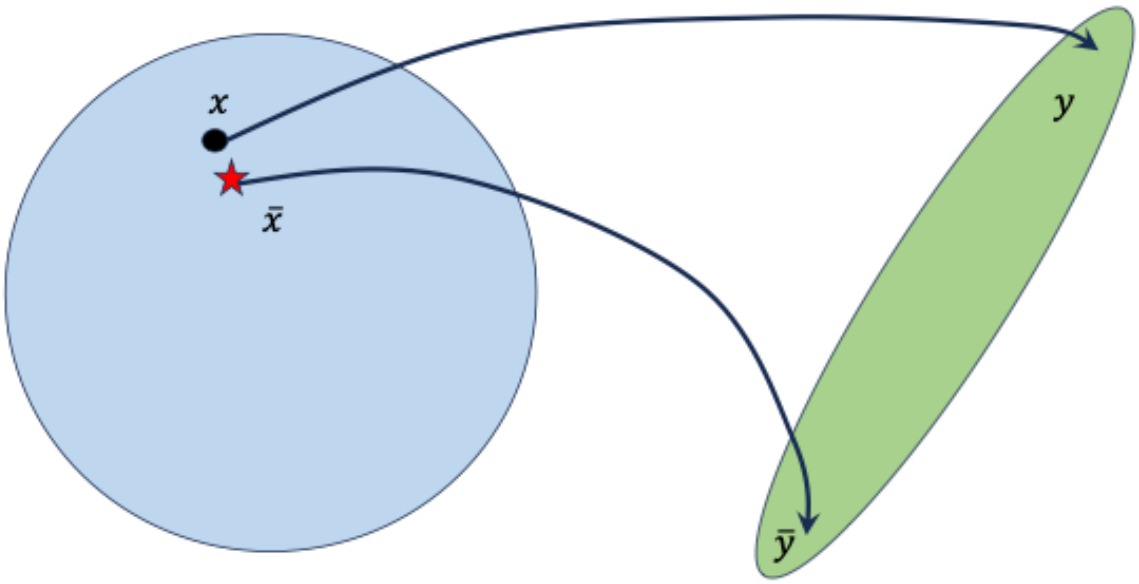


Figure 6.1: Ill-conditioned problem where small changes in the input can lead to large changes in the output

The second concept is that of stability in an algorithm. Here we say that an algorithm is ***stable*** if the output generated is the exact result for a slightly perturbed input. Another way of phrasing this is that a stable algorithm solves a ***nearby problem.***

Example 6.1. Let's take a look at a simple example of a well-conditioned problem first. Consider the function $f(x) = \sqrt{1+x}$ with $f'(x) = \frac{1}{2\sqrt{1+x}}$.

Let's suppose that we fix $|x| << 1$ and consider $\bar{x} = 0$ as a small perturbation of x. Then

$$\bar{y} = f(\bar{x}) = \sqrt{1+0} = 1,$$

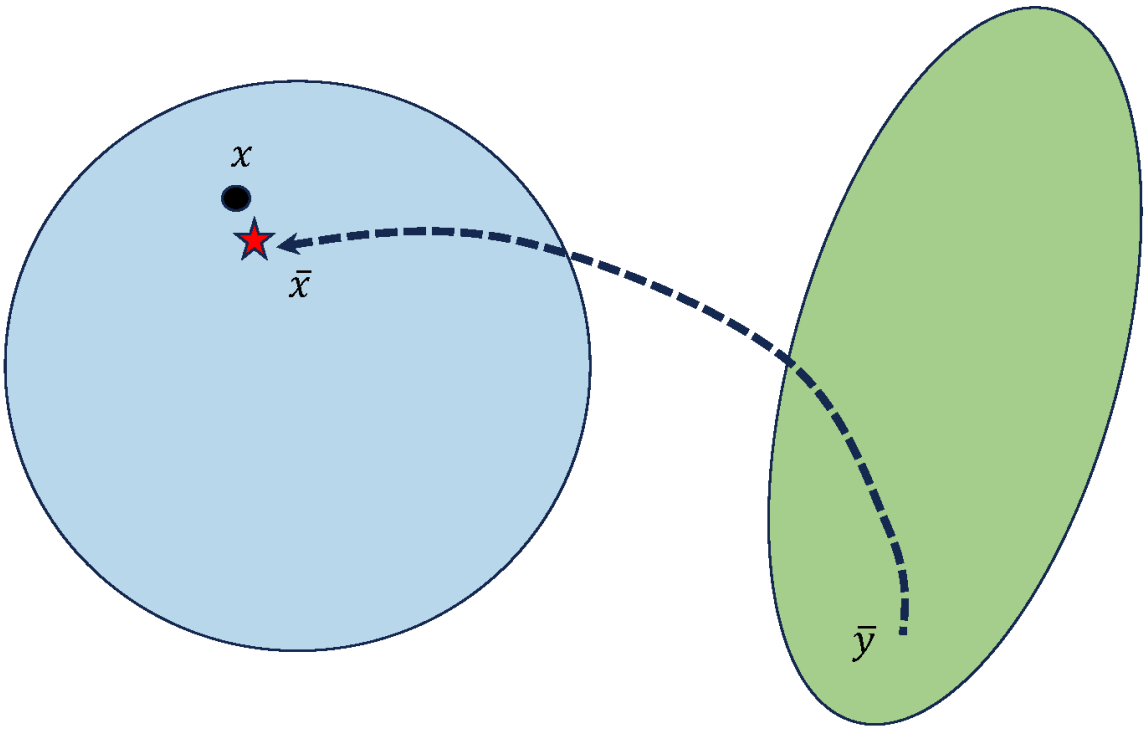


Figure 6.2: A stable algorithm is one that is the exact solution to a slightly perturbed problem.

and the change in the output can be given by

$$y - \bar{y} = \sqrt{1+x} - 1.$$

Now let's approximate $\sqrt{1+x}$ by the first 2 terms of a Taylor series about 0, i.e.

$$f(x) \approx 1 + \frac{x}{2}.$$

Substituting into the prior equation we can write the change in the output by:

$$y - \bar{y} \approx \left(1 + \frac{x}{2}\right) - 1 = \frac{x}{2}.$$

This can then be rewritten in terms of the change in the inputs as:

$$y - \bar{y} \approx \frac{1}{2}(x - \bar{x}).$$

Notice that if $x = 0.001$ then the change in the output is only $y - \bar{y} \approx 0.0005$, one half of what the change in input was.

We're now ready for a more rigorous definition of conditioning. First, note that when we introduced the concept of conditioning is was to describe the sensitivity of a problem (function) to change in the input data x. As such it seems natural to look at:

$$\frac{\text{Relative change in } f(x)}{\text{Relative change in } x}.$$

Mathematically we can translate this into:

$$\begin{aligned}
\left(\frac{\frac{f(x)-f(\bar{x})}{f(x)}}{\frac{x-\bar{x}}{x}}\right) &= \frac{f(x) - f(\bar{x})}{f(x)} \cdot \frac{x}{x - \bar{x}}, \\
&= \frac{f(x) - f(\bar{x})}{x - \bar{x}} \cdot \frac{x}{f(x)}, \\
&\approx \frac{f'(x) \cdot x}{f(x)}.
\end{aligned}$$

This leads us to the following definition:

Definition 6.2. The quantity

$$\kappa = \left| \frac{f'(x) \cdot x}{f(x)} \right|.$$

is called the *(relative)* ***condition number*** of the problem.

Let's now discuss the concept of stability. It is natural to expect that any algorithm will produce some roundoff error. In fact, it is common that accumulation of roundoff error will occur as we progress through a computation.

If the error grows slowly as the computation proceeds it won't be a problem, but if it grows too fast then we need to be more careful.

Let E_n be the error at the nth step of some computation.

If the error $E_n \approx C \cdot nE_0$, where C is a constant and E_0 is the original error, then the growth of error is said to be ***linear***.

If the error $E_n \approx C^n E_0$, where C is a constant and E_0 is the original error, then the growth of error is said to be ***exponential***.

An algorithm that has an exponential error growth rate is said to be ***unstable***. We should note that an algorithm could be stable for solving one problem, but unstable for solving another problem.

6.4 Tips on designing stable algorithms

While there are no hard and fast rules, there are several things to watch out for in designing an algorithm. The tips below are paraphrased from the excellent discussion given in Higham (Nicholas J. Higham 2002).

- Watch out for cancellation errors. Try to avoid subtracting quantities of near equal magnitude, especially if they are contaminated by error.
- Look for different formulations that are mathematically equivalent, but perhaps numerical better.
- It can be useful to write formulas in the form of:

$$x_{new} = x_{old} + \Delta x$$

where Δx is a small correction to the previous (old) value. You'll see that many numerical methods take this form.

- Minimize the size of intermediate results relative to the final result. It's also a good idea to check the intermediate results when first writing a code.
- Avoid ill-conditioned transformations of the problem.
- Take precautions against underflow and overflow.

6.5 Summary

This lesson covered the fundamental concepts of conditioning of numerical problems and stability of algorithms. Some of the takeaway messages include:

- The condition of a problem, which speaks to the sensitivity of the outputs to the inputs is related to the relative change in the output of a problem to the relative change in the inputs.
- All algorithms will generate some errors as a consequence of doing finite digit arithmetic. If the error growth rate is reasonable, for example linear with respect to the number of steps in the calculation, then we can expect good results (if the problem is also well-conditioned).
- If the error growth rate is exponential instead, then the algorithms is said to be unstable.
- An algorithm can be stable for one type of problem but unstable for another type of problem.

6.6 Advanced

The condition number of a problem is especially useful in numerical linear algebra, where it can be used to estimate the accuracy of the solutions for systems of linear equations, such as $Ax = b$. The stability of algorithms will become important in our study of initial value problems (ODEs).

6.7 References

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

Part II

Nonlinear Equations

Nonlinear equations

Suppose that we have a scalar, nonlinear equation $f : \mathbb{R}^1 \to \mathbb{R}^1$ on an interval $[a, b]$

If $x^* \in \mathbb{R}$ is such that

$$f(x^*) = 0, \tag{6.1}$$

then we will call x^* a *zero* or *root* of f.

For the time being, let's assume that $f(x)$ is continuous on the closed interval $[a, b]$. As it turns out, except for some special cases, nonlinear equations will not have a closed form solution, so we are naturally led to developing alternative means for solving Equation 6.1.

We could have just as easily defined the problem in higher dimensions, i.e. $f : \mathbb{R}^n \to \mathbb{R}^p$, for $n > 1, p > 1$, and n not necessarily equal to p. This would most likely be the case for a real-world problem, but while many of the ideas (and algorithms) can generalized to higher dimensions, the concepts are easier to understand in one dimension to begin with.

We note in passing that if $n = p$, then this is an example of finding the roots of a ***nonlinear system of equations***. If $n \neq p$, then this is an example of a ***nonlinear least squares*** problem.

> **Relation to optimization**
>
> Many of the methods that we study in this section are also applicable to the problem of optimization. For example:
>
> $$\min f(x)$$
>
> since a minimum will occur at points where $f'(x) = 0$. Of course, special care must be taken that the method approaches a minimum and not a maximum or an inflection point, but this is usually not hard to handle.

> **Examples**
>
> 1. $f(x) = 6x^2 - 7x + 2 = (2x - 1)(3x - 2)$
> 2. $f(x) = 2^{x^2} - 10x + 1$
> 3. $f(x) = \cosh(\sqrt{x^2 + 1} - e^x) + \log|\sin(x)|$
> 4. $f(x) = \blacksquare \leftarrow$ some black box

Most nonlinear equations cannot be solved analytically and as a result most approaches involve some form of iteration, which can be described at a high-level as:

1. First guess a solution,
2. Check to see how accurate it is, and if not satisfied,
3. Update the guess and try again, i.e. go back to step 1

This is the essence of an ***iterative method.***

Using an iterative technique to find roots of equations can be tricky however. The initial guess can sometimes be important, and if not chosen properly can lead to slow convergence or even non-convergence.

How one updates the guess is also important. Without some theory behind the updating scheme, an iterative method is nothing more than trial and error.

Finally, there is always the question as to when to terminate an iterative method, in other words when is an estimate of the root "good enough"? For all of these reasons, having an algorithm based on sound theoretical foundations is of fundamental importance.

In the next few sections, we discuss some of the more popular iterative methods for finding the roots of a nonlinear equation and provide some general guidance for when to use them. In particular, we will study the ***bisection method*** (Section 7.1), ***Newton's*** method (Section 8.1), ***secant method*** (Section 9.1) and ***fixed-point iterations*** (Section 10.1).

7 Bisection Method

7.1 Bisection

One of the simplest methods for solving a nonlinear equation is known as the bisection method. The main advantage is the robustness of the method - if the method is applicable to the problem, it is guaranteed to find a solution. On the other hand, the method can often take far more iterations than some of the other methods we will discuss.

Let's first describe the general method.

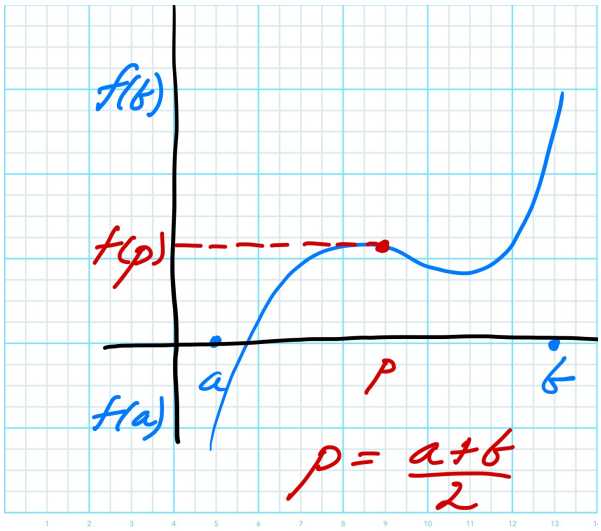


Figure 7.1: Bisection Method

Idea

Suppose that we happen to know two points where the function is of opposite sign as in Figure 7.1. Using the Intermediate Value Theorem we also know that if the function is continuous, then the function must take every value in between the two values and, in particular, ***it must be equal to 0 somewhere in the interval***.

Mathematically, if $f(x) \in C[a,b]$ and $f(a) \cdot f(b) < 0$, then there exists an $x^* \in [a,b]$ such that $f(x^*) = 0$. This leads us to the following general procedure:

1. Start with an interval $[a, b]$ such that $f(a) \cdot f(b) < 0$.
2. Cut the interval in half (bisect).
3. Evaluate the function f at the midpoint of the interval.
4. Choose whichever sub-interval contains a sign change.
5. Repeat as necessary.

Writing this in pseudocode might lead to something like the following:

Given a_0, b_0 s.t. $f(a_0)f(b_0) < 0$
for k=0, ..., maxiter **do**
  $mid \leftarrow (a_k + b_k)/2$
  if $f(a_k)f(mid) < 0$ **then**
    $a_{k+1} \leftarrow a_k; b_{k+1} \leftarrow mid$
  else
    $a_{k+1} \leftarrow mid; b_{k+1} \leftarrow b_k$
  end if
end for

Figure 7.2: Bisection Algorithm

Example 7.1. Consider $f(x) = x^3 + 4x^2 - 10 = 0$ on $[1, 2]$ where $x^* = 1.365$. We should first check that the function has opposite signs on the given interval:

$$\begin{aligned} f(1) &= 1 + 4 - 10 = -5 \\ f(2) &= 8 + 16 - 10 = 14 \\ f(1) \cdot f(2) &< 0 \qquad\qquad YES! \end{aligned}$$

1. Compute $p = \frac{a+b}{2} = \frac{1+2}{2} = 1.5$
2. Evaluate $f(1.5) = 3.375 + 9 - 10 = 2.375$
3. $f(a) \cdot f(p) = f(1) \cdot f(1.5) < 0$ so we choose left interval and set $b = p$
4. Repeat

In the book, after 13 iterations $p = 1.365112305$, $f(p) = -0.00194$, and $|b_{14} - a_{14}| = 0.000122070$.

Let's take a look at how this pseudocode could be implemented in python.

```
import numpy as np

def bisect(x0, a0, b0, ftol):
  ak = a0
  bk = b0
  converged = False

  k = 0

  while (not converged):
    k = k+1
    mid = (ak + bk)/2
    fmid = fx(mid)
    if ( fx(ak)*fmid < 0):
      bk = mid
    else:
      ak = mid
    if (np.abs(fmid) < ftol):
      converged = True
  else:
      print("Bisection converged after %d iterations at: \n x = %e with f(x) = %e" % (k, mid

  return mid
```

Let's define a function and call the bisection algorithm. For this example, let's use the function:

$$f(x) = (x - 1.5)^3 - x + 2$$

on the interval $I = [a, b] = [0, 2.5]$.

For initial values we'll use an initial starting guess of $x_0 = 2.0$ and a function tolerance of $ftol = 10^{-6}$

```
# Example function

def fx(x):
  fvalue = (x-1.5)**3 - x + 2
  return fvalue

#Initialize
```

```
a = 0.0
b = 2.5
x0 = 2.0
ftol = 1.e-6


# Check assumptions
fa  = fx(a)
print ("fa = %f" %(fa))
```

```
fa = -1.375000
```

```
fb  = fx(b)
print ("fb = %f" %(fb))
```

```
fb = 0.500000
```

```
# Call Bisection function
xstar = bisect(x0, a, b, ftol)
```

```
Bisection converged after 22 iterations at:
 x = 3.085119e-01 with f(x) = -8.565410e-07
```

7.2 Stopping (Convergence) Criteria

Before proceeding further, we should discuss when and how to terminate an iterative algorithm. This decision is one of great importance in real-world applications because many of the problems are expensive or time consuming. Both the expense and time are usually a result of the complexity of the function being evaluated. In some cases it could take hours if not days of computer time to yield one function evaluation. In these cases, it is not unusual that a scientist or an engineer will decide to terminate an algorithm based simply on how much computer time they are willing to use.

In practice, there are numerous possibilities for convergence criteria, each with pros and cons. The most obvious would be to check to see how close we are to our desired solution, but in general this would be impossible since we don't know what the solution is ahead of time (except for academic exercises). On the other hand, it is not unreasonable to assume that we might have some sort of bound, for example in some cases, the solution might be known to be positive or have a maximum value.

Suppose that an iterative algorithm has produced a sequence of iterates starting with an initial guess:

$$x_0,\ x_1,\ x_2,\ \ldots,\ x_k.$$

One logical approach is to take a look at the magnitude of $f(x_k)$ and check to see how close we are to zero:

$$|f(x_k)| < atol$$

Another frequent approach is to stop an iteration when "sufficient" progress has been made. In other words, an engineer is simply interested in reducing the initial function value by some fraction, i.e.

$$\frac{|f(x_k)|}{|f(x_0)|} < ftol$$

In general, these are good approaches, but there are cases for which progress towards the solution may be slow and little is to be gained from each new iteration. In this case, we may decide to stop if we believe we are not making sufficient progress towards a solution. This could be construed, for example, if the difference between successive iterates becomes small:

$$|x_{k+1} - x_k| < xtol$$

Here, we should note that it might also make sense to check that the relative step size is small:

$$\frac{|x_{k+1} - x_k|}{|x_{k+1}|} < rtol$$

In practice, many algorithms will employ some combination of these (and sometimes others). Experience and knowledge of the specific problem are usually needed to ensure that we don't stop too early or waste time iterating for too long.

7.3 Convergence of Bisection method

One of the first questions one should ask about any iterative method is when and under what conditions do we expect that it might converge to a solution. In the case of the bisection method, it turns out that the only condition we need to have is that the function $f(x)$ is continuous, given that the two initial points yield functions values with opposite signs.

The alert student should recognize that this is really just a consequence of Theorem 1.2 - the Intermediate Value Theorem.

Moreover, if these conditions hold, we can prove that the error bound between the solution and the iterates x_n can be given by:

$$|x^* - x_k| \leq \frac{b-a}{2^k}, \quad k = 0, 1, \ldots . \tag{7.1}$$

Notice that for $k = 0$ all we're really saying is that the root must lie within the given interval $[a_0, b_0]$. At the next step, the interval and hence the error is cut in half so that

$$|x^* - x_1| \leq \frac{b_0 - a_0}{2}.$$

At each iteration, the interval is cut in half by construction, so that at the kth iteration we get our desired result.

As constructed then, ***the bisection method cannot fail***. This type of method is known as a ***robust*** algorithm.

Additionally, we can use the error bound to estimate the number of iterations required to achieve a certain accuracy, ϵ.

Using Equation 7.1 we want

$$|x^* - x_k| \leq \frac{b-a}{2^k} \leq \epsilon.$$

Taking logs of both side we have

$$\log(b-a) - \log 2^k \leq \log \epsilon,$$

or rearranging

$$\log(b-a) - \log \epsilon \leq k \log 2.$$

If we would like to have the error be less than ϵ then solving for the number of iterations gives us

$$k \geq \frac{\log[(b-a)/\epsilon]}{\log 2}. \tag{7.2}$$

For example, let's set $\epsilon = 0.001$ and $a = 1, b = 2$. Then solving for k, we have

$$k \geq \frac{\log[1/0.001]}{\log 2} = \frac{3}{0.301} \approx 9.97,$$

so $k = 10$ iterations should suffice to achieve an error tolerance of $\epsilon = 0.001$.

7.4 Summary

To recap, bisection is a robust algorithm for finding the zero of a nonlinear function. It needs to have two initial points where the function value is of opposite sign, but it is guaranteed to converge. The advantages and disadvantages can be summarized as follows:

Table 7.1: **Bisection Method Summary**

Advantages	Disadvantages
Always converges, i.e. ***robust*** algorithm	Need to provide a specific interval, with 2 points where function is of opposite sign
Error bound easily derived and can be used to estimate number of iterations need to achieve a desired tolerance	Slow convergence - error only decreases by 1/2 at each iteration
Can be used to start other methods	Not clear how to generalize to higher dimensions

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

8 Newton's Method

8.1 Newton's Method

In the last lecture we saw that while the bisection method was robust and would always converge to a solution given the right set of initial values, it could exhibit slow convergence. As a result, most solvers used other types of root-finding algorithms. In this section we will study two such methods that can provide much faster convergence to a root.

Newton's method is likely the most popular and certainly the most powerful method for solving nonlinear equations (Meza 2011). The idea behind Newton's method is to use the slope of the function at the current iterate to compute a new iterate. Naturally, this requires that we first assume that the given function $f(x)$ is differentiable.

Idea

One approach for deriving Newton's method is to think about building a ***linear model of the function*** at the current iterate. Let's consider the linear model $m(x)$:

$$m(x) = f(x_0) + f'(x_0)(x - x_0). \tag{8.1}$$

Notice that at $x = x_0$ the model agrees with the function $f(x)$, in other words $m(x_0) = f(x_0)$. The idea is to then solve for the root of Equation 8.1 and use the root as the next guess of our iterative method:

Using this idea let's solve for the root of the linear model, i.e.

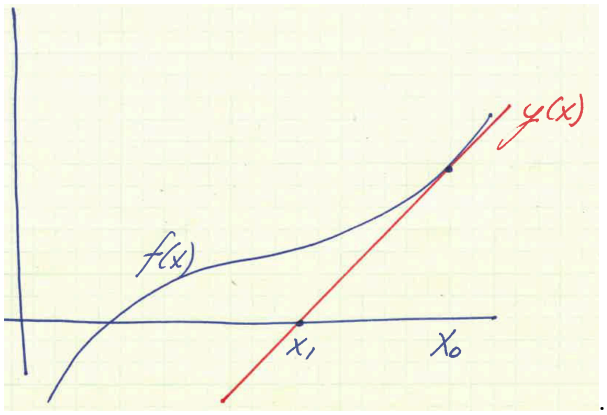
$$\begin{aligned} & f'(x_0)(x - x_0) + f(x_0) = 0, \\ \implies & x^\star = x_0 - \frac{f(x_0)}{f'(x_0)}. \end{aligned}$$

We can then set $x_1 = x^\star$ as the next iterate in our sequence and repeat the process. This gives us the general procedure for Newton's method:

Newton's Method

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad k = 0, 1, ... \tag{8.2}$$

We could have derived the same equation by just noting that we could take the derivative at the current iterate and use that to set up a linear equation, which we can solve for the new iterate.

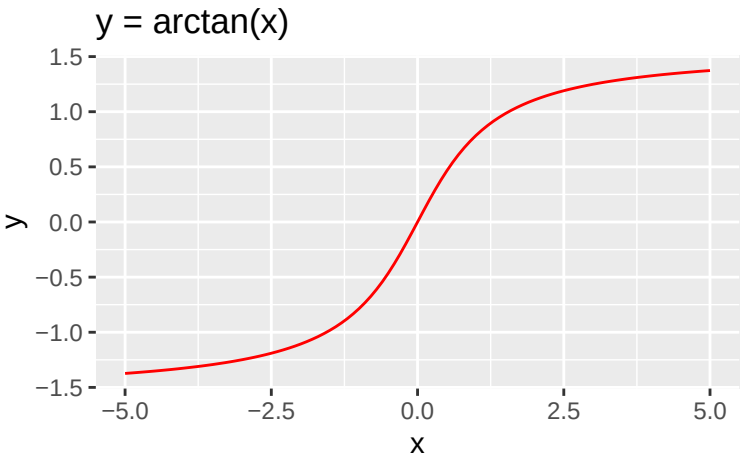


.

Finally, we will also note in passing that another derivation (which we went through in class) was to use Taylor's theorem to approximate our function $f(x)$ out to the first degree with a remainder term that included the second derivative. We then chose to ignore the second derivative term based on the argument that when we are near the solution the term would be small. When we solved for our new iterate, we derived the same equation for Newton's method.

Remark

A natural question to ask is under what conditions does Newton's method converge. In fact, it isn't hard to show that if the initial point x_0 is not chosen properly (i.e. close enough to a root), Newton's method will diverge. A typical example would be $y = \arctan(x)$, where if x_0 isn't close enough to the root the iterates quickly diverge to infinity.



Stopping criteria for Newton's method (and all of its variants) are similar to the ones discussed in the section on Bisection Method (Section 7.2.)

Example 8.1. Let $f(x) = x^6 - x - 1 = 0$ and let $x_0 = 1.5$. It is easy to verify that one root is given by $x^* = 1.134724$.

To use Newton's method we first need to calculate the derivative - $f'(x) = 6x^5 - 1$.

Using Equation 8.2 allows us to compute the $k+1$ iteration:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)},$$
$$x_{k+1} = x_k - \frac{x_k^6 - x_k - 1}{6x_k^5 - 1}.$$

Proceeding in the natural way from x_0, we can generate the following sequence of iterates:

k		x_k
0	1.5	1.5
1	$x_1 = x_0 - \frac{x_0^6 - x_0 - 1}{6x_0^5 - 1} = 1.5 - \frac{8.8906}{44.5625}$	1.3005
2	$x_2 = x_1 - \frac{x_1^6 - x_1 - 1}{6x_1^5 - 1} = 1.3005 - \frac{2.5373}{21.3197}$	1.1815
3	$x_3 = x_2 - \frac{x_2^6 - x_2 - 1}{6x_2^5 - 1} = 1.1815 - \frac{0.5387}{12.8140}$	1.1395

Notice that after only 3 iterations, the iterates is already correct to 3 significant digits.

Several questions one might consider at this point include:

- Under what conditions might we expect (local) convergence? Here by local we mean that the algorithm will converge if we start sufficiently close to a root. We will define this more carefully later.
- If Newton's method converges, how fast can we expect the convergence to be?

8.2 Error Analysis for Newton's Method

Let's consider the Taylor expansion about $x = x^*$.

$$0 = f(x_k) + (x^* - x_k)f'(x_k) + \frac{(x^* - x_k)^2}{2}f''(\xi).$$

Dividing by $f'(x_k)$ (we will assume for the time being that it's not equal to zero for any x_k) we get:

$$0 = \frac{f(x_k)}{f'(x_k)} + (x^* - x_k) + \frac{f''(\xi)}{f'(x_k)}\frac{(x^* - x_k)^2}{2}.$$

Using the equation for Newton's method we see that the first term is nothing but $x_k - x_{k+1}$ and substituting into the above equation we get:

$$0 = x_k - x_{k+1} + (x^* - x_k) + \frac{f''(\xi)}{f'(x_k)}\frac{(x^* - x_k)^2}{2}.$$

We see that the x_k terms cancel out. Rearranging to put the error on the left-hand side of the equation yields:

$$x^* - x_{k+1} = \frac{f''(\xi)}{2f'(x_k)}\ (x^* - x_k)^2. \tag{8.3}$$

The quantity on the left-hand side of the equation is just the error at the $k+1$ iteration, while the last term on the right-hand side is the error at the k iteration. As a result, we can interpret the equation to mean that the error at the $k+1$ iteration is proportional to the square of the error at the k iteration. This type of error bound is called ***quadratic convergence*** (see Definition 11.1).

> **Remark**
>
> If $f \in C^2[a, b]$ and $f'(x^*) = 0$, then Newton's method still converges but just not as rapidly. Consider for example $f(x) = x^4$, which has a root at $x = 0$, but where the first derivative is also equal to 0.

8.2.1 Summary for Newton's Method

Table 8.2: **Newton's Method Summary**

Advantages	Disadvantages
Doesn't require interval with function sign change	Need to have derivatives
Fast convergence rate – quadratic	May not converge from all starting points
Can generalize to higher dimension	Can be expensive (especially in higher dimensions)

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

9 Secant Method

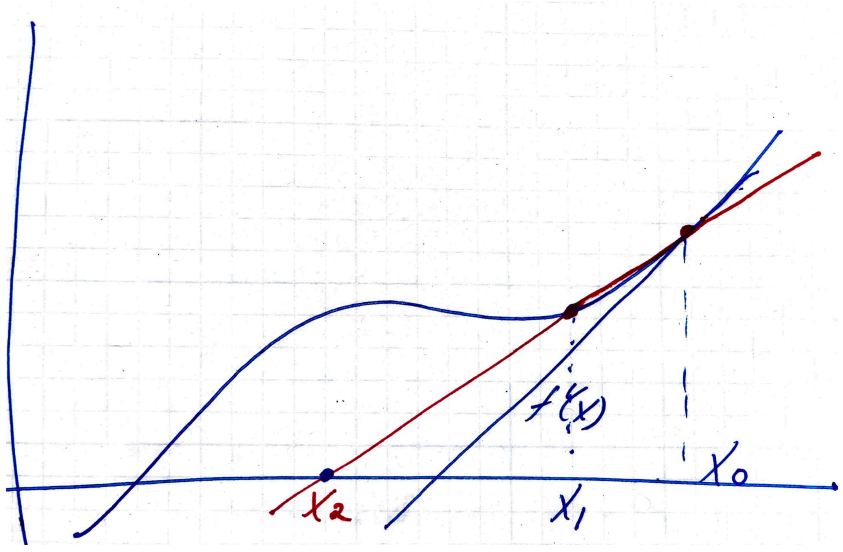
9.1 Secant Method

Recall that Newton's method uses the derivative of the function whose roots we seek. That is both its power and its main disadvantage. In many real-world problems, the derivative may be difficult to compute. In other cases, it could be expensive. And in the worst case, it may not even be available.

The secant method tries to address this disadvantage through an approximation to the derivative $f'(x)$ that uses two points close to each other, i.e. the secant. Using the secant, a new iterate is computed in a fashion similar to Newton's method.

> **Historical Note**
>
> The secant method is one of the oldest methods for solving nonlinear equations, and has an interesting history that can be traced back to the Rule of Double False Position described in the 18th-century BCE Egyptian Rhind Papyrus(Papakonstantinou 2009).

Consider a line through two points $(x_0, f(x_0))$ and $(x_1, f(x_1))$. Let x_2 be the x intercept of this line.



Then it follows that

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

But notice that $f(x_2) = 0$, which leads to

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

Rearranging and solving for x_2 yields

$$x_2 = x_1 - \left[\frac{(x_1 - x_0)}{f(x_1) - f(x_0)}\right] f(x_1)$$

which is used as the next guess in our sequence.

This then yields the form for the general ***secant method***:

Secant Method

$$x_{k+1} = x_k - \left[\frac{(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}\right] f(x_k), \quad k = 0, 1, ... \tag{9.1}$$

Remark

Another way to view this is to note that the term in the brackets $\frac{(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}$ approximates the derivative of a function (or rather in this case, the inverse). Therefore, one could interpret the secant method as just Newton's method with a finite difference approximation to the derivative.

9.1.1 Summary for Secant Method

Table 9.1: **Secant Method Summary**

Advantages	Disadvantages
Do not need to have derivatives	Need to provide 2 initial points.
Can have fast convergence (although not quadratic)	May not converge from all starting points
Generalizes to higher dimensions	Can be expensive in higher dimensions

Regula Falsi

Given that both bisection and secant method require two points, it may not be surprising to learn that the two methods can be combined into a new method, where the updated points in the secant method are chosen in a manner similar to bisection. This method goes by several names including the ***method of false position*** and ***regula falsi.***

9.2 Advanced: Root Finding in Higher Dimensions

Finding roots of nonlinear functions in dimensions higher than one has a long and rich history. So far of the methods that we have discussed: 1) bisection, 2) Newton's, and 3) Secant, only Newton's method has an obvious path forward. This section gives a brief overview on how one proceeds in the case of Newton's method, and also provides a more general iterative procedure that is used in many applications.

Recall that Newton's method is based on approximating the next iterate in the sequence of approximations by using the following equation:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad k = 0, 1, ...$$

First, let's rewrite the equation as follows:

$$x_{k+1} = x_k - f'(x_k)^{-1} f(x_k), \quad k = 0, 1, ...$$

Let's now assume that $F : \mathbb{R}^n \to \mathbb{R}^n$, where $n > 1$. We can still take the derivative of this function, following all the usual rules. In this case, it results in a matrix, which is called the ***Jacobian*** and is given by:

$$J(x_k) = F'(x_k) = \left(\frac{\partial f_i(x_k)}{\partial x_j} \right) \quad i, j = 1, ..., n.$$

Newton's method can then be written as:

$$x_{k+1} = x_k - J(x_k)^{-1} f(x_k), \quad k = 0, 1, ...$$

where the inverse is interpreted as *matrix inversion.*

> **Remark**
>
> It is a fundamental precept in numerical analysis that one rarely computes the inverse of a matrix. As such, the usual method for stating Newton's method in higher dimensions is as follows – at each iteration k solve for the step $s_k = (x_{k+1} - x_k)$ by solving the linear equation:
>
> $$J(x_k)s_k = F(x_k).$$

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

10 Fixed Point Method

10.1 Fixed Point

Methods for root finding are hard to analyze, especially if we include derivatives of f. Another approach is to reformulate the problem to allow for easier analysis using what is known as a fixed point formulation. This approach has a long history and is also known as functional iteration, Picard iteration, and successive substitution.

> **Note:**
>
> We will do everything in $\mathbb{R}^1$ but the ideas hold in higher dimensions with appropriate generalizations.

First let's start with a definition.

Definition 10.1. The point $x^* \in \mathbb{R}$ is said to be a ***fixed point*** of $g : \mathbb{R} \to \mathbb{R}$ if $g(x^*) = x^*$.

Using the definition it is easy to see that finding fixed points is equivalent to finding roots of an equation.

Theorem 10.1 (Fixed Point)**.** *Suppose x^* is not a zero of $g(x)$. Then x^* is a zero of $f(x)$ if and only if x^* is a fixed point of $g(x)$.*

Proof. If x^* is a zero of $f(x)$, i.e. $f(x^*) = 0$, then clearly $g(x^*) = x^*$. On the other hand, if $g(x^*) = x^*$ then

$$x^* = x^* + g(x^*)f(x^*)$$

from which it follows that

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

and therefore $f(x^*) = 0$ since we assumed that $g(x^*) \neq 0$ □

> **Note**
>
> Note that once we know how to find fixed points, we can find zeros of a function.

The general idea is that if we have an approximation to the fixed point x^*, then if we take $x_1 = g(x_0)$, our hope is that x_1 will be an even better approximation. There are many choices we can make for $g(x)$ for any given $f(x)$. And not too surprisingly, the choice of $g(x)$ will be extremely important in determining whether we produce a good algorithm or not.

In the rest of the section, we will look into some of the details of this approach and what conditions are needed to ensure that the iteration converges.

For now, let's state our algorithm as follows:

```
x = x0
for (i in 1:maxiter) {
  xplus <- g(x)
  if (g(xplus) < ftol) {break}
  x <- xplus
}
# Either we met the maxiter criteria or the ftol criteria
xsol <- xplus
```

Example 10.1. Let's suppose that the sqrt key on your calculator is broken and you need to compute $\sqrt{3}$. This is equivalent to finding a solution of the function $f(x) = x^2 - 3 = 0$.

First attempt:

$g(x) = 3/x, x_0 = 1$

Using the algorithm above we get the following sequence of iterates:

$$\begin{aligned} x_0 &= 1 \\ x_1 &= 3/x_0 = 3 \\ x_2 &= 3/x_1 = 1 \\ x_3 &= 3/x_2 = 3 \\ &\vdots \end{aligned}$$

Apparently, this is not a good choice!

Second attempt:

$g(x) = \frac{1}{2}(x + 3/x), x_0 = 1$

This time the sequence of iterates is:

$$
\begin{aligned}
x_0 &= 1 \\
x_1 &= \frac{1}{2}(x_0 + 3/x_0) = 2 \\
x_2 &= \frac{1}{2}(x_1 + 3/x_1) = 1.75 \\
x_3 &= \frac{1}{2}(x_2 + 3/x_2) = 1.73214 \\
x_4 &= \frac{1}{2}(x_3 + 3/x_3) = 1.7320508 \\
&\vdots
\end{aligned}
$$

Note that after only 4 iterations, our solution appears to be a great approximation to $\sqrt{3}$. Later on, we'll find out why this choice of $g(x)$ is a good choice.

Exercise 10.1 (In class exercise:)**.** Compute $\sqrt{5}$ using the same procedure as above with $g(x) = \frac{1}{3}(2x + 5/x^2),\ x_0 = 1$.

$$
\begin{aligned}
x_0 &= 1 \\
x_1 &= \frac{1}{3}(2x_0 + 5/x_0^2) = \\
x_2 &= \\
x_3 &= \\
x_4 &= \\
&\vdots
\end{aligned}
$$

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

11 Fixed Point Existence and Uniqueness

11.1 Existence and Uniqueness of Fixed Points

At this point, there are several questions that arise naturally:

1. Is there a fixed point x^* in the interval $[a, b]$?
2. If yes, is it unique?
3. Does a given fixed point iteration generate a sequence of iterates $\{x_k\} \to x^*$?
4. And if yes, how fast will the iterates converge to the fixed point?

We'll take each of these points in turn starting with (1) and (2) using the following theorem.

Theorem 11.1. *If $g \in C[a, b]$ and $g(x) \in [a, b]$ $\forall x \in [a, b]$, then there exists a fixed point $x^* \in [a, b]$.*

If in addition, $g'(x)$ exists on (a, b) and there exists a positive constant $k < 1$ with $|g'(x)| \leq k \quad \forall x \in (a, b)$ then there exists exactly 1 fixed point in $[a, b]$.

Proof. The first part of the theorem states the condition for the existence of a fixed point. The second part states the conditions necessary for uniqueness. Let's take them one at a time.

Existence.

There are two cases to consider. The first is if the fixed point is one of the endpoints of the interval $[a, b]$. The second case, is if the fixed point is in the interior.

If $g(a) = a$ or $g(b) = b$, then clearly the fixed point $x^* = a$ or b, so we're done.

If not, then if the fixed point exists it must lie in the interior, i.e. $g(a) \neq a$ and $g(b) \neq b$. By assumption $g(x)$ maps the interval $[a, b]$ onto itself, so we can deduce that

$$g(a) > a \quad \text{and} \quad g(b) < b.$$

Let's now consider the function

$$h(x) = g(x) - x$$

First note that $h(x)$ is continuous on $[a, b]$ since $g(x)$ is continuous. Next, it is easy to see that it also satisfies the conditions:

$$h(a) = g(a) - a > 0$$
$$h(b) = g(b) - b < 0$$

Now, by the Intermediate Value Theorem, there exists a point $x^* \in (a, b)$ such that $h(x^*) = 0$. But using the definition of $h(x)$ that means that

$$h(x^*) = 0 = g(x^*) - x^*$$
$$\implies g(x^*) = x^*$$

So x^* must be a fixed point in $[a, b]$.

Uniqueness To show that a unique fixed point exists we will have need of the second condition

$$|g'(x)| \leq k < 1$$

The proof will be by contradiction. Let's assume that we have two fixed points x^*, y^* and that $x^* \neq y^*$. Using the Mean Value Theorem we can say that:

$$\frac{g(x^*) - g(y^*)}{x^* - y^*} = g'(\xi) \quad \xi \in [x^*, y^*] \subset [a, b].$$

Let's now consider $|x^* - y^*|$:

$$|x^* - y^*| = |g(x^*) - g(y^*)| = |g'(\xi)| \cdot |x^* - y^*| < k \cdot |x^* - y^*|$$

The first equation is true by the definition of a fixed point, and the second from the equation above derived from the MVT. The final inequality is due to the assumption on the bound of the derivative of g. But since the constant $k < 0$, this reduces to:

$$|x^* - y^*| < |x^* - y^*|$$

which is a contradiction. Therefore, x^* must be unique. ■ □

11.2 More on convergence of sequences

The question of convergence is not simply a matter of deciding when one should stop an iterative method. When one has a choice of different algorithms to pick from, it would make sense to choose the one that is fastest, where fastest can be loosely defined to be the one that is likely to take the fewest number of iterations. Towards that end, let's discuss convergence in a bit more detail.

Definition 11.1. We say that the ***rate of convergence*** of x_k to x^* is of ***q-order*** $p > 0$ if

$$\lim_{k\to\infty} \frac{|x_{k+1} - x^*|}{|x_k - x^*|^p} < \infty$$

For the special case of $p = 1$ we ask instead that

$$\lim_{k\to\infty} \frac{|x_{k+1} - x^*|}{|x_k - x^*|} < 1$$

This is known as a ***linear rate*** of convergence.

In addition, if for some sequence $\{c_k\} \to 0$ we have

$$|x_{k+1} - x^*| \leq c_k \, |x_k - x^*|$$

then the sequence $\{x_k\}$ is said to converge ***q-superlinearly*** to x^*.

> **Remark**
>
> If $p = 2$, we say the rate of convergence is ***quadratic***. All other things being equal, the larger the value of p, the faster an algorithm will converge to a solution. In other words, a quadratic rate of convergence is superior to a linear rate of convergence. Superlinear rate of convergence lies between quadratic and linear.

11.2.1 Examples.

1. $x_k = a^k, \quad 0 < a < 1$
2. $x_k = a^{2^k}, \quad 0 < a < 1$
3. $x_k = \left(\frac{1}{k}\right)^k$

11.2.2 Solutions:

(1) linear, (2) quadratic, (3) superlinear.

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

12 Fixed Point Iteration Convergence (Part 1)

12.1 Fixed Point Iteration Convergence (Part 1)

Now that we've answered questions (1) and (2) on the existence and uniqueness of fixed points, we will turn our attention to the last two questions dealing with the convergence of the fixed point iteration itself.

3. Does a given fixed point iteration generate a sequence of iterates $\{x_k\} \to x^*$?
4. And if yes, how fast will the iterates converge to the fixed point?

Let's first recall what a fixed point iteration looks like:

```
x = x0
for (i in 1:maxiter) {
  xplus <- g(x)
  if (g(xplus) < ftol) {break}
  x <- xplus
}
# Either we met the maxiter criteria or the ftol criteria
xsol <- xplus
```

In order to better understand the possible cases, let's take a look at a couple of pictures.

> **Note**
>
> ***To do: Insert pictures here:***

What the third case appears to show is that if the slope of the function $g(x)$ is too large, then the iteration might not converge. Another way to think about this is that the derivative must be bounded in some way for the fixed point iteration to converge.

We can in fact prove this rigorously. Your textbook proves convergence by making an assumption on the derivative of $g(x)$ and specifically that there is a constant $k < 1$ such that:

$$|g'(x)| \leq k \quad \forall x \in [a, b]$$

Here, we will use another concept that will prove useful in later lectures. It also has the additional benefit of not requiring the assumption that the derivative of $g(x)$ exists.

Definition 12.1. A function $g : \mathbb{R} \to \mathbb{R}$ is said to be ***Lipschitz continuous*** if

$$|g(x) - g(y)| \leq L\ |x - y| \quad \forall x \in [a, b].$$

L is called the ***Lipschitz constant.***

As it turns out there is a close relation of the Lipschitz constant to the derivative. Recall that by the MVT

$$g(x) - g(y) = g'(\xi)(x - y) \quad \xi \in (x, y) \subset [a, b].$$

Taking absolute values of both sides we get:

$$\big|g(x) - g(y)\big| = \big|g'(\xi)\big| \cdot \big|x - y\big| \quad \xi \in (x, y) \subset [a, b].$$

Now, if we let

$$L = \max_{\xi \in [a,b]} \big|g'(\xi)\big|,$$

it follows that:

$$\big|g(x) - g(y)\big| \leq L \cdot \big|x - y\big| \quad x \in [a, b].$$

Intuitively this makes sense as bounding the derivative is similar to bounding the change in the function values.

To begin our proof, we will need one final definition.

Definition 12.2. We say that $g : \mathbb{R} \to \mathbb{R}$ is a ***contraction mapping*** if

$$|g(x) - g(y)| \leq L\ |x - y| \quad \forall x \in [a, b].$$

with $L < 1$.

12.1.1 Example:

$$f(x) = e^x - 2x - 1 \quad \text{on } [1, 2] g(x) = \ln(2x + 1)$$

12.1.2 Solution:

Outline:

1. Compute g'
2. Note that g' is monotonically decreasing on given interval
3. Show bound on g', and in particular that the bound is < 1.

We're now ready to show our first convergence theorem for fixed point iterations.

Theorem 12.1 (Fixed Point Convergence)**.** *Suppose $g : \mathbb{R} \to \mathbb{R}$, with $g(x) \in [a, b] \quad \forall x \in [a, b]$ be a continuous contraction mapping. Then there exists a unique fixed point x^* with $g(x^*) = x^*$ and the fixed point iteration converges to x^* for any starting point $x_0 \in [a, b]$.*

Proof. Existence and uniqueness of a fixed point was proven earlier. To prove convergence of the fixed point iteration we need to show that $\{x_k\} \to x^*$ for the fixed point iteration:

$$x_{k+1} = g(x_k)$$

Let's consider:

$$x_k - x^* = g(x_{k-1}) - g(x^*)$$

Taking absolute values and using the fact that g is a contraction mapping we can write

$$\left|x_k - x^*\right| = \left|g(x_{k-1}) - g(x^*)\right| \leq L \left|x_{k-1} - x^*\right|.$$

Now we apply this inductively:

$$\left|x_k - x^*\right| \leq L^k \left|x_0 - x^*\right|.$$

But since $L < 1$ we know that $L^k \to 0, k \to \infty$ and therefore

$$\left\|x_k - x^*\right\| \to 0, k \to \infty$$

as we set out to show. ■ □

Remark: As it turns out, while a contraction mapping is a useful tool, it is often hard to verify in a real-world application. Instead what is usually assumed is that either the function is Lipschitz continuous in a neighborhood of the fixed point x^*, or that the function has a bounded derivative at the root.

Another thing to keep in mind is that it might be unrealistic to assume a contraction mapping property for all $x \in [a, b]$. An alternative is to assume that we have some property that holds at or near the solution. In this case, then we have a different type of convergence.

Definition 12.3. The iterative process $x_{k+1} = g(x_k)$ is said to be ***locally convergent*** to the fixed point x^* if there exists $\delta > 0$ such that $x_k \to x^*$ for any x_0 satisfying $|x^* - x_0| \leq \delta$.

Perhaps the simplest interpretation of this definition is to say, if we start ***close enough*** to a fixed point, then the method will converge to it. What defines close enough, will depend not only on the initial point, but also on the function we use.

With this background, we can state the following theorem.

Theorem 12.2. *Suppose $g : \mathbb{R} \to \mathbb{R}$ is continuously differentiable and:*

1. $x^* = g(x^*)$
2. $|g'(x^*)| < 1$

Then the iterative process $x_{k+1} = g(x_k)$ is **locally convergent** *to x^*.*

Remark: Note the difference here that we are only asking for a bound on g' at a single point, namely the fixed point, x^*.

Proof. The proof follows much the same as before, but now we have to also show that $|g'(x)| < 1$ for all x in a neighborhood of x^*. Since we are assuming that g is continuously differentiable, this is fairly easy to show. □

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

13 Fixed Point Iteration Convergence (Part 2)

13.1 Fixed Point Iteration Convergence (Part 2)

Theorem 13.1. *Suppose $g : \mathbb{R} \to \mathbb{R}$ is twice continuously differentiable and the iterative process $x_{k+1} = g(x_k)$ is locally convergent.*

Then the convergence rate is linear if

$$|g'(x^*)| \neq 0$$

and the convergence rate is at least quadratic if

$$|g'(x^*)| = 0.$$

Proof. We've already shown that the fixed point iteration converges (Theorem 12.1). To show that we have linear convergence, we can either show that we can bound the derivative in a neighborhood of the fixed point or we can use the Lipschitz continuity condition, to show that the error at any iteration is bounded by $L < 0$, which gives us a linear rate of convergence.

To show that the convergence rate is at least quadratic, let's first expand $g(x)$ In a Taylor polynomial about the point x^*.

$$g(x) = g(x^*) + g'(x^*)(x - x^*) + \frac{g''(\xi)}{2}(x - x^*)^2. \tag{13.1}$$

By assumption we know that:

$$g(x^*) = x^* \text{ and } g'(x^*) = 0.$$

Substituting into (Equation 13.1), we get:

$$g(x) = x^* + 0 + \frac{g''(\xi)}{2}(x - x^*)^2.$$

Setting $x = x_k$, we have:

$$g(x_k) = x^* + \frac{g''(\xi_k)}{2}(x_k - x^*)^2,$$

with ξ_k between x_k and x^*.

Using the definition of the fixed point iteration: $x_{k+1} = g(x_k)$, we have

$$x_{k+1} = x^* + \frac{g''(\xi_k)}{2}(x_k - x^*)^2,$$

Rearranging we have:

$$\frac{x_{k+1} - x^*}{(x_k - x^*)^2} = \frac{g''(\xi_k)}{2},$$

Finally, by assumption, the fixed point iteration is locally convergent which means that

$$\{x_k\} \to x^* \implies \{\xi_k\} \to x^*$$

since ξ_k is between x_k and x^*. Therefore

$$\frac{x_{k+1} - x^*}{(x_k - x^*)^2} = \frac{g''(x^*)}{2} = C < \infty,$$

which proves that the iterates are converging q-quadratically. □

Remark: This theorem shows us that if we want quadratic convergence, we should look for fixed point methods such that $g'(x^*) = 0$.

Let's consider the general form

$$g(x) = x - \phi(x)f(x) \tag{13.2}$$

where $\phi(x)$ is differentiable and to be chosen later. Taking derivatives we have:

$$g'(x) = 1 - \phi'(x)f(x) - f'(x)\phi(x).$$

Letting $x = x^*$, such that $f(x^*) = 0$, we get:

$$g'(x^*) = 1 - f'(x^*)\phi(x^*).$$

By our theorem, if we want quadratic convergence we need to have $g'(x^*) = 0$, hence

$$0 = 1 - f'(x^*)\phi(x^*),$$

and solving for ϕ we get the result:

$$\phi(x^*) = \frac{1}{f'(x^*)}.$$

Substituting back into our general form (Equation 13.2) we have:

$$g(x) = x - \frac{f(x)}{f'(x)}.$$

But that's just Newton's method!

As a result, an immediate corollary to (Theorem 13.1) is that Newton's method is quadratically convergent.

Example 13.1. Recall our earlier example (Example 10.1) where we wanted to find the root of $x^2 - 3$, that led us to try different fixed point iterations.

1. $g(x) = 3/x, \implies g'(x) = -3/x^2.$

$$g'(x^*) = g'(\sqrt{3}) = -1 \quad \implies \quad \text{No convergence}$$

2. $g(x) = \frac{1}{2}(x + 3/x), \implies g'(x) = \frac{1}{2}(1 - 3/x^2).$

$$g'(x^*) = g'(\sqrt{3}) = 0 \quad \implies \quad \text{q-quadratic convergence}$$

3. $g(x) = x - \frac{x^2-3}{7}, \implies g'(x) = 1 - \frac{1}{7}(2x).$

$$g'(x^*) = g'(\sqrt{3}) = 0.505 \quad \implies \quad \text{Linear convergence}$$

13.2 Final note on convergence

Sometimes it is useful to estimate the number of iterations required to achieve a certain reduction in the error. We were able to do this in the case of the bisection method as each iteration reduced the interval in half, so therefore it was easy to compute the number of iterations needed to reduce the error by a certain factor (Equation 7.2).

We can do something similar for other methods, although it can be a bit trickier. Suppose that we know that we have q-linear convergence with a constant λ, i.e.

$$\lim_{k\to\infty} \frac{|x_{k+1} - x^*|}{|x_k - x^*|} = \lambda < 1$$

One way to approach this is to consider our proof of convergence for the fixed point iteration (Theorem 12.1). We were able to show that

$$\left|x_k - x^*\right| \leq L^k \left|x_0 - x^*\right|.$$

Suppose we would like to reduce the error by a factor of 10 after k iterations. Setting $\lambda = L$, this means that we want to have:

$$\lambda^k = 10^{-1}$$

Taking logs of both sides we have:

$$k \ \log_{10} \lambda = -1$$

or equivalently

$$k = -\frac{1}{\log_{10} \lambda}$$

The value of k can be interpreted to mean that it approximates the number of iterations required to reduce the error by a factor of 10.

Example 13.2. Suppose that we knew that a certain fixed point iteration g had a Lipschitz constant of $L = 2/3$. Then setting $\lambda = L$.

$$k = -\frac{1}{\log_{10}(2/3)} \approx \frac{1}{0.176} = 5.68$$

which we can interpret to mean that it would take approximately $k = 6$ iterations to reduce the error by a factor of 10.

13.3 Comparison of nonlinear equation methods

Let's take a step back and summarize our main results:

Table 13.1: Comparison of Different Solution Methods for Finding Roots of an Equation

Method	Assumptions	Advantages	Disadvantages
Bisection	f is continuous; requires 2 starting points where function has opposite sign	Robust; easy to implement	Linear convergence
Newton	f is continuously differentiable	Fast convergence	Requires derivatives

Method	Assumptions	Advantages	Disadvantages
Secant	f is continuous	Doesn't require derivatives; superlinear convergence	Might encounter cancellation error
Fixed Point	Need to have a good $g(x)$	Easy to implement	Linear convergence

We have not yet talked about the stability of the algorithms, but this will prove to be an important characteristic of any method we choose. More on this later.

Corollary 13.1. *Consider $f : \mathbb{R} \to \mathbb{R}$, twice continuously differentiable. Suppose:*

1. $f(x^*) = 0$
2. $|f'(x^*)| \neq 0$

Then Newton's method is locally convergent to x^*.

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

Part III

Interpolation and Approximation

Introduction

Wherein we cover methods for interpolation, approximation, and extrapolation. We will be following your textbook covering mostly sections 3.1, 3.3, and 3.5. Time permitting we may also some material on Chebyshev polynomials and least squares approximations.

Concepts Covered

- Approximation versus interpolation
- Basis Functions
- Construction and evaluation of interpolants
- **Theorem** - Polynomial Interpolant Uniqueness
- Lagrange Polynomials
- Divided Differences
- **Theorem** - Divided differences and derivatives
- **Theorem** - Polynomial Interpolant Error
- Piecewise Interpolation
- Cubic Splines

14 Polynomial Interpolation

Lecture 14: October 17, 2023 Note: October 19 was taken for review of Midterm makeup quiz

14.1 Interpolation and Extrapolation

14.1.1 Motivation

Although interpolation is usually not an end product in numerical analysis, it is used in many other techniques that we will study, so having a good understanding of the concepts in approximating functions or interpolating some given data will prove useful later.

Let's consider approximation first. There are two basic problems: 1) data fitting, and 2) approximating functions.

14.1.2 Data fitting

For ease, we will only consider the case of $\mathbb{R}^1$ here, although (as before) many of the ideas translate into higher dimensions. Suppose we are given a set of data points

$$\{(x_i, y_i)\}_{i=0}^{n}.$$

Our goal is to find a function $v(x)$ that fits the data in some yet to be determined way. As a quick aside, the points x_i are sometimes called the ***node points*** or simply just ***nodes***.

If the data is believed to be accurate, we could also insist that $v(x)$ match the data exactly, i.e. that

$$v(x_i) = y_i \quad i = 0, 1, ... n.$$

If this is the case, then we say that $v(x)$ ***interpolates*** the data.

This still leaves open the question of what we mean by *fit*, and also what might constitute a good function, i.e. what properties do we want in a function that fits the data.

The second area we will study is that of approximating a function. This situation might arise if we had a complicated or computationally expensive function. Our goal here is to find a

simpler or cheaper function that we could use to approximate our expensive function. The techniques will be similar to the earlier case, but with the difference that here, we might be able to choose the points ourselves.

One typical use for ***approximating*** functions widely used in practice is to predict the value of the function at points other than those given (or chosen). If the point at which we want to predict the function value is inside the interval set by the data points, then we call it ***interpolation***. If the point is chosen outside the interval, then we say it is ***extrapolation***.

Another use for approximating functions arises in the context of other numerical methods. The most common example is in helping us to take derivatives or compute integrals of other functions.

14.2 General Representation

We will first consider the general case of a linear form, in which we will write the approximating (interpolating) function as:

$$v(x) = \sum_{j=0}^{n} c_j \phi_j(x)$$

Here we call $\{c_j\}_{j=0}^n$ the unknown coefficients, and $\{\phi_j\}_{j=0}^n$ the ***basis functions***.

We will further assume that $\{\phi_j\}_{j=0}^n$ are linearly independent.

> **i** Linear form
>
> When we say that we will take the linear form, we mean that $v(x)$ is written as a linear combination of the basis functions and not that $v(x)$ is a linear function of x.

Example 14.1. Examples of basis functions

1. Polynomial (monomial): $\phi_j(x) = x^j \quad j = 0, 1, \ldots n$
2. Trigonometric: $\phi_j(x) = \cos(jx) \quad j = 0, 1, \ldots n$

There are many other forms that can be chosen, most of which are used for specific applications or problems with a special structure. For now we will restrict ourselves to polynomial interpolants. Some of the reasons that polynomials are a good first choice include:

- Easy to both construct and evaluate the interpolants.
- Easy to sum and multiply polynomials

- Easy to differentiate and integrate
- And despite their simple appearance, they can fit many different types of data.

Before we go to the next section, we should also say that there are two main stages when using polynomial interpolation.

1. ***Constructing an interpolant.*** This usually entails computing the unknown coefficients given a particular set of data points. This can be expensive, but it is done only once for a given data set.
2. ***Evaluating an interpolant.*** Once the polynomial is constructed, we are then able to evaluate the polynomial at some point or some set of points. This is typically much less expensive per evaluation, but we may want to do it many times.

14.3 Polynomial (Monomial) Interpolation

Polynomials of one form or another are ubiquitous in numerical analysis. We will see that they can be used to approximate data, they are used to approximate derivatives in other contexts, and even to help us evaluate integrals numerically. Some of the uses of polynomials include approximating commonly used functions and in computer graphics to smooth curves and surfaces of geometric objects.

Let's recall that we can write a polynomial as:

$$\begin{aligned} p(x) &= \sum_{j=0}^{n} c_j x^j, \\ &= c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n, \end{aligned} \tag{14.1}$$

for a given set of $n+1$ data points:

$$(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n).$$

Our goal is to find the $n+1$ coefficients $c_0, c_1, \ldots, c_n$, such that we satisfy the interpolating conditions:

$$p(x_i) = y_i, \quad i = 0, 1, \ldots, n.$$

> **Note**
>
> **Reminder:** a polynomial of degree n has $n+1$ coefficients.

It will also be necessary to assume that the data points are distinct, i.e.

$$x_i \neq x_j \quad \forall \quad i \neq j.$$

Example 14.2 (Construction of Interpolating Polynomial)**.** Suppose we are given the data $(x_0,y_0) = (1,1), (x_1, y_1) = (2,3),(x_2, y_2) = (4,3), $ and we wish to interpolate the first two data points using a first degree polynomial $n = 1$. In other words we want to construct the interpolating linear polynomial

$$p_1(x) = c_0 + c_1 x. \tag{14.2}$$

The interpolating conditions give us:

$$\begin{aligned} p_1(x_0) &= c_0 + 1 \cdot c_1 = 1 \quad (= y_0), \\ p_1(x_1) &= c_0 + 2 \cdot c_1 = 3 \quad (= y_1). \end{aligned}$$

This is a set of two equations in two unknowns, which you can easily verify has the solution:

$$c_0 = -1 \quad c_1 = 2.$$

These coefficients can then be substituted into our polynomial form given by Equation 14.2 to yield the desired polynomial interpolant:

$$p_1(x) = 2x - 1.$$

As we mentioned earlier, now that we have constructed the polynomial, we are free to evaluate it at any number of other points.

Remark: An alternate derivation of this form can be found in the supplemental section at the end of this section, see Section 14.4.2.

In class exercise:

Construct interpolating polynomial of degree $n = 2$, using the additional third point given above $(x_2, y_2) = (4, 3)$.

Solution:

$$p_2(x) = \frac{1}{3}(-2x^2 + 12x - 7). \tag{14.3}$$

14.3.1 Vandermonde Matrices

In solving the exercise problem above you will have noticed that you had to set up a set of equations that looked like:

$$\begin{aligned} p_2(x_0) = c_0 + c_1 x_0 + c_2 x_0^2 &= y_0 \\ p_2(x_1) = c_0 + c_1 x_1 + c_2 x_2^2 &= y_1 \\ p_2(x_2) = c_0 + c_1 x_1 + c_2 x_2^2 &= y_2. \end{aligned}$$

These can be written more concisely in matrix notation as:

$$\begin{bmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix}. \tag{14.4}$$

or simply as

$$Xc = y.$$

This system of linear equations can now be solved for the coefficients c_0, c_1, c_2, which determine our interpolating polynomial. The matrix X in Equation 14.4 is an example of a ***Vandermonde*** matrix. It can be shown that the $\det X \neq 0$, as long as the given data points are distinct. This implies that X is nonsingular and hence the linear system has a unique solution.

It is relatively easy to show that the general form of the Vandermone matrix for $n+1$ data points is given by:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & x_2 & x_2^2 & \cdots & x_n^n \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \cdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \cdots \\ y_n \end{bmatrix}. \tag{14.5}$$

This observation leads to the following theorem.

Theorem 14.1 (Polynomial Interpolant Uniqueness)**.** *For any real data points $\{(x_i, y_i)\}_{i=0}^n$ such that x_i are distinct there exists a unique polynomial $p(x)$ of degree at most n that satisfies the interpolating conditions $p(x_i) = y_1$ for $i = 0, 1, \ldots, n$.*

14.3.2 Summary

Let's quickly summarize the use of this particular type of basis functions (monomials) as an interpolating function.

- We introduced the concepts of approximation and interpolation - fitting a given set of points with a function, with interpolation being a special case of approximation.
- Considered the general linear form for construction our approximating function and looked at our first case using polynomials and specifically monomials, i.e. $\phi_j(x) = x^j$.
- This led to the construction of the Vandermonde matrix, which is extremely easy to set up..
- The coefficients can be computed by solving one system of linear equations, which can be done in $O(n^3)$ operations, and the evaluation of the interpolating polynomials can be done in $O(n)$ operations.
- A big disadvantage is that the Vandermonde matrices are notoriously ill-conditioned for even medium dimensions. For $n = 10, \kappa(A) \approx 10^{10}$. As a result, using this approach is highly discouraged. (Notwithstanding this comment, for small n, this approach might be useful.)
- The resulting coefficients don't have a clear relation to the y_i values, which is sometimes desired. This will come up when we study numerical differentiation and integration.

14.4 Supplemental Materials

14.4.1 Weierstrass Approximation

A natural question to ask is if or when a general function $f(x)$ can be approximated by a polynomial. This was answered by Weierstrass (1885) in a theorem bearing his name.

Theorem 14.2 (Weierstrass Approximation)**.** *Suppose $f(x)$ is defined and continuous on a closed interval $[a, b]$, and $\epsilon > 0$ is given. Then there exists a polynomial $p(x)$ on $[a, b]$ such that*

$$|f(x) - p(x)| < \epsilon \quad \forall x \in [a, b].$$

The most common way of stating this is that any continuous function on a closed interval can be approximated with arbitrary precision by a polynomial.

14.4.2 Alternate derivation of linear interpolation using monomial basis functions

Let's begin with some simple cases - in particular the linear case. Suppose we are given two points (x_0, y_0) and (x_1, y_1), and $x_0 \neq x_1$. Then the straight line passing through these two points is given by the linear polynomial:

$$p_1(x) = \frac{(x_1 - x)y_0 + (x - x_0)y_1}{x_1 - x_0}. \tag{14.6}$$

Let $x_0, x_1, \ldots, x_n$ be $n+1$ distinct points on some interval $[a, b]$.

We say that $p_1(x)$ ***interpolates*** the value y_i at the points $x_i, i = 0, 1$ if

$$p_1(x_i) = y_i \quad i = 0, 1.$$

It isn't too hard to see that Equation 14.6 satisfies this property by construction. Specifically note that:

$$\begin{aligned} p_1(x_1) &= \frac{(x_1 - x_1)y_0 + (x_1 - x_0)y_1}{x_1 - x_0} \\ &= \frac{(0)y_0 + (x_1 - x_0)y_1}{x_1 - x_0} \\ &= \frac{(x_1 - x_0)}{x_1 - x_0} y_1 \\ &= y_1 \end{aligned}$$

You can use a similar argument to show that $p_1(x_0) = y_0$.

14.4.3 Computational Tip

> **Shifted Power Form**
>
> While the form given by Equation 14.1 is the most convenient for analysis, it can lead to numerical errors and we will often see instead the *shifted power form*:
>
> $$p(x) = c_0 + c_1(x - a) + c_2(x - a)^2 + \cdots + c_n(x - a)^n,$$

14.5 References

1. Interpolation and Approximation. (Davis 1975)

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

15 Lagrange Polynomial Interpolation

Lecture 15: October 24, 2023 Note: October 19 was taken for review of Midterm makeup quiz

15.1 Lagrange Polynomials

As a result of the advantages and disadvantages listed above, other approaches have been developed that attempt to address the disadvantages. One such approach for constructing an interpolating polynomial consists of using what are known as the Lagrange polynomials.

Proceeding as before, let's try to construct a polynomial using the general linear form using the following equation:

$$\begin{aligned} p(x) = p_n(x) &= \sum_{j=0}^{n} y_j L_j(x), \\ &= y_0 L_0(x) + y_1 L_1(x) + y_2 L_2(x) + \cdots + y_n L_n(x), \end{aligned} \tag{15.1}$$

where $L_j(x)$ are called ***Lagrange polynomials*** with the following properties:

$$L_j(x_i) = \begin{cases} 0, & i \neq j \\ 1, & i = j \end{cases} \quad j = 0, 1, \ldots, n. \tag{15.2}$$

Using this definition, it isn't hard to show that

$$p_n(x_i) = y_i, \quad i = 0, 1, 2, \ldots, n.$$

In other words, $p_n(x)$ is an interpolating polynomial.

> **Kronecker notation**
>
> You may also see the Lagrange polynomials written in the more concise notation:
>
> $$L_j(x_i) = \delta_{ij}$$
>
> where δ_{ij} denotes the Kronecker δ function. In simple terms, the Lagrange polynomials

are equal to 1 at their corresponding node point. For example, $L_0(x_0) = 1$ and $L_0(x_i) = 0$ at all of the other node points not equal to x_0. (see Figure 15.1).

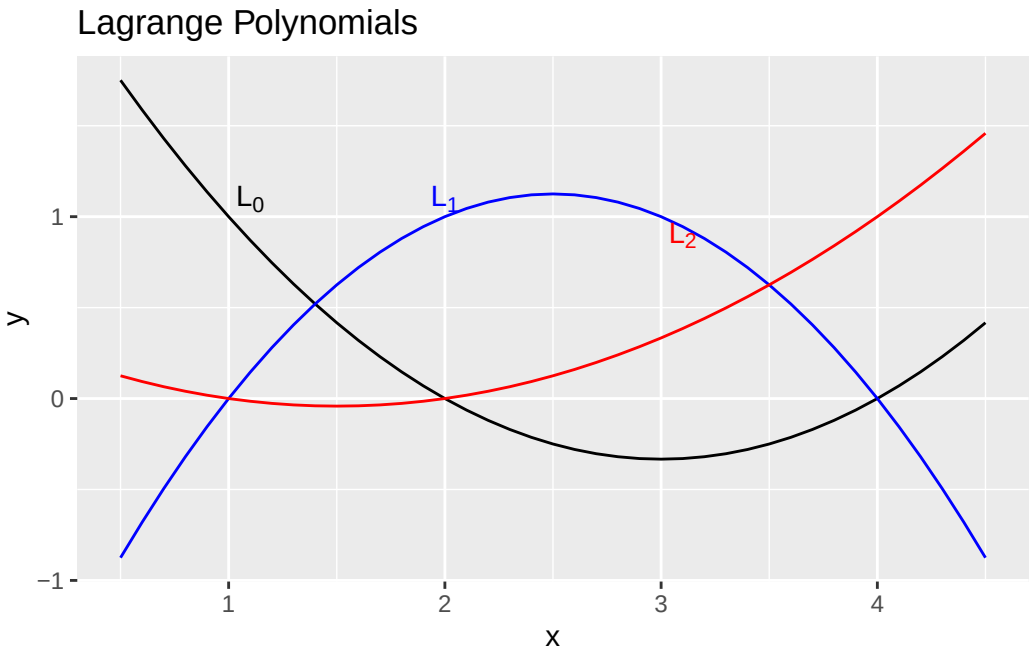
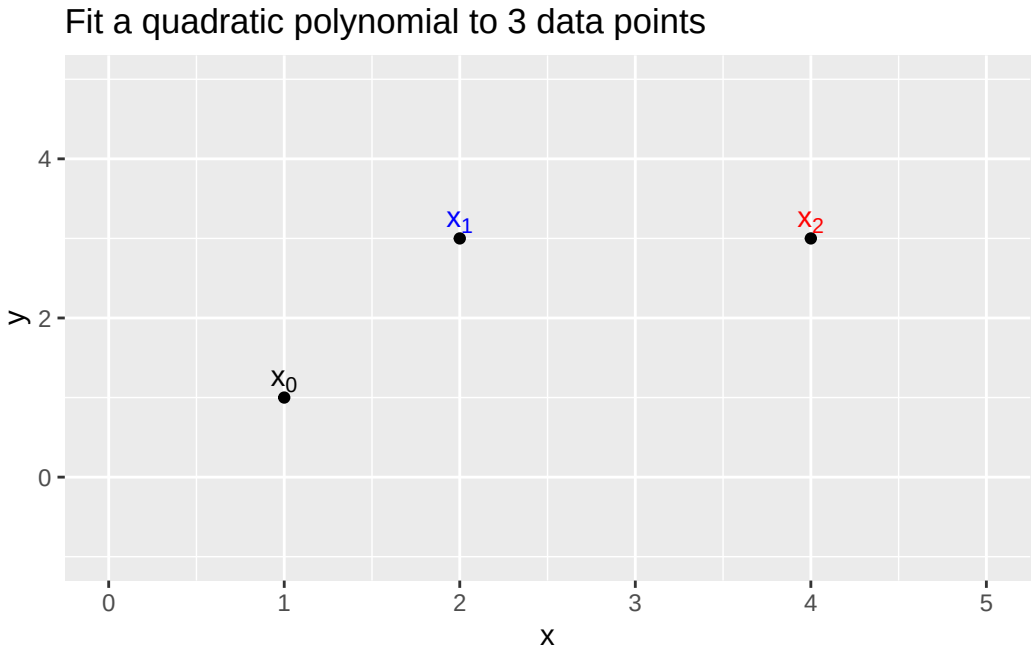


Figure 15.1: Lagrange Coefficient Polynomials

Example 15.1. Construct $p_2(x)$ for data points $(1, 1), (2, 3), (4, 3)$ using the Lagrange polynomial formulation.



Solution. The first step Is to compute the individual $L_j(x)$ for the given data points $(1,1),(2,3),(4,3)$.

Let's start with the first Lagrange polynomial $L_0(x)$. By definition (Equation 15.2), we know that the polynomial must satisfy the following conditions for $j = 0$.

$$L_0(x_i) = \begin{cases} 0, & i \neq 0 \\ 1, & i = 0 \end{cases}.$$

We also know that L_0 must be a quadratic polynomial (***Why?***) As a result, we know that the general form for L_0 must be:

$$L_0(x) = a \cdot (x - r_1)(x - r_2),$$

where r_1, r_2 are the two roots. To satisfy condition $L_0(x) = 0$ at the two nodes other than x_0, means that the two other nodes, $x_1 = 2, x_2 = 4$ must be the two roots of the quadratic.

$$L_0(x) = a \cdot (x - 2)(x - 4).$$

We can now use the remaining condition $L_0(x_0) = 1$ to see that:

$$\begin{aligned} L_0(x_0) = 1 &= a \cdot (x_0 - 2)(x_0 - 4), \\ &= a \cdot (1 - 2)(1 - 4) \\ &= 3a \end{aligned}$$

Therefore $a = 1/3$ giving us the solution:

$$L_0(x) = \frac{1}{3}(x-2)(x-4)$$

Using the same procedure, we can compute:

$$L_1(x) = -\frac{1}{2}(x-1)(x-4).$$

and finally:

$$L_2(x) = \frac{1}{6}(x-1)(x-2).$$

I leave the computation of these two for you to practice on.

Using (Equation 15.1), we can now combine the 3 Lagrange polynomials and multiply by the corresponding y_j values to gives us the desired interpolating polynomial :

$$\begin{aligned} p_2(x) = \frac{y_0}{3}(x-2)(x-4) - \frac{y_1}{2}(x-1)(x-4) \\ + \frac{y_2}{6}(x-1)(x-2) \end{aligned}$$

Substituting for the values of y we get:

$$\begin{aligned} p_2(x) = \frac{1}{3}(x-2)(x-4) - \frac{3}{2}(x-1)(x-4) \\ + \frac{3}{6}(x-1)(x-2) \end{aligned} \tag{15.3}$$

Figure 15.2 shows all three of the Lagrange polynomials in addition to the final interpolating polynomial $p_2(x)$ of degree 2 for the 3 given data points.

! Uniqueness (again)

One should note at this point that while the two approaches appear different, they yield the same second degree interpolating polynomial. In other words Equation 14.3 = Equation 15.3 .

Remark: It can be easily shown that there is only one quadratic interpolating polynomial for any three (distinct) points.

Proof. Suppose there were 2 polynomials $p_2(x), q_2(x)$ that interpolate the given points. Then define

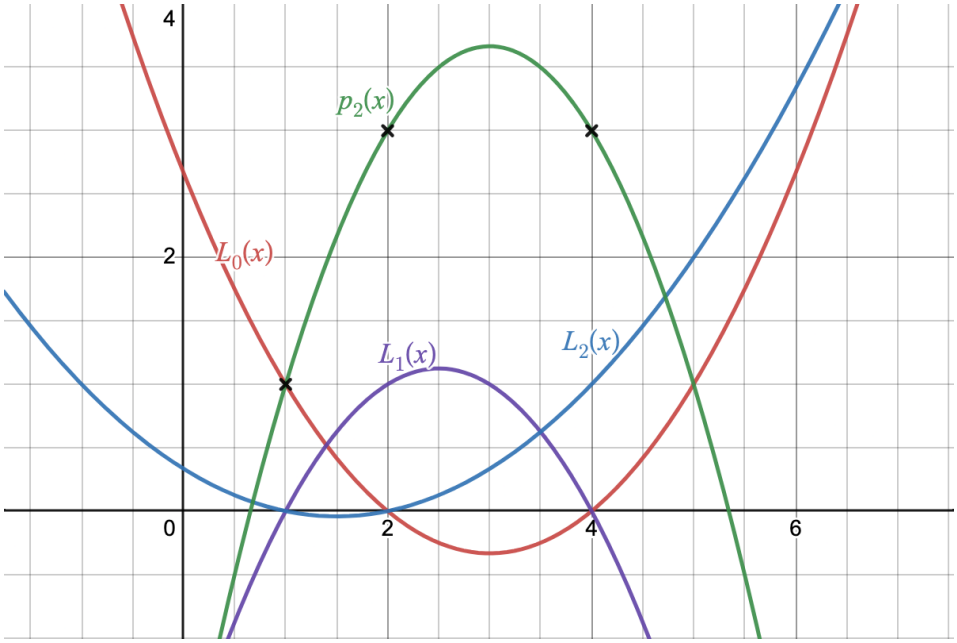


Figure 15.2: Lagrange Polynomials and resulting 2nd degree polynomial

$$r(x) = p_2(x) - q_2(x).$$

First note that $r(x)$ is also of degree ≤ 2. But

$$r(x_i) = p_2(x_i) - q_2(x_i) = y_i - y_i = 0, \quad i = 0, 1, 2$$

This means that $r(x)$ has three distinct roots and has degree ≤ 2. That isn't possible unless $r(x) \equiv 0$ by the Fundamental Theorem of Algebra. Therefore $p_2(x) = q_2(x)$.

15.2 General Form of Lagrange Polynomials

To generalize this idea to a higher number of data points, first notice that we can write the Lagrange polynomial fitting the n points as:

$$\begin{aligned} L_j(x) &= \frac{(x - x_0)(x - x_1)\cdots(x - x_{j-1})(x - x_{j+1})\cdots(x - x_n)}{(x_j - x_0)(x_j - x_1)\cdots(x_j - x_{j-1})(x_j - x_{j+1})\cdots(x_j - x_n)} \\ &= \prod_{i=0}^{n} \frac{x - x_i}{x_j - x_i}. \end{aligned} \tag{15.4}$$

Note that the term in the numerator:

$$(x - x_0)(x - x_1) \cdots (x - x_{j-1})(x - x_{j+1}) \cdots (x - x_n) = 0 \quad \forall\ x_i \neq x_j$$

is missing the jth term, hence for all other node points it will be equal to 0. In addition, if $x_i = x_j$, then the numerator and the denominator are equal, hence $L_j(x)$ satisfies the conditions for a Lagrange polynomial.

Given this formulation, the construction and evaluation of the interpolating polynomials proceeds as before.

1. ***Construction of*** $L_j(x)$.

To simplify notation, we define the following terms:

$$\begin{aligned} \rho_j &= \prod_{i \neq j}^{n} x_j - x_i \quad j = 0, 1, \ldots, n, \\ w_j &= \frac{1}{\rho_j}, \end{aligned} \tag{15.5}$$

where the w_j are called the ***barycentric weights***. ((Berrut and Trefethen 2004))

Remark: Having computed these weights that's it for construction!

Notice also that this computation can be done in $O(n^2)$ flops.

2. ***Evaluation of*** $p_n(x)$

To evaluate the polynomial we just need to bring the different parts together. In order to simplify this process, first notice that if we define

$$\begin{aligned} \Psi(x) &= (x - x_0)(x - x_1) \cdots (x - x_n) \\ &= \prod_{i=0}^{n} (x - x_i) \end{aligned} \tag{15.6}$$

then we can write the interpolating polynomial as:

$$p_n(x) = \Psi(x) \sum_{j=0}^{n} \frac{w_j \cdot y_j}{(x - x_j)}. \tag{15.7}$$

Here all we've done is notice that the numerator of Equation 15.4 is nothing more than Equation 15.6 divided by the missing term $(x - x_j)$. When we insert this into the formula for the interpolating polynomial then $\Psi(x)$, which doesn't depend on j can be pulled out of the summation sign. This also has the advantage that we just need to compute that term once! The rest of the evaluation can be done if $O(n)$ flops.

Let's do one quick example using the data given below:

Table 15.1: Data

	$i=0$	$i=1$	$i=2$	$i=3$
x	-1.1	1.1	2.2	0.0
y	0.0	6.75	0.0	0.0

First we construct the ***barycentric weights***, $w_j = 1/\rho_j,\ j = 0, 1, 2, 3.$

- $j = 0$

$$\begin{aligned}\rho_0 &= \prod_{i\neq j}^{n}(x_j - x_i), \qquad j = 0\\ &= (x_0 - x_1)(x_0 - x_2)(x_0 - x_3)\\ &= (-1.1 - 1.1)(-1.1 - 2.2)(-1.1 - 0)\\ &= (-2.2)(-3.3)(-1.1)\\ &= -7.986\end{aligned}$$

- Similarly for $j = 1$

$$\begin{aligned}\rho_1 &= \prod_{i\neq j}^{n}(x_j - x_i), \qquad j = 1\\ &= (x_1 - x_0)(x_1 - x_2)(x_1 - x_3)\\ &= (1.1 - (-1.1))(1.1 - 2.2)(1.1 - 0)\\ &= (2.2)(-1.1)(1.1)\\ &= -2.662\end{aligned}$$

I'll leave the computation of the other two as an exercise.

Solution.

$$\begin{aligned}\rho_2 &= (x_2 - x_0)(x_2 - x_1)(x_2 - x_3) \qquad j = 2\\ &= 7.986\\ \rho_3 &= (x_3 - x_0)(x_3 - x_1)(x_3 - x_2) \qquad j = 3\\ &= 2.662\end{aligned}$$

Having computed the ρ values, the barycentric weights can now be easily computed.

An interesting feature of this formulation is that the construction of the interpolating polynomial ***does not depend on the values of*** y - we only need the data nodes x_i.

Tip

Once we have computed the weights for a given set of x values, we can use them for any function whatsoever (as long as we know values at those nodes). This also means that since we never assumed any order of the nodes, the weights are independent of the order that the nodes are in. Based on this, it is not hard to show that Equation 15.7 can be re-written in the more elegant form of:

$$p_n(x) = \frac{\sum_{j=0}^{n} \frac{w_j \cdot y_j}{(x-x_j)}}{\sum_{j=0}^{n} \frac{w_j}{(x-x_j)}}. \tag{15.8}$$

where the term $\Psi(x)$ has been eliminated.

15.2.1 Summary

Let's summarize the use of Lagrange basis functions as an interpolating function.

- We introduced the concept of a Lagrange polynomial that can be used as a basis for an interpolating function
- We also showed how to construct an interpolating polynomial using a special set of weights called barycentric weights.
- The construction of this polynomial can be shown to be $O(n^2)$, and does not depend on the values of y or $f(x)$.
- Once the construction has been completed, the interpolating polynomial can be used for any number of different functions, each evaluation costing $O(n)$ flops.
- It can also be shown that if we choose the data point properly, then the algorithm for constructing and evaluating this polynomial is stable. ((Nicholas J. Higham 2004))

15.3 References

1. Barycentric Lagrange Interpolation. (Berrut and Trefethen 2004)
2. The numerical stability of barycentric Lagrange interpolation. (Nicholas J. Higham 2004)


```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

16 Divided Differences and Newton Interpolating Polynomials

Lecture 16: October 26, 2023

16.1 Divided Differences

As noted earlier, although an interpolating polynomial for a given set of points is unique, there are several algebraic representations we can use. We have one more form for interpolating polynomials that is frequently used.

It has the advantage that the coefficients can be easily computed as new data becomes available. This could prove especially useful in an experimental setup where one may not know how much data will be available at the beginning of the experiment.

We'll need some new notation first in order to write down this new form for an interpolating polynomial. Let's first recall the shifted form for expressing a polynomial:

$$\begin{aligned} p_n(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + ... \\ + c_n(x - x_0)(x - x_1)...(x - x_{n-1}) \end{aligned}$$

Note that we could determine the coefficients through the following procedure:

$$\begin{aligned} &p_n(x_0) = c_0 = f(x_0) \\ &p_n(x_1) = c_0 + c_1(x_1 - x_0) = f(x_1) \implies c_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0} \\ &\quad ... \end{aligned}$$

The following notation will prove useful.

Definition 16.1. The ***zeroth divided difference*** of a function f with respect to x_i is defined as $f[x_i] = f(x_i)$. Using the procedure above as our guide, the ***first divided difference*** is defined as:

$$f[x_i, x_{i+1}] = \frac{f[x_{i+1}] - f[x_i]}{x_{i+1} - x_i}$$

Higher order divided differences can likewise be defined ***recursively*** in terms of lower order divided differences. For example the ***second-order divided difference*** can be expressed as:

$$f[x_i, x_{i+1}, x_{i+2}] = \frac{f[x_{i+1}, x_{i+2}] - f[x_i, x_{i+1}]}{x_{i+2} - x_i},$$

so specifically for $i = 0$ we would have:

$$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$$

In general, we can express the divided difference can be written as:

$$f[x_0, x_1, x_2, \dots, x_j] = \frac{f[x_1, x_2, \dots, x_j] - f[x_0, x_1, \dots, x_{j-1}]}{x_j - x_0} \tag{16.1}$$

Equation 16.1 is also known as the ***Newton divided difference*** form.

> **Note**
>
> Divided differences have some useful properties that we will use later on. The first important one is that the order of the data points doesn't matter. In other words:
>
> $$f[x_{i_0}, x_{i_1}, x_{i_2}, \dots, x_{i_n}] = f[x_0, x_1, x_2, \dots, x_n]$$
>
> where$(i_0, i_1, i_2, \dots, i_n)$ is a permutation of $0, 1, 2, \dots, n$. One can easily show that, for example:
>
> $$f[x_1, x_0] = f[x_0, x_1].$$

Example 16.1. Compute the first and second divided differences for $f(x) = cos(x)$, using $x_0 = 0.2, x_1 = 0.3, x_2 = 0.4$.

$$
\begin{aligned}
f[x_0, x_1] &= \frac{f[x_1] - f[x_0]}{x_1 - x_0} \\
&= \frac{\cos(0.3) - \cos(0.2)}{0.3 - 0.2} \\
&= \frac{0.955336489 - 0.980066578}{0.1} \\
&= -0.24730089
\end{aligned}
$$

Likewise

$$
\begin{aligned}
f[x_1, x_2] &= \frac{f[x_2] - f[x_1]}{x_2 - x_1} \\
&= \frac{\cos(0.4) - \cos(0.3)}{0.1} \\
&= \frac{0.921060994 - 0.955336489}{0.1} \\
&= -0.34275495
\end{aligned}
$$

The second divided difference is therefore given by:

$$
\begin{aligned}
f[x_0, x_1, x_2] &= \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0} \\
&= \frac{(-0.34275495) - (-0.24730089)}{0.4 - 0.2} \\
&= -0.477727030
\end{aligned}
$$

Solution. Example 16.1

```
x0 = 0.2; x1 = 0.3; x2 = 0.4

# first 2 divided differences
fx0x1 = (cos(x1) - cos(x0))/(x1 - x0)
fx1x2 = (cos(x2) - cos(x1))/(x2 - x1)

# second order divided difference
fx0x1x2 = (fx1x2 - fx0x1)/(x2 - x0)
fx0x1x2
```

```
[1] -0.4772703
```

```
# actual
fc = -cos(x1)/2
fc
```

```
[1] -0.4776682
```

16.2 Newton Interpolating Polynomials

While divided differences have many applications, the one we will use here is as a means to write down an interpolating polynomial with an important computational property.

Suppose as before that we have an interpolating polynomial of degree n such that $p_n(x) = f(x_i), i = 0, 1, \dots, n$ with distinct data points. Then an interpolating polynomial can be written using Newton divided differences as follows:

$$\begin{aligned}
p_1(x) &= f(x_0) + f[x_0, x_1](x - x_0) \\
p_2(x) &= f(x_0) + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) \\
&\vdots \\
p_n(x) &= f(x_0) + f[x_0, x_1](x - x_0) + \dots \\
&\quad + f[x_0, x_1, \dots, x_n](x - x_0)(x - x_1) \cdots (x - x_{n-1})
\end{aligned}$$

Note that for all interpolating polynomial of degree $n>1$, they can all be constructed by using the previous interpolating polynomial of degree $n-1$, with one additional term. For example:

$$\begin{aligned}
p_1(x) &= f(x_0) + f[x_0, x_1](x - x_0) \\
p_2(x) &= p_1(x) + f[x_0, x_1, x_2](x - x_0)(x - x_1) \\
&\vdots \\
p_n(x) &= p_{n-1}(x) + f[x_0, x_1, \dots, x_n](x - x_0)(x - x_1) \cdots (x - x_{n-1})
\end{aligned}$$

In general we can write:

$$p_{k+1}(x) = p_k(x) + f[x_0, x_1, \dots, x_k](x - x_0)(x - x_1) \cdots (x - x_k)$$

The beauty of this form is that we can easily go from a polynomial of degree k to degree $k+1$, by adding one ($k-th$ order) divided difference.

> **Note**
>
> ***Remark:*** As before, it is not difficult to show that the Newton divided difference interpolating polynomial is exactly the same as the one we derived earlier (if perhaps in a slightly disguised form).

The following theorem will prove useful in future analysis.

Theorem 16.1. *Let $n \geq 1$ and assume that $f(x) \in C^n[a,b]$. Let $x_0, x_1, \ldots, x_n$ be $n+1$ distinct points in $[a,b]$. Then*

$$f[x_0, x_1, x_2, \ldots, x_n] = \frac{1}{n!} f^n(\xi),$$

for some ξ between the minimum and maximum of $x_0, x_1, \ldots, x_n$.

Remark: This might look familiar as it looks similar to the remainder term in the Taylor polynomial. As a result, it won't come as a surprise that it will be useful in our error analysis (Section 17.1)

16.3 Summary

We've now seen three different approaches for computing interpolating polynomials. Let's briefly summarize some of the most important features of each of them.

Table 16.1: Interpolating Polynomials Summary

Basis	$\phi_j(x)$	Construction Cost	Evaluation Cost	Distinguishing Feature
Monomial	x^j	$O(n^3)$	$O(n)$	Simple, easy
Lagrange	$L_j(x)$	$O(n^2)$	$O(n)$	$c_j = y_j$, most stable
Newton	$\prod_{i=0}^{j-1}(x - x_i)$	$O(n^2)$	$O(n)$	Adaptive (can add new points)

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

17 Interpolation Error

Lecture 17: October 31, 2023

17.1 Error in Polynomial Interpolation

If we have an interpolating polynomial for a given function at a set of points, we know that by definition, it must match the function exactly at the nodes. However, this also brings up a related question - How does $p_n(x)$ behave at points other than the nodes $\{x_i\}$?

In order to consider this question we will make a few assumptions to help us with the analysis, specifically:

1. $f(x)$ is defined on $[a, b]$.
2. f has all needed derivatives and they are bounded.

Let's first define the error function for the $n-th$ degree interpolating polynomial:

$$e_n(x) = f(x) - p_n(x), \quad x \in [a, b].$$

The question we would like to ask is what does the error look like for a point in the interval $[a, b]$ that is not one of the node points.

Here we will use the simple observation that any x that is not a node can be treated as a new interpolation point. As such, we can use the Newton Interpolation formula for adding a new point, i.e.

$$f(x) = p_{n+1}(x) = p_n(x) + f[x_0, x_1, \dots, x_n, x]\Psi_n(x),$$

where

$$\Psi_n(x) = (x - x_0)(x - x_1)\cdots(x - x_n).$$

Substituting this into the error function we get:

$$e_n(x) = f(x) - p_n(x) = f[x_0, x_1, \dots, x_n, x]\Psi_n(x).$$

This seems a little unsatisfying however as the error seems to depend both on the data points and an individual point x.

Using Equation 16.1 we can replace the divided difference with the derivative:

Theorem 17.1.

$$f[x_0, x_1, x_2, \dots, x_n, x] = \frac{f^{n+1}(\xi)}{(n+1)!},$$

to give us the following formula for the error function:

$$e_n(x) = f(x) - p_n(x) = \frac{f^{n+1}(\xi)}{(n+1)!}\Psi_n(x).$$

This leads us directly to the following theorem.

Theorem 17.2 (Polynomial Interpolation Error)**.** *Let $n \geq 0$ and $f \in C^n[a, b]$. Suppose we are given $x_0, x_1, \dots, x_n$ distinct points in $[a, b]$. Then*

$$f(x) - p_n(x) = \frac{f^{n+1}(\xi)}{(n+1)!}\Psi_n(x), \tag{17.1}$$

with $\Psi_n(x) = \prod_{i=0}^{n}(x - x_i)$., for $a \leq x \leq b$, where $\xi(x)$ is an unknown between the min and max of $x_0, x_1, \dots, x_n$ and x.

Furthermore

$$\max |f(x) - p_n(x)| \leq \frac{1}{(n+1)!} \max_{a \leq t \leq b} \left|f^{(n+1)}(t)\right| \cdot \max_{a \leq s \leq b} \prod_{i=0}^{n} |s - x_i|$$

This theorem is nice, but in order to proceed much further, we would need to know more about both f and $\Psi_n(x)$. Nonetheless, it does suggest that in order to minimize the error, it would be good to keep the new point x close to one of the interpolating points.

A final note is in order. We used the Newton form to obtain this error bound, but the bound will apply to other polynomial forms since we know we have a unique polynomial, and the various forms we have used are merely disguised versions of each other.

Example 17.1. To Do:

- Include one example here for theorem, e.g. f(x) = exp(x)
- Include one example here for previous theorem, e.g. f(x) = cos(x)

17.2 Practical Tips

1. In understanding the error we might encounter when using interpolation, we can use Equation 17.1, to give us a sense of how the error behaves throughout the interval. Clearly, at the node points themselves, the error will be zero, but what about the rest of the points in $[a, b]$?
2. When considering the error bounds above, the interpolation error is likely to be smaller when evaluated at points close to the middle of the domain.
3. In practice, high degree polynomials with equally spaced nodes are not suitable for interpolation because of this oscillatory behavior.
4. However, if a set of suitably chosen data points that are not equally spaced may be useful in obtaining polynomial approximations of some functions.

Example 17.2. An excellent example of the type of behavior that can occur when using equally spaced nodes was described by Runge in 1901. Using the seemingly innocuous function: $f(x) = 1/(1 + x^2), -5 \leq x \leq 5$. If one uses equally spaced nodes, for example $x_i = \frac{2i}{n} - 1$ on the interval $[-1, 1]$, then it can be shown that the interpolation error grows without bound at the ends of the interval.

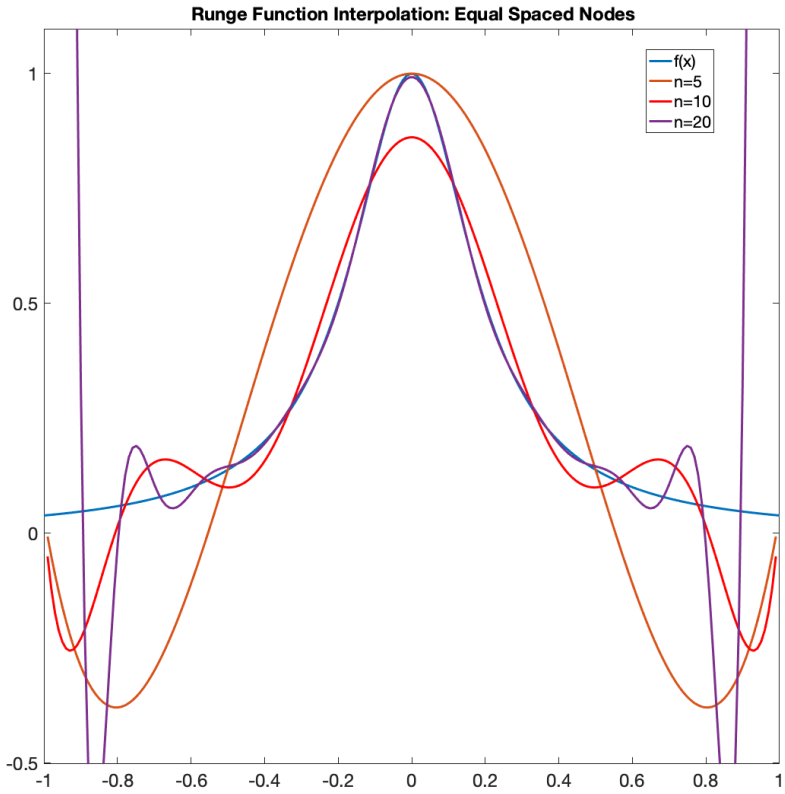


Figure 17.1: Example: Runge Function using equally spaced nodes

> **!** High-Degree Polynomials
>
> In general, one should be wary of using a high-degree polynomial as they can oscillate drastically and care must be taken whenever used. In general, other approaches will prove to be more useful in situations where more accuracy is required or more data points are given.

17.3 Chebyshev Points

The Chebyshev points are defined on the interval $[-1, 1]$ by:

$$x_i = \cos\left(\frac{2i+1}{2(n+1)}\pi\right), \quad i = 0, \ldots n. \tag{17.2}$$

To generate the desired points for a general interval $[a, b]$, one then uses the affine transformation on Equation 17.2:

$$\tilde{x_i} = a + \frac{b-a}{2}(x_i + 1), \quad i = 0, \ldots n. \tag{17.3}$$

These new points have the feature that they are clustered near the end points of the interval rather than being uniformly spaced across the interval.

If one uses, for example, the Lagrange polynomial form with the Chebyshev points defined by Equation 17.3 then it can be shown that the interpolation error is greatly reduced.

> **Note**
>
> Chebyshev points and interpolation have many interesting properties and many different applications. Unfortunately, we will not have time to go over most of these advanced topics.

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

18 Piecewise Polynomials and Cubic Splines

18.1 Piecewise Polynomials

Lecture 18: November 2, 2023

18.1.1 Motivation

So far we've spent a lot of time discussing the good properties of polynomial interpolants. However, they also have several shortcomings. Among these, we list four here:

1. The error term may not be small. Theorem 17.2 showed that the error bound depended on both the size of the interval as well as the higher derivatives, which could be large.
2. Higher order polynomials oscillate (wiggle) a lot. When fitting data that doesn't have a lot of oscillations the polynomial will not match the data in certain areas of the interval (usually near the ends of the intervals).
3. Data are often only piecewise smooth, but polynomials are infinitely differentiable. Asking a polynomial to fit data that isn't as smooth as itself may not be fair.
4. Changing a single data point could drastically alter the entire interpolant

As we discussed in the practical tips section (Section 17.2), it is best to think of using 1) low-order polynomials, 2) within small intervals, 3) and only think of them as local approximations.

This leads us to think about using an alternative approach, which can be briefly described as:

> **Idea**
>
> Instead of finding one single polynomial to fit all the data find a set of polynomials for different regions within the given interval.

In more detail, this approach can be described as:

1. Divide the interval $[a, b]$ into a set of smaller subintervals (elements)

$$a = t_0 < t_1 < ... < t_r = b.$$

   The t_i are often referred to as break points, or sometimes just ***knots***.

2. Fit a low-degree polynomial $s_i(x)$ in each of the subintervals $[t_i, t_{i+1}]$, $\quad i = 0, ... r - 1$

3. Patch (glue) the polynomials together so that

$$v(x) = s_i(x), \quad t_i \leq x \leq t_{i+1} \quad i = 0, ... r - 1.$$

18.2 Piecewise Linears

Example 18.1. Suppose we were given the following data points

x	1	2	4	5	6
y	1	1.8	2	1.8	0.5

Consider using the Newton form for a linear polynomial ***within*** each of the sub-intervals.

$$v(x) = s_i(x) = f(x_i) + f[x_i, x_{i+1}](x - x_i), \quad t_i \leq x \leq t_{i+1}, \quad 0 \leq i \leq 4.$$

Here note that for now, $t_i = x_i$

For example, for $i = 0$, we would have:

$$\begin{aligned} s_0(x) &= f(x_0) + f[x_0, x_1](x - x_0), \\ &= 1 + \frac{1.8 - 1}{2 - 1}(x - 1), \\ &= 1 + 0.8(x - 1). \end{aligned}$$

Likewise, we would then compute s_1, s_2, s_3, s_4.

18.3 Error Bounds

It turns out to be fairly easy to compute an error bound for this case (we've done most of the heavy lifting in the previous sections already).

First let's provide some notation to help us in this new situation.

Let

$$\begin{aligned} n &= r && \text{number of subintervals} \\ t_i &= x_i \\ h &= \max_{1 \le i \le n} (t_i - t_{i+1}) && \text{maximum subinterval size} \end{aligned}$$

Claim:

$$\left| f(x) - v(x) \right| \le \frac{h^2}{8} \max_{a \le \xi \le b} f''(\xi),$$

for any $x \in [a, b]$.

Proof: First, let's note that for any $x \in [a,b], $ it must lie in some interval, say i, therefore $t_{i-1} \le x \le t_i$. We can now apply Equation 17.1, and since we are using linear interpolation, $n = 1$, the bound states that:

$$f(x) - v(x) = \frac{f''(\xi)}{2!}(x - t_{i-1})(x - t_i). \tag{18.1}$$

Next we note that the maximum of the quantity $(x - t_{i-1})(x - t_i)$ occurs at the point

$$x = \frac{t_{i-1} + t_i}{2}.$$

(We did this in class, so we won't repeat it here).

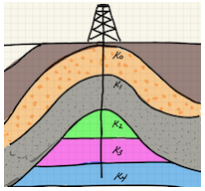
Therefore we can say that:

$$\begin{aligned} \left| (x - t_{i-1})(x - t_i) \right| &\le \left| \left(\frac{t_i - t_{i-1}}{2} \right)^2 \right|, \\ &\le \frac{h^2}{4}. \end{aligned}$$

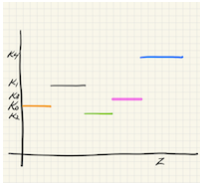
The result now follows by substituting this back into Equation 18.1.

18.4 Piecewise Constants

Before moving to higher order piecewise interpolation, it might be good to note that sometimes it is useful to consider piecewise constants. This approach could be used, for example in applications where the data is known to have discontinuities. One application we presented in class was that of modeling the subsurface of the earth in oil reservoir and geophysical exploration models.



(a) Reservoir Model with different layers

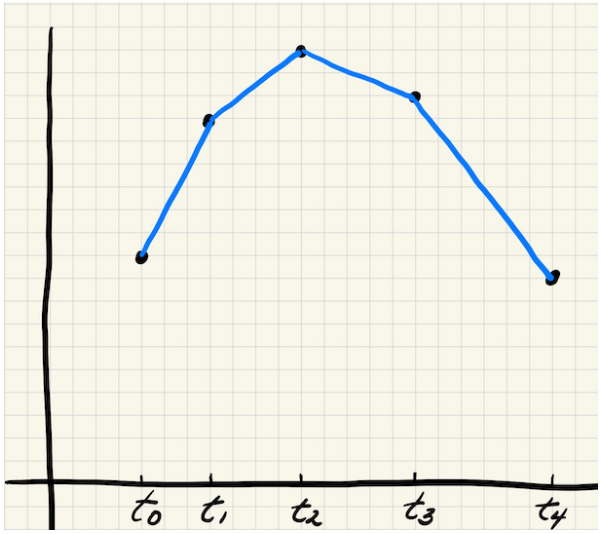


(a) Piecewise constant data for layers

Figure 18.2: Oil Reservoir Model with piecewise constant data

18.5 Cubic Splines

Piecewise linear polynomials appear to be a good compromise but they do have one clear disadvantage – the final interpolant will likely have corners at the knots.



What if we want to have a ***smoother*** interpolant? This could be quite important if we are trying to approximate a function that is known to have certain smoothness properties, or if

we are modeling some physical or engineering problem that we wish to have smoothness, such as an airplane wing or a car body.

The most popular approach for creating a smooth piecewise interpolant is known as ***cubic splines***.

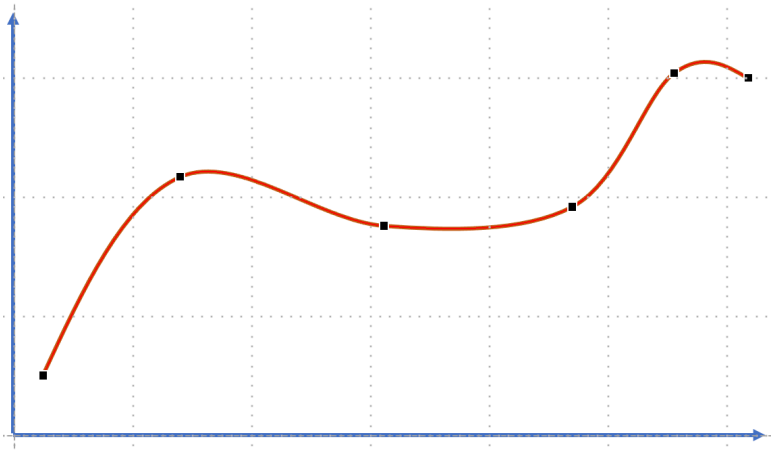


Figure 18.3: Cubic Spline with 6 data points

Let's consider a cubic interpolant for the ith interval, which we can write as:

$$v(x) = s_i(x) = a_i + b_i(x - t_i) + c_i(x - t_i)^2 + d_i(x - t_i)^3.$$

Note that there are 4 unknowns a_i, b_i, c_i, d_i per sub-interval. Hence there are $4n$ unknowns in total. That means if we want to have a unique solution, we need to also have $4n$ equations (conditions) specified.

The usual approach is to generate these equations through a combination of:

1. interpolation conditions, and
2. continuity conditions

Let's first note that in the linear case we had:

$$\begin{aligned} s_i(t_i) &= f(t_i), \quad & i = 0, 1, \dots, n-1, \\ s_i(t_{i+1}) &= f(t_{i+1}), \quad & i = 0, 1, \dots, n-1. \end{aligned} \tag{18.2}$$

This gave us $2n$ conditions for the $2n$ unknowns. In addition, continuity was implied because

$$s_i(t_{i+1}) = f(t_{i+1}) = s_{i+1}(t_{i+1}).$$

With cubic splines, we can use the interpolating conditions (Equation 18.2) to give us $2n$ conditions. The question before us now is how to choose the additional $2n$ conditions required to give us a unique solution.

Idea

Use remaining $2n$ conditions so as to satisfy $v(x) \in C^2[a, b]$. In other words, ensure that the spline is twice-continuously differentiable, i.e.

- $s_i(x)$ is continuous at the knots
- $s_i'(x)$ is continuous at the knots
- $s_i''(x)$ is continuous at the knots

Mathematically, this idea translates into:

$$\begin{aligned}
s_i(t_i) &= f(t_i), & i &= 0, 1, \dots, n-1, & n &\text{ conditions} \\
s_i(t_{i+1}) &= f(t_{i+1}), & i &= 0, 1, \dots, n-1, & n &\text{ conditions} \\
s_i'(t_{i+1}) &= s_{i+1}'(t_{i+1}), & i &= 0, 1, \dots, n-2, & n-1 &\text{ conditions} \\
s_i''(t_{i+1}) &= s_{i+1}''(t_{i+1}), & i &= 0, 1, \dots, n-2, & n-1 &\text{ conditions}
\end{aligned}$$

It is important to note that the last two conditions only hold at the internal knots since that is where two splines meet and need to be aligned to maintain continuity of the derivatives. Counting up the conditions therefore leaves us with only $4n - 2$ conditions.

There are two popular approaches to resolving this problem:

1. ***Free boundary*** (natural spline):

$$v''(t_0) = v''(t_n) = 0$$

2. ***Clamped boundary***:

$$\begin{aligned}
v''(t_0) &= f'(t_0), \\
v''(t_n) &= f'(t_n).
\end{aligned}$$

The free boundary approach is the easiest to implement and apply. However, it is rather arbitrary and there is no *a priori* reason to expect it to be true.

The clamped boundary approach is more realistic, but has the disadvantage of requiring the second derivative of the function. If the second derivative is known (or can be approximated) however, this approach would be preferred.

18.6 Summary

This section covered the idea of using piecewise interpolating polynomials within subintervals of the domain as opposed to using one single polynomial over the entire domain. This approach has several advantages over using one polynomial including the ability to take into account more of the structure of the problem as well as producing smoother interpolants over the entire region.

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

Part IV

Numerical Differentiation

Motivation - Why/When do we use numerical differentiation

- Approximating Derivatives in ODEs and PDEs
- Computing Derivatives from data (experimental or computed)

Goals

- Introduce formulas for computing derivatives numerically
- Richardson Extrapolation
- Analyze errors when computing numerical derivatives
- Understand stability of numerical algorithms

19 Finite Differences

19.1 Motivation

Computing derivatives numerically arises in many situations in numerical analysis as well as scientific computing. Some of the most common examples include:

- Numerically solving ordinary differential equations (ODEs), partial differential equations (PDEs), nonlinear equations and optimization (e.g. Newton's method).
- Computing derivatives of complicated functions.
- Situations where f is not known explicitly or only as a black box.

The basic tools used for numerical derivatives include one tool that we've been using extensively (Taylor series), and one more recently (polynomial interpolation).

19.2 Taylor Series Approach

Let $f : R^1 \to R^1$ with as many derivatives as we need.

Recall the *Taylor series expansion* for a function

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(\xi), \qquad \xi \in [x, x+h]$$

Rearranging we can write:

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{h}{2}f''(\xi)$$

This leads to the ***forward difference*** approximation, which can be written as:

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

where we will denote the *truncation error* by

$$-\frac{h}{2}f''(\xi)$$

Notice that the truncation error can be expressed in Big O notation as $O(h)$ and we therefore say that forward differences are an $O(h)$ approximation or that it is ***first order accurate***.

> **!** Truncation Error is NOT Roundoff Error
>
> *The truncation error should not be confused with roundoff error!*

Similarly the ***backward difference*** approximation can be written as

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

with a similar error term. Clearly, backward differences are also first order accurate.

Which one should I use?

There is no major difference between the two; it mostly comes down to a matter of convenience or preference. There are however some situations, where it makes more sense to use one over the other. One recent example is when we had to impose a condition on the cubic spline interpolant and we needed to have the derivative of the first and last spline math the derivative of the function. In that case, we should use the forward difference approximation for the first spline and the backward difference approximation for the last spline.

> **i** Review - Big O Notation
>
> Remember that we denote a quantity x as $O(h)$ if it is at most proportional to h, for example Ch for some constant C. One way to think of it as
>
> $$\lim_{h \to 0} \frac{O(h)}{h} = C < \infty$$
>
> Notice that according to our definition, both forward and backward difference formulas have truncation error that is $O(h)$.
> ***Question:***
>
> - What is the order of
>
> $$\frac{h}{2} f''(x) + \frac{h^2}{3} f'''(x)?$$
>
> - Just need to take a look at the lowest power of h. If the power of h is 1, we say that it is *first order.* If the power of h is 2, we say that it is *second order,* and so on.
>
> *All other things being equal, we want as high a power of h for our error term as we can achieve!*

Example 19.1. Computation of forward difference for $\exp(x), x = 1$

```
x <- c(1, 1, 1, 1, 1, 1)
h <- c(.1, .05, .025, .0125, 0.00625, 0.003125)
xph <- x + h
fx <- exp(x)
fxph <- exp(xph)
ffwd <- (fxph-fx)/h
fprime <- fx
err <- abs(ffwd-fprime)
```

Table 19.1: Forward Difference for exp(x), x = 1

h	fprime	F.D	err (F.D)
0.100000	2.718282	2.858842	0.1405601
0.050000	2.718282	2.787386	0.0691040
0.025000	2.718282	2.752545	0.0342635
0.012500	2.718282	2.735342	0.0170603
0.006250	2.718282	2.726794	0.0085124
0.003125	2.718282	2.722534	0.0042517

Exercise 19.1. Compute the forward difference of $f(x) = tan(x), x = \pi/4, h = 0.1, 0.05, 0.025, 0.125, 0.00625, 0.003125$

Solution:

```
library(knitr)
x <- (pi/4)*c(1, 1, 1, 1, 1, 1)
h <- c(.1, .05, .025, .0125, 0.00625, 0.003125)
xph <- x + h
xmh <- x - h
fx <- tan(x)
fxph <- tan(xph)
fxmh <- tan(xmh)
ffwd <- (fxph-fx)/h
fbwd <- (fx - fxmh)/h
fprime <- 1 / (cos(x)^2)
err1 <- abs(fprime-ffwd)
err2 <- abs(fprime-fbwd)

ex1data <- data.frame(h, fprime, ffwd, fbwd, err1, err2)
```

```
names(ex1data) <- c("h", "fprime", "F.D", "B.D", "err (F.D)", "err (B.D)")
kable(ex1data,caption = 'Forward Difference for tan(x), x = pi/4')
```

Table 19.2: Forward/Backward Difference for tan(x), x = pi/4

h	fprime	F.D	B.D	err (F.D)	err (B.D)
0.100000	2	2.230489	1.823712	0.2304888	0.1762881
0.050000	2	2.107112	1.906275	0.1071118	0.0937249
0.025000	2	2.051721	1.951616	0.0517205	0.0483838
0.012500	2	2.025423	1.975410	0.0254233	0.0245897
0.006250	2	2.012605	1.987603	0.0126050	0.0123966
0.003125	2	2.006276	1.993776	0.0062761	0.0062241

19.3 Central Differences

Consider the Taylor series at $x+h$ and $x-h$

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(\xi_1(x)) \\ f(x-h) &= f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(\xi_2(x)) \end{aligned} \tag{19.1}$$

for some $\xi_1 \in [x, x+h], \xi_2 \in [x-h, x]$.

Subtracting the second equation from the first gives us:

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{h^3}{6}[f'''(\xi_1(x)) + f'''(\xi_2(x))]$$

and solving for f' we get:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6} \cdot \frac{1}{2}[f'''(\xi_1(x)) + f'''(\xi_2(x))]$$

The last term that includes the third derivative at two different points can be replaced by using the Intermediate Value Theorem:

> **i** Intermediate Value Theorem:
>
> Suppose that *(i)* f is continuous on the closed finite interval $[a, b]$ and *(ii)* $f(a) < c < f(b)$. Then there exists some point $x \in [a, b]$ such that $f(x) = c$.

Notice that, since the average value must lie in between the value at the two end points a straightforward application of the IVT says:

$$\frac{1}{2}[f'''(\xi_1(x)) + f'''(\xi_2(x))] = f'''(\xi), \quad \xi \in [x-h, x+h]$$

Substituting for the f''' terms yields:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6} f'''(\xi(x)) \tag{19.2}$$

Equation 19.2 is called the central (or centered) difference approximation.

Example 19.2. Compare central vs. forward differences for $f(x) = \tan(x), x = \pi/4$

Table 19.3: Forward v. Central Difference for tan(x), x = pi/4

h	F.D	err (F.D)	C.D	err (C.D.)
0.100000	2.230489	0.2304888	2.027100	0.0271004
0.050000	2.107112	0.1071118	2.006693	0.0066934
0.025000	2.051721	0.0517205	2.001668	0.0016683
0.012500	2.025423	0.0254233	2.000417	0.0004168
0.006250	2.012605	0.0126050	2.000104	0.0001042
0.003125	2.006276	0.0062761	2.000026	0.0000260

Practical Tip

It is not too difficult to generate other 3- and 5- point formulats using similar techniques. One must remember that we need to balance accuracy with the additional work needed and to be aware of any special conditions or structure available - for example the case of specifying derivative conditions for cubic splines at the end points, discussed earlier in this section.

19.4 Second Derivative Formulas

Using similar techniques as for the centered difference formulas we can develop approximations for the second derivative of a function. Consider once again the Taylor series for a function about a point:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f^{(4)}(\xi_1(x))$$
$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f^{(4)}(\xi_2(x))$$

Here, as we want to have a formula for the second derivative, our goal is to get rid of the other terms, and in particular the first derivative. Hence, let's add the two equations:

$$f(x+h) + f(x-h) = 2f(x) + h^2 f''(x) + \frac{h^4}{24}\left[f^{(4)}(\xi_1(x)) + f^{(4)}(\xi_2(x))\right]$$

Solving for f'' we get:

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} - \frac{h^4}{24}\left[f^{(4)}(\xi_1(x)) + f^{(4)}(\xi_2(x))\right]$$

Now using the same trick we used before by appealing to the IVT, we have the following formula for numerically computing the second derivative.

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} - \frac{h^2}{12}f^{(4)}(\xi).$$

Again, this approximation is clearly $O(h^2)$ and we say that it is a *second order approximation*.

Example 19.3. Second Derivatives using finite differences

```
x <- c(1, 1, 1, 1, 1)
h <- c(.1, .05, .025, .0125, 0.00625)
xph <- x + h
xmh <- x -h
fx <- exp(x)
fxph <- exp(xph)
fxmh <- exp(xmh)
f2prime <- (fxph - 2*fx + fxmh)/ (h^2)
fprimeExact <- fx
err <- abs(f2prime-fprimeExact)
```

Table 19.4: Second Derivate for exp(x), x = 1

h	f(x)	f2prime	err
0.10000	2.718282	2.720548	0.0022660
0.05000	2.718282	2.718848	0.0005664
0.02500	2.718282	2.718423	0.0001416
0.01250	2.718282	2.718317	0.0000354
0.00625	2.718282	2.718291	0.0000088

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

20 Finite Differences via Lagrange Polynomials (Optional)

20.1 ($n+1$)-point Formulas

General Idea

Another approach for producing general finite difference formulas relies on using interpolating polynomials. The key idea is to first replace the function $f(x)$ by a Lagrange interpolating polynomial. Then in place of the derivative of $f(x)$ we will use the derivative of the Lagrange polynomial. Appropriate error estimates can also be produced.

The starting point for this strategy is the formula for the Lagrange interpolating polynomials.

Interpolating polynomials

Let $[x_0, x_1, \ldots, x_n]$ be $n+1$ distinct points on some interval $[a, b]$.

Then we can approximate $f(x)$ by

$$f(x) = \sum_{k=0}^{n} f(x_k) L_k(x) + \frac{(x - x_o)(x - x_1) \cdots (x - x_n)}{(n+1)!} f^{(n+1)}(\xi(x)), \tag{20.1}$$

where $\xi(x) \in [a, b]$, $L_k(x)$ is the kth Lagrange coefficient polynomial, and is written as

$$L_k(x) = \prod_{i=0, i \neq k}^{n} \frac{(x - x_i)}{(x_k - x_i)}$$

The second step is to differentiate Equation 20.1. Letting D_x denote the derivative with respect to x, we can rewrite Equation 20.1 as:

> **Note**
>
> *Except for special cases we cannot estimate the truncation error as Equation 20.2 is currently written.*

$$f'(x) = \sum_{k=0}^{n} f(x_k)L'_k(x) + D_x\left[\frac{(x-x_o)(x-x_1)\cdots(x-x_n)}{(n+1)!}\right] f^{(n+1)}(\xi(x)) + \frac{(x-x_o)(x-x_1)\cdots(x-x_n)}{(n+1)!} D_x[f^{n+1)}(\xi(x))] \tag{20.2}$$

Let's first consider the last term in Equation 20.2. Suppose we take $x = x_j$, where x_j is one of the given node points. With this choice, it is easy to see that the last term vanishes, i.e.:

$$\frac{(x-x_o)(x-x_1)\cdots(x-x_n)}{(n+1)!} D_x[f^{n+1)}(\xi(x))] = 0$$

since for any $x = x_j$, one of the product terms in the numerator will be equal to 0.

The second thing to note is that for $x = x_j$ we can simplify the second term

$$D_x\left[\frac{(x-x_o)(x-x_1)\cdots(x-x_n)}{(n+1)!}\right] = \prod_{k=0,k\neq j}^{n} (x_j - x_k)$$

The easiest way to see this is to note that the numerator is nothing but a product of linear terms. So using the chain rule we get a sum of terms each one of which has the original product terms minus the one corresponding to the one we take the derivative with respect to $x_k, k = 0, \ldots n$.

Substituting x_j for x, all the summand terms go to zero except for the one where $k \neq j$.

Together this means we can simplify Equation 20.2 to:

$$f'(x_j) = \sum_{k=0}^{n} f(x_k)L'_k(x_j) + \frac{f^{(n+1)}(\xi(x_j))}{(n+1)!} \prod_{k=0,k\neq j}^{n} (x_j - x_k) \tag{20.3}$$

We call this is the $(n+1)$-point formula (Equation 4.2, p. 176, textbook).

Remark: *In general, more points leads to higher accuracy, but at the expense of more function evaluations. The extra computations will likely lead to more roundoff error as well.*

20.2 Three-Point Formulas

The most common versions are the 3-point and 5-point formulas. Let's consider the specific case $n = 2$ (i.e. the 3-point case) applied to Equation 20.3

$$\begin{aligned} f'(x_j) =& f(x_0)L_0'(x_j) + f(x_1)L_1'(x_j) + f(x_2)L_2'(x_j) + \\ & \frac{f^{(3)}(\xi(x_j))}{6} \prod_{k=0, k\neq j}^{2} (x_j - x_k) \end{aligned} \tag{20.4}$$

Consider $L_0(x)$, which can be written as:

$$L_0(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)}$$

Taking the derivative with respect to x yields

$$L_0'(x) = \frac{(2x - x_1 - x_2)}{(x_0 - x_1)(x_0 - x_2)} \tag{20.5}$$

Likewise for L_1' and L_2' we can write:

$$L_1'(x) = \frac{(2x - x_0 - x_2)}{(x_1 - x_0)(x_1 - x_2)}, \qquad L_2'(x) = \frac{(2x - x_0 - x_1)}{(x_2 - x_0)(x_2 - x_1)} \tag{20.6}$$

Let's try to simplify matters a bit.

To start with, let's assume *equally spaced points* $[x_0, x_1, x_2]$, i.e. $[x_0, x_0 + h, x_0 + 2h]$. Since we can choose x_j to be any of the node points let's arbitrarily choose $x_j = x_0$.

This means several of the terms in the Lagrange polynomial can be further simplied, in particular notice that:

$$x_1 - x_0 = x_2 - x_1 = h \quad \text{and} \quad x_2 - x_0 = 2h$$

Substituting this into the formulas for the derivatives of the Lagrange polynomials (Equation 20.5, Equation 20.6) means we can write those terms as:

$$L_0'(x_0) = \frac{(2x_0 - x_1 - x_2)}{2h^2}, \quad L_1'(x_0) = \frac{(2x_0 - x_0 - x_2)}{-h^2}, \quad L_2'(x_0) = \frac{(2x_0 - x_0 - x_1)}{2h^2}$$

Now all we need to do is gather all the terms together and substitute them into Equation 20.4.

$$
\begin{aligned}
f'(x_0) = & f(x_0)\left[\frac{(2x_0 - x_1 - x_2)}{2h^2}\right] + \\
& f(x_1)\left[\frac{(2x_0 - x_0 - x_2)}{-h^2}\right] + \\
& f(x_2)\left[\frac{(2x_0 - x_0 - x_1)}{2h^2}\right] + \\
& \frac{f^{(3)}(\xi(x_j))}{6} 2h^2
\end{aligned}
$$

Here we are also using the fact that:

$$\prod_{k=0, k\neq j}^{2} (x_j - x_k) = (x_0 - x_1)(x_0 - x_2) = 2h^2$$

Now note that due to the equal spacing of the nodes the coefficient terms of $f(x_i), i = 0, 1, 2$ can also be simplified. For example, for the $f(x_0)$ term we see that:

$$
\begin{aligned}
2x_0 - x_1 - x_2 &= 2x_0 - (x_0 + h) - (x_0 + 2h) \\
&= 2x_0 - x_0 - h - x_0 - 2h \\
&= -3h
\end{aligned}
$$

A similar algebra exercise reduces the terms multiplying $f(x_1)$ and $f(x_2)$, resulting in:

$$2x_0 - x_0 - x_2 = -2h; \quad 2x_0 - x_0 - x_1 = -h$$

Substituting and gathering like terms yields:

$$f'(x_0) = \frac{1}{h}\left[\frac{-3}{2} f(x_0) + 2f(x_1) - \frac{1}{2} f(x_2)\right] + \frac{h^2}{3} f^{(3)}(\xi_0)$$

This leads directly to the *3-point endpoint finite difference* approximation

$$f'(x) = \frac{-3f(x) + 4f(x + h) - f(x + 2h)}{2h} + \frac{h^2}{3} f^{(3)}(\xi) \tag{20.7}$$

Using a similar set of arguments, we can derive the *3-point midpoint finite difference* formula

$$f'(x) = \frac{f(x + h) - f(x - h)}{2h} - \frac{h^2}{6} f^{(3)}(\xi) \tag{20.8}$$

Remark: *While the 3-point endpoint and midpoint formulas are both $O(h^2)$, the endpoint formula requires an additional function evaluation over the midpoint formula. Also note that the mid-point formula is sometimes called the central difference formula.*

```r
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

21 Richardson Extrapolation and Finite Difference Stability

21.1 Richardson Extrapolation

The methods we've studied so far for computing numerical derivatives are good and generally lead to good approximations. But what if we need to have greater accuracy? This section covers one approach to generating higher-order (more accurate) approximations to derivatives using an idea dating back to 1927. In spirit, it is not unlike what we did when we derived the central difference approximation by noticing that if we took the two $O(h)$ approximations for the forward and backward difference and combined them in such a way as to cancel out one of the error terms we could get a higher-order approximation $O(h^2)$.

> **Idea**
>
> Combine 2 approximations with similar error terms to obtain a more accurate approximation. Can be used in many different contexts including interpolation (Aitken), quadrature (Romberg, adaptive methods), IVP, and even to accelerate convergence of sequences.

Let's consider how we arrived at the 3-point formula for the second derivative:

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} - \frac{h^2}{12} f^{(4)}(\xi).$$

In that derivation we started with the Taylor series about $x+h$ and $x-h$. This time let's also consider one additional term, which would give us:

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} - \frac{h^2}{12} f^{(4)}(x) - \frac{h^4}{360} f^{(6)}(\xi) + O(h^5). \tag{21.1}$$

Now replace h by $2h$.

$$f''(x) = \frac{f(x+2h) - 2f(x) + f(x-2h)}{4h^2} - \frac{4h^2}{12} f^{(4)}(x) - \frac{16h^4}{360} f^{(6)}(\xi) + O(h^5). \tag{21.2}$$

Notice that the second formula has the same term for the fourth derivative except it is multiplied by 4. This leads to the natural idea of multiplying the first equation by 4 and subtracting

the second equation from it, which will cancel that part of the error term. If we do this, we will (after a bit of simple algebra) get to the following formula:

$$f''(x) = \frac{[-f(x+2h) + 16f(x+h) - 30f(x) + 16f(x-h) - f(x-2h)]}{12h^2} + \frac{h^4}{90}f^{(6)}(x) + O(h^5).$$

This formula for the second derivative is now ***fourth order*** accurate.

If we take another approach, it will help us generalize for future applications.

First, let's define some terms to simplify the derivation. Let:

$$M = f''(x)$$
$$N(h) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

Using these we can write Equation 21.1 and Equation 21.2 as:

$$M = N(h) - \frac{h^2}{12}f^{(4)}(x) - \frac{h^4}{360}f^{(6)}(\xi) + O(h^5).$$
$$M = N(h/2) - \frac{4h^2}{12}f^{(4)}(x) - \frac{16h^4}{360}f^{(6)}(\xi) + O(h^5).$$

As before, let's multiply the first equation by 4 and subtract the first:

$$\begin{aligned} 4M &= 4N(h) - \frac{4h^2}{12}f^{(4)}(x) - \frac{4h^4}{360}f^{(6)}(\xi) + O(h^5) \\ M &= N(h/2) - \frac{4h^2}{12}f^{(4)}(x) - \frac{16h^4}{360}f^{(6)}(\xi) + O(h^5) \\ \hline 3M &= 4N(h) - N(h/2) + \frac{12h^4}{360}f^{(6)}(\xi) + O(h^5) \end{aligned}$$

Solving for M, leads to:

$$M = \frac{1}{3}[4N(h) - N(h/2)] + \frac{h^4}{90}f^{(6)}(\xi) + O(h^5)$$

One final adjustment is usually made - we will split $4N(h) = 3N(h) + N(h)$ to give us the final form of the formula that is commonly used.

$$M = N(h) - \frac{1}{3}[N(h) - N(h/2)] + O(h^4).$$

The procedure to obtain a fourth order accurate approximation using the two second order formulas is then easily accomplished through the following procedure:

1. compute $N(h)$, for some given h
2. compute $N(h/2)$
3. Combine the two using the formula $M = N(h) - \frac{1}{3}[N(h) - N(h/2)]$

> **Remark**
>
> Notice that using this approach wasn't specific to the second derivative formula. All we needed was an approximation to some quantity where we knew the error term. Therefore we could use this same approach anytime we can write an approximation as:
>
> $$M = N(h) + K_1 h + K_2 h^2 + K_3 h^3 + \dots .$$
>
> The trick is then to find the right combination to cancel out the leading error term, when evaluating the equation at two different points, e.g. h and $h/2$.

There are advantages and disadvantages to Richardson extrapolation as for all numerical methods.

On the plus side, it is simple and general so we can apply the technique to many different problems. You will find applications in interpolation, numerical integration, and even differential equations. A great survey paper that describes many of these and others is

It also leads to formulas for higher-order approximations for derivatives, which are useful in certain applications. On the other hand, the technique requires more points at which to evaluate the function and it makes an assumption that the higher-order derivatives are nicely bounded, which may or may not hold true.

21.2 Stability

Sometimes an algorithm will fail to yield a good result due to stability. Computing derivatives by using finite differences is a prime example of such a possibility. This is due to the properties of computer arithmetic. In practice there is a delicate balance between truncation error and roundoff error. As a result one needs to be careful when choosing the ***step size***, h.

We briefly looked at an example in Section 3.1 where we looked at the accuracy of the forward difference approximation as a function of h. Let's look at a similar example, but this time using both forward and central differences.

Example 21.1. Let's take $f(x) = \cos(x)$ and $h = 10^{-3} - 10^{-15}, x = \pi/6$

Let's try to analyze what is happening there.

Consider the central difference formula for $f'(x)$ at some point x_0.

$$f'(x_0) = \frac{f(x_0+h) - f(x_0-h)}{2h} - \frac{h^2}{6} f^{(3)}(\xi(x))$$

When evaluating a function on a computer, we know that we do not have infinite precision, hence suppose:

$$f(x_0+h) = \hat{f}(x_0+h) + e(x_0+h)$$
$$f(x_0-h) = \hat{f}(x_0-h) + e(x_0-h)$$

where $e(x)$ is the roundoff error as a result of computing $f(x)$ and $\hat{f}(x)$ is the computed value of the function.

The error in the finite difference approximation can then be written as:

$$f'(x_0) - \left(\frac{f(x_0+h) - f(x_0-h)}{2h} \right) = \frac{e(x_0+h) - e(x_0-h)}{2h} - \frac{h^2}{6} f^{(3)}(\xi(x))$$

Now assume that:

$$e(x_0 \pm h) < \tau, \quad \tau > 0,$$
$$f^{(3)}(\xi) < M \quad \xi \in [x_0 - h, x_0 + h].$$

The error in the finite difference approximation can be bounded by

$$\left| \frac{\tau}{h} + \frac{h^2}{6} M \right|$$

Remark: *As $h \to 0$ the second term goes to 0, but the first term blows up.*

We need to find the "sweet spot" to minimize the error in the approximation.

Can show that the optimal h^* is given by:

$$h^* = \sqrt[3]{\frac{3\tau}{M}}$$

Unfortunately, one rarely knows either ϵ or M.

However a good rule of thumb is to choose h so that you only perturb half the digits of x. That suggests

$$h^* = \sqrt{\epsilon}$$

where ϵ is called machine precision. On most modern computers $\epsilon = 10^{-16}$, so this translates into

$$h \approx 10^{-8}$$

> **Caution**
>
> Finite Difference approximations are an example of unstable algorithms.

21.3 Summary

Some of the key takeaways

- Taylor series can be used to generate finite difference approximations to first derivatives, second derivatives, etc.
- We can also use Lagrange polynomials to generate similar formulas. (covered in book as well as in the optional lecture in Section 20.1).
- Richardson extrapolation is a simple and general approach that combines lower-order formulas to generate higher-order approximations (at additional function evaluation costs).
- ***Numerical differentiation is inherently unstable*** - but it's also (essentially) the only game in town. Care must be taken in choosing good step sizes.

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

Part V

Numerical Integration (Quadrature)

Introduction

This section covers methods for numerically evaluating definite integrals of the form

$$I(f) = \int_a^b f(x)dx,$$

where we do not have an explicit antiderivative or the antiderivative might not be easy to find or obtain. Numerical integration arises in numerous science and engineering (S&E) applications. These methods are most useful when the function doesn't have explicit antiderivative or doesn't have a closed form solution.

For example, think about

$$\int_0^\pi x^\pi sin(\sqrt{}(x))dx,$$

or

$$\int_{-1}^1 \exp(x)T_2(x)dx, \quad T_2 \text{ is a Chebyshev polynomial.}$$

In many real-world problems the integrals are in one, two or three dimensions. Here we will assume that our functions are scalar, that is $f : R^1 \to R^1$ and with as many derivatives as we need.

We will note in passing, that there are many important cases for which the dimension of the problem can be quite large. These situations require special methods and we will not be covering them here, but see the references for further information.

Other special cases involve integrals where one or both of the limits aren't definite, which is to say that either a or b could be infinite. These cases can sometimes be handled with the methods we will discuss here, but one should be careful when approaching the solution to these types of problems.

Finally, some functions are highly oscillatory. Again special care must be taken when trying to solve these types of problems. Here we will only point to several references that discuss these types of problems.

Practical Applications

- Statistics - probability distribution, stochastic equations, Bayesian statistics, etc.
- Integral transforms - uses in differential equations, signal processing, Fourier analysis, etc.
- Finite element methods - pdes
- Boundary Integral methods for solving pdes

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

22 Numerical Integration Basics

22.1 Introduction

Let's start with a bit of background. What is the first thing you think of when you see:

$$\int_a^b f(x)dx \tag{22.1}$$

If you said, the area under the curve, you're right! This leads us to the **general strategy**:

Approximate

$$I(f) = \int_a^b f(x)dx \approx \sum_{i=0}^{n} a_i f(x_i)$$

for some yet to be determined coefficients a_i.

We will call this *numerical quadrature*.

To do this we will follow the same strategy we used for numerical differentiation, i.e. we will replace the function whose integral we seek with one whose integral can be more easily evaluated – an *interpolating polynomial*.

> Approach
>
> Our overall goal is to approximate the integral Equation 22.1 by computing $\sum_{i=0}^{n} a_i f(x_i)$ through the following 3 steps:
>
> 1. Write $f(x)$ as an interpolating polynomial
> 2. Integrate the polynomial
> 3. Understand/analyze the truncation error

Before we start, let's first develop some intuition on what we're doing. Consider Figure 22.1 in which we've plotted a generic function. A natural idea is to use the well-known trapezoidal rule to approximate the area under the curve.

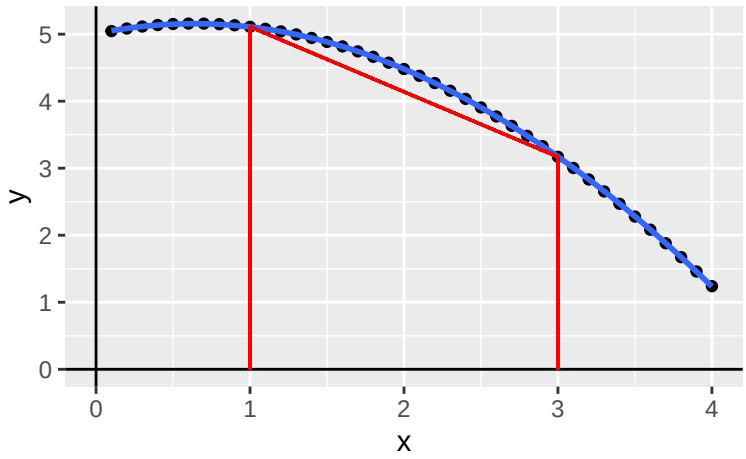


Figure 22.1: Trapezoidal Rule.

If we do this, it would make sense to approximate the integral as:

$$\int_1^3 f(x)dx \approx \frac{h}{2}[f(x_0) + f(x_1)], \; x_0 = 1, x_1 = 3,$$

where h is defined as the interval width, i.e. $h = b - a = 3 - 1$. Notice also, that in Figure 22.1 all that we did was to approximate the function by using a linear approximation using the two endpoints. It is natural to conjecture for what type of functions would the trapezoidal rule be exact for? Can you guess?

Remark

In order to get a more accurate approximation, we could subdivide the total region into smaller trapezoids and sum over all of them. All of this is by way of developing some intuition on what we should do. To make this more rigorous we will need to develop our framework and compute error estimates for our approximations. We will return to this idea in Section 24.1 *on* **Composite Integration**

22.2 Interpolating Polynomials

Step 1. Write down our function as a Lagrange interpolating polynomial along with its truncation error.

Let

$$f(x) = \sum_{i=0}^{n} f(x_i)L_i(x) + \prod_{i=0}^{n}(x - x_i)\frac{f^{(n+1)}(\xi(x))}{(n+1)!}. \tag{22.2}$$

For convenience, let's denote

$$\Psi_n(x) = \prod_{i=0}^{n}(x - x_i).$$

We can then write Equation 22.2 as:

$$f(x) = \sum_{i=0}^{n} f(x_i)L_i(x) + \Psi_n(x)\ \frac{f^{(n+1)}(\xi(x))}{(n+1)!}.$$

Step 2. Integrate the interpolating polynomial:

$$\int_a^b f(x) = \int_a^b \sum_{i=0}^{n} f(x_i)L_i(x) + \int_a^b \Psi_n(x)\ \frac{f^{(n+1)}(\xi(x))}{(n+1)!}.$$

Notice we can move the integral under the sum for the first term. Now (again for convenience) let's denote

$$c_i = \int_a^b L_i(x)dx,\ \ i = 0, \ldots, n. \tag{22.3}$$

Rearranging we get:

$$\int_a^b f(x) = \sum_{i=0}^{n} c_i f(x_i) + E(f),$$

where we denote the truncation error $E(f)$ by:

$$E(f) = \frac{1}{(n+1)!}\int_a^b \Psi_n(x)\ f^{(n+1)}(\xi(x))dx. \tag{22.4}$$

Notice we are partway to our goal of having written down the integral in the form we wanted:

$$I(f) = \int_a^b f(x)dx \approx \sum_{i=0}^{n} a_i f(x_i).$$

Remark

The a_i and c_i (Equation 22.3) *are not the same, but they are related in some yet to be determined way!*
We're now left with only two steps:

2. Compute c_i for a specific interpolating polynomial, and
3. Understand/analyze the error function $E(f)$

Specific Case: $n = 1$ (Linear Interpolating Polynomial)

Let's take the easiest case, $n = 1$, a linear Lagrange interpolating polynomial. To be consistent with our earlier notation we'll also let $a = x_0$ and $b = x_n = x_1$ for this case.

Recall, the first degree Lagrange polynomial takes the form:

$$P_1(x) = \frac{(x - x_1)}{x_0 - x_1} f(x_0) + \frac{(x - x_0)}{x_1 - x_0} f(x_1), \tag{22.5}$$

and the truncation error Equation 22.4 reduces to

$$E(f) = \frac{1}{2} \int_{x_0}^{x_1} (x - x_0)(x - x_1) f''(\xi(x)) dx. \tag{22.6}$$

Let's consider the truncation error first.

It would be nice if we could take $f''(\xi(x))$ term outside the integral to simplify the integral. Let's first define $g(x) = (x - x_0)(x - x_1)$, and notice that $g(x)$ doesn't change sign on $[x_0, x_1]$.

That means we can apply the Weighted Mean Value Theorem (WMVT) for integrals and pull the $f''(\xi(x))$ term outside of the integral.

Note

Weighted Mean Value Theorem for Integrals (): Suppose that $f \in C[a, b]$, the Riemann integral of g exists on $[a, b]$, and $g(x)$ does not change sign on $[a, b]$. Then there exists a number $c \in (a, b)$ such that

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

That simplifies $E(f)$ so that

$$E(f) = \frac{1}{2} f''(\xi) \int_{x_0}^{x_1} (x - x_0)(x - x_1) dx,$$

for some $\xi \in [x_0, x_1]$.

That just leaves us with integrating the quadratic inside the integral, which reduces to:

$$E(f) = \frac{1}{2} f''(\xi) \left[\frac{x^3}{3} - \frac{(x_1 + x_0)}{2} x^2 + x_0 x_1 x \right]_{x_0}^{x_1}. \tag{22.7}$$

To simplify the calculations, let's first do a change of variable, $x' = x - x_0$. Also recall that $x_1 = x_0 + h$. As a result, the limits reduce to $x_0 \to 0, x_1 \to h$, and as a result, the term in brackets in Equation 22.7 evaluates to:

$$\left[\frac{h^3}{3} - \frac{(h)h^2}{2} + 0 \right] = \frac{-h^3}{6}.$$

That simplifies $E(f)$ to

$$E(f) = -\frac{h^3 f''(\xi)}{12}. \tag{22.8}$$

That takes care of **Step 3 - Understand/Analyze the Truncation Error $E(f)$!**

To reach our final goal, we just need to complete the second step we need to compute the integrals of Equation 22.5, i.e.

$$\begin{aligned} \int_{x_0}^{x_1} P_1(x) &= \int_{x_0}^{x_1} \frac{(x - x_1)}{x_0 - x_1} f(x_0) + \frac{(x - x_0)}{x_1 - x_0} f(x_1), \\ &= \left[\frac{(x - x_1)^2}{2(x_0 - x_1)} f(x_0) + \frac{(x - x_0)^2}{2(x_1 - x_0)} f(x_1) \right]_{x_0}^{x_1}. \end{aligned}$$

Notice that for the upper value x_1 the first term in the sum drops out, and likewise for the lower value x_0 the second term drops out, leaving only 2 terms.

Evaluating these two terms we get:

$$\begin{aligned}
\int_{x_0}^{x_1} P_1(x) &= \left[\frac{(x_1-x_0)^2}{2(x_1-x_0)}f(x_1) - \frac{(x_0-x_1)^2}{2(x_0-x_1)}f(x_0)\right], \\
&= \left[\frac{(x_1-x_0)}{2}f(x_1) - \frac{(x_0-x_1)}{2}f(x_0)\right], \\
&= \left[\frac{(x_1-x_0)}{2}f(x_1) + \frac{(x_1-x_0)}{2}f(x_0)\right], \\
&= \left(\frac{x_1-x_0}{2}\right)[f(x_1)+f(x_0)], \\
&= \frac{h}{2}[f(x_1)+f(x_0)],
\end{aligned}$$

where the last line is due to the fact that $x_1 = x_0 + h$.

Pulling everything together leads us to our desired result:

Trapezoidal Rule

$$\int_a^b f(x)dx = \frac{h}{2}[f(x_0)+f(x_1)] - \frac{h^3}{12}f''(\xi),$$

Trapezoidal Rule exact for polynomials of degree $\leq$ 1.
Since the truncation error is given by $\frac{h^3}{12}f''(\xi)$ we expect that for any function whose second derivative is identical to zero that the Trapezoidal Rule will be *exact*, and in particular for any polynomial of degree 1 or less.

22.3 Simpson's Rule

i Optional

The derivation for Simpson's Rule follows the one for the Trapezoidal Rule, with some minor modifications. The following is included for completeness (and for practice), but you may also want to skip down to the final formula (Equation 22.13) and discussion of the major properties of Simpson's Rule.

In a similar fashion to our approach for deriving the Trapezoid Rule, if we integrate the second-degree Lagrange polynomial, we can derive Simpson's Rule.

Conjecture

1. For what degree polynomials will Simpson's rule be exact?
2. What do you think the order of the truncation error will be for Simpson's method?

22.3.1 Setting up the integral of the interpolating polynomial.

As before, we will first approximate the integrand by an interpolating polynomial. In this case we will take 3 equally spaced points x_0, x_1, x_2 such that $x_0 = a, x_1 = x_0 + h, x_2 = b$, where $h = (b-a)/2$.

We assumed that $f(x)$ had as many derivatives as we needed. This time let's write the Taylor expansion for $f(x)$ about x_1 going out to the 4th derivative term. The reason for going out to the 4th derivative will become clear in a minute.

$$\begin{aligned} f(x) = f(x_1) + f'(x_1)(x-x_1) + \frac{1}{2}f''(x_1)(x-x_1)^2 \\ + \frac{1}{6}f'''(x_1)(x-x_1)^3 + \frac{1}{24}f^{(4)}(\xi), \qquad \xi \in [x_0, x_2] \end{aligned}$$

Now let's integrate this equation:

$$\begin{aligned} \int_{x_0}^{x_2} f(x) = \Big[f(x_1)(x-x_1) + \frac{1}{2}f'(x_1)(x-x_1)^2 \\ + \frac{1}{6}f''(x_1)(x-x_1)^3 + \frac{1}{24}f'''(x_1)(x-x_1)^4 \Big]_{x_0}^{x_2} \\ + \frac{1}{24}\int_{x_0}^{x_2} f^{(4)}(\xi(x))(x-x_1)^4 dx. \end{aligned} \tag{22.9}$$

As before let's consider the error term first and notice that it would be nice to take the $f^{(4)}(\xi(x))$ in the last term outside of the integral.

Using the previous trick, we note that $(x-x_1)^4$ doesn't change sign in the interval $[x_0, x_1]$, so we can again use the Weighted Mean Value Theorem for Integrals to pull the $f^{(4)}$ term out from inside the integral.

$$\begin{aligned} E(f) &= \frac{1}{24}\int_{x_0}^{x_2} f^{(4)}(\xi(x))(x-x_1)^4 dx, \\ &= \frac{f^{(4)}(\xi_1)}{24}\int_{x_0}^{x_2} (x-x_1)^4 dx, \\ &= \frac{f^{(4)}(\xi_1)}{120}(x-x_1)^5\Big|_{x_0}^{x_2}. \end{aligned}$$

for some $\xi_1 \in [x_0, x_2]$.

Now we can use the fact that $h = x_2 - x_1 = x_1 - x_0$ to reduce the equation to:

$$E(f) = \frac{h^5 f^{(4)}(\xi_1)}{60}. \tag{22.10}$$

22.3.2 Evaluating the integral of the interpolating polynomial.

Our only remaining task is to evaluate the first term in Equation 22.9 to arrive at formulas for the $a_i(x)$ coefficients and produce the approximation for $I(f)$. :

$$\Big[f(x_1)(x-x_1) + \frac{1}{2}f'(x_1)(x-x_1)^2 + \frac{1}{6}f''(x_1)(x-x_1)^3 + \frac{1}{24}f'''(x_1)(x-x_1)^4\Big]_{x_0}^{x_2}$$

> **Tip:**
>
> When faced with a daunting amount of algebra (recall our experience with the Trapezoid Rule), it helps to remember both the goal and the assumptions we've made that might help simplify the task. For example, recall our assumption of equally spaced nodes. We can replace $x_2 - x_1 = x_1 - x_0$ by h – which reduces our formula to:

$$\Big[f(x_1)h + \frac{1}{2}f'(x_1)h^2 + \frac{1}{6}f''(x_1)h^3 + \frac{1}{24}f'''(x_1)h^4\Big] - \Big[f(x_1)(-h) + \frac{1}{2}f'(x_1)h^2 + \frac{1}{6}f''(x_1)(-h)^3 + \frac{1}{24}f'''(x_1)h^4\Big].$$

Nicely, the h^2 and h^4 terms cancel out, leaving us with

$$2hf(x_1) + \frac{h^3}{3}f''(x_1). \tag{22.11}$$

> **Remark:**
>
> We could just as easily have noticed that (under the assumption of equally spaced nodes), $(x_2 - x_1)^k - (x_0 - x_1)^k = 0$ for even k and that $(x_2 - x_1)^k - (x_0 - x_1)^k = 2h^k$ for odd k.

Now remember our goal was to write our approximation in terms of only $f(x_i)$, meaning we should look for a way to replace the second derivative term in Equation 22.11.

For this task, we can take advantage of one last substitution, which is to use our finite difference approximation for the second derivative (derived in our previous lecture), i.e.

$$f''(x_1) = \frac{f(x_0) - 2f(x_1) + f(x_2)}{h^2} - \frac{h^2}{12} f^{(4)}(\xi_2)$$

and not forgetting to include its own error term.

When substituted into Equation 22.11 we get:

$$2hf(x_1) + \frac{h^3}{3}\left[\frac{f(x_0) - 2f(x_1) + f(x_2)}{h^2} - \frac{h^2}{12} f^{(4)}(\xi_2)\right] \tag{22.12}$$

Combining Equation 22.12 with Equation 22.10 and simplifying terms we arrive at our final result, which is called Simpson's Rule:

Simpson's Rule

$$\int_{x_0}^{x_2} f(x)dx = \frac{h}{3}[f(x_0) + 4f(x_1) + f(x_2)] - \frac{h^5}{90} f^{(4)}(\xi). \tag{22.13}$$

Caution: Other formulas use $\frac{b-a}{6}$ instead of $\frac{h}{3}$ in the definition of Simpson's Rule. Just remember that here, we defined $h = x_2 - x_1 = x_1 - x_0$, so $h = \frac{b-a}{2}$, as a result the two definitions are equivalent.

I leave as an exercise how to combine the two $f^{(4)}$ terms from Equation 22.10 and Equation 22.12 into one, i.e. show that

$$-\frac{h^5}{60} f^{(4)}(\xi_1) - \frac{h^5}{36} f^{(4)}(\xi_2) = -\frac{h^5}{90} f^{(4)}(\xi), \quad \xi \in (a, b)$$

Remark: In some formulations, the leading term in Simpson's Rule is $h/3$ and sometimes it's $(b-a)/6$. This is a result of how one defines h. Here we set it equal to $x_2 - x_1 = x_1 - x_0$, hence $h = (b-a)/2$.

An important result is that instead of the expected $O(h^4)$ error term, we might have expected from going from a linear interpolant to a quadratic interpolant we have instead gained an additional order of accuracy in the error term!

Definition 22.1. The ***precision*** (also degree of accuracy) of a quadrature formula is defined as the largest positive integer n such that the quadrature formula is exact for x^k, for $k = 0, 1, \ldots, n$.

Tip

In the case of Simpson's rule, it is exact for any polynomial of degree 3 or less, hence the precision is 3. Similarly, the precision for the Trapezoid rule is 1. The easiest way to remember this is to take a look at the derivative in the error term and subtract one order. Please do not confuse this with the order of accuracy, which can be seen from the power in the h term!

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

23 Newton-Cotes Methods

23.1 Newton-Cotes

The basic quadrature rules derived so far are generally good, but what if we wanted to have formulas with greater accurarcy. The general approach we used still holds and leads to a family of quadrature formulas known as ***Newton-Cotes*** formulas. These are classified under either open or closed depending on whether the formulas include the end points or not.

Closed Newton-Cotes include the endpoints of closed interval $[a, b]$ as nodes.

Open Newton-Cotes do not include the endpoints.

To be specific, for a closed Newton-Cotes quadrature formula we would choose the node points x_i through the formula:

$$x_i = a + i\frac{b-a}{n-1}, \quad i = 0, 1, \ldots, n-1. \tag{23.1}$$

For an open Newton-Cotes quadrature formula we would use the formula:

$$x_i = a + (i+1)\frac{b-a}{n+1}, \quad i = 0, 1, \ldots, n-1. \tag{23.2}$$

Example 23.1. Suppose, we choose $n = 5$ on the interval [a,b] = [0,1].

Then the closed Newton-Cotes formula would generate the points:

$$\begin{aligned} x_i &= a + i \ \cdot \frac{b-a}{n-1}, \\ &= 0 + i\frac{1}{4}, \\ &= \frac{i}{4}, \quad i = 0, 1, \ldots, 4, \end{aligned}$$

thereby yielding the set of nodes: $\{x\} = \{\ 0, .25, .5, .75, 1.0\ \}$.

Similarly for the open Newton-Cotes formula would generate the points:

$$
\begin{aligned}
x_i &= a + (i+1) \cdot \frac{b-a}{n+1}, \\
&= 0 + (i+1)\frac{1}{6}, \\
&= \frac{i+1}{6}, \quad i = 0, 1, \ldots, 4,
\end{aligned}
$$

which generates the set of nodes: $\{x\} = \{\ 1/6,\ 2/6,\ 3/6,\ 4/6,\ 5/6\ \}$.

One example of an Open Newton-Cotes that we've already seen is the midpoint rule

$$\int_a^b f(x)dx = 2hf(x_0) + \frac{h^3}{3}f''(\xi)\ \xi \in (a,b)]$$

where x_0 is the midpoint between a and b. Likewise, both Trapezoidal and Simpson's rules, which we introduced in Section 22.1 can be categorized as Closed Newton-Cotes.

There are many different formulas of both the Closed and Open variety all with corresponding error terms. All of them can be derived by the methods we've used for Trapezoid and Simpson's rule, so there is little to be gained by re-deriving them.

Instead we will present them here because an interesting pattern arises that is worth knowing about:

23.1.1 Closed Newton-Cotes formulas:

$n = 2$ (Trapezoid)

$$I(f) = \frac{b-a}{2}[f(x_0) + f(x_1)] \tag{23.3}$$

$n = 3$ (Simpson's)

$$I(f) = \frac{b-a}{6}[f(x_0) + 4f(x_1) + f(x_2)] \tag{23.4}$$

$n = 4$ (Simpson's 3/8)

$$I(f) = \frac{b-a}{8}[f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)]$$

$n = 5$ (Boole's rule)

$$I(f) = \frac{b-a}{90}[7f(x_0) + 32f(x_1) + 12f(x_2) + 32f(x_3) + 7f(x_4)]$$

Notation

The formulas here are written using $b-a$ versus h to make them easier to compare. However, you will see these formulas written in terms of h in many other places. You should be careful in understanding exactly what h represents as it often is taken to mean $h=(b-a)/(n-1),\ n\geq 1$, which corresponds to the number of node points used in the qudrature formula.

In theory, we could go as high as we wanted (and people have) in generating higher-order quadrature formulas, and of course with additional computational work. However, for large n the formulas can be shown to become numerically unstable ($n\geq 11$.) One can actually prove that formulats do not converge for all integrands that are analytic. In practice, we tend to only use low-order formulas since they can still give us good accuracy (especially over small intervals (see Exercise 23.1 below).

23.1.2 Open Newton-Cotes formulas:

$n=1$ (Midpoint)

$$I(f)=(b-a)f(x_0)$$

$n=2$

$$I(f)=\frac{b-a}{2}[f(x_0)+f(x_1)]$$

$n=3$

$$I(f)=\frac{b-a}{3}[2f(x_0)-f(x_1)+2f(x_2)]$$

Similarly to the closed Newton-Cotes formulas, we could continue and derive higher-order formulas - with the same consequences.

23.2 Error Estimates

In both the closed and open Newton-Cotes cases, the formulas have error terms, which we have summarized in the table below, along with the precision of each:

Table 23.1: Summary of Error Terms for Newton-Cotes quadrature formulas

Name	Npts	Error	Precision
Trapezoid	2	$-\frac{(b-a)^3}{12}f^{(2)}(\xi)$	1
Simpson's	3	$-\frac{(b-a)^5}{2880}f^{(4)}(\xi)$	3
Simpson's 3/8	4	$-\frac{(b-a)^5}{6480}f^{(4)}(\xi)$	3
Boole	5	$-\frac{(b-a)^7}{1935360}f^{(6)}(\xi)$	5
Midpoint	1	$\frac{(b-a)^3}{24}f^{(2)}(\xi)$	1
	2	$\frac{(b-a)^3}{36}f^{(2)}(\xi)$	1
	3	$\frac{(b-a)^5}{23040}f^{(4)}(\xi)$	3

Important

An interesting feature of the quadrature formulas is that whenever N is odd then the precision of the formula $= N$. But when N is even then the precision is only $N-1$. We lose one order in the precision whenever N is even! Or we could also say that we gain one order of precision for N odd.

As a final look into these methods, let's compare several of the methods on a simple function to gain further insight into the behavior of the formulas.

Exercise 23.1. Compute the value of

$$\int_0^1 e^x dx$$

using the Trapezoid and Simpson's Rule for:

1. a = 0, b = 1
2. a = 0.9, b = 1

Trapezoid Rule:

$$\int_a^b f(x)dx = \frac{(b-a)}{2}\Big[f(x_0) + f(x_1)\Big],$$

Figure 23.1: Solution

Simpson's Rule:

$$\int_a^b f(x)dx = \frac{(b-a)}{6}[f(x_0) + 4f(x_1) + f(x_2)]$$

Figure 23.2: Solution

The lesson from this example is that all of the formulas have a rather large error when we compute the integral over a large interval, whereas when we considered a smaller interval, the error was in fact quite small.

23.3 Summary

Let's take a step back and summarize the main results:

- We can use a simple approach towards deriving basic quadrature rules by replacing the integrand with an interpolating polynomial and integrating the polynomial on a chosen set of N points.
- Using Taylor's theorem, we can also generate corresponding error terms that can provide us with estimates on how well the quadrature formula approximated the given integral.
- The precision of a quadrature formula is the highest degree of the polynomial for which the formula is exact. When N is odd, the precision is also N; but when N is even, the precision is $N - 1$.
- Higher-order quadrature formulas yield greater accuracy, but at greater additional computational work as well as a fundamental assumption on the higher-order derivatives being nicely behaved (i.e. bounded).
- Basic (and low-order) formulas can be quite accurate, but usually require a small interval. This observation will prove useful in the next sections.

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

24 Composite Quadrature Rules

24.1 Composite Integration

In practice we can't use Newton-Cotes over large intervals as it would require high degree polynomials, which would be unsuitable due to the highly oscillatory nature. Another disadvantage is that we would need to have equally spaced intervals, which are not suitable for many physical applications.

Idea

Instead of trying to approximate the integral accurately over the entire interval with one polynomial, break up the domain into smaller regions and use low-order polynomials on each of them.

Divide and Conquer (Redux)

The idea behind breaking the original problem into several smaller problems is used in many situations and is often referred to as a divide-and-conquer strategy. Recall, that we used a similar approach in Section 18.1 when we chose to break up the problem of interpolation via piecewise functions over a set of sub-intervals instead of one single interpolating polynomial.

The strategy is to use something simple like Trapezoid or Simpson's Rule on each of the subregions (often called ***panels***), and then sum up the individual contributions to arrive at the solution to the original problem. As a quick note, there is nothing in this approach that directs us to use subregions of equal size, but for exposition, we will assume that they are for the time being. We will come back to this point in the next section on adaptive methods Section 25.1.

In the general case, let's assume that we have sub-divided the interval $[a, b]$ Into r subregions, each of equal length $h = \frac{b-a}{r}$. See Figure 24.1 for the case $r = 5$.

Then our composite quadrature rule could be written as:

$$\int_a^b f(x)dx = \sum_{i=1}^{r} \int_{t_{i-1}}^{t_i} f(x)dx,$$

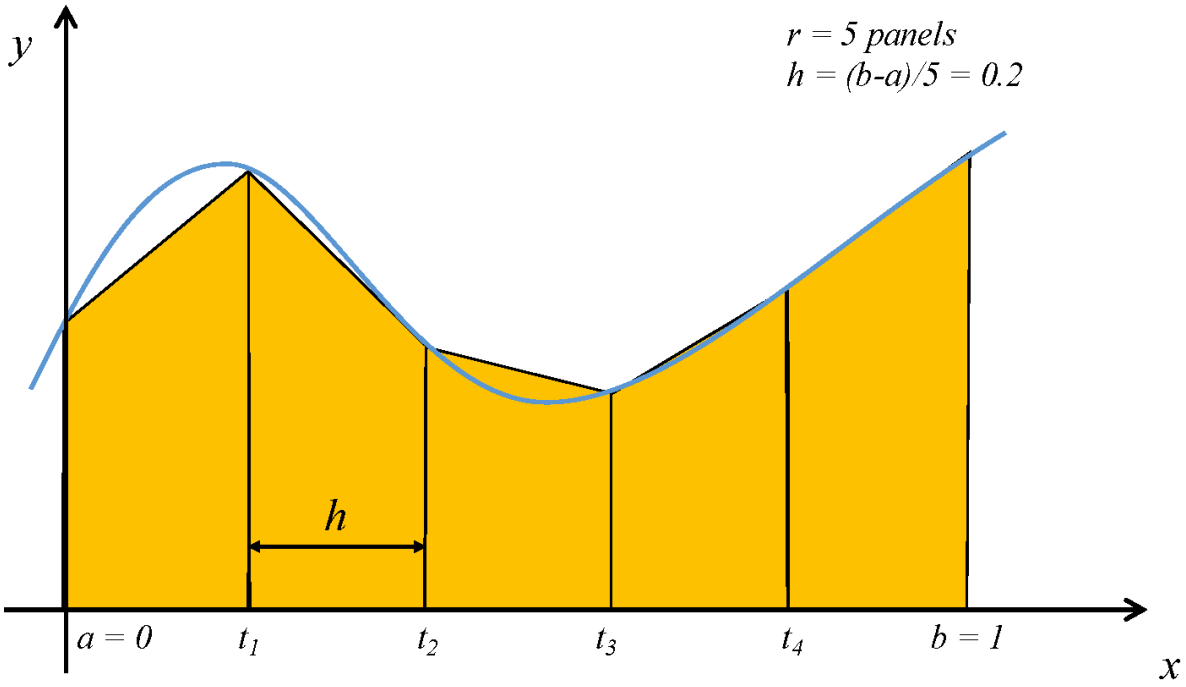


Figure 24.1: Approximating the integral by sudviding the region into multiple "panels" and using the trapezoid rule will lead to higher accuracy

where $t_i = a + ih$. All we need do now is to apply one of our earlier quadrature formulas to the integrals in each of the subintervals $[t_{i-1}, t_i]$.

Let's work out an example in the simple case of the Trapezoid Rule:

$$\begin{aligned}
\int_a^b f(x)dx &\approx \sum_{i=1}^{r} \int_{t_{i-1}}^{t_i} f(x)dx, \\
&\approx \sum_{i=1}^{r} \frac{h}{2}\left[f(t_{i-1}) + f(t_i)\right], \\
&= \frac{h}{2}\left[f(t_0) + 2f(t_1) + 2f(t_2) + ... + 2f(t_{r-1}) + f(t_r)\right], \\
&= \frac{h}{2}\left[f(a) + 2f(t_1) + 2f(t_2) + ... + 2f(t_{r-1}) + f(b)\right],
\end{aligned}$$

This leads to the Composite Trapezoid Rule:

Composite Trapezoidal Rule

$$\int_a^b f(x)dx = \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{r-1} f(t_i) + f(b)\right] - \frac{(b-a)}{12}h^2 f''(\mu), \tag{24.1}$$

where $h = (b-a)/r$; $t_i = a + ih,\ i = 0, 1, ..., r$ and $\mu \in (a, b)$.

A similar exercise will lead to the Composite Simpson's Rule:

Composite Simpson's Rule

$$\begin{aligned}
\int_{x_0}^{x_2} f(x)dx = \frac{h}{3}[f(a) + 2\sum_{i=1}^{r/2-1} f(t_{2i}) + 4\sum_{i=1}^{r/2} f(t_{2i-1}) + f(b)] \\
- \frac{(b-a)}{180}h^4 f^{(4)}(\mu).
\end{aligned} \tag{24.2}$$

where $h = (b-a)/r$; $t_i = a + ih,\ i = 0, 1, ..., r$ and $\mu \in (a, b)$.

The derivation for Composite Simpson's rule isn't difficult, and mostly a matter of getting the right indices. One observation that is helpful in deriving the formula is that because of the need for 3 nodes in Simpson's rule we should consider the panels in pairs. For this reason, the number of panels for Simpson's rule must always be an even number.

Example: Let's consider the simplest case with 4 panels, which we can group into two pairs, as depicted in the figure below. Here you should think of the first integration region spanning the first two panels so as to encompass the area in the interval $[t_0, t_2]$. Likewise, the second region will cover the interval $[t_2, t_4]$.

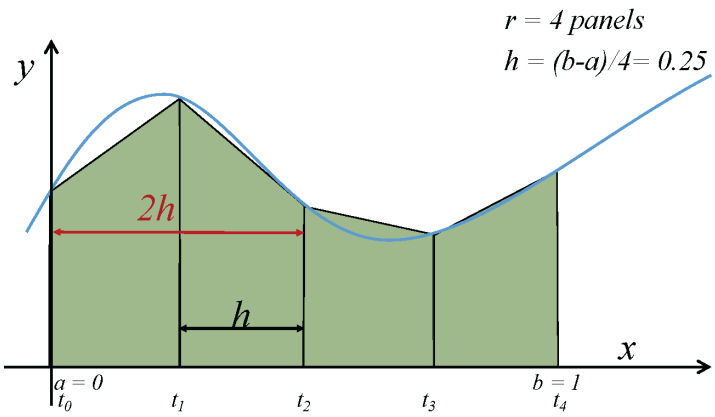


Figure 24.2

Recall Simpson's rule (Equation 23.4),

$$I(f) = \frac{b-a}{6}[f(x_0) + 4f(x_1) + f(x_2)],$$

which we can apply to each one of the two sub-intervals, $[t_0, t_2]$ and $[t_2, t_4]$. Let's call the integrals over the two sub-intervals, S_1 and S_2 :

$$S_1 = \frac{t_2 - t_0}{6}[f(t_0) + 4f(t_1) + f(t_2)],$$
$$S_2 = \frac{t_4 - t_2}{6}[f(t_2) + 4f(t_3) + f(t_4)].$$

The total integral is then just the sum of these two. Here we need to notice that each sub-interval is of length $2h$, allowing us to also simplify the sums a bit.

$$\begin{aligned} I(f) &= S_1 + S_2 \\ &= \frac{h}{3}[f(t_0) + 4f(t_1) + f(t_2)] + \\ &\quad \frac{h}{3}[f(t_2) + 4f(t_3) + f(t_4)]. \end{aligned}$$

Now, rearraning the terms we get:

$$\begin{aligned} I_{Simp}(f) &= \frac{h}{3}[f(t_0) + 4f(t_1) + 2f(t_2) + 4f(t_3) + f(t_4)], \\ &= \frac{h}{3}[f(t_0) + 4[f(t_1) + f(t_3)] + 2f(t_2) + f(t_4)]. \end{aligned}$$

It should be clear how the general pattern follows.

> **!** Important
>
> *An immediate consequence of this approach is that* ***the quadrature formulas lose one order of*** h ***in their approximations.*** *This should not be surprising as even though each individual panel has the original truncation error, when we add up all the panels, the errors from each of the individual panels add up and we lose one order of magnitude. However, since* h *is smaller, the overall result is a win!*

24.2 Error Analysis and Stability

While we have a fairly good handle on the error analysis of quadrature formulas and the order of convergence, one question we haven't addressed yet is the stability of quadrature formulas.

First, we know that the truncation error is well-behaved and will go to zero at varying degrees depending on the degree of the interpolating polynomial we use. What caused problems before was that roundoff error could increase dramatically leading to unstable algorithms.

Let's proceed as before, by performing a floating point error analysis similar to the one we used for numerical differentiation. Recall, that we first assumed that a computation of a function value always incurs some roundoff error, in other words, we can write:

$$f(t_i) = \hat{f}(t_i) + e_i, \ i = 0, 1, \ldots, n,$$

where $\hat{f}$ is the floating point representation and e_i is the roundoff error incurred when computing the function value. As before, we will assume that the errors are uniformly bounded:

$$e_i < \epsilon, \quad \epsilon > 0.$$

for all i.

If we substitute into the Composite Simpson's rule we can write the error as:

$$\begin{aligned} |e(h)| &= \Big|\frac{h}{3}(e_0 + 2\sum_{i=1}^{r/2-1} e_{2i} + 4\sum_{i=1}^{r/2} e_{2i-1} + e_r)\Big|, \\ &\leq \frac{h}{3}(\epsilon + 2(\frac{r}{2} - 1)\epsilon + 4(\frac{r}{2})\epsilon + \epsilon), \\ &\leq \frac{h}{3}(3r\epsilon) = rh\epsilon. \end{aligned}$$

Finally, let's remember that $h = (b-a)/r$, which means that the error is bounded by:

$$|e(h)| \leq (b-a)\ \epsilon,$$

which is independent of both h and r.

> Important
>
> ***In other words, Simpson's rule is stable!***

Furthermore, it should be obvious that we can do the same analysis for any of the open or closed Newton-Cotes quadrature formulas. Unlike numerical differentiation, quadrature is generarlly more stable.

Exercise 24.1. Determine values of h that will ensure an approximation error of < 0.00002 when approximating

$$\int_0^{\pi} \sin x dx$$

using

1. Composite Trapezoidal Rule
2. Composite Simpson's Rule

Solution:

Composite Trapezoid. Our starting point is the formula for the truncation error. For composite Trapezoid we use Equation 24.1:

$$E(f) = -\frac{b-a}{12} h^2 f''(\xi).$$

We would like this term to be less than the tolerance $\epsilon = 2 \cdot 10^{-5}$

$$\begin{aligned} |E(f)| &= \left| \frac{\pi}{12} h^2 \sin(\xi) \right| \\ &= \frac{\pi}{12} h^2 \left| \sin(\xi) \right| \\ &\leq \frac{\pi h^2}{12} < 2 \cdot 10^{-5} \end{aligned}$$

with the last inequality because $|\sin(x)| \leq 1$.

Solving for h is then an easy matter:

$$
\begin{aligned}
\frac{\pi h^2}{12} &< 2 \cdot 10^{-5}, \\
h^2 &< \frac{24 \cdot 10^{-5}}{\pi}, \\
h &< \sqrt{\frac{24 \cdot 10^{-5}}{\pi}}. \\
\implies h &\approx 0.00874.
\end{aligned}
$$

To achieve the desired accuracy, would then require $r = (b-a)/h$ panels or $r \approx 360$ panels.

Composite Simpson. As before, our starting point is the formula for the truncation error. For composite Simpson we use Equation 24.2:

$$
E(f) = -\frac{b-a}{180} h^4 f^{(4)}(\xi).
$$

We would like this term to be less than the tolerance $\epsilon = 2 \cdot 10^{-5}$

$$
\begin{aligned}
|E(f)| &= \left| \frac{\pi}{12} h^4 \sin(\xi) \right|, \\
&= \frac{\pi}{12} h^4 \left| \sin(\xi) \right|, \\
&\leq \frac{\pi h^4}{180} < 2 \cdot 10^{-5},
\end{aligned}
$$

with the last inequality because $|\sin(x)| \leq 1$.

Solving for h:

$$
\begin{aligned}
\frac{\pi h^4}{180} &< 2 \cdot 10^{-5}, \\
h^4 &< \frac{360 \cdot 10^{-5}}{\pi}, \\
h &< \sqrt[4]{\frac{360 \cdot 10^{-5}}{\pi}}. \\
\implies h &\approx 0.18399
\end{aligned}
$$

To achieve the desired accuracy, would then require $r = (b-a)/h$ panels or $r \approx 18$ panels. For this example, it is clear that Composite Simpson is a much better choice than Composite Trapezoid.

24.3 Summary

There are several choices to be made when considering which quadrature method to use and what size of h to choose. Deciding between a lower order method versus a higher order method can also be tricky. The error term will depend on both the size of h as well as the magnitude of the highest derivative required. Choosing h will require (as in the case of numerical differentiation) a balance between truncation error and roundoff error.

Practical Tips

1. The smaller h is, the greater the accuracy for all methods; however that also implies we have more panels and hence a higher computational load.
2. If the function that is to be integrated is smooth then higher order methods are likely to work well.
3. However, if the the function is rapidly changing, then the higher derivatives will likely be large and one might want to consider lower order methods, especially if the integral interval is small.

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

25 Adaptive Quadrature

25.1 Adaptive Quadrature

Composite quadrature formulas can be quite effective as we discussed in the last section. There is one drawback however - so far we have only used a uniform spacing for the nodes. We did this mainly to simplify the analysis, and to highlight the main ideas. However, there was no underlying need to do so. In fact, there are many situations where it is clear that a uniform spacing might not be optimal. Consider for example, the following function:

$$f(x) = e^{-3x}\sin(4x),\ x \in [0, 10]$$

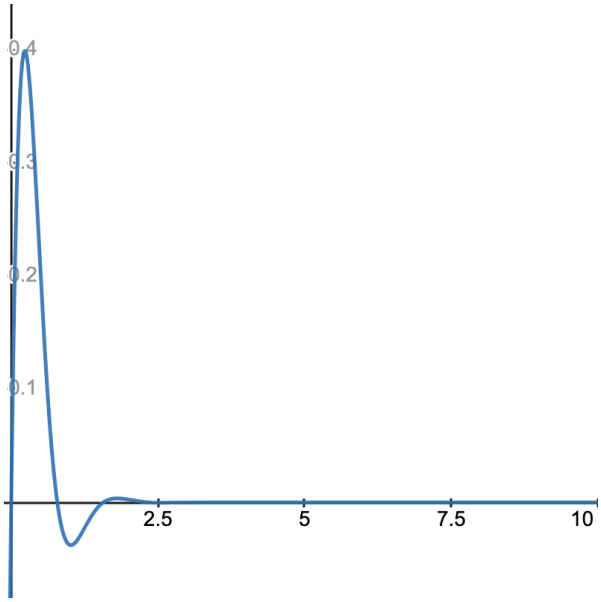


Figure 25.1

If we attempt to use a uniform spacing that tries to approximate the integral of this function, we will be caught between two competing interests. To capture the behavior of the function towards the right end of the interval a spacing of $h = 0.5$ or even $h = 1.0$ would likely be adequate. However, if we want to capture the behavior of the function towards the left end of

the interval, then it looks likely that we would need to have a spacing of h that is much smaller. And if we choose the smaller h then we will be committed to doing additional computational work that is not needed on the right end of the interval.

This problem can be exacerbated when working in two or three dimensions, where the computational work could increase dramatically, if we have to choose a uniform h in all of the dimensions. In the above example, suppose we had to choose one h for the entire region. The table below depicts the size of the problem in terms of the number of "cells" one would have to compute over the interval $[0, 10]$ with a uniform grid, and a nonuniform grid where the majority of the points are chosen in a small subinterval, say $x \in [0, 2]$. The difference isn't particularly noteworthy in one dimension but when you reach a three-dimensional problem, there is an additional factor of 1000 to consider when using a uniform grid.

Type	h	N	N^2	N^3
Uniform	0.1	100	$\approx 10^4$	$\approx 10^6$
	0.01	1000	$\approx 10^6$	$\approx 10^9$
Nonuniform	0.1	≈ 20	≈ 400	≈ 8000
	0.01	≈ 110	$\approx 10^4$	$\approx 10^6$

The solution is obvious, which is to find a value of h that is adapted to what the function is doing over a particular subinterval. But the big question is how do we know this without evaluating the function at many different points and how do we choose a good value of h that gives us good accuracy without also increasing the computational workload too much.

> Idea
>
> If we could predict the variation in the function, then we could choose a smaller h in only those regions that need it to attain the accuracy we want! Our strategy will be to leverage our error analysis to help us predict the variation.

Let's consider the Composite Trapezoid Rule first. Recall that we can write the quadrature formula as:

$$I(f) = \int_a^b f(x)dx = \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{r-1} f(t_i) + f(b)\right] - \frac{(b-a)}{12}h^2 f''(\mu),$$

Suppose we write the approximation to the integral as:

$$R_1 = \frac{h}{2}\Big[f(a) + 2f(a+h) + ... + 2f(b-h) + f(b)\Big] + Kh^2,$$

Now let's suppose we cut h in half and write the Trapezoid rule again:

$$R_2 = \frac{h}{4}\Big[f(a) + 2f(a+h/2) + ... + 2f(b-h/2) + f(b)\Big] + K\Big(\frac{h}{2}\Big)^2,$$

Let's now consider the error in each of these formulas:

$$\begin{aligned} I(f) - R_1 &\approx Kh^2, \\ I(f) - R_2 &\approx K\Big(\frac{h}{2}\Big)^2 = \frac{1}{4}Kh^2. \end{aligned}$$

The next step requires us to make an assumption, namely that the terms that constitute the constant K in both of the above terms are approximately equal. This should be true if the fourth derivative terms in each of the constants are comparable, which seems reasonable since they are from the same function and over similar intervals:

$$f^{(4)}(\xi_1) \approx f^{(4)}(\xi_2).$$

Substituting the first equation into the second we get:

$$I(f) - R_2 \approx \frac{1}{4}[I(f) - R_1]. \tag{25.1}$$

Let's now consider the error for R_1:

$$\begin{aligned} I(f) - R_1 &= (I(f) - R_2) + (R_2 - R_1), \\ &\approx \frac{1}{4}[I(f) - R_1] + (R_2 - R_1) \end{aligned} \tag{25.2}$$

$$\implies I(f) - R_1 \approx \frac{4}{3}(R_2 - R_1).$$

Finally, we can combine Equation 25.1 and Equation 25.2 to get:

$$\begin{aligned} I(f) - R_2 &\approx \frac{1}{4}[I(f) - R_1] \\ &\approx \frac{1}{4}\Big[\frac{4}{3}(R_2 - R_1)\Big] \end{aligned} \tag{25.3}$$

$$\implies I(f) - R_2 \approx \frac{1}{3}\Big[(R_2 - R_1)\Big]$$

The important thing to note is that once everything on the right hand sides (specifically R_1, R_2), have been computed, we can generate an estimate for the error in both quadrature approximations. These types of computations are known as ***a posteriori error*** estimates.

In a similar manner we can produce a posteriori error estimates for composite Simpson's rule and write:

Simpson *a posteriori* error estimates

$$I(f) - S_1 \approx \frac{16}{15}\Big(S_2 - S_1\Big),$$
$$I(f) - S_2 \approx \frac{1}{15}\Big(S_2 - S_1\Big).$$

We can interpret this to mean that the error from the quadrature approximation with $h/2$ (S_2) should be about 1/15 of the difference between the two Simpson's rule approximations at h and $h/2$.

This observation can lead us to develop a strategy for deciding when to subdivide a panel and when to stop. Specifically, suppose we want the error to be less than a certain tolerance, ϵ. Then we can ask whether for a given panel

$$\frac{1}{15}\Big(S_2 - S_1\Big) < \epsilon.$$

If this is true, then we can stop for that given panel. If however, the difference doesn't satisfy the tolerance, that is an indication that we should subdivide the region in half again.

A general algorithm might look something like:

1. Initialize by computing Simpson's on $[a, b]$, i.e. S_1
2. For $i = 1, 2, ...$
    1. subdivide the interval into 2 sub-regions and compute S_{i+1} by applying Simpson on each subinterval
    2. If $|S_{i+1} - S_i| < 15\epsilon$
        1. converged
        2. else repeat.

25.2 Summary

This is just a brief introduction into adaptive quadrature. There are many other techniques one can use to enhance both the accuracy and the efficiency of these methods. An interested reader, can find many references under topics such as adaptive mesh refinement.

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

Part VI

Initial-Value Problems for ODEs

Part VII

Initial-Value Problems for ODEs

The areas we will cover include:

1. General statement of Initial Value Problems, systems of ODEs, etc.
2. Euler's Method including a simple error analysis
3. Higher-order methods
4. Multi-step methods

The material on the existence and uniqueness of the IVP was moved to the Appendix, including:

1. Concepts of Lipschitz continuity and convex sets
2. Fundamental Existence and Uniqueness of solutions to the IVP
3. Concept of well-posed problems

Let's first start with a roadmap for the lectures to follow.

As we discussed in class last time, we will:

- be mostly concerned with introducing methods for the solution of IVPs and providing advantages and disadvantages of them
- not discuss problems that are ill-posed, stiff ODE's, or have other structure

Our reason for this particular focus is that there is a lot of good software available for these problems, so you may never need to actually implement one of these methods. Nevertheless, it will be important to know the differences between the methods, what types of problems they can be used on, and the pros and cons of each method.

26 Euler's Method

26.1 Introduction

The scalar ***initial-value problem*** (IVP) has the form:

$$y' = \frac{dy}{dt} = f(t, y(t)), \quad a \leq t \leq b, \quad y(a) = \alpha. \tag{26.1}$$

In the general case, we would consider a *system* of ODEs, i.e.

$$y' = \frac{dy}{dt} = f(t, y), \quad a \leq t \leq b, \quad y(a) = \alpha,$$

where

$$y' = \begin{bmatrix} y_1' \\ \vdots \\ y_n' \end{bmatrix} \quad f(t, y) = \begin{bmatrix} f_1(t, y_1, \dots, y_n) \\ \vdots \\ f_n(t, y_1, \dots, y_n) \end{bmatrix} \tag{26.2}$$

Since all of the solution techniques we will study can be generalized to a system of ODEs, we will keep it simple for now and assume we have an IVP of the form given by Equation 26.1.

Reduction of higher-order ODE to a system of ODEs

It can be shown that a general ***n-th*** order ODE:

$$y^{(m)}(t) = f(t, y(t), y'(t), \dots, y^{(m-1)}(t)).$$

can be written in the form of a system of ODEs.

Example 26.1. Consider the simple second order IVP given by:

$$y'' + ay' + by = f$$

If we let:

$$y_1 = y$$
$$y_2 = y'$$

then we can rewrite the IVP as two first order ODEs in the form of Equation 26.2:

$$y_1' = y_2$$
$$y_2' = f - (ay_2 + by_1).$$

> Autonomous Systems
>
> An IVP where the function f does not depend explicitly on t is said to be in ***autonomous form***, i.e.
>
> $$y' = f(y)$$
>
> Many software packages for the solution of IVPs assume that the function is given in this form. This is generally achieved through the addition of an additional equation of the form $t' = 1$.

26.2 Euler's Method

We will now present the simplest method for solving an IVP.

> Note
>
> For the remainder of the discussion we will assume that our IVP is well-posed. See Section B.1 for details.

Recall we are looking for solutions to the IVP:

$$\frac{dy}{dt} = f(t, y), \quad a \leq t \leq b, \quad y(a) = \alpha.$$

As in previous lectures, we will take an approach for numerically solving this problem by approximating it on a discrete grid.

In this case, the points are referred to as ***mesh points*** and are typically of the form:

$$t_i = a + ih, \; i = 0, 1, 2, \ldots, N,$$

where the t_i are assumed to be equally spaced. Here, the distance between two consecutive points $t_{i+1} - t_i$ is called the step size and is given by:

$$t_{i+1} - t_i = \frac{(b-a)}{N} = h.$$

Terminology

We will denote the step size by h. Many other references denote it by Δt since we are usually referring to the time evolution of the IVP. Also note that the step size is sometimes called the ***time step***, again in reference to the time variable.

In order to derive Euler's method we can either make use of Taylor's Theorem or just use the numerical approximation for the first derivative that we used in Section 19.1 :

$$y(t_{i+1}) = y(t_i) + hy'(t_i) + \frac{h^2}{2}y''(\xi_i) \quad i = 0, 1, 2, \dots, N-1,$$

where $h = t_{i+1} - t_i$, and $\xi_i \in [t_i, t_{i+1}]$.

Now remember that y' satifies the IVP. As a result, we can rewrite the above equation as:

$$y(t_{i+1}) = y(t_i) + hf(t_i, y(t_i)) + \frac{h^2}{2}y''(\xi_i) \tag{26.3}$$

Taking the first 2 terms on the right hand side of this equation as our approximation to $y(t_{i+1})$ leads us to propose the following algorithm, which we will call:

Euler's Method

$$\begin{aligned} y_0 &= \alpha \\ y_{i+1} &= y_i + hf(t_i, y_i), \quad i = 0, 1, \dots, N-1 \end{aligned} \tag{26.4}$$

Here $y_i \approx y(t_i)$.

This type of equation is known as a ***difference equation***. You can think of it as being derived from a forward difference approximation to the derivative. Another interpretation is that it is the discretization (in time) of the continuous differential equation.

Example 26.2. Solve the IVP given by:

$$y' = f(t, y) = y - t^2 + 1 \quad y(0) = 0.5 \quad 0 \leq t \leq 2,$$

with $h = 0.5$.

I find it easier before I start, to write down a table with some of the important variables, where I can keep track of the steps. Something like the following is helpful:

Table 26.1: Euler Computations

i	t_i	y_i
0	$t_0 = a$	$y_0 = y(a)$
1	$t_1 = a + h$	$y_1 = ...$
2	$t_2 = a + 2h$	
3		
...	...	...
N	$t_N = b$	

I then fill in the initial conditions in the first row and as I compute subsequent y_i I fill in the table with those values.

Let's first code the function, specifically $f(t, y) = y - t^2 + 1$. Note that this IVP has an exact solution given by $(t + 1)^2 - 0.5 \exp(t)$

```
ftyex1 <- function(t, y, pars) {
  # Note that we don't use pars, so it's only here as a dummy parameter required by the ODE s
    yprime <- y - t^2 + 1
    return(list(c(yprime)))
}
# define the exact solution as well
yexact <- function(t,y) {
    yexact <- (t+1)^2 - 0.5*exp(t)
    return(yexact)
}
```

Let's now solve the IVP with a step size of $h = 0.5$ using the built-in ODE solver with method chosen to be "euler". We can also compute the true solution and the associated error generated by Euler's method.

```
# Call the ODE solver with several values for the initial condition
library(deSolve)

# Set time integration limits and initial condition
a <- 0
b <- 2
y0 = 0.5
```

```r
# Set up the ode solver
h <- 0.5
parms <- c() # Set some of the parameters
times1 <- seq(a,b, by=h)   # Create the timestep mesh
init <- c(y = y0)          # Set up the initial condition for the solver

ex1out0 <- as.data.frame(ode(init, times1, ftyex1, parms, method = "euler"))

# Compute exact solution for comparison
ytrue <- data.frame(ex1out0$time,yexact(times1,0))
colnames(ytrue) <- c("time", "y")
yerr <- data.frame(ex1out0$time, abs(ytrue$y - ex1out0$y))
colnames(yerr) <- c("time", "yerr")
```

The table below summarizes the output from the ode solver and compares it to the exact solution. What do you notice about the error, especially as time increases?

Exact Solution versus Euler's method Solution

```
  time   ysol   yexact      yerr
1  0.0 0.5000 0.500000 0.0000000
2  0.5 1.2500 1.425639 0.1756394
3  1.0 2.2500 2.640859 0.3908591
4  1.5 3.3750 4.009155 0.6341555
5  2.0 4.4375 5.305472 0.8679720
```

Let's plot the solution from Euler alongside the exact solution

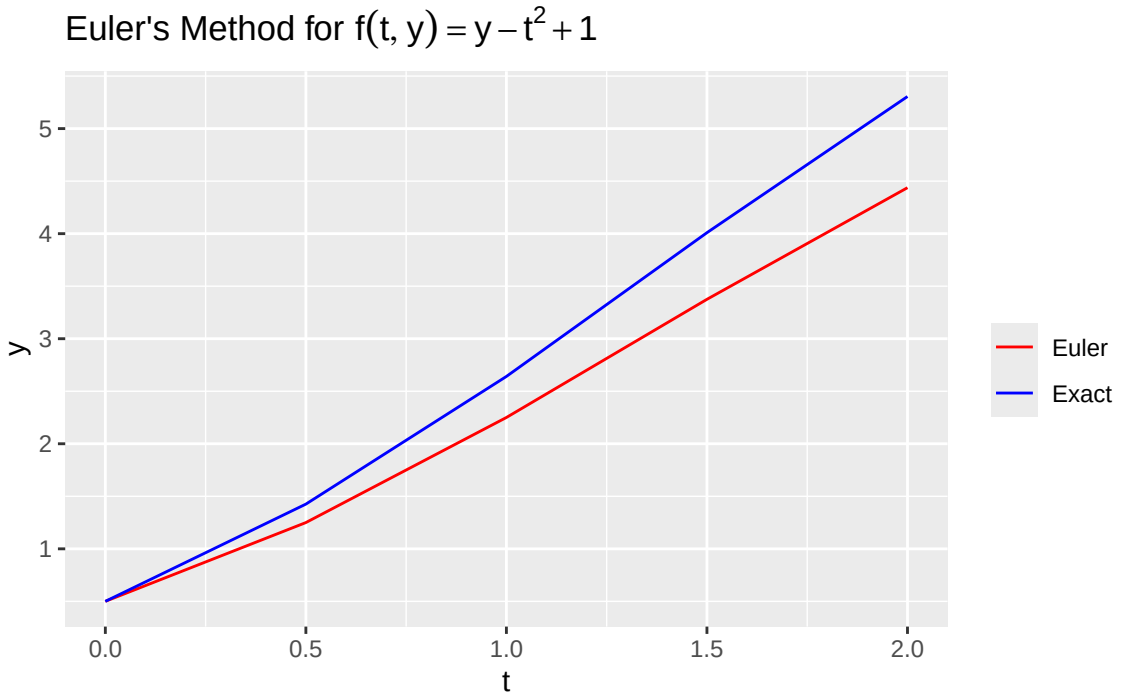


Figure 26.1: Euler's method example

Let's explore what happens to the solutions by solving the IVP with several different initial conditions (see the plot below).

One immediate observation is that the IVP generates a family of solutions that can be parameterized by the specific initial condition chosen. In this case, also notice that the curves do not converge to a single line. What implications would this have on the solutions generated by Euler's method? We'll have more to say in later lectures on what is happening and what we can do to help us attain more accurate solutions.

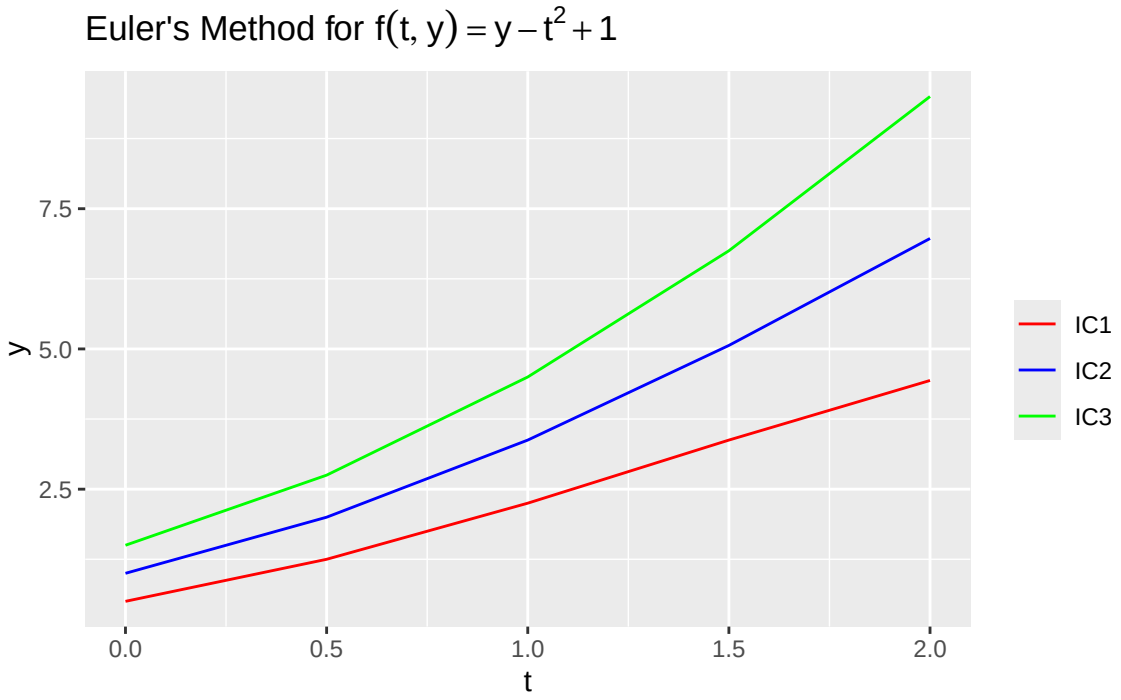


Figure 26.2: IVP with different ICs

Exercise 26.1. Solve the IVP given by:

$$y' = f(t, y) = 1 + (t - y)^2 \quad y(2) = 1.0 \quad 2 \leq t \leq 3$$

with $h = 0.5$ Fill out the table below with your calculations.

Solution:

Table 26.2: In class exercise

i	t_i	y_i
0		
1		
2		
3		
...		
N		

26.3 Backward Euler

What if we had used a backward difference formula to approximate the derivative of y? In other words:

$$y'(t_i) = \frac{y(t_i) - y(t_{i-1})}{h} = f(t_i, y(t_i))$$

Following the same procedure as before we would have:

$$y(t_i) = y(t_{i-1}) + hf(t_i, y(t_i))$$

Since we really want to compute the approximation at the next time step, let's shift the index by 1:

$$y(t_{i+1}) = y(t_i) + hf(t_{i+1}, y(t_{i+1}))$$

Again, letting the approximation to the true solution be denoted by $y_i \approx y(t_i)$, leads to what is known as ***Backward Euler:***

Backward Euler

$$\begin{aligned} y_0 &= \alpha \\ y_{i+1} &= y_i + hf(t_{i+1}, y_{i+1}), \quad i = 0, 1, \ldots, N-1 \end{aligned}$$

This seems like a straightforward alternative, but now notice that the computation of y_{i+1} will depend implicitly on itself since it appears on both the right and left hand sides of this equation. This type of method is known as an ***implicit method*** and will require some sort of iterative method to be able to compute the solution at the next time step.

Explicit/Implicit Methods

(Forward) Euler's Method is an example of a type of method called an ***explicit*** *method, because everything we need to compute a quantity at time* t_{i+1} is given by known quantities at the previous time step t_i. Backward Euler on the other hand is an example of an ***implicit*** method since we have y_{i+1} on both sides of the equation. There are advantages and disadvantages to both approaches. In general, one can take longer timesteps with an implicit method. On the other hand, an implicit method will generally require the solution of a nonlinear system.

26.4 Key Points

- Initial Value Problems arise in many scientific and engineering problems
- Euler's method can be used to solve the IVP by using a forward difference approximation to the derivative of y.
- Using the backward difference approximation yields a similar method, but requires having to solve the difference equation *implicitly*.
- Forward Euler is easy to implement and relatively cheap.

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

27 Error Analysis for Euler's Method

27.1 Error Analysis

In order to discuss the error in Euler's method as well as its convergence, we will need to define a few terms.

First let's recall that we are seeking to approximate the solution to the IVP at a set of discrete points in time or ***mesh points*** typically of the form:

$$t_i = a + ih,\ i = 0, 1, 2, \ldots, N.$$

We start with a few definitions.

Definition 27.1. The ***difference method***

$$\begin{aligned} y_0 &= \alpha \\ y_{i+1} &= y_i + h\phi(t_i, y_i), \quad i = 0, 1, \ldots, N-1 \end{aligned}$$

has ***local truncation error***

$$d_{i+1} = \frac{y_{i+1} - (y_i + h\phi(t_i, y_i))}{h},$$

where y_i denotes the solution of the difference equation at t_i, and $\phi(t, y)$ is a given function.

We say that a method is ***consistent (or accurate) of order*** q if q is the lowest positive integer such that

$$\max_i\ |d_i| = O(h^q).$$

Finally, the ***global error*** is defined as

$$e_i = y(t_i) - y_i \quad i = 0, 1, \ldots N,$$

where $y(t_i)$ is the true solution at time, t_i.

Example 27.1. Show that Euler's method has local truncation error of $O(h)$.

For Euler's method (Equation 26.4) $\phi(t_i, y_i) = f(t_i, y_i)$. As such we can write the local truncation error as:

$$\begin{aligned} d_{i+1} &= \frac{y_{i+1} - (y_i + h\phi(t_i, y_i))}{h}, \\ &= \frac{h}{2} y''(\xi_i), \quad \xi_i \in (t_i, t_{i+1}), \end{aligned} \tag{27.1}$$

where the second equation is as a result of Equation 26.3. If we assume that the second derivative of y is bounded by some constant M, then we have:

$$|d_{i+1}| \leq \frac{h}{2} M,$$

$$\implies d_{i+1} = O(h) \implies \text{Euler is first order accurate,}$$

and hence the local truncation error is $O(h)$.

Remark: *We call d_{i+1} local because it measures the accuracy of the solution at a specific point (step) in time. Notice also that the error will depend on 1) the ODE, and 2) the step size.*

By the same argument, it is easy to see that Backward Euler is also first order accurate.

27.2 Convergence and Global Error Estimates

We now state a theorem that provides error bounds on the approximations generated by Euler's method.

Theorem 27.1 (Euler Method Convergence.)**.** *Suppose $f(t, y)$ is continuous and Lipschitz continuous in y, with constant L on a region $D = \{(t, y) |\ a \leq t \leq b,\ -\infty < y < \infty\}$.*

Let $y_1, \ldots, y_N$ be approximations generated by Euler's method for some integer $N > 0$. Then Euler's method converges and its global error decreases linearly in h.

Furthermore if a constant M exists with

$$|y''| \leq M \quad \forall t \in [a, b],$$

then the global error satisfies

$$|e_i| \leq \frac{hM}{2L} \left[e^{L(t_i - a)} - 1 \right] \ \forall i = 0, 1, 2, \ldots, N \tag{27.2}$$

Proof. See Section 27.4 □

Let's take a quick look to see what the error bound is saying.

Note that:

- the bound is exactly zero for $t_i = a$, which makes sense since $y_0 = y(a)$, the given initial condition.
- the last term depends on the Lipschitz constant, as well as the term $t_i - a$. But $t_i - a$ is bounded by $b - a$, so the entire term is also bounded.
- The bound depends on both the Lipschitz constant as well as the bound, M, on the second derivative of $y(t)$.
- ***the error bound is linear in*** h.

To summarize: the good news is that we can bound the error at each time step. Nonetheless it is clear that the error bound will increase at each time step t_i. Our hope is that by choosing a small enough h we can compensate for the other terms and make the error bound small enough to generate an accurate approximation to $y(t)$.

Remark

Note that the theorem requires a bound on the second derivative. We can sometimes use some knowledge of the partial derivatives to obtain an error bound. The important aspect is that the ***error bounds are linear in*** h. *Not surprisingly, as the number of computations grow so will the roundoff error.*

27.3 Roundoff Error Analysis

Advanced

This section was not discussed in class, but follows the textbook closely.)

As in the numerical differentiation lectures, we can derive an error analysis that includes the roundoff error. This leads us to the following error bound:

$$|y(t_i) - y_i| \leq \frac{1}{L}\left(\frac{hM}{2} + \frac{\delta}{h}\right)\left[e^{L(t_i-a)} - 1\right] + |\delta_0|e^{L(t_i-a)},$$

where δ, δ_0 are constants representing the amount of roundoff error incurred at each time step.

Notice that we have the same situation as before with numerical differentiation – one of the terms is going to 0 while the second term blows up as $h \to 0$.

A similar type of calculation as in the case of numerical differentiation, yields an optimal h:

$$h = \sqrt{\frac{2\delta}{M}}$$

that will depend on both δ and M. If we assume that $\delta \approx \epsilon$, i.e machine epsilon, then depending on the value of M, this implies h should be roughly the square root of machine epsilon. For IVPs, the more important question is stability, which will depend on choosing an appropriate h. Unfortunately, we don't have time to cover that topic here.

27.4 Proof of Convergence for Euler's Method

We had to bypass the proof of the convergence of Euler's method. It is not a difficult proof and it uses standard techniques. If you're interested this section will provide a brief overview of the proof.

First, we will need a few lemmas that are used in the proof of the convergence of Euler's method. They are included here for completeness.

Lemma 5.7. For all $x \geq 1$ and any positive m we have

$$0 \leq (1+x)^m \leq e^{mx} \tag{27.3}$$

Proof. Straightforward application of Taylor's Theorem to $f(x) = e^x$ about $x_0 = 0$.

Lemma 5.8. If

- s, t are positive real numbers
- $\{a_i\}_{i=0}^k$ is a sequence satisfying $a_0 \geq -\frac{t}{s}$
- $a_{i+1} \leq (i+s)a_i + t \quad i = 0, 1, \ldots, k-1$

Then

$$a_{i+1} \leq e^{(i+1)s}(a_0 + \frac{t}{s}) - \frac{t}{s}.$$

Proof. Left as an exercise. In case you're interested in trying to prove it, the idea is to use a geometric series to show that under our assumptions that

$$a_{i+1} \leq (i+s)^{i+1}(a_0 + \frac{t}{s}) - \frac{t}{s}$$

followed by an application of Equation 27.3, with $x = s$ to show result.

Proof. Convergence for Euler's method Theorem 27.1.

A method is said to *converge* if the maximum global error tends to 0 as h tends to 0 (assuming that an exact solution exists and is sufficiently smooth. For Euler' method, which is $O(h)$, we would then expect that $e_i = y(t_i) - y_i$ should be of the same order.

Let's consider the local truncation error first:

$$\begin{aligned} d_i &= \frac{y(t_{i+1}) - y(t_i)}{h} - f(t_i, y(t_i)), \\ 0 &= \frac{y_{i+1} - y_i}{h} - f(t_i, y_i), \end{aligned}$$

Subtracting the two, gives us a difference formula for the local truncation error:

$$d_i = \frac{e_{i+1} - e_i}{h} - [f(t_i, y(t_i)) - f(t_i, y_i)]$$

Solving for the error at t_{i+1}, we have:

$$e_{i+1} = e_i + h\left[f(t_i, y(t_i)) - f(t_i, y_i)\right] + hd_i$$

Now taking absolute values:

$$|e_{i+1}| \leq |e_i| + h|\left[f(t_i, y(t_i)) - f(t_i, y_i)\right]| + |hd|,$$

where d is the maximum of $|d_i|$ over all time steps.

Since f is Lipschitz continuous with constant L, we can simplify the error difference equation to:

$$\begin{aligned} |e_{i+1}| &\leq |e_i| + hL|e_i| + |hd|, \\ &\leq (1 + hL)|e_i| + hd. \end{aligned}$$

Almost there (hang in there) ...

Now note that we can do the same estimate with e_i, which would gives us:

$$
\begin{aligned}
|e_{i+1}| &\leq (1+hL)|e_i| + hd, \\
&\leq (1+hL)[(1+hL)|e_{i-1} + hd| + hd = (1+hL)^2|e_{i-1}| + (1+hL)hd + hd \\
&\leq \cdots \leq (1+hL)^{i-1}|e_0| + hd\sum_{j=0}^{i}(1+hL)^j \\
&\leq d\left[e^{L(t_i-a)} - 1\right]/L
\end{aligned}
,
$$

where we've used the Lemmas above to compute the sum and the fact that $e_0 = 0$.

The final step is to note that by definition of we have that:

$$
\begin{aligned}
d &\geq \max_{0\leq i\leq N-1} |d_i| \\
d_i &= \frac{h}{2}y''(\xi_i)
\end{aligned}
$$

So if we can bound the second derivative such that:

$$
M = \max_{a\leq t\leq b} |y''(t)|,
$$

then we can set

$$
d = \frac{h}{2}M.
$$

which gives us our desired error bound:

$$
|e_i| \leq \frac{Mh}{2L}\left[e^{L(t_i-a)} - 1\right], \quad i = 0, 1, \ldots, N.
$$

□

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

28 Higher-order Methods

28.1 Higher-order Taylor Methods

In a similar vein to what we did to derive Euler's method, we can derive higher-order methods by extending the Taylor series approximation to include the higher derivatives. This will yield methods that are more accurate, but at a cost of having to compute the higher derivatives.

There will always be a tradeoff between higher accuracy methods and computational cost. Here computational cost is usually measured in terms of function evaluations. This is important to remember because in a real-world problem each function evaluation could cost hours of computer time.

We note in passing that while higher-order Taylor methods can be generated, they are rarely used in practice, as the requirement of having higher-order derivatives is rarely met in real-world problems.

> Idea
>
> *If Euler's method used a Taylor series expansion truncated after the first derivative (i.e. $n = 1$), let's try higher values of n.*

Let's start by writing down the Taylor series expansion of $y(t)$ about the current time step.

$$y(t_{i+1}) = y(t_i) + hy'(t_i) + \frac{h^2}{2}y''(t_i) + ... + \frac{h^{(n)}}{n!}y^{(n)}(t_i) + \frac{h^{(n+1)}}{(n+1)!}y^{(n+1)}(\xi_i),\ \xi_i \in [t+i, i_{i+1}]$$

Using the statement of the IVP, we know that $y' = f(t, y)$, so we can replace the derivatives of y by the corresponding derivative of $f(t, y)$

This leads to:

$$y(t_{i+1}) = y(t_i) + hf(t_i, y(t_i)) + \frac{h^2}{2}f'(t_i, y(t_i)) + ... + \\ \frac{h^{(n)}}{n!}f^{(n-1)}(t_i, y(t_i)) + \frac{h^{(n+1)}}{(n+1)!}f^{(n)}(t_i, y(t_i)),\ \xi_i \in [t+i, i_{i+1}]$$

For convenience let's denote the function $T^{(n)}$ as:

$$T^{(n)} = f(t_i, y(t_i)) + \frac{h}{2}f'(t_i, y(t_i)) + ... + \frac{h^{(n-1)}}{n!}f^{(n-1)}(t_i, y(t_i)).$$

In a manner similar to Euler's method, this leads us to propose the following algorithm for higher-order Taylor methods

$$y_0 = \alpha$$
$$y_{i+1} = y_i + hT^{(n)}(t_i, y_i),\ i = 0, 1, ..., N-1.$$

Remark: *Clearly Euler's method is the special case of* $n = 1$.

Remark: Also, *by the definition of the local truncation error, we can see that the higher-order Taylor method will have a local truncation error of* $O(h^n)$, using the same argument that we used for Euler's method.

For example, if we wanted to have a second order method, we would use $T^{(2)}$ in our difference equation:

$$T^{(2)} = f(t_i, y(t_i)) + \frac{h}{2}f'(t_i, y(t_i))$$

28.2 Runge-Kutta Order 2 Methods

Another approach we can use leads to the well-known class of Runge-Kutta methods.

> Idea
>
> The idea for the Runge-Kutta methods is to have the high-order local truncation error of Taylor methods ***without the need to compute and evaluate the derivatives*** **of** $f(t, y)$. Ideally, we should only need function evaluations.

Let's first rewrite the IVP as an integral:

$$y(t_{i+1}) = y(t_i) + \int_{t_i}^{t_{i+1}} f(t, y(t))dt.$$

One simple idea is to now use our methods from the numerical integration sections to evaluate the integral, for example the Trapezoid rule:

$$y_{i+1} = y_i + \frac{h}{2}(f(t_i, y_i) + f(t_{i+1}, y_{i+1})). \tag{28.1}$$

As with our discussion of Backward Euler, we can see that this is an implicit equation as y_{i+1} is both on the right and left hand sides of the equation and would require the solution of a nonlinear equation.

But what if we could approximate the value of y_{i+1} on the right hand side with an explicit method? Since we only know one explicit method (Euler's method), let's try it out and see what happens.

Let's define:

$$Y = y_i + hf(t_i, y_i),$$

and substitute Y for y_{i+1} into Equation 28.1:

$$y_{i+1} = y_i + \frac{h}{2}(f(t_i, y_i) + f(t_{i+1}, Y)).$$

This equation is known as the ***explicit trapezoidal method***. This method is also known as a two-stage method - the second stage comes about because we have to evaluate the function at a second point, namely $f(t_{i+1}, Y)$. We can summarize this by the following steps:

Explicit Trapezoid

$$\begin{aligned} Y &= y_i + hf(t_i, y_i) \\ y_{i+1} &= y_i + \frac{h}{2}(f(t_i, y_i) + f(t_{i+1}, Y)) \end{aligned} \tag{28.2}$$

It can be shown that this method is second order accurate.

A similar process but using the midpoint rule instead of the trapezoid rule will also work and yields the following two-stage algorithm, known as the ***explicit midpoint*** method:

Explicit Midpoint

$$\begin{aligned} Y &= y_i + \frac{h}{2}f(t_i, y_i) \\ y_{i+1} &= y_i + hf(t_{i+1/2}, Y) \end{aligned} \tag{28.3}$$

Programming Tip

When coding the algorithms above, for example the ***explicit trapezoidal method***, the algorithm is usually stated in terms of the following:

$$\begin{aligned} k_1 &= f(t_i, y_i) \\ k_2 &= f(t_{i+1}, y_i + hk_1) \\ y_{i+1} &= y_i + \frac{h}{2}(k_1 + k_2) \end{aligned}$$

You should convince yourself that these are the same.

28.3 Runge Kutta Order 4

We can derive higher-order Runge-Kutta methods using similar techniques. One of the most popular is the 4th order method, which can be written as:

$$\begin{aligned} k_1 &= f(t_i, y_i), \\ k_2 &= f(t_{i+1/2}, y_i + \frac{h}{2}k_1), \\ k_3 &= f(t_{i+1/2}, y_i + \frac{h}{2}k_2), \\ k_4 &= f(t_{i+1}, y_i + hk_3), \\ y_{i+1} &= y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4). \end{aligned}$$

As the name implies, this method is $O(h^4)$.

Example 28.1. Solve our favorite IVP problem using the Explicit Midpoint Method and the Explicit Trapezoidal method.

$$\begin{aligned} y' &= y - t^2 + 1, \quad 0 \le t \le 2 \\ y(0) &= 0.5 \end{aligned}$$

Note: The complete worked out example for both methods can be found in the notebook Math130-IVP-RK.Rmd. Include Rmd file here???

28.4 Summary

- Higher-order Taylor methods can be derived, extending the accuracy of Euler's method, although these methods are rarely use in practice.
- Another approach is to use the same idea but replace higher derivatives of f with other approximations.
- Leads to multi-stage methods such as explicit Trapezoidal, explicit midpoint methods and the widely used and popular class of methods known as Runge-Kutta methods.
- Choosing a good method will depend on the problem, how smooth the functions are, and the desired accuracy.

28.5 General form for Runge Kutta methods

Advanced - not covered in class

The general form of the Runge-Kutta equations with s stages is given by:

$$y_{i+1} = y_i + h \sum_{j=1}^{s} b_j k_j, \tag{28.4}$$

where the k_i terms are given by:

$$\begin{aligned} k_1 &= f(t_i, y_i), \\ k_2 &= f(t_i + hc_2, y_i + a_{21}k_1), \\ k_3 &= f(t_i + hc_3, y_i + a_{31}k_1 + a_{32}k_2), \\ &\vdots \\ k_s &= f(t_i + hc_s, y_i + a_{s1}k_1 + a_{s2}k_2 + ... + a_{s,s-1}k_{s-1}), \end{aligned} \tag{28.5}$$

The coefficients, a_{ij}, b_i, c_i, are determined by the process we introduced earlier that compares partial derivatives.

In addition, for consistency, we ask that the coefficients satisfy the equations:

$$\sum_{j=1}^{i-1} a_{ij} = c_i, \quad i = 2, ..., s$$

and

$$\sum_{j=1}^{s} b_j = 1,$$

An easy way to visualize the coefficients is to use the Butcher tableau:

$$\begin{array}{c|c} c & A \\ \hline & b^T \end{array}$$

where c, b are vectors of coefficients, and A is a matrix of coefficients, as defined in Equation 28.4 and Equation 28.5.

Using this tableau, we can, for example, summarize the formula for the Runge-Kutta method of order 2 (RK2) as:

$$\begin{array}{c|cc} 0 & 0 & \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array}$$

Substituting into Equation 28.5 yields the formulas:

$$\begin{aligned} k_1 &= f(t_i, y_i), \\ k_2 &= f(t_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_1), \\ y_{i+1} &= y_i + hk_2. \end{aligned}$$

We can also derive a similar set of formulas for a Runge-Kutta method with 4 stages that will yield a method of $O(h^4)$. The corresponding tableau is given by.

$$\begin{array}{c|cccc} 0 & 0 & & & \\ \frac{1}{2} & \frac{1}{2} & 0 & & \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

Again, substituting into Equation 28.5, yields the Runge-Kutta method of order 4:

$$\begin{aligned} k_1 &= f(t_i, y_i), \\ k_2 &= f(t_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_1), \\ k_3 &= f(t_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_2), \\ k_4 &= f(t_i + h, y_i + hk_3), \\ y_{i+1} &= y_i + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4). \end{aligned}$$

Recall that our goal was to replace the need for higher order derivatives while retaining the higher order local discretization error. The price we had to pay was in the form of extra function evaluations.

In the case of RK-2, it was an additional 2 function evaluations. For RK-4, it was 4 extra function evaluations. It can be shown that the following relationship holds between the additional function evaluations and the local discretization error holds for Runge-Kutta methods:

Table 28.1: Function Evals vs. Local Error

Function Evaluations	Local Discretization Error
2	$O(h^2)$
3	$O(h^3)$
4	$O(h^4)$
$5 \leq n \leq 7$	$O(h^{n-1})$
$8 \leq n \leq 9$	$O(h^{n-2})$
$10 \leq n$	$O(h^{n-3})$

As you can see, when $n > 4$, there is no big advantage to increasing the order in terms of increasing the order of the local discretization error. As a consequence, the most popular of the Runge-Kutta methods is the one of order 4.

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

29 Multi-Step Methods for IVP

29.1 Motivation

All the methods so far belong to a class of methods known as one-step methods, which is to say that all of the information used in the computation of the approximation at the next time step only used information from the immediately prior time step.

However, you might ask yourself, can we use information from other previous steps to improve our approximation. This would be especially useful in the case when each of the function evaluations are computationally expensive. This leads us to proposing a set of methods that attempt to take advantage of all of this additional information already available to us.

Let's first start with some notation and a definition. As before, let's assume that we have equally spaced time steps such that $t_i = a + ih, i = 0, 1, N$.

Definition: *an s-step linear multistep method for solving the IVP has a difference equation of the form:*

$$\sum_{j=0}^{s} \alpha_j y_{i+1-j} = h \sum_{j=0}^{s} \beta_j f_{i+1-j}, \tag{29.1}$$

where we let $f_{i+1-j} = f(t_{i+1-j}, y_{i+1-j})$. Without loss of generality, we can assume that $\alpha_0 = 1$ since we can rescale all of the equations.

It will also be useful to distinguish between cases that need the function value $f(t_{i+1}, y_{i+1})$ at the next time step to compute y_{i+1}. In particular, if $b_0 = 0$ the method is called an **explicit (open) method** and we can write Equation 29.1 as:

$$y_{i+1} = -\sum_{j=1}^{s} \alpha_j y_{i+1-j} + h \sum_{j=1}^{s} \beta_j f_{i+1-j}$$

Notice that we can recover Euler's method from the explicit form of this equation by setting $\beta_0 = 0, s = 1, \alpha_1 = -1, \beta_1 = 1$.

The second case is if we let $b_0 \neq 0$ and the method is then called **implicit (closed)** as y_{i+1} appears on both sides of Equation 29.1 so it is only implicitly defined.

$$y_{i+1} - h\beta_0 f_{i+1} = -\sum_{j=1}^{s} \alpha_j y_{i+1-j} + h\sum_{j=1}^{s} \beta_j f_{i+1-j}$$

Remark

In general implicit methods are more accurate than explicit methods and we can get by with larger time steps. The disadvantage is that we need to solve a system of linear (or nonlinear) equations at each time step. Additionally, the solution may not be unique (or even exist).

How then should be develop higher-order formulas for solving the IVP. We've already decided that it would be good to use some of the past information and in particular, we should try to use the past values of y_i that we have already computed. This leads to the following idea:

Idea

Use some number of past values of y_i (e.g. $y_i, y_{i-1}, y_{i-2}, ...$) to fit an ***interpolating polynomial to*** f, which can then be used to derive a higher order method using techniques similar to the ones we used to derive the multi-stage (i.e. Explicit Trapezoid, Runge-Kutta, etc.) methods

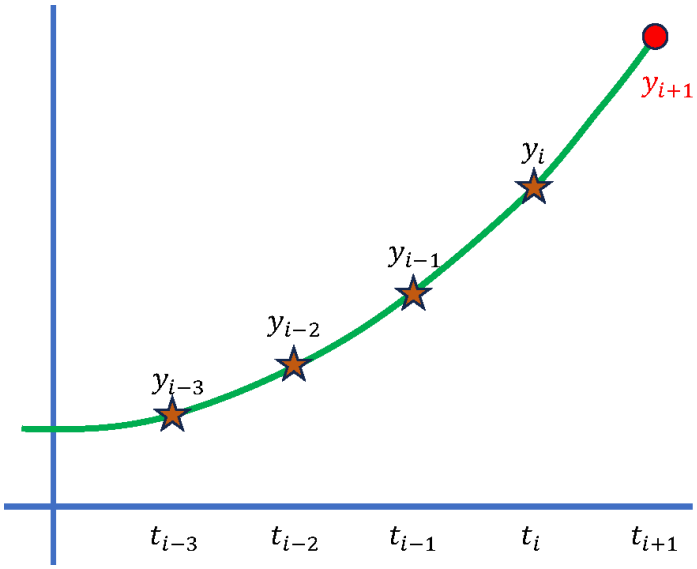


Figure 29.1: Multi-step method uses past computed values

There are many methods one could use to solve the IVP and we will give examples of several of the more popular multi-step methods including the Adams-Bashforth (explicit) and Adams-

Moulton (implicit) methods. We will not derive the following methods here, but if you're interested we give a brief derivation in the supplemental section Section 29.4.

Adams-Bashforth fourth-order

$$\begin{aligned} y_0 = \alpha_0,\ y_1 = \alpha_1,\ y_2 = \alpha_2,\ y_3 = \alpha_3, \\ y_{i+1} = y_i + \frac{h}{24}\left[55f(t_i, y_i) - 59f(t_{i-1}, y_{i-1}) + 37f(t_{i-2}, y_{i-2}) - 9f(t_{i-3}, y_{i-3})\right] \end{aligned}$$

Adams-Moulton fourth-order

$$\begin{aligned} y_0 = \alpha_0,\ y_1 = \alpha_1,\ y_2 = \alpha_2, \\ y_{i+1} = y_i + \frac{h}{24}\left[9f(t_{i+1}, y_{i+1}) + 19f(t_i, y_i) - 5f(t_{i-1}, y_{i-1}) + f(t_{i-2}, y_{i-2})\right] \end{aligned}$$

> Remark
>
> Note that in both cases, one needs to supply additional initial values. For example, in the case of Adams-Bashforth fourth-order method, we need to have 4 initial values in total. These are usually computed through an explicit method, for example a Runge-Kutta method.

29.2 Demo: Basic SIR Model

We demonstrated the use of a simple ODE/IVP solver by solving a problem of predicting the breakout of an epidemic using data from Merced County COVID cases taken from: USA Facts Merced County, California coronavirus cases and deaths

To model an epidemic of an infectious disease, the usual approach is to use what is known as the SIR Model.

The SIR equations are given by:

$$\begin{aligned} \frac{dS}{dt} &= -\alpha SI \\ \frac{dI}{dt} &= \alpha SI - \gamma I \\ \frac{dR}{dt} &= \gamma I \\ N &= S + I + R \end{aligned}$$

where S is the number of **susceptible** (healthy) individuals, I represents the number of **infected** individuals, R is the number of people who have **recovered** from the disease, and N is the total population.

The demo we presented had 4 parameters you can play with: initial population (N), the number of days to run the simulation for, and the 2 parameters that represent the rates between susceptible and infected (α) and between infected and recovered (γ).

The solver used comes from the deSolve package in R called ode. The default solver is "lsoda" (Petzold & Hindmarsh), but other choices are available. Calling the ode solver requires the initial conditions (init), the times at which to compute the solution (times, the function to evaluate the ode (sir in this case), and a list of parameters that the ode solver passes along to the ode function.

The original data taken from the site gave us the following plot:

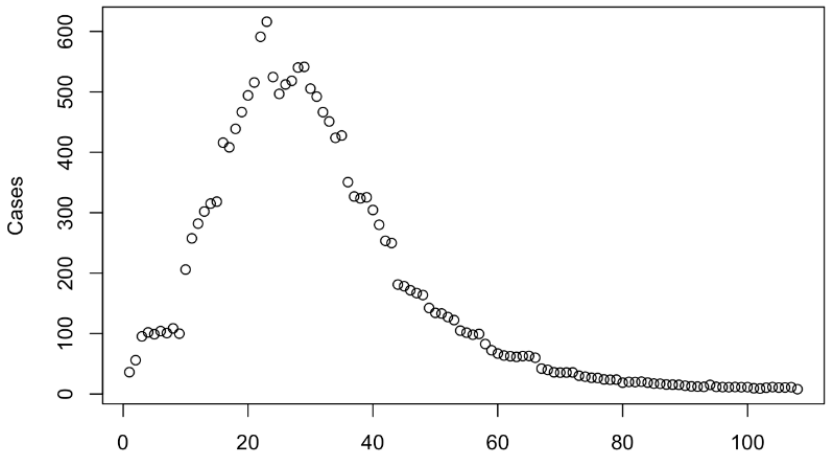


Figure 29.2: Merced County Covid Cases (weekly average)

In the demo, we played around with the parameters for the SIR model to match the data as best we could. According to the model, the basic Reproduction number $R_0 = 8.5$. For comparison, for measles one of the more contagious diseases, $R_0 = 12 - 18$ while the normal flu has $R_0 \approx 1.28$.

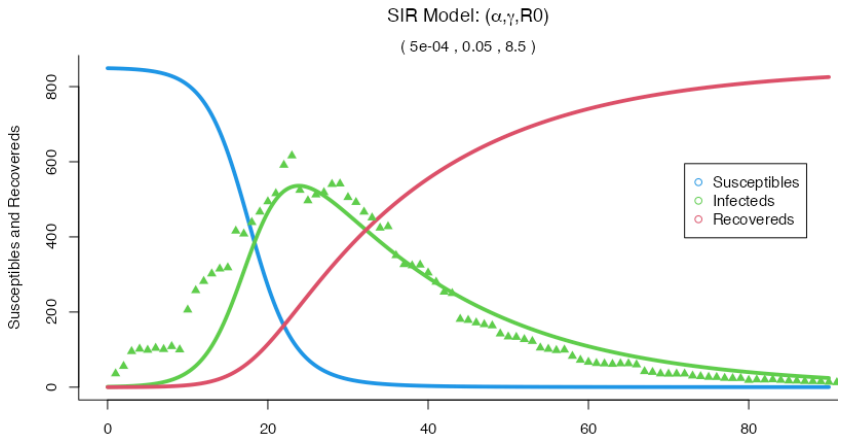


Figure 29.3: SIR demo using Merced County data

29.3 Summary of Methods Studied

Let's take a step back and summarize our main results:

Table 29.1: Comparison of Different Solution Methods for IVP

Method	Local Truncation Error	Explicit/Implicit	Stability
Euler	$\frac{hM}{2L}\left[e^{L(t_i-a)}-1\right]$	E	
Backward Euler	$O(h)$	I	
Higher Order Taylor	$O(h^n)$	E	
Midpoint	$O(h^2)$	E	
Runge-Kutta Order 2	$O(h^2)$	E	
Runge-Kutta Order 4	$O(h^4)$	E	
Adams-Bashforth	$O(h^4)$	E	
Adams-Moulton	$O(h^4)$	I	

We did not talk much about the stability of the algorithms, but this will prove to be an important characteristic of any method we choose for an IVP.

29.4 Derivation of multi-step methods

> Advanced: Not covered in class
>
> This derivation closely follows the proof in Burden and Faires, pages 304-305.

First we write:

$$\begin{aligned} y(t_{i+1}) - y(t_i) &= \int_{t_i}^{t_{i+1}} y'(t)dt \\ &= \int_{t_i}^{t_{i+1}} f(t, y(t))dt \end{aligned}$$

Let $y_i \approx y(t_i)$, and rearrange the equation to give us an expression for the approximation at the next time step t_{i+1}

$$y(t_{i+1}) \approx y_i + \int_{t_i}^{t_{i+1}} f(t, y(t))dt \tag{29.2}$$

This should remind us of a similar problem we studied earlier, namely the numerical approximation for an integral, i.e. quadrature. Recall that in the earlier case, we chose to replace the function by a polynomial, for which it will be easier to compute the integral. In that case, we used a Lagrange interpolating polynomial.

In this case, it will be more convenient to use a ***Newton backward-difference polynomial*** because we can more easily incorporate previously calculated values.

As reminder we can write the $m-1$ degree interpolating polynomial as (ref: equation 3.13, p. 130 textbook):

$$P_{m-1}(t) = \sum_{k=0}^{m-1} (-1)^k \binom{-s}{k} \nabla^k f(t). \tag{29.3}$$

We can then use this polynomial (along with the remainder term) as an approximation to $f(t, y)$

$$f(t, y) = P_{m-1}(t) + \frac{1}{m!} f^{(m)}(\xi_i, y(\xi_i))(t - t_i)(t - t_{i-1}) \dots (t - t_{i+1-m}) \tag{29.4}$$

where $\xi_i \in (t_{i+1-m}, t_i)$.

Substituting Equation 29.3 and Equation 29.4 into Equation 29.2, and taking the integral of both sides, yields:

$$\begin{aligned}\int_{t_i}^{t_{i+1}} f(t,y(t))dt = &\int_{t_i}^{t_{i+1}} \sum_{k=0}^{m-1} (-1)^k \binom{-s}{k} \nabla^k f(t_i, y(t_i))dt \\ &+ \int_{t_i}^{t_{i+1}} \frac{1}{m!} f^{(m)}(\xi_i, y(\xi_i))(t-t_i)(t-t_{i-1})\ldots(t-t_{i+1-m})dt.\end{aligned} \tag{29.5}$$

The integral is easier to solve by using the variable substitution:

$$\begin{aligned}t &= t_i + sh \\ dt &= hds\end{aligned}$$

in Equation 29.5

$$\begin{aligned}\int_{t_i}^{t_{i+1}} f(t,y(t))dt = h &\left[\sum_{k=0}^{m-1} \nabla^k f(t_i, y(t_i))\ (-1)^k \int_0^1 \binom{-s}{k} ds\right] \\ &+ \frac{h^{m+1}}{m!} \int_0^1 (s)(s+1)\ldots(s+m-1) f^{(m)}(\xi_i, y(\xi_i))ds\end{aligned}$$

The integrals involving the binomial function are easily computed. Using the values of the computed integrals in Table 5.12, we can write the formula as

$$\begin{aligned}\int_{t_i}^{t_{i+1}} f(t,y(t))dt = h &\left[f(t_i,y_i) + \frac{1}{2}\nabla f(t_i,y_i) + \frac{5}{12}\nabla^2 f(t_i,y_i) + \frac{3}{8}\nabla^3 f(t_i,y_i) + \ldots\right] \\ &+ \frac{h^{m+1}}{m!} \int_0^1 (s)(s+1)\ldots(s+m-1) f^{(m)}(\xi_i, y(\xi_i))ds\end{aligned}$$

Note: *The header in Table 5.12 in your textbook incorrectly states the second column. It should read* $(-1)^k \int_0^1 \binom{-s}{k} ds$

> **Note**
>
> Recall that $\nabla p_n = p_n - p_{n-1}, n \geq 1$ and $\nabla^k p_n = \nabla(\nabla^{k-1} p_n), k \geq 2$ (see p. 130, textbook)

The last step is to recognize that:

$$\begin{aligned}
\nabla^0 f(t_i, y_i) &= f(t_i, y_i) \\
\nabla^1 f(t_i, y_i) &= f(t_i, y_i) - f(t_{i-1}, y_{i-1}) \\
\nabla^2 f(t_i, y_i) &= \nabla(\nabla^1 f(t_i, y_i)) \\
\nabla^3 f(t_i, y_i) &= \nabla(\nabla^2 f(t_i, y_i))
\end{aligned}$$

to expand the backward difference terms in the integral, followed by collecting like terms to arrive at the Adams-Bashforth Four-Step ($m = 4$) Method:

$$\begin{aligned}
y_0 = \alpha_0,\ y_1 = \alpha_1,\ y_2 = \alpha_2,\ y_3 = \alpha_3, \\
y_{i+1} = y_i + \frac{h}{24}\left[55f(t_i, y_i) - 59f(t_{i-1}, y_{i-1}) + 37f(t_{i-2}, y_{i-2}) - 9f(t_{i-3}, y_{i-3})\right]
\end{aligned}$$

29.5

```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

30 Summary

Thank you for checking out this book. Any comments or suggestions would be appreciated. Please feel free to write to me at jcmeza@ucmerced.edu.

Happy Computing!

References

Apostol, T. M. 1974. *Mathematical Analysis*. Addison-Wesley Series in Mathematics. Addison-Wesley. https://books.google.com/books?id=Le5QAAAAMAAJ.

Bailey, David H. 1993. "Algorithm 719: Multiprecision Translation and Execution of FORTRAN Programs." *ACM Transactions on Mathematical Software* 19 (3): 288–319. https://doi.org/10.1145/155743.155767.

Berrut, Jean-Paul, and Lloyd N. Trefethen. 2004. "Barycentric Lagrange Interpolation." *SIAM Review* 46 (3): 501–17. https://doi.org/10.1137/S0036144502417715.

Davis, Philip J. 1975. *Interpolation and Approximation*. Courier Corporation.

Goldberg, David. 1991. "What Every Computer Scientist Should Know about Floating-Point Arithmetic." *ACM Computing Surveys (CSUR)* 23 (1): 548.

Hennessy, John L., and David A. Patterson. 2011. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Higham, Nicholas J. 2021. "The Mathematics of Floating-Point Arithmetic." *LMS Newsletter* 493: 35–41. https://nickhigham.files.wordpress.com/2021/04/high21m.pdf.

Higham, Nicholas J. 2002. *Accuracy and Stability of Numerical Algorithms*. 2nd ed. Philadelphia: Society for Industrial; Applied Mathematics.

———. 2004. "The Numerical Stability of Barycentric Lagrange Interpolation." *IMA Journal of Numerical Analysis* 24 (4): 547–56. https://doi.org/10.1093/imanum/24.4.547.

Knuth, Donald E. 1997. *The Art of Computer Programming: Fundamental Algorithms, Volume 1*. Addison-Wesley Professional.

Meza, Juan C. 2011. "Newton's Method." *Wiley Interdisciplinary Reviews: Computational Statistics* 3 (1): 75–78.

Overton, Michael L. 2001. *Numerical Computing with IEEE Floating Point Arithmetic*. Society for Industrial; Applied Mathematics. https://doi.org/10.1137/1.9780898718072.

Papakonstantinou, JM. 2009. "Historical Development of the BFGS Secant Method and Its Characterization Properties," April, 1167.

A Big O Notation

A.1 Introduction

Big O Notation is frequently used both in mathematics and computer science. While it can be confusing at first, the thing to remember is that it is mostly a means to characterize and understand how an algorithm or an approximation behaves asymptotically. In fact, one of the chief advantages to using this notation is that it hides some of the detailed information that isn't usually useful to the analysis.

The goals of this worksheet are to provide a few examples of Big O notation and to illustrate the usefulness of this notation in numerical analysis. There are two main uses that you will see for Big O Notation in this class. The first use is to understand how an approximation to a given function behaves - for example, when we use Taylor's Theorem to approximate a function. In these cases, we usually use h or x for our notation and implicitly assume that h or x are small or tending to zero.

The second use is to assess the computational workload of an algorithm as a function of the dimension of the problem, typically denoted by n. This case is much more prominent in computer science applications when analyzing the complexity of an algorithm, but we will have use for it as well in numerical analysis. In these cases, we usually assume that n is large or tending to infinity.

The power of Big O notation is that it is a high-level description that allows us to quickly say something about an algorithm or an approximation without getting into all the messy details. As such, it can be used to compare two algorithms to see which one is more efficient or to compare two approximations to see which one is more accurate.

But beware – in both cases the statements are understood to be ***in the limit.*** For specific cases there may be (and usually are) counterexamples.

A.2 Big O for approximations

Let's first review what we studied in the Error Lectures. There, we introduced the notation, $O(h)$ (read Big O of h), to mean that a quantity is proportional to the value of h (or x).

For example, let's suppose we have a quantity, $g(h)$ that depends on the parameter h. If

$$\lim_{h \to 0} \frac{|g(h)|}{h} = C < \infty \tag{A.1}$$

for some constant C, then we say that g is $O(h)$.

> **Other formulations**
>
> You may also see Big O notation written in a different but equivalent form:
>
> $$|g(h)| \leq C \cdot h, \tag{A.2}$$
>
> for some constant C as h goes to 0. Also note that in some applications, the quantity we are interested in is known to always be positive, so the absolute values are sometimes left off.

In class, we looked at an approximation to the derivative of a function. After writing down our Taylor polynomial, we discarded all higher order terms, and found that we could approximate the first derivative at x_0 by:

$$\left| f'(x_0) - \frac{f(x_0 + h) - f(x_0)}{h} \right| \approx \left| \frac{h}{2} f''(x_0) \right|.$$

Letting

$$g(h) = \left| \frac{h}{2} f''(x_0) \right|.$$

and using Equation A.1 we see that:

$$\lim_{h \to 0} \frac{|g(h)|}{h} = \lim_{h \to 0} \frac{\left| \frac{h}{2} f''(x_0) \right|}{h} = \left| \frac{f''(x_0)}{2} \right| = C < \infty$$

as long as the second derivative exists and is bounded at x_0.

Another use for Big O notation is to describe terms that we throw away when making an approximation. Let's take for example the Taylor series approximation for $\arctan(x)$:

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} + \cdots$$

as $x \to 0$. One can imagine truncating the series after a different number of terms, which would give us different degrees of accuracy, each one of which can be described in Big O notation.

$$\begin{aligned}\arctan(x) &= x - \frac{x^3}{3} + \frac{x^5}{5} - \cdots \\ &= x - \frac{x^3}{3} + O(x^5), \quad \text{... truncate after cubic term} \\ &= x + O(x^3). \quad \text{... truncate after linear term}\end{aligned}$$

> **Tip**
>
> The easiest way to remember this is that the term used in the Big O notation is the first term of all of the quantities that were thrown away - x^5 for the first approximation, and x^3 for the second approximation, *without any additional constants*. Note also, that the sign is not important in the last formulation.

The rationale for using Big O notation is that if $x \to 0$, then of the terms we're discarding, the error can be described predominantly by the first term discarded as all of the other terms will go to 0 much faster.

> **Equals doesn't mean equal**
>
> A word of caution. As mathematicians, we're used to thinking that $=$ means that the two sides of the equation are equal. When using Big O notation, this interpretation doesn't necessarily hold, so one needs to be careful when comparing terms. Donald Knuth calls these *one-sided equalities*. For a fuller explanation, check out the wikipedia entry on Big O Notation.

A.3 Big O for computational workload

In computer science, Big O notation is also used, but the applications are mostly to characterize the computational workload of an algorithm. Since many algorithms can be quite complicated, it can be difficult to compare one against another. In addition, counting every single arithmetic operation could be tedious (and with modern computers not as important).

Nonetheless, it is important to understand how the workload will behave as the size of a problem increases. An algorithm that works fine on a problem of dimension $1, 2$, or 3, can be prohibitively expensive when applied to a problem with dimension $1,000,000$. As such, computer scientists have used Big O notation to express how this workload will grow with the size of the problem without diving into all of the details of the algorithm. Recall that unlike the first case, when we use Big O notation here, n is assumed to be large.

Other formulations

Much like before you may see Big O notation written in a different form for the case using the problem size. According to Knuth(Knuth 1997), the number x_n is said to be $O(f(n))$:

$$|x_n| \leq M\ |f(n)|, \quad \forall \text{ integers } n \geq n_0$$

for some constants $M > 0, n_0 > 0$. It should also be noted that neither one of the constants M, n_0 are known.

Let's take a simple example. Suppose we wanted to compute the mean of a set of n numbers. The formula is:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

It is easy to see that the formula entails adding n numbers, which amounts to $n-1$ additions. We then have to divide by n, so there's 1 division. The total is therefore n floating point operations (*flops*).

Everything else being equal, that's also the computational workload for this simple algorithm. If $n = 100$, then we have 100 *flops*; if $n = 1,000,000$ then we have $1,000,000$ *flops*.

We say that this algorithm is $O(n)$, because the workload increases linearly with the dimension of the problem n.

Important

An important point to remember is that we are only interested with how the computational workload of an algorithm behaves as the dimension increases, and ***not with the details of the computation***.

Consider the following modification to the problem.

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} w_i x_i,$$

where the w_i are weights assigned to each of the x_i. If we counted *flops,* then in addition to the count before, we also have n multiplications to consider, one for each of the weights times the x_i. That increases the cost of the computation, but it only grows by a ***constant factor*** - the flop count is now $2n$. **The important point is that the algorithm is still $O(n)$.**

Finally, let's consider a slightly more complicated problem. Let's suppose we want to multiply a matrix A by a vector x. As you know, we can write this as:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

where

$$b_i = a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n = \sum_{k=1}^{n} a_{ik}x_k, \qquad i = 1, 2, \ldots, n.$$

Were we to write this in code to compute b we would need to have a ***nested loop*** to compute all of the elements of b.

> **Nested Loops**
>
> When we say a ***nested*** loop, we mean that a ***for*** loop is ***inside*** another ***for*** loop. The first loop is called the ***outer loop*** versus the one inside, which is called the ***inner loop***. That means whatever computational work is being done in the inner loop has to be done at each iteration of the outer loop.

In python, the matrix-vector multiply might look something like this:

```
# Example: Multiply an nxn matrix a by a vector x of dimension n
# (assume all vectors and matrices have been initialized somewhere else)
#
for i in range (0,n): # outer loop
  for k in range (0,n): # inner loop
    b[i] = b[i] + a[i,k]*x[k]
```

Here, we see that each of the b_i terms will require n multiplications and $n-1$ additions using the same argument as before. But we have to do this for all n of the b_i. So the total work is $n*(n+n-1) = 2n^2 - n$ *flops*. We say that this algorithm Is $O(n^2)$ since it grows as the square of the dimension of the problem. Of course the linear term is still there, but the n^2 term will dominate the computational work. Here, one can see that the workload will increase much faster with the dimension of the problem than the previous example.

> **Tip**
>
> One easy way to determine the Big O for computational workload is to look at the number of **nested *for*** loops required to compute a quantity. Notice that in the first 2 examples, we could compute the mean using one ***for*** loop. In the second example of a matrix-vector

multiply, we had to have a ***for*** loop inside another ***for*** loop. That usually means the algorithm is $O(n^2)$, the number 2 coming from the 2-level nested ***for*** loop and assuming both loops are of the same size.

A.4 Summary: Why is this important?

As we said before, the power of Big O notation is that it is a high-level description that allows us to say something about an algorithm or an approximation. This is useful when comparing the computational efficiency of algorithms or when comparing the accuracy of two numerical approximations. Most of all, it is critical to understand that the Big O notation only gives us a sense of how the algorithm/approximation behaves ***in the limit***.

For example, say we want to quickly estimate and compare running times of two different algorithms. Suppose for the sake of argument that the time to compute a given quantity was 1 second on a computer of your choosing for a problem of size $n = 10$.

If I use an $O(n)$ algorithm, and if the problem grows by a factor of 1000, then the algorithm should only take about 1×1000 seconds or about **16.67 minutes**. Again, there may be constants involved, but this should be a good estimate.

But if the algorithm is $O(n^2)$, the time will grow as the **square** of the size of the problem. That same problem would take about 1×1000^2 seconds or about **11.6 days**.

A.5 References

1. Math 130 Lecture Notes
2. Big O notation, https://en.wikipedia.org/wiki/Big_O_notation
3. The Art of Computer Programming, (Knuth 1997)


```
today <- Sys.Date()
format(today, format="Revised: %B %d %Y")
```

```
[1] "Revised: May 13 2024"
```

B Existence and Uniqueness of IVP

B.1 The Initial Value Problem (IVP)

The ***initial-value problem*** has the form:

$$y' = \frac{dy}{dt} = f(t, y), \quad a \leq t \leq b, \quad y(a) = \alpha \tag{B.1}$$

In the general case, we would consider a *system* of ODEs, i.e.

$$y' = \frac{dy}{dt} = f(t, y), \quad a \leq t \leq b, \quad y(a) = \alpha$$

where

$$y' = \begin{bmatrix} y_1' \\ \vdots \\ y_n' \end{bmatrix} \quad f(t, y) = \begin{bmatrix} f_1(t, y_1, \dots, y_n) \\ \vdots \\ f_n(t, y_1, \dots, y_n) \end{bmatrix} \tag{B.2}$$

Since all of the solution techniques we will study can be generalized to a system of ODEs, we will keep it simple for now and assume we have an IVP of the form given by Equation B.1.

Reduction of higher-order ODE to a system of ODEs

It can be shown that a general **n-th** order ODE can be written in the form of a system of ODEs.

$$y^{(m)}(t) = f(t, y(t), y'(t), \dots, y^{(m-1)}(t)).$$

Advanced

Most software packages assume that the IVP is given in ***autonomous form****.*

$$\frac{dy}{dt} = f(y)$$

i.e. f does not depend explicitly on t. This is generally achieved through the addition of an additional equation of the form …

B.2 Lipschitz Condition and Convex Sets

To start off with, let's present a few definitions that will prove useful in our analyses.

> **i** Remark:
>
> *One way to think about a Lipschitz condition (in a loose sense) is as being somewhere between continuity and continuously differentiable. Another way to think about it is as a strong form of uniform continuity.*

Definition. A function f(t,y) is said to satisfy a *Lipschitz condition* in the variable y on a set $D \subset \mathbb{R}^2$ if a constant $L > 0$ exists with

$$|f(t, y_1) - f(t, y_2)| \leq L|y_1 - y_2|, \tag{B.3}$$

whenever (t, y_1) and (t, y_2) are in D. The constant L is called a *Lipschitz constant* for f.

Knowing whether a function satisfies a Lipschitz condition, will often prove to be useful in our analysis of IVP.

Example.

Let $f(t, y) = t|y|$ on $D = (t, y)|1 \leq t \leq 2,\ -3 \leq y \leq 4$.

For any two points $(t, y_1), (t, y_2)$, we can write

$$\begin{aligned} |f(t, y_1) - f(t, y_2)| &= |\ t|y_1| - t|y_2|\ |, \\ &= t\ |\ |y_1| - |y_2|\ |, \\ &\leq 2\ |y_1 - y_2|. \end{aligned}$$

> **i** Note
>
> The reverse triangle inequality states that $\big|\ |y_1| - |y_2|\ \big| \leq |y_1 - y_2|$ (e.g., see (Apostol 1974))

The last line is true because $t \leq 2$ on the interval and using the (reverse) triangle inequality.

As a result, f satisfies a Lipschitz condition on D in the variable y with constant 2.

Next we develop the notion of a convex set.

Definition. A set $D \subset \mathbb{R}^2$ is said to be **convex** if whenever (t, y_1) and (t, y_2) belong to D, then

$$((1-\lambda)t_1 + \lambda t_2, (1-\lambda)y_1 + \lambda y_2) \tag{B.4}$$

also belongs to D for every λ in $[0, 1]$.

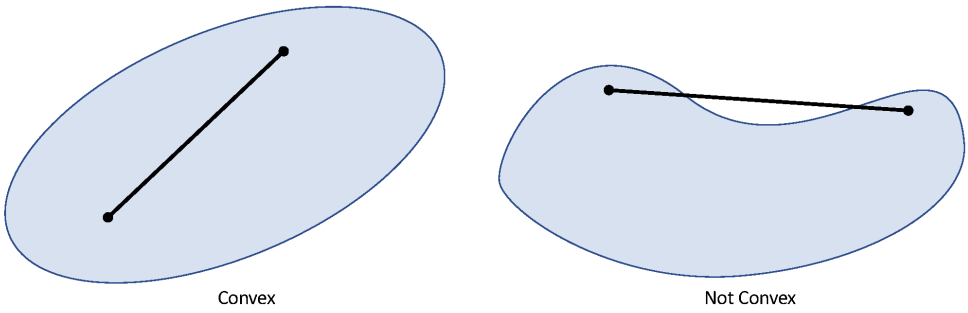


Figure B.1: Convex Set

Geometrically, (see Figure B.1) what this says is that whenever 2 points belong to D , then every point in the straight-line segment connecting the 2 points also belongs to D.

Example.

In addition to the previous 2 definitions, we will sometimes use the following characterization to help us show Lipschitz continuity under certain assumptions on the IVP.

Theorem 5.3 Suppose $f(t, y)$ is defined on a convex set $D \subset \mathbb{R}^2$. If a constant $L > 0$ exists with

$$\left|\frac{\partial f}{\partial y}(t,y)\right| \leq L, \quad \forall\ (t,y) \in D, \tag{B.5}$$

then f satisfies a *Lipschitz condition* on D in the variable y with Lipschitz constant L**.**

Proof. See Exercise 6 (Straightforward application of MVT).

Remark: *It will be useful to know when a function is Lipschitz continuous and Equation B.5 is sometimes easier to apply.*

Remark: *It should be noted that Equation B.5 is only a* ***sufficient*** *condition. In our previous example, $f(t,y) = t|y|$ is Lipschitz continuous but the partial derivative* ***does not*** *exist at $y = 0$.*

B.3 Fundamental Existence and Uniqueness of IVP

We're now in a position to state the **fundamental existence and uniqueness theorem** for the IVP problem Equation B.1 (or Equation B.2).

Theorem 5.4 Suppose $D = \{(t,y)|\ a \leq t \leq b,\ -\infty < y < \infty\}$, and that $f(t,y)$ is continuous on D. If f satisfies a Lipschitz condition on D in the variable y , then the initial-value problem

$$y'(t) = f(t,y), \quad a \leq t \leq b, \quad y(a) = \alpha$$

has a unique solution $y(t)$ for $a \leq t \leq b$.

Proof. See any text on ODEs (e.g. Birkhoff and Rota). For those interested see a brief sketch of the proof below.

Example: Show that the following IVP has a unique solution.

$$\begin{aligned} y' &= 1 + tsin(ty) \quad 0 \leq t \leq 2 \\ y(a) &= y(0) = 0 \end{aligned}$$

Solution:

To start with, we can see that $f(t,y)$ is continuous on D. So all we need to show is that it also satisfies a Lipschitz condition. To show this, first hold t constant and apply the MVT to $f(t,y) = 1 + tsin(ty)$.

For $y_1 < y_2$ the MVT states that there exists $\xi \in (y_1, y_2)$ such that

$$\begin{aligned}\frac{f(t,y_2)-f(t,y_1)}{y_2-y_1} &= \frac{\partial}{\partial y}f(t,\xi)\\ &= t^2 cos(\xi t)\end{aligned}$$

Rearranging and taking absolute values on both sides we get:

$$\begin{aligned}|f(t,y_2)-f(t,y_1)| &= |y_2-y_1|\cdot|t^2 cos(\xi t)|\\ &\leq 4|y_2-y_1|\end{aligned}$$

since $0 \leq t \leq 2$ and $|cos(\xi t)|$ is bounded by 1. This shows that f satisfies a Lipschitz condition on y with $L = 4$. Since f is continuous on D, by our theorem the IVP has a unique solution.

In class example: Show uniqueness for

$$\begin{aligned}y' &= \frac{sin(2t)-2ty}{t^2} \quad 1 \leq t \leq 2\\ y(1) &= 2\end{aligned}$$

Sketch of proof [@shampine1975], pp. 10-12

1. Assume that a solution extends from $y(a)$ to t_i.
2. It can be shown that the solution has to extend somewhere on the boundary of the box region R, here denoted as a box.
3. We then have to show that the solution can't reach the top or the bottom of R. Here we would use the Lipschitz condition.
4. So the solution has to reach the right side of the box R, with δ bounded away from 0.
5. Repeat steps 1-4 until you reach $t = b$.

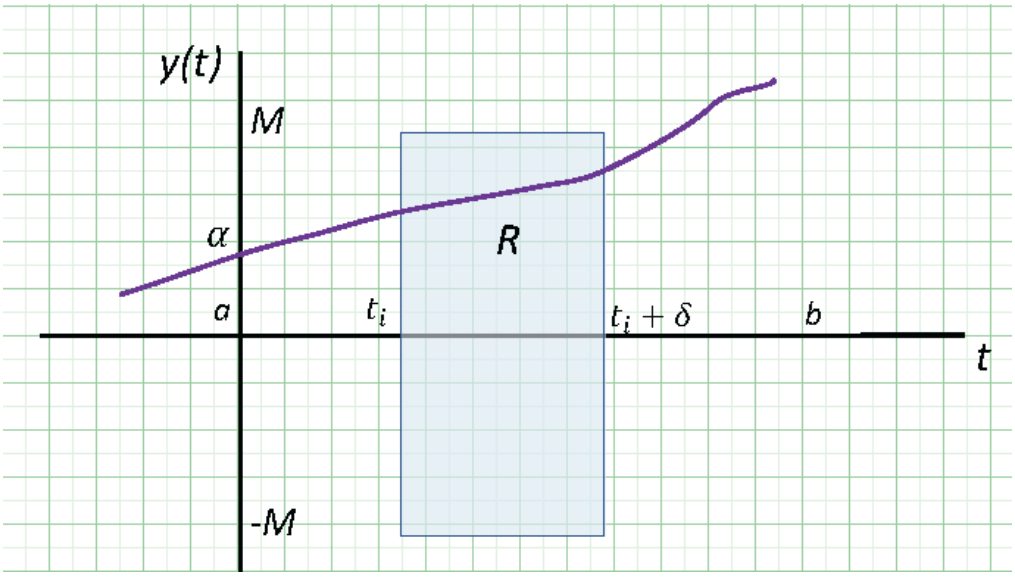


Figure B.2: Sketch of Proof

B.4 Well-Posed Problems

It will be important to understand when we can expect to be able to compute a solution to a problem in the situation when there are small changes to the statement of the IVP.

To do this we will need one more definition.

Definition. The IVP (Equation B.1) is said to be **well-posed** if:

- A unique solution, $y(t)$, to the problem exists, and
- There exist constants ϵ_0 and $k > 0$ such that for any ϵ with $\epsilon_0 > \epsilon > 0$, whenever $\delta(t)$ is continuous with $|\delta(t)| < \epsilon$ for all t in $[a, b]$, and when
- $|\delta_0| < \epsilon$,

the IVP

$$\frac{dz}{dt} = f(t, z) + \delta(t),\ a \leq t \leq b,\ z(a) = \alpha + \delta_0, \tag{B.6}$$

has a unique solution $z(t)$ that satisfies

$$|z(t) - y(t)| < k\epsilon\ \forall t \in [a, b]$$

Equation B.6 is called a **perturbed problem** associated with the original IVP given by Equation B.1.

> **Note**
>
> Numerical methods always solve a perturbed problem due to roundoff error on every computer. This leads to the notion that a numerical algorithm always solves (at best) a ***"nearby problem".***

> **Important**
>
> **Remark:** *The idea of "well-posed" refers to a property of a problem not the algorithm. For algorithms we talk about stability instead.*

This leads very naturally to ask, under what conditions is an initial value problem "well-posed"? We will tackle this problem next with a theorem that provides one criteria that can be used based on being able to show that $f(t, y)$ satisfies the Lipschitz condtion.

Theorem 5.6 Suppose $D = \{(t, y) | \ a \leq t \leq b, \ -\infty < y < \infty\}$. If f is continuous and satisfies a Lipschitz condition in the variable y on the set D, then the initial-value problem

$$\frac{dy}{dt} = f(t, y), \quad a \leq t \leq b, \quad y(a) = \alpha \tag{B.7}$$

is ***well-posed.***

In class example Show that the IVP defined by:

$$\begin{aligned} y' &= \frac{1+y}{1+t} \quad 0 \leq t \leq 1 \\ y(0) &= 1 \end{aligned}$$

Solution

Use Theorem 5.6 to show the IVP is well-posed. Consider

$$\begin{aligned} \frac{\partial f}{\partial y}(t, y) &= \frac{\partial}{\partial y}\left[\frac{1+y}{1+t}\right] \\ \frac{\partial f}{\partial y}(t, y) &= \frac{1}{1+t} \\ \left|\frac{\partial f}{\partial y}(t, y)\right| &= \left|\frac{1}{1+t}\right| \leq 1, \end{aligned}$$

since $0 \leq t \leq 1$. Since the partial derivative is bounded, by Theorem 5.3, f is Lipschitz, hence by Theorem 5.6, the IVP is well-posed.

C DataSets

This is a future section that will contain pointers to various open source data sets that are used in the book for the examples.