PROF. DR.-ING. MARK SCHUTERA

NUMERICAL METHODS

UNFINISHED LECTURE NOTES

Copyright © 2026 Prof. Dr.-Ing. Mark Schutera

PUBLISHED BY UNFINISHED LECTURE NOTES

Combobulated with the help of multiple large language model driven tools. Licensed under the Creative Commons Attribution-NonCommercial 4.0 International License ("CC BY-NC-SA 4.0"). You may not use this file for commercial purposes. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original You must obtain explicit permission from the author for uses beyond those permitted by this license. To view a copy of this license, visit https://creativecommons.org/licenses/by-nc-sa/4.0/. Unless required by applicable law or agreed to in writing, distributed material is provided on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the license for details.

These notes are, by their very nature, unfinished, and they improve with every reader. If you spot an error, disagree with a framing, or want to add a source, a question, or a position, open a pull request at https://github.com/Quillstacks/LectureMaterial/tree/main/lecturenotes/notes_numerischemethoden. Every contribution is welcome.

2026-05-06 · feisty cranberry Hermelin

"AN INFINITELY ACCURATE APPROXIMATION

IS NO LONGER AN APPROXIMATION."

PROBABLY SOMEONE SMART

Contents

Enter Numerical Methods 9

Floating-Point Arithmetic 17

Error Analysis 25

Newton Methods 35

Global Optimization 47

Numerical Integration 57

Index 71

Introduction

Numerical Methods are essential for solving mathematical problems that cannot be addressed analytically. This course will cover fundamental concepts, error analysis, problem conditioning, stability, and various numerical techniques to equip you with the skills needed to implement reliable and efficient algorithms in scientific computing and engineering applications.

This course will teach you everything you need to know to become proficient with numerical methods, and how to put them to good use in machine learning, data science, and engineering contexts.

Enter Numerical Methods

2026-04-05 · regal coconut Wachtel

Tapping into Computational Power

The Why

Numerical Methods [1] [2] [3] [4] are essential for solving mathematical problems that cannot be addressed analytically. Numerical methods are a subfield of mathematics in which we calculate our solutions not analytically exactly, but approximately.

Further Reading

[1] T. Arens, F. Hettlich, C. Karpfinger, U. Kockelkorn, K. Lichtenegger, and H. Stachel. *Mathematik*. Springer

[2] W. Dahmen and A. Reusken. *Numerik für Ingenieure und Naturwissenschaftler*. Springer

[3] M. P. Deisenroth, A. A. Faisal, and C. S. Ong. *Mathematics for Machine Learning*. Cambridge University Press

[4] P. Deuflhard and A. Hohmann. *Numerische Mathematik 1 - Eine algorithmisch orientierte Einführung*. De Gruyter

And we have good reason to do so.

- Many problems cannot be solved analytically, or are too complex to be practical.
- We can tap into computational power to get approximate solutions efficiently.

Approximation comes from Latin *approximare*, meaning "to come near to".

In machine learning and artificial intelligence, numerical methods are crucial for training models, optimizing parameters, and simulating complex systems where analytical solutions are infeasible. They enable efficient handling of large datasets and complex algorithms, ensuring that models can learn effectively from data while managing computational resources. Especially in deep learning, where models involve millions of parameters and require extensive computations, numerical methods facilitate the optimization processes that underpin model training, making them indispensable for advancing models.

Moore's Law states that computing power doubles approximately every two years. As of today consumer GPUs have thousands of cores and can perform trillions of floating point operations per second.

G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965

GPT-2: 1.5B release
https://openai.com/index/gpt-2-1-5b-release/

Hands On Experience

Later in this course, you will learn the theoretical foundations of the numerical methods that power modern AI and ML development. For now let's get a feeling why these technologies lean so heavily on numerical methods.

THE LIMITS OF SCALING ANALYTICAL SOLUTIONS become apparent when dealing with large-scale problems. Let's consider a simple example: Solving a system of two linear equations analytically with substitution.

ANALYTICS includes methods like substitution, elimination, matrix inversion, etc. you would learn in linear algebra.

$$\begin{bmatrix} 2 & 1 \\ 5 & 1 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} 11 \\ 13 \end{bmatrix} \tag{1}$$

$$\begin{aligned}
\text{From the first equation:} & \\
2w_1 + w_2 &= 11 \\
w_2 &= 11 - 2w_1 \\
\text{Substitute into the second equation:} & \\
5w_1 + 1(11 - 2w_1) &= 13 \\
5w_1 + 11 - 2w_1 &= 13 \\
3w_1 + 11 &= 13 \\
3w_1 &= 2 \\
w_1 &= \frac{2}{3} \\
\text{Then:} & \\
w_2 &= 11 - 2\left(\frac{2}{3}\right) \\
&= 11 - \frac{4}{3} \\
&= \frac{33 - 4}{3} \\
&= \frac{29}{3} \\
\text{Verify 1st equation:} & \\
2\left(\frac{2}{3}\right) + \left(\frac{29}{3}\right) &= \frac{4}{3} + \frac{29}{3} = \frac{33}{3} = 11 \\
\text{Verify 2nd equation:} & \\
5\left(\frac{2}{3}\right) + 1\left(\frac{29}{3}\right) &= \frac{10}{3} + \frac{29}{3} = \frac{39}{3} = 13
\end{aligned}$$

COMPUTATIONAL COMPLEXITY increases with the size of the system. Now take a shot and solve this larger system of three equations with three unknowns:

$$\begin{bmatrix} 2 & 1 & 3 \\ 1 & 4 & 2 \\ 3 & 2 & 5 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} 14 \\ 20 \\ 32 \end{bmatrix} \tag{2}$$

However, solvable for 2 equations, as the size of the system increases (e.g., thousands of equations with thousands of unknowns), analytical solutions become impractical due to computational complexity and time constraints.

The limits of analytical solutions then are also a general issue in nonlinear equations:

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{3}$$

Transcendental functions cannot be expressed as finite combinations of algebraic operations (addition, subtraction, multiplication, division, and roots) and thus lack closed-form solutions. The exponential term makes it impossible to isolate x using elementary functions like polynomials, rationals, or trigonometric functions.

Sigma $\sigma(x)$ is the sigmoid activation function commonly used in neural networks, and a transcendental function. A closed-form solution for $\sigma(x) = 0$ does not exist. $\sigma(x)$ approaches 0 asymptotically as x approaches negative infinity, but never actually reaches 0 for any finite value of x.

The Learning Objectives of this chapter aim at providing you with the abilities to:

- Explain when we deploy numerical methods and the problems that come with solving problems analytically.
- Discretization by transforming continuous mathematical problems into discrete, computer-solvable approximations.
- Apply basic numerical techniques to simple problems by hand, and crunch larger problems on a computer.

Discretization and Approximation

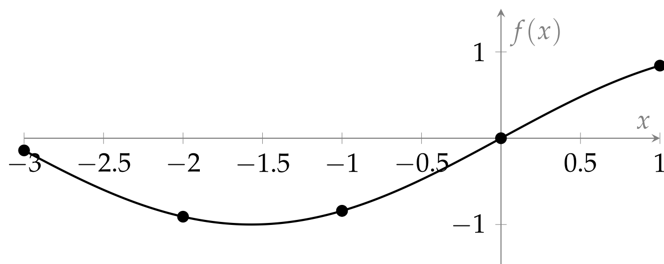

Figure 1: Continuous curve $f(x) = \sin(x)$ and its discretized version (black dots) on $[-3, 1]$ with step size $h = 1$.

DISCRETIZATION involves breaking continuous domains, such as time, space, or other functions, into finite steps or grids, and evaluating these functions at discrete points with finite precision:

REMEMBER, the brackets "[" and "]", are called a closed interval and mean that they include the values a and b in the definition set.

$$\begin{aligned} \text{Continuous function:} \quad & f(x), \quad x \in [a, b], \\ \text{Discretized function:} \quad & f(x_i), \quad x_i = a + ih, \quad i = 0, 1, \ldots, N, \end{aligned}$$

where

$$h = \frac{b-a}{N}, \tag{4}$$

is the step size, with N being the number of steps on the interval.

TO ANALYTICALLY FIND the Minimum of $\sin(x)$ on $[-3, 1]$ we seek $\min_{x \in [-3,1]} \sin(x)$. The minimum of $\sin(x)$ occurs where its derivative vanishes and the second derivative is positive.

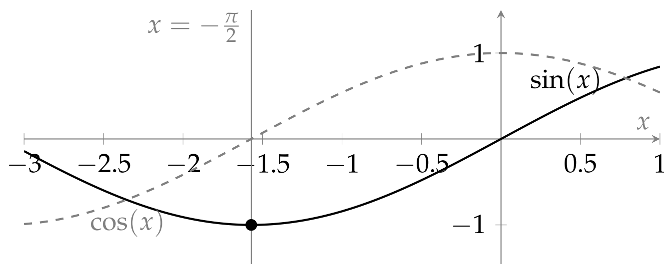

Figure 2: Continuous curve $f(x) = \sin(x)$ (black), its derivative $f'(x) = \cos(x)$ (gray, dashed), and the analytical minimum (black dot) on $[-3, 1]$. The dashed gray line marks the critical point where $\cos(x) = 0$ and the minimum of $\sin(x)$.

$$\begin{aligned} & f(x) = \sin(x) \\ & f'(x) = \cos(x) = 0 \implies x^* = \frac{\pi}{2} + k\pi, \; k \in \mathbb{Z} \end{aligned}$$

TRIGONOMETRIC RULES give us this general solution for $\cos(x) = 0$. If you forget you can always derive it from the unit circle.

Within $[-3, 1]$, the critical points are:

$$x_1 = -\frac{\pi}{2} \approx -1.5708$$
$$x_2 = \frac{\pi}{2} \approx 1.5708 \ (> 1, \text{ not in interval})$$

The minimum (or maximum) of a function on a closed interval $[a, b]$ can occur at a critical point (where the derivative is zero or undefined) inside the interval, or at the endpoints a or b themselves, even if the derivative at the endpoints is not zero. This is why, when searching for extrema on a closed interval, you must check both the critical points and the endpoints.

$$\begin{aligned}\text{Check endpoints and } x_1:&\\ \sin(-3) &\approx -0.1411\\ \sin(-\frac{\pi}{2}) &= -1\\ \sin(1) &\approx 0.8415\end{aligned}$$

Thus, the minimum is -1 at $x = -\frac{\pi}{2} \approx -1.5708$.

BRUTE FORCE means we try out all possible options and select the best one. Here with a grid search.

ON TO THE NUMERICAL SOLUTION, here we use a brute force grid search. When doing a grid search, we evaluate the function at discrete points over the interval and select the point with the minimum value. Discretizes $[-3, 1]$ with step size $h = 1$.

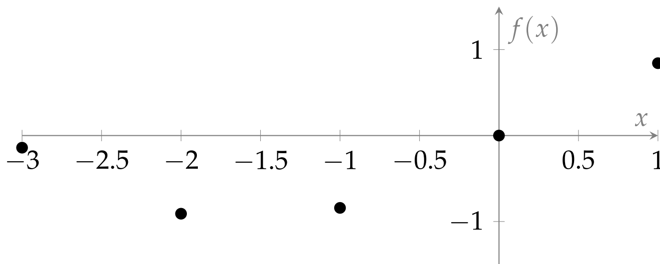

Figure 3: Continuous curve $f(x) = \sin(x)$ and its discretized version (black dots) on $[-3, 1]$ with step size $h = 1$.

$$\begin{aligned} x_0 &= -3, & \sin(-3) &\approx -0.1411\\ x_1 &= -2, & \sin(-2) &\approx -0.9093\\ x_2 &= -1, & \sin(-1) &\approx -0.8415\\ x_3 &= 0, & \sin(0) &= 0\\ x_4 &= 1, & \sin(1) &\approx 0.8415 \end{aligned}$$

Thus, the minimum among these is $\sin(-2) \approx -0.9093$ at $x = -2$.

THE MINIMUM among these is $\sin(-2) \approx -0.9093$ at $x = -2$. It is easy to see that the choice of step size h affects the accuracy of the approximation. A smaller step size would yield a closer approximation to the true minimum. Further the interval selection matters, in a sense that it .

APPROXIMATION ERROR is the difference between the analytical and numerical solutions. Here: $|-1-(-0.9093)| = 0.0907$.

Examples & Excercises

REMEMBER, our example from above?

$$\begin{bmatrix} 2 & 1 \\ 5 & 1 \end{bmatrix} \begin{bmatrix} w \\ b \end{bmatrix} = \begin{bmatrix} 11 \\ 13 \end{bmatrix} \tag{5}$$

EXCERCISES are for practice and reinforcing concepts. Try to solve them on your own first, try things, play with it, discuss, this is not a time trial. And there is no shame in not ending up at the right answer, in the same sense, that uncovering great questions and tossing them around is usually pretty fruitful on the long run.

THIS WE can also solve via brute force discretization and approximation. We discretize the variables w and b over a grid of possible values, solve according to these values and select the solution that minimizes the error.

LINEAR REGRESSION see how $xw + b$ forms a linear model, which can also be thought of as the most basic form of a single unit neural network θ.

LET'S PERFORM A GRID SEARCH for w and b with step size 2.5 over the range $w, b \in \{0, 2.5, 5, 7.5, 10\}$. For each combination, we compute the error:

ERROR is defined as the sum of absolute differences between the left and right sides of the equations:

$$\begin{aligned} \text{error}_1 &= |2w + b - 11|\,, \\ \text{error}_2 &= |5w + b - 13|\,. \end{aligned} \tag{6}$$

We evaluate all combinations and select the (w, b) pair with the smallest accumulated error.

THERE IS VALUE IN doing this by hand, as it helps to understand the mechanics of numerical methods. Also it gives an intuition on the computeational cost of brute-force methods, and later on we will find more efficient approaches. This was only 100 operations, and it does not fail to bring the message home that this is something you will want to hand over and automate on a computer. Which on the other hand does not even struggle with this size of a problem.

ENTER THE MACHINE. A lot of numerical methods are designed to be implemented on computers. In this lecture course, we will often switch between hand calculations for small examples and computer implementations for larger problems.

CODE IS HOSTED AS NOTEBOOKS and is to be followed up here `https://enlitenment.schutera.com/landing`.

YOU GET a jupyter notebook pre-hosted at your fingertips. With it you get a python template for this excercise. Use the self-reflection questions below to guide your while exploring and experimenting with the implementation.

w	b	$2w + b$ ($error_1$)	$5w + b$ ($error_2$)	**Error**
0	0	0 (11)	0 (13)	24
0	2.5	2.5 (8.5)	2.5 (10.5)	19
0	5	5 (6)	5 (8)	14
0	7.5	7.5 (3.5)	7.5 (5.5)	9
0	10	10 (1)	10 (3)	4*
2.5	0	5 (6)	12.5 (0.5)	6.5
2.5	2.5	7.5 (3.5)	15 (2)	5.5
2.5	5	10 (1)	17.5 (4.5)	5.5
2.5	7.5	12.5 (1.5)	20 (7)	8.5
2.5	10	15 (4)	22.5 (9.5)	13.5
5	0	10 (1)	25 (12)	13
5	2.5	12.5 (1.5)	27.5 (14.5)	16
5	5	15 (4)	30 (17)	21
5	7.5	17.5 (6.5)	32.5 (19.5)	26
5	10	20 (9)	35 (22)	31
7.5	0	15 (4)	37.5 (24.5)	28.5
7.5	2.5	17.5 (6.5)	40 (27)	33.5
7.5	5	20 (9)	42.5 (29.5)	38.5
7.5	7.5	22.5 (11.5)	45 (32)	43.5
7.5	10	25 (14)	47.5 (34.5)	48.5
10	0	20 (9)	50 (37)	46
10	2.5	22.5 (11.5)	52.5 (39.5)	51
10	5	25 (14)	55 (42)	56
10	7.5	27.5 (16.5)	57.5 (44.5)	61
10	10	30 (19)	60 (47)	66

Table 1: Grid search for w, b in $\{0, 2.5, 5, 7.5, 10\}$: values of $2w + b$ and $5w + b$ with errors to 11 and 13 in parentheses. The last column shows the accumulated error (sum of absolute errors). The minimum error occurs at $w = 0$, $b = 10$ with an error of 4 (marked with *).

Self-Reflection and Recap

Self-Reflection Questions which can guide your thoughts during the excercises and afterwards:

- Why is the choice of step size h important when discretizing a continuous function, and how does it affect the accuracy and the compute time of the numerical solution?
- How does the selection of the interval $[a, b]$ influence the results of discretization and the location of extrema found numerically?
- What are the main differences between a continuous function and its discretized version, and what are the implications for solving mathematical problems numerically?

RECAP of Key Concepts:

- Numerical Methods are essential for solving complex mathematical problems that lack analytical solutions.
- Discretization transforms continuous problems into discrete approximations suitable for computational methods.
- We can do numerical computations by hand for small problems to understand the mechanics, but computers are essential for larger problems.

ERRORS EVERYWHERE. Mathematical models are simplifications of reality, and numerical methods introduce additional errors through approximation.

TEASER. Can you think of a simple way to improve the accuracy to compute ratio of our grid search example from above?

$$f(x) \approx f(x_i), \quad x_i = a + ih \tag{7}$$

Numerical computation introduces a few types of errors, which we will need to understand in order to be able to fully harness this new methodology.

Floating-Point Arithmetic

2026-04-29 · lively date Hase

Getting used to Errors Everywhere

The Why

We saw that numerical methods introduce errors through approximation.

Where $f(x)$ be a continuous function on $[a, b]$. The discretized version $f(x_i)$ approximates $f(x)$ at discrete points x_i with an error that depends on the step size h and the smoothness of f. Numerical values can further only be stored approximately in a computer's memory, in floating-point representation. This leads to rounding errors when performing arithmetic operations, adding up to the truncation errors we already experienced when approximating infinite processes with finite ones, it is also called approximation error.

Modeling Error occurs as all models are simplifications of reality, and the difference between the model and the real-world system introduces an error. We will not dive deeper into this type of error in this course. Yet, mind the fine difference between what we term $f(x)$ and what actually is a $\tilde{f(x)}$.

This gives us good reason to understand these errors and their interplay with our machines.

- Numerical methods introduce rounding and truncation errors.
- Based on our machines these errors play out differently, and can amplify and accumulate.

In machine learning and artificial intelligence, you already are aware thatnumerical methods are crucial for training models, but of course floating-point arithmetic

is as relevant in model inference. Especially in compute-sparse environments on the edge, or when deploying quantized models [5], understanding floating-point arithmetic and the underlying mechanics comes handy.

On the Edge models use lower-precision arithmetic, such as 8-bit integers or even binary weights and activations.

Binary neural networks (BNNs) use weights and activations constrained to $\{-1, +1\}$, drastically reducing memory and computation requirements, but making floating-point representation and rounding effects critical.

[5] M. Courbariaux and Y. Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016. URL http://arxiv.org/abs/1602.02830

Hands On Experience

Let's get a feeling for rounding error with a simple example.

CONSIDER DIVIDING 10 BY 3. Write out the decimal expansion:

$$10 \div 3 = 3 \quad \text{remainder: } 1$$
$$\text{Bring down a 0: } 10 \div 3 = 3 \quad \text{remainder: } 1$$
$$\text{Bring down a 0: } 10 \div 3 = 3 \quad \text{remainder: } 1$$
$$\text{Bring down a 0: } 10 \div 3 = 3 \quad \text{remainder: } 1$$
$$\text{... and so on}$$
$$\Rightarrow \frac{10}{3} = 3.333\ldots$$

THE 3S REPEAT FOREVER. But when you write it down or enter it into a calculator or computer, you have to stop at some point:

$$\frac{10}{3} \approx 3.333 \quad \text{(rounded or truncated after 3 digits)} \tag{8}$$

What you will end up with is the rounding error: The difference between the true value and the value you get when you cut off (truncate) the expansion. No matter how many digits you write, as soon as you stop, you introduce an error:

$$\text{Rounding error} = |\,3.333333\ldots - 3.333\,| = 0.000333\ldots \tag{9}$$

The more digits you keep, the smaller the error, but it never disappears completely unless you write infinitely many digits, which well, you know is impossible.

THIS IS ROUNDING ERROR: Computers always store numbers with a finite number of digits, so rounding errors will inevitably show up and need to be managed.

Because as you will see, these small errors can accumulate and amplify and lead to significant inaccuracies in computations, especially in iterative algorithms common in numerical methods and machine learning.

THE LEARNING OBJECTIVES of this chapter aim at providing you with the abilities to:

- Understand the different types of numerical errors: Modeling, truncation, and rounding errors.
- Comprehend floating-point representation, machine epsilon, and loss of significance.
- Be able to handle numerical errors in practical computations.

TYPES of numerical representations in computers include:
Integer (int):
3
Floating-point (float):
3.3333333
Double precision (double):
3.3333333333333333
Fixed-point (e.g. 4-digits):
3.3333

IMPOSSIBLE? Mathematical annotation helps us with period: $3.\overline{3}$.

Floating Point Representation and Precision

Floating-point numbers are a way for computers to represent real numbers using a finite number of bits. The IEEE 754[6] standard is the most widely used format. A floating-point number is typically stored as:

$$x = (-1)^s \cdot m \cdot 2^e, \tag{10}$$

where s is the sign bit, m is the mantissa (or significand) which determines the precision, and e is the exponent which determines the scale (or magnitude) of the number. This allows for a wide range of values, but only a finite set of real numbers can be represented exactly. In IEEE 754 single precision (float), 32 bits are divided into 1 sign bit, 8 exponent bits, and 23 mantissa bits.

[6] IEEE Computer Society. Ieee standard for floating-point arithmetic. https://ieeexplore.ieee.org/document/4610935, August 2008. IEEE Std 754-2008 (Revision of IEEE Std 754-1985)

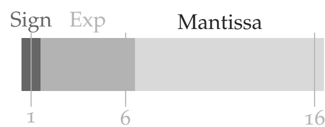

Figure 4: Bit layout of IEEE 754 **HALF (16)**: sign (dark), exponent (medium), mantissa (light).

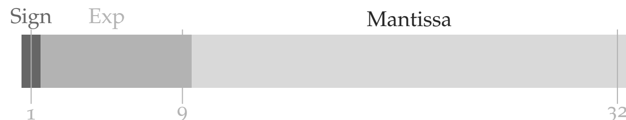

Figure 5: Bit layout of IEEE 754 **SINGLE (32)**: sign (dark), exponent (medium), mantissa (light).

Figure 6: Bit layout of IEEE 754 **DOUBLE (64)**: sign (dark), exponent (medium), mantissa (light).

A Bit, short for binary digit, is the most basic unit of information in computing and digital communications. It can have a value of either 0 or 1.

The exponent is stored with a bias to allow both positive and negative exponents. This allows floating-point numbers to represent both very small and very large magnitudes. For example, in IEEE 754 single precision, the exponent uses 8 bits and a bias of 127. The stored exponent E is related to the true exponent e by $e = E - 127_{10}$. The reason for the bias of 127 is that with 8 bits, the exponent field can store values from 0 ($2^0 - 1$) to 255 ($2^8 - 1$). By subtracting the bias (127), the actual exponent e can take both positive and negative values, centered around zero. This makes encoding and comparison of floating-point numbers easier in hardware.

CONSTRUCTING SCALE in floating-point representation:

$$10^1 = 10_{10} = 1010_2 = 1.010 \times 2^3, \quad E = 10000010_2$$
$$10^2 = 100_{10} = 1100100_2 = 1.100100 \times 2^6, \quad E = 10000101_2$$
$$10^3 = 1000_{10} = 1111101000_2 = 1.111101000 \times 2^9, \quad E = 10001000_2$$

THE LARGEST NUMBER IN SINGLE PRECISION is about 10^{38}, set by the largest exponent $e = +127$. The decimal exponent 38 comes from $\log_{10}(2^{128}) \approx 38.5$.

REMEMBER, mega (10^6), giga (10^9), tera (10^{12}), peta (10^{15}), exa (10^{18}), zetta (10^{21}), yotta (10^{24}), ronna (10^{27}), quetta (10^{30}),
no official SI prefix for (10^{33}).

THE MANTISSA determines how finely numbers can be represented between powers of two.

2-BIT MANTISSA EXAMPLE (UNNORMALIZED):

$$\text{Mantissa bit combinations:} \quad 00,\ 01,\ 10,\ 11$$
$$\text{Unnormalized significand:} \quad 0.xx_2$$

$$\begin{aligned} 00: &\quad 0.00_2 = 0 + 0 \times 2^{-1} + 0 \times 2^{-2} = 0.0 \\ 01: &\quad 0.01_2 = 0 + 0 \times 2^{-1} + 1 \times 2^{-2} = 0.25 \\ 10: &\quad 0.10_2 = 0 + 1 \times 2^{-1} + 0 \times 2^{-2} = 0.5 \\ 11: &\quad 0.11_2 = 0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 0.75 \end{aligned}$$

A TWO BIT MANTISSA means we can represent four distinct values between any two powers of two. A three bit mantissa allows eight distinct values, and so on. In general, with a mantissa of t bits, we can represent 2^t distinct values between any two powers of two, scaled up by the exponent.

MACHINE EPSILON ($\varepsilon_{\text{mach}}$) is the smallest positive number such that $1 + \varepsilon_{\text{mach}} \neq 1$ in the computer's arithmetic. It quantifies the upper bound on relative error due to rounding in floating-point arithmetic:

$$\varepsilon_{\text{mach}} = 2^{-t}, \tag{11}$$

where t is the number of bits in the mantissa.
In our 2-bit Mantissa example this is:

$$\varepsilon_{\text{mach}} = 2^{-2} = 0.25$$

For IEEE 754 single precision, $\varepsilon_{\text{mach}} \approx 1.19 \times 10^{-7}$; for double precision, $\varepsilon_{\text{mach}} \approx 2.22 \times 10^{-16}$.

THIS MEANS that the relative precision of floating-point numbers is approximately 2^{-t}, while the absolute precision depends on the magnitude of the number being represented:

$$\varepsilon_{\text{mach}} \cdot |x|. \tag{12}$$

ABSOLUTE PRECISION is just about to become clear, $1 : 2 = 0.5$ but $10 : 2 = 5$. Same number of steps (relative precision), but gaps of 0.5 vs 5.

In other words large numbers have larger absolute gaps between representable values than small numbers.

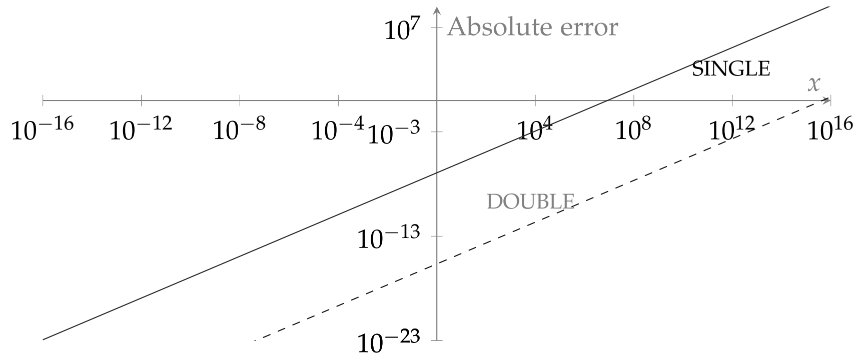

Figure 7: Absolute error for single (black, labeled SINGLE) and double (gray, dashed, labeled DOUBLE) precision as a function of the represented value x. The error grows linearly with x and is proportional to machine epsilon for each format.

Fixed-point representation is another way to store real numbers in computers, especially when you want predictable precision and performance. In fixed-point, you decide in advance how many bits are used for the integer part and how many for the fractional part. This means the gap between representable numbers (the precision) is always the same, no matter how large or small the value. Which gives more control.

Example: Suppose you want to store numbers between -1000 and 1000 using a 32-bit signed integer. Normally, a 32-bit integer can represent values from -2147483648 to 2147483647, which is much more than you need. To get more precision, you can use a scaling factor: For example, multiply every real number by 10^6 and store the result as an integer. So, the number 1.234567 becomes 1234567 in storage.

Precision, with a scaling factor of 10^{-6}, is equal to the smallest difference you can represent is 0.000001. The maximum rounding error is half a step, or $0.5 \times 10^{-6} = 0.0000005$.

Examples & Excercises

PSEUDO-ACCURACY in general is a unjustifiably high level of detail, creating a misleading, and artificial sense of accuracy.

LET'S TAKE A CLOSER LOOK at loss of significance, also called catastrophic cancellation.

This occurs when subtracting two nearly equal numbers, causing leading digits to cancel and leaving only the less significant, rounding-error-prone digits. This can greatly amplify rounding errors. Let's say we have:

SUPPOSE WE HAVE TWO NUMBERS a and b that are both stored in a computer with limited precision. Let's say each is rounded to 8 significant digits as a fixed-point representation:

$$a = 12345678.5$$
$$b = 12345678.0$$

But with 8 significant digits, they are stored as:

$$\tilde{a} = 12345679$$
$$\tilde{b} = 12345678$$

Now, subtract:

$$\tilde{a} - \tilde{b} = 12345679 - 12345678 = 1$$

Compare to the true difference:

$$a - b = 12345678.5 - 12345678.0 = 0.5$$

THE ERROR IN THE RESULT is 0.5, which is equal to the rounding error in $\tilde{a}$ or $\tilde{b}$ individually at the machine epsilon level 0.5. Let's have a look at what happens with a minimal deviation in the numbers.

$$a = 12345678.4$$
$$b = 12345678.0$$

But with 8 significant digits, they are stored as:

$$\tilde{a} = 12345678$$
$$\tilde{b} = 12345678$$

Now, subtract:

$$\tilde{a} - \tilde{b} = 12345678 - 12345678 = 0$$

Compare to the true difference:

$$a - b = 12345678.4 - 12345678.0 = 0.4$$

INFINITY in mathematics there is an infinity between 0 and 1. The difference between something and nothing.

THE DIFFERENCE IN THE TRUE RESULTS is 0.1, but comparing the machine results, we see that the result is 0 in one case and 1 in the other, which is a huge relative error. This shows how loss of significance can lead to large errors in computations, especially when the numbers being subtracted are very close to each other.

CODE is again to be found here https://github.com/Quillstacks/lecturecode_numericalmethods.git.

HANDS ON MACHINE EPSILON. But just before you head over to your machine, think about how you would determine the machine epsilon of any given system,

for a specific floating-point format, by deploying a program. Revisit how machine epsilon is defined. Write down your thoughts in pseudo-code. What would such a program show when run on a fixed-point system vs a floating-point system? Write down your thoughts. Then try different types on your machine in code. Write down things that you observe and reflect on them.

OVERFLOW, occurs when numbers with large magnitude are approximated as $+\infty$ or $-\infty$.

UNDERFLOW, occurs when numbers near zero are rounded to zero.

When you would build a calculator, which type would you choose and why [7]?

IS DOUBLE ENOUGH?

[7] D. Blochinger. Numerische methoden – foliensatz. Zentrum für Angewandte Ökonomik (ZAÖ), DHBW Ravensburg, 2025. URL https://www.economicon.de/repository/index.html. Illustration: Prof. Dr. Daniel Blochinger. Lizenz: CC BY-NC-SA 4.0. Stand: 28. Mai 2025. Weitere Materialien: https://www.economicon.de/repository/index.html

LOSS OF SIGNIFICANCE EXAMPLE. Now, think about how you would demonstrate loss of significance on your machine. Write down your thoughts in pseudo-code, before you read beyond this point.

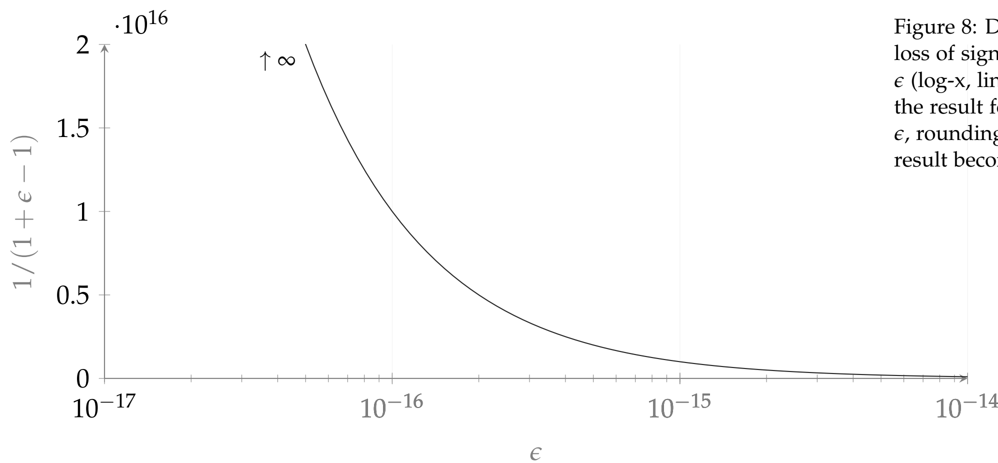

Figure 8: Demonstration of catastrophic loss of significance: $1/(1+\epsilon-1)$ vs ϵ (log-x, linear-y scale). For large ϵ, the result follows $1/\epsilon$. For very small ϵ, rounding error dominates and the result becomes infinite.

REASON ABOUT how computing $f(\epsilon) = 1/(a+\epsilon-b)$ for small ϵ and $a = b$ would do the job. What do you expect to see when ϵ is very small?

WHAT HAPPENS for $a > b$? Think along the lines of significant digits.

Self-Reflection and Recap

SELF-REFLECTION Questions which can guide your thoughts during the excercises and afterwards:

- How is floating-point representation structured, and what are its components?
- What is machine epsilon, and how does it relate to numerical precision?
- How do these concepts impact numerical computations in practice?

RECAP of Key Concepts:

- Floating-point representation allows computers to store a wide range of real numbers using a finite number of bits, but introduces rounding errors.
- Machine epsilon quantifies the smallest difference that can be represented in floating-point arithmetic, affecting the precision of numerical computations.
- Loss of significance occurs when subtracting nearly equal numbers, amplifying rounding errors and leading to inaccurate results.

KNOWING WHAT CAN GO WRONG. We are now close to understanding how we can define what is good and the quality of our methods. We now know that there is a true function f and an approximated function $\hat{f}$, further we have a true input x and a rounded input $\tilde{x}$. These effect and characterize our numerical methods.

TEASER. Can you think of metrics for numerical methods, based on the approximations and errors we discussed?

Error Analysis

2026-04-29 · lively date Hase

Some call it Error. I call it Character.

The Why

Conditioning, Stability, Consistency and Convergence. are fundamental concepts in numerical analysis that help us understand the behavior of numerical algorithms and their reliability in solving mathematical problems.

One last word on model error. With $f(\cdot)$ we imply the exact model of a system. In practice, models are simplifications of reality, for a distance between two points A and B we might use a simplified Manhattan or Euclidean model that does not account for terrain, mode of travel, or obstacles. So keep in mind that our $f(\cdot)$ here, is another $\hat{f}(\cdot)$.

Understanding Concepts to describe the behavior and characteristics of numerical methods, let us:

- Assess the sensitivity to small changes in the input data.
- Evaluate the sensitivity of the numerical solution to the small changes in the input data.
- Ensure that numerical methods produce accurate and reproducible results.
- Guarantee that our approximations converge to the true solution.

In machine learning and artificial intelligence, especially when training large models on vast datasets, understanding these concepts is crucial. Latest now, it should have become clear to you that training neural networks and other models, involves solving large-scale optimization problems, often requiring iterative numerical methods. In Deep Learning convergence has the center stage, then almost as an afterthought comes Complexity - the number of operations and amount of time it takes to get there.

In general you will be able to later on evaluate and describe the characteristics of AI models θ with exactly these concepts. So the maths aside, this will come in handy.

Deep Learning by Goodfellow, does a great job kick-starting you into numerical computation (Ch. 4) for machine learning. Check it out for further reading: .

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org

Hands On Experience

For starters let's get an intuitive feel for these concepts with some simple examples. Each concept will be illustrated with two quick exercises you can do by hand or in your head. It is two excercises as we want to highlight two sides of the same coin for each concept, introducing somewhat extremes on the spectrum. Because you already know it, let's revisit our discretized sine function example from Chapter 1.

CONDITIONING is about how sensitive the solution is to small changes in the input data.

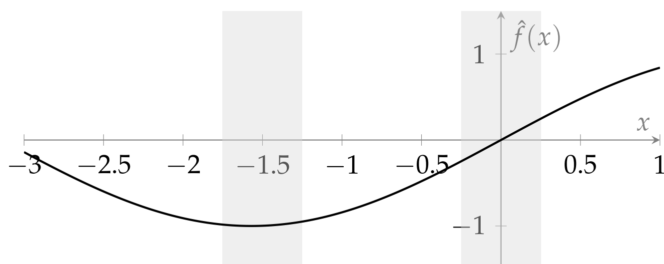

Figure 9: Continuous curve $\hat{f}(x) = \sin(x)$ on $[-3, 1]$ with two highlighted transparent vertical bands at $x = -1.5$ and $x = 0$ (width 0.5).

Let's assume for a moment that our approximated function equals the true function of the sine curve, however $\tilde{x}$ is a slightly perturbed version of x due to measurement errors or rounding (± 0.25). Now, look at the two gray bands at $x = -1.5$ and $x = 0$. The first is a well-conditioned region, where small changes in x lead to small changes in $f(x)$. The second band around $x = 0$ is ill-conditioned, where small changes in x can lead to large relative changes in $f(x)$.

STABILITY refers to the sensitivity of the numerical solution to the small changes in the input data.

THIS IS ABOUT FLOATING-POINT REPRESENTATION and not about step size or optimal approximation. Make sure to wrap your head around that, before moving on.

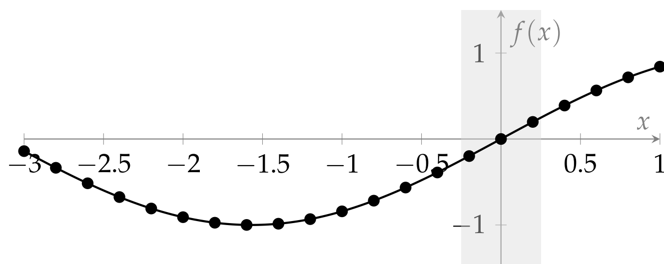

Figure 10: Continuous curve $f(x)_{.2} = \sin(x)$ on $[-3, 1]$ with a highlighted transparent vertical band at $x = 0$ (width 0.5), and discretized points with step size $h = 0.2$.

When paying attention to the gray band around $x = 0$, we can see that in the first figure with step size $h = 0.2$, the discretized points closely follow the sine curve, indicating a instability due to the changes in $\tilde{x} \pm 0.25$, resulting in fluctuations in the computed values of $\hat{f}(\tilde{x})$. In contrast, the second figure with step size $h = 1$

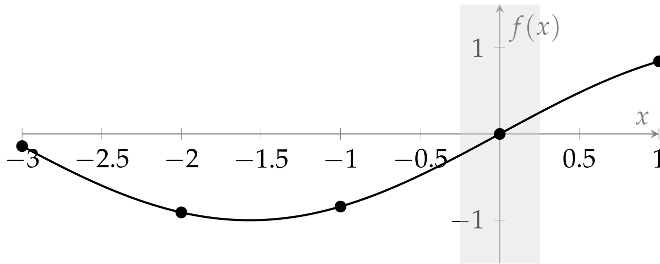

Figure 11: Continuous curve $f(x)_{1.} = \sin(x)$ on $[-3, 1]$ with a highlighted transparent vertical band at $x = 0$ (width 0.5), and discretized points with step size $h = 1$.

shows only a single discretized point within the gray band, leading to a stable approximation of the sine curve in that region - regardless of the deviations in $\tilde{x}$.

CONSISTENCY quantifies how well our numerical method matches the exact solution of the original problem. To make it easy to follow, we introduce a discretization with a step size of $h = 1$ and one with a step size of $h = 0.2$.

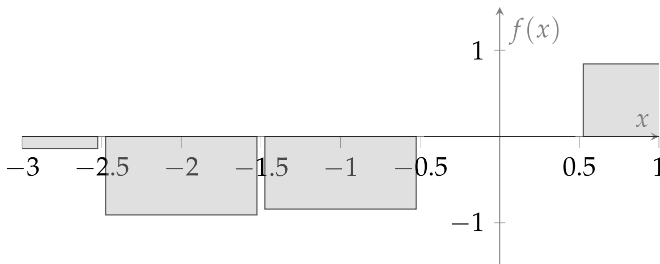

Figure 12: Bar plot of discretized values $f(x)_{1.} = \sin(x)$ on $[-3, 1]$ with step size $h = 1$.

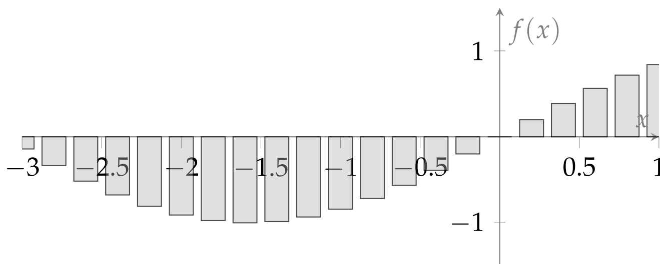

Figure 13: Bar plot of discretized values $f(x)_{.2} = \sin(x)$ on $[-3, 1]$ with step size $h = 0.2$.

Again, it is easy to see that for infinitesimally small step sizes $h \to 0$, the discretized points will get closer and closer to the true sine curve. In the end resulting in a perfect match between the numerical method and the exact solution of the original problem, which means optimal consistency.

INTUITIVE CONVERGENCE EXAMPLE WITH SINE, wanted. If you have a better idea to have the examples stick to the sine context, let me know.

CONVERGENCE emerges when we combine the ideas of stability and consistency. A numerical method is convergent if, as we refine our approximation $\hat{f}(x)$ (for example, by decreasing the step size h), the

computed solution approaches the exact solution of the problem regardless of the deviations in $\tilde{x}$, or by implicitly accounting for them.

The Learning Objectives of this chapter aim at providing you with the abilities to:

- Have an intuition about the concepts of conditioning, stability, consistency, and convergence in numerical methods.
- Analyze simple numerical problems quantitatively with respect to these concepts.

Quantitative Characterization of Numerical Methods

WE INTRODUCE the following notation. Let $f(\cdot)$ denote the exact (analytical) solution of a problem, and let $\hat{f}(\cdot)$ represent the numerical method (algorithm) that provides an approximation. The variable x stands for the exact input data, while $\tilde{x}$ refers to the actual input data used, which may be perturbed, for example, by measurement errors or rounding.

HAT VERSUS TILDE. The hat $\hat{}$ marks the numerical method (algorithm), while the tilde $\tilde{}$ marks a perturbed input. So $\hat{f}(\tilde{x})$ reads as: numerical method evaluated on perturbed input data.

CONDITIONING describes how sensitive a problem's solution is to small changes in the input data. Formally, the conditioning of a problem at x can be quantified by the condition number κ:

$$\kappa = |f(x) - f(\tilde{x})| . \quad (13)$$

κ is often normalized to express it as a relative measure.

STABILITY is given if small errors in the input or intermediate steps do not result in disproportionately large errors in the output. Mathematically, a stable method ensures that the error in the computed solution $\hat{f}(\tilde{x})$ remains bounded by a constant multiple of the error in the input data:

$$\left|\hat{f}(\tilde{x}) - \hat{f}(x)\right| \leq s \cdot |\tilde{x} - x| , \quad (14)$$

where s is a constant. For $s \approx 1$, the error in the computed solution develops linearly with the error in the input data.

CONSISTENCY means how well our numerical solution approximates the exact solution of the original problem:

$$\left|\hat{f}(x) - f(x)\right| \leq c \quad (15)$$

where c is a constant that quantifies the consistency error.

CONVERGENCE usually aims for zero error margin, which is often the exact solution $f(x)$. However, often we will experience non-zero convergence. If the method converges to a value different from $f(x)$, this indicates a systematic error (bias) in the method.

CONVERGENCE in general refers to our approximation reaching a specific stable limit. A method is convergent if:

$$\left|\hat{f}(\tilde{x}) - f(x)\right| \rightarrow \lim \quad (16)$$

That is, as both the real solution is approximated and deviations in data are controlled by an error margin.

Examples & Excercises

Finding the Minimum of the sine function on the interval $[-3, 1]$ one more time. Again discretize the function with two different step sizes $h = 1$ and $h = 0.2$. Further assume that the input data x is perturbed by $\tilde{x} = x \pm 0.25$ due to measurement errors and has a precision of two decimal places. Now analyze the conditioning, stability, consistency, and convergence of the numerical method used to find the minimum in both cases. Quantify the characteristics using the formulas provided in the previous section.

Conditioning is independent of the numerical method used, as it describes the sensitivity of the problem itself to changes in input data. Thus, the conditioning can be analyzed by purely examining how small changes in x affect the value of $\sin(x)$. We can see that around $x = -1$, where the minimum is, the sine function is relatively flat, indicating good conditioning. For sake of comparison we also consider $x = 0$, where the sine function changes rapidly, indicating poor conditioning. Solving this on paper we focus on these two regions to quantify the conditioning.

$$\begin{aligned}
\kappa(-1, +0.25) &= |\sin(-1) - \sin(-0.75)| \\
&\approx |-0.8415 - (-0.6816)| \\
&= 0.1599
\end{aligned}$$

$$\begin{aligned}
\kappa(-1, -0.25) &= |\sin(-1) - \sin(-1.25)| \\
&\approx |-0.8415 - (-0.9489)| \\
&= 0.1074
\end{aligned}$$

Which shows well-conditioning with $0.1074 \leq \kappa \leq 0.1599$.

$$\begin{aligned}
\kappa(0, +0.25) &= |\sin(0) - \sin(0.25)| \\
&\approx |0 - 0.2474| \\
&= 0.2474
\end{aligned}$$

$$\begin{aligned}
\kappa(0, -0.25) &= |\sin(0) - \sin(-0.25)| \\
&\approx |0 - (-0.2474)| \\
&= 0.2474
\end{aligned}$$

Which shows ill-conditioning with $\kappa \leq 0.247$.

STABILITY depends on the numerical method used to approximate the sine function and find its minimum. For the discretization with step size $h = 1$, the method is stable as small changes in $\tilde{x}$ lead to small changes in the computed values of $\hat{f}(\tilde{x})$. For the discretization with step size $h = 0.2$, the method is less stable, as small changes in $\tilde{x}$ can lead to larger fluctuations in the computed values of $\hat{f}(\tilde{x})$. Quantifying stability for both cases involves calculating the constant s in the stability inequality:

STABILITY, equals the conditioning of the system, for $h \to 0$. Show it.

$$\left|\hat{f}(\tilde{x}) - \hat{f}(x)\right| \leq s \cdot |\tilde{x} - x| \tag{17}$$

Due to symmetry we only consider the case $\tilde{x} = x + 0.25$.

A) See how the step size $h = 1$ leads to the same function value at both points. For reference, Fig. 12

For $h = 1$:

$$\begin{aligned}
s_{h=1,x=-1} \cdot |\tilde{x} - x| &= |\hat{f}(\tilde{x}) - \hat{f}(x)| \\
&= |\hat{f}_1(-0.75) - \hat{f}_1(-1)| \\
&= |\hat{f}_1(-1) - \hat{f}_1(-1)| \text{ see A)} \\
s_{h=1,x=0} &\approx 0 \div 0.25 \\
&\approx 0
\end{aligned}$$

For $h = 0.2$:

$$\begin{aligned}
s_{h=0.2,x=-1} \cdot |\tilde{x} - x| &= |\hat{f}(\tilde{x}) - \hat{f}(x)| \\
&= |\hat{f}_{0.2}(-0.75) - \hat{f}_{0.2}(-1)| \\
&= |\hat{f}_{0.2}(-0.8) - \hat{f}_{0.2}(-1)| \\
&= |-0.71736 - (-0.84147)| \\
s_{h=0.2,x=-1} &= 0.12411 \div 0.25 \\
&\approx 0.4964
\end{aligned}$$

Be aware that stability has anomalies in border regions of the discretization. This is especially a problem for piece-wise constant approximations.

CONSISTENCY is determined by comparing the numerical solution $\hat{f}(x)$ with the exact solution $f(x)$ for the minimum of the sine function in $[-3, 1]$. To again quantify consistency for both step sizes, we calculate the constant c in the consistency inequality:

$$\left|\hat{f}(x) - f(x)\right| \leq c \tag{18}$$

REMEMBER the analytical solution of min(sin(x)) on $[-3, 1]$ - revisit Chapter 1.

For $h = 1$:

$$\begin{aligned} c_1 &\geq \left| \hat{f}_1(-\frac{\pi}{2}) - f(-\frac{\pi}{2}) \right| \\ &\geq \left| \hat{f}_1(-1) - f(-\frac{\pi}{2}) \right| \\ &\geq |-0.84147 - (-1)| \\ &\geq 0.15853 \end{aligned}$$

For $h = 0.2$:

$$\begin{aligned} c_{0.2} &\geq \left| \hat{f}_{0.2}(-\frac{\pi}{2}) - f(-\frac{\pi}{2}) \right| \\ &\geq \left| \hat{f}_{0.2}(-1.2) - f(-\frac{\pi}{2}) \right| \\ &\geq |-0.94898 - (-1)| \\ &\geq 0.05102 \end{aligned}$$

Consistency improves with smaller step sizes, as expected.

CONVERGENCE combines the ideas of stability and consistency. To quantify convergence we calculate the total error between the numerical solution $\hat{f}(\tilde{x})$ and the exact solution $f(x)$:

$$\left| \hat{f}(\tilde{x}) - f(x) \right|. \tag{19}$$

Due to symmetry we only consider the case $\tilde{x} = x + 0.25$.
For $h = 1$:

$$\begin{aligned} \left| \hat{f}_1(\tilde{x}) - f(x) \right| & \left| \hat{f}_1(-1) - f(-\frac{\pi}{2}) \right| \\ & \left| \hat{f}_1(-0.75) - f(-\frac{\pi}{2}) \right| \\ & \left| \hat{f}_1(-1) - f(-\frac{\pi}{2}) \right| \\ & |-0.84147 - (-1)| \\ & = 0.1599 \end{aligned}$$

For $h = 0.2$:

$$\begin{aligned} \left| \hat{f}_{0.2}(\tilde{x}) - f(x) \right| & \left| \hat{f}_1(-1.2) - f(-\frac{\pi}{2}) \right| \\ & \left| \hat{f}_{0.2}(-0.95) - f(-\frac{\pi}{2}) \right| \\ & \left| \hat{f}_{0.2}(-1) - f(-\frac{\pi}{2}) \right| \\ & |-0.84147 - (-1)| \\ & = 0.1599 \end{aligned}$$

DUE TO the stability issues in the $h = 0.2$ case, convergence does not improve with smaller step sizes in this operation point. This allows us to understand two things, both numerical solutions convergence to the same limit. Judging from these two results we could also say that the method of iteratively reducing the step size h converges to the same limit as well. Notice that we use the characteristics for both a numerical solution and the numerical method.

HANDS ON COMPUTE-DRIVEN ERROR ANALYSIS. Let's brute-force our way through the error analysis of solving the two-equation system of Chapter 1. You will determine and optimize the conditioning, stability, consistency, and convergence of the numerical solution by optimizing the numerical method.

CODE is again to be found here `https://github.com/Quillstacks/lecturecode_numericalmethods.git`.

Self-Reflection and Recap

SELF-REFLECTION Questions which can guide your thoughts during the excercises and afterwards:

- Compared to the numerical solution to the minimum of the sine function, how well conditioned is the two-equation system?
- Does stability converge with smaller step sizes in the two-equation system? Against what?
- How do perturbations in the input data affect the numerical solution of the two-equation system? Is there an interplay with step size?
- Compare convergence with stability and consistency. What do you observe? Can you express your observations in terms of bias and variance?
- For ever smaller step sizes, do you observe a limit to the accuracy of the numerical solution? If so, why?
- What is another problem you observe while refining the step size?

RECAP of Key Concepts:

- Conditiioning, Stability, Consistency, and Convergence are fundamental concepts in numerical analysis that help us understand the behavior of numerical algorithms.
- These concepts can be expressed qualitatively and quantitatively to analyze numerical solutions and methods.

TEASER. So far we did brute-force numerical solutions. Can you think of a way to refine brute-force methods to get better results with less effort? Use the code of this lecture to design and test your idea.

Now that we understand and can characterize the behavior of individual numerical solutions, we can move on to understanding how to analyze entire numerical methods and algorithms. So far we have taken the assumption of where to look for a solution, inside a specific interval, for granted. This is usually not a given, and we will need to move away from local optimization methods in the next lecture.

Newton Methods

2026-04-05 · regal coconut Wachtel

A Step in the Right direction.

The Why

In the previous chapter, we characterized numerical methods with conditioning, stability, consistency, and convergence. But our brute-force grid search is fundamentally limited: it explores the solution space blindly, requiring $O(N^d)$ evaluations for N grid points in d dimensions.

The key insight is deceptively simple: instead of asking "what is the value here?" at every grid point, we also ask "which way should I go next?". The function's slope, its derivative, tells us the direction towards the solution. This transforms search fundamentally.

Understanding these methods allows us to:

- Use local information to make sophisticated steps instead of brute-force searches
- Achieve faster and more optimal convergence.

In Machine Learning and Deep Learning, Newton's method simplified to first order is also known as gradient descent. One could argue that the entire field of deep learning is built on the idea of using local gradient information to navigate toward minima in a high-dimensional loss landscape, to train models that are optimized towards specific objectives.

Backpropagation is of course as vital to distribute gradient information through the layers of a neural network, but the core optimization step is still a form of gradient-based navigation.

Hands On Experience

For starters let's get an intuitive feel for the power of using local information. We compare grid search with Newton's method on the same sine function from Chapter 3, but now we're finding where it crosses zero: $\sin(x) = 0$ on the interval $[2.5, 4]$ (the solution is near $\pi \approx 3.14159$).

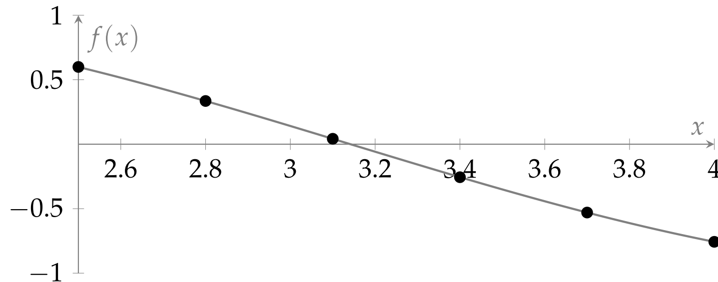

Figure 14: Continuous curve $f(x) = \sin(x)$ on $[2.5, 4]$, and discretized points with step size $h = 0.3$.

GRID SEARCH on $[2.5, 4]$ with step $h = 0.3$ evaluates blindly:

$$\begin{aligned}
&f(2.5) \approx 0.599 \\
&f(2.8) \approx 0.335 \\
&f(3.1) \approx 0.042 \quad \leftarrow \text{closest to zero} \\
&f(3.4) \approx -0.256 \\
&f(3.7) \approx -0.530 \\
&f(4.0) \approx -0.757
\end{aligned}$$

We found $x \approx 3.1$ with 6 evaluations, with an error $|3.1 - \pi| \approx 0.042$.

NEWTON'S METHOD has an adaptive approach to search. At every point we evaluate the function and its derivative, and use this local information to make a step towards the solution.

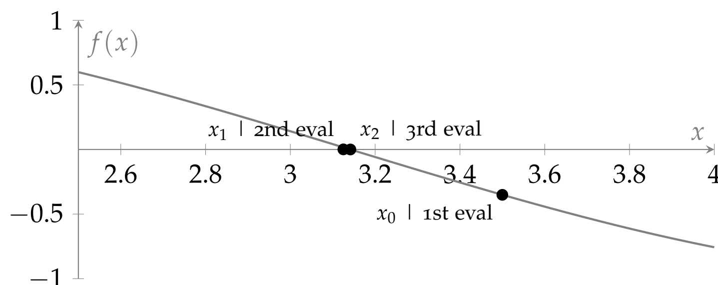

Figure 15: Newton's method iterations: starting from $x_0 = 3.5$, converging rapidly to $\pi \approx 3.14159$.

DERIVATIVE AS SOURCE OF LOCAL INFORMATION $f'(x) = \cos(x)$ gives the slope of the tangent at a given point. There are multiple

WE PAY FOR THIS SOPHISTICATION with more hyperparameters that we need to determine and more complex computations (derivative evaluation and division), but we gain in convergence speed.

ways to use this information to derive the direction and the size of the next step: We for sure want to move in the direction where the function decreases, then we could take a step with fixed size, or we could make the step size proportional to the derivative. Let's just take the next guess where the tangent crosses zero for now - which is the solution to the linear approximation of f at x_n.

Starting at $x_0 = 3.5$:

$$\begin{aligned}
x_1 &= x_0 - \frac{\sin(x_0)}{\cos(x_0)} \\
&= 3.5 - \frac{-0.351}{-0.936} \\
&= 3.5 - 0.375 \\
&= 3.125 \\
x_2 &= 3.125 - \frac{\sin(3.125)}{\cos(3.125)} \\
&= 3.125 - \frac{0.0008}{-0.9999} \\
&\approx 3.14159 \\
x_3 &\approx 3.14159 \\
\text{and } \sin(3.14159) &\approx 0
\end{aligned}$$

After just 2 iterations, to be fair this means 4 function/derivative evaluations, we have $x \approx 3.14159$ with error $< 10^{-5}$. The derivative told us where to look way more efficiently than we have been used to.

THE LEARNING OBJECTIVES of this chapter aim at providing you with the abilities to:

- Derive and apply Newton-Raphson iteration for root finding in 1D
- Derive and understnad the Taylor expansion and its role in Newton's method
- Analyze convergence rates and failure modes for Newton's method
- Understand and apply the Secant method when derivatives are unavailable

"NEWTON-RHAPSON" because Isaac Newton came up with the general idea, while Joseph Raphson simplified the approach into a practical iterative method.

Newton-Raphson and Taylor Expansion

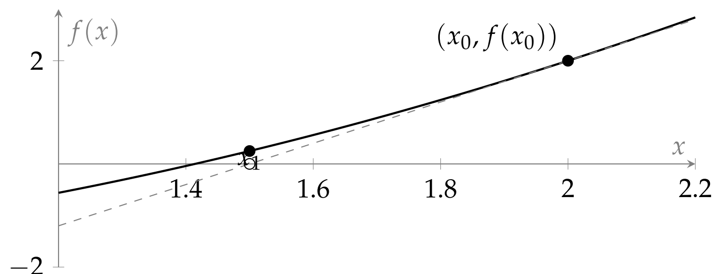

Figure 16: Newton's method geometric intuition: For $f(x) = x^2 - 2$, the tangent line at $(x_0, f(x_0))$ intersects the x-axis at x_1, our next approximation.

ABOVE you can see the geometric intuition behind Newton's method for a single iteration. Let's formalize the approach of finding x^* such that $f(x^*) = 0$.

THE LINEAR APPROXIMATION IS EFFECTIVELY given as:

$$f(x) \approx f'(x_n) \cdot (x - x_n) \quad \text{for } x \text{ close to } x_n + f(x_n) \tag{20}$$

NOTE: This linear approximation is the first-order Taylor expansion of f around x_n. We'll return to Taylor expansions later when we study accuracy and approximation formally.

FINDING THE NEXT GUESS: We want to find where this linear approximation crosses zero, because this is our best guess for where the actual function crosses zero. Setting the approximation to zero:

$$\begin{aligned} 0 &= f(x_n) + f'(x_n) \cdot (x_{n+1} - x_n) \\ f'(x_n) \cdot (x_{n+1} - x_n) &= -f(x_n) \\ x_{n+1} - x_n &= -\frac{f(x_n)}{f'(x_n)} \end{aligned}$$

This gives us the Newton-Raphson iteration formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{21}$$

Algorithm 1: Newton-Raphson Method

Require: Initial guess x_0, function f, derivative f', tolerance ε

1: $x \leftarrow x_0$
2: **while** $|f(x)| > \varepsilon$ **do**
3: Compute derivative: $d \leftarrow f'(x)$
4: **if** $|d| < \varepsilon_{\text{machine}}$ **then**
5: **error** "Derivative too small"
6: **end if**
7: Update: $x \leftarrow x - f(x)/d$
8: **end whilereturn** x

This error occurs when the derivative $f'(x_k)$ approaches zero, which would cause division by zero or numerical instability in Newton's method.

Taylor Expansion

So far we did just trust the linear approximation at x_n:

$$f(x) \approx f'(x_n)(x - x_n) + f(x_n), \tag{22}$$

to quickly find the next guess x_{n+1} and finally converge to the root, but is that trust well placed?
The answer lies in the Taylor expansion, a fundamental tool that tells us how well polynomials approximate smooth functions at a given point.

Let's derive it from first principles to understand why the linear approximation is a good choice for Newton's method, and how we can systematically improve it if needed. To illustrate let's approximate the sine function step by step.

Smooth: A function is smooth if it has derivatives of all orders, and is thus differentiable

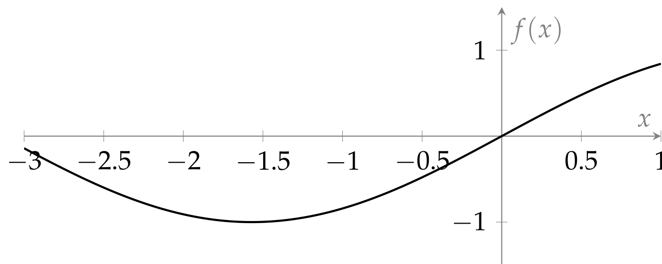

Figure 17: Continuous curve $f(x) = \sin(x)$ and its discretized version (black dots) on $[-3, 1]$ with step size $h = 1$.

0th Order: Match the function value. We want a polynomial $P(x)$ that approximates $f(x)$ near x_n. Start with matching at the point itself:

$$P(x_n) = f(x_n) \tag{23}$$

This comes very close to what grid search does: it only looks at the function value at discrete points, without any information about how the function behaves between those points.

1st Order: Match the first derivative. Near x_n, the function's slope matters. We want $P'(x_n) = f'(x_n)$.

Adding a linear term:

$$P(x) = f(x_n) + f'(x_n)(x - x_n) \tag{24}$$

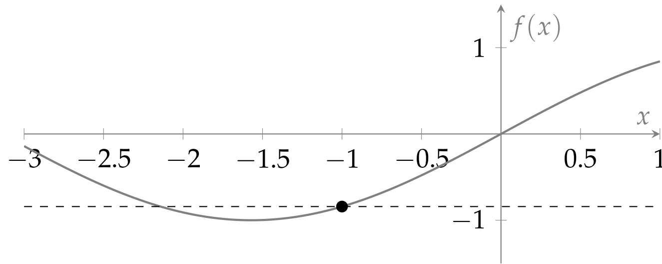

Figure 18: Continuous curve $f(x) = \sin(x)$ with constant approximation $P(x) = f(-1)$ (black dashed line) anchored at $x = -1$ (black dot).

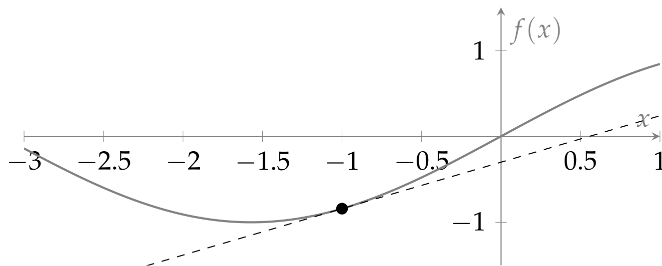

Figure 19: Continuous curve $f(x) = \sin(x)$ with linear approximation $P(x) = f(-1) + f'(-1)(x+1)$ (black dashed tangent line) anchored at $x = -1$ (black dot).

2nd Order: Match the second derivative. The curvature captures how the slope changes. We want $P''(x_n) = f''(x_n)$. Adding a quadratic term:

$$P(x) = f(x_n) + f'(x_n)(x - x_n) + \frac{f''(x_n)}{2}(x - x_n)^2 \tag{25}$$

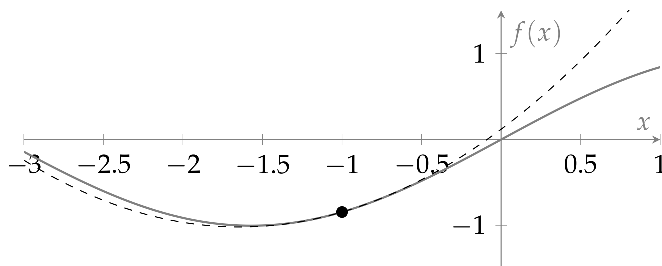

Figure 20: Continuous curve $f(x) = \sin(x)$ with quadratic approximation $P(x) = f(-1) + f'(-1)(x+1) + \frac{f''(-1)}{2}(x+1)^2$ (black dashed parabola) anchored at $x = -1$ (black dot).

nth Order: General pattern.

Why divide by 2? When we differentiate $(x - x_n)^2$ twice, we get 2. So if we want $P''(x_n) = f''(x_n)$, we need to divide by 2 to cancel this factor.

The Taylor expansion of a function f around a point x_n is given by:

$$f(x) = f(x_n) + f'(x_n)(x - x_n) + \frac{f''(x_n)}{2!}(x - x_n)^2 + \frac{f'''(x_n)}{3!}(x - x_n)^3 + \cdots$$

Or more compactly:

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(x_n)}{k!}(x - x_n)^k \tag{26}$$

Around a point. Visually we can see that the approximations are anchored on a specific point x_n. The Taylor expansion is a local approximation.

WHY LINEAR IS ENOUGH FOR NEWTON?
We want to solve $f(x^*) = 0$, not compute $f(x)$ as best as we can. Improving an approximation of course comes with a cost: We need to compute higher derivatives and evaluate more complex polynomials. In numerical computations we always need to decide where to cut off the Taylor series. We do so at some finite order, and the linear term is the first non-constant term that gives us directional information.

DERIVING THE CONVERGENCE RATE. Recall from Chapter 3, that a method is convergent if our approximation reaches a specific stable limit. For Newton's method finding a root x^* where $f(x^*) = 0$, we want:

$$|x_n - x^*| \to 0 \quad \text{as } n \to \infty \tag{27}$$

Thanks to the Taylor expansion, we can quantify how fast the error decreases.
Let $e_n = x_n - x^*$ be the error at iteration n. Applying a 2nd Order Taylor expansion to f around the true root x^*:

$$f(x_n) = f(x^*) + f'(x^*)(x_n - x^*) + \frac{f''(\xi)}{2}(x_n - x^*)^2, \tag{28}$$

where ξ is some point between x_n and x^*. Since $f(x^*) = 0$, this simplifies to:

$$f(x_n) = f'(x^*) \cdot e_n + \frac{f''(\xi)}{2}e_n^2 \tag{29}$$

Where $e_n = x_n - x^*$ is the error at iteration n.

NOW APPLY NEWTON'S FORMULA:

$$\begin{aligned} e_{n+1} &= x_{n+1} - x^* \\ &= x_n - \frac{f(x_n)}{f'(x_n)} - x^* \\ &= (x_n - x^*) - \frac{f(x_n)}{f'(x_n)} \\ &= e_n - \frac{f(x_n)}{f'(x_n)} \end{aligned}$$

and substitute the simplified Taylor expansion for $f(x_n)$:

$$e_{n+1} = e_n - \frac{f'(x^*) \cdot e_n + \frac{f''(\xi)}{2}e_n^2}{f'(x_n)}. \tag{30}$$

FOR POINTS CLOSE TO x^*, we can make the assumption that $f'(x_n) \approx f'(x^*)$,

$$e_{n+1} \approx e_n - \frac{f'(x^*) \cdot e_n + \frac{f''(\xi)}{2}e_n^2}{f'(x^*)} = e_n - e_n - \frac{f''(\xi)}{2f'(x^*)}e_n^2 \tag{31}$$

NOTICE THE ASSUMPTION PLAY OUT: $10^{-1} \to 10^{-2} \to 10^{-3} \to 10^{-6} \to 10^{-12}$. Once we're close enough and our assumptions hold (after iteration 2), the error squares perfectly, demonstrating quadratic convergence.

which makes the first order error term cancel out, leaving us with:

$$e_{n+1} \approx -\frac{f''(\xi)}{2f'(x^*)}e_n^2 \tag{32}$$

In more abstract terms, we can write this as:

$$|e_{n+1}| \leq C \cdot |e_n|^2 \tag{33}$$

Which means that the error at the next iteration is approximately proportional to the square of the current error, where C is a constant that depends on the function's curvature and slope at the root.

This is quadratic convergence.

Let's come back to our introductory example of computing $\sqrt{2}$ with Newton starting from $x_0 = 2$:

n	x_n	$\lvert e_n \rvert$ **(error)**
0	2.000000000	5.86×10^{-1}
1	1.500000000	8.58×10^{-2}
2	1.416666667	2.45×10^{-3}
3	1.414215686	2.13×10^{-6}
4	1.414213562	4.5×10^{-12}

Table 2: Newton's method for computing $\sqrt{2}$. See how the error approximately squares each iteration.

Newton's method can fail in several ways, which we will endure for now, and for which we will find solutions later on. As a brain teaser, it is a nice exercise to visualize the failure modes in this plot:

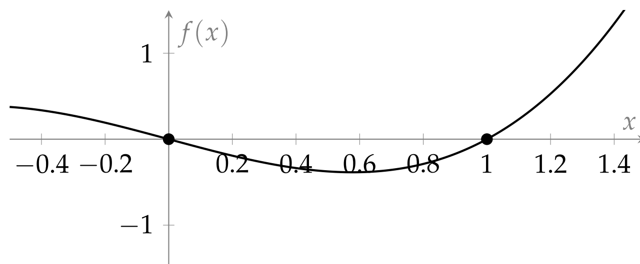

Figure 21: $f(x) = x^3 - x$ has multiple roots. Newton's method can fail in several ways: zero derivative (if $f'(x_n) = 0$, the tangent is horizontal and doesn't cross the axis), oscillation (especially for symmetrical functions Newton can cycle between points), and ambiguous starting point selection (x_0, heavily influences which root is found).

Now something we can not endure, is if we can't compute $f'(x)$.

This can happen because:

- The derivative is expensive to compute.
- The function is given as a black box (e.g., a simulation).
- The function isn't differentiable everywhere.

Secant Method

Here comes a clever way to approximate the derivative using only function evaluations, eliminating the need for an explicit derivative.

A POOR MAN'S DERIVATIVE. Approximate the derivative using two recent points:

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

Visually this is the slope of the secant line connecting the points $(x_{n-1}, f(x_{n-1}))$ and $(x_n, f(x_n))$ on the graph of f:

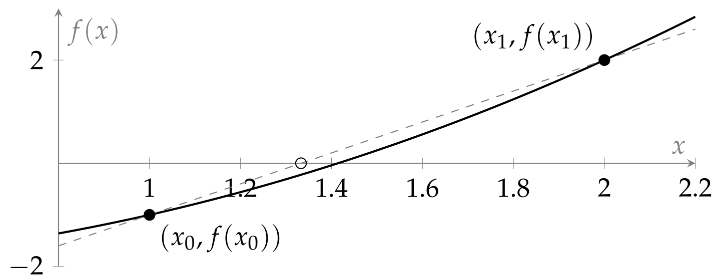

Figure 22: Secant method: the line through $(x_0, f(x_0))$ and $(x_1, f(x_1))$ gives the next approximation x_2.

NOTE, that this also means that we need two initial guesses x_0 and x_1 to start the iteration, instead of just one for Newton's method.

Substituting into Newton's formula, gives us the iterative Secant method:

$$x_{n+1} = x_n - f(x_n) \cdot \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \tag{34}$$

AND HERE'S THE ALGORITHMIC VERSION FORMULATED AS A LOOP:

Algorithm 2: Secant Method

Require: Initial guesses x_0, x_1, function f, tolerance ε
1: **while** $|f(x_1)| > \varepsilon$ **do**
2: Compute secant slope: $s \leftarrow (f(x_1) - f(x_0))/(x_1 - x_0)$
3: Store previous: $x_{\text{old}} \leftarrow x_1$
4: Update: $x_1 \leftarrow x_1 - f(x_1)/s$
5: $x_0 \leftarrow x_{\text{old}}$
6: **end whilereturn** x_1

Examples & Exercises

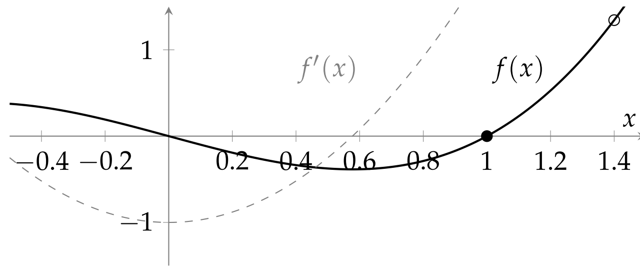

Figure 23: $f(x) = x^3 - x$ has multiple roots. We assume the root at $x = 1$ is the target. The gray dashed line shows the derivative $f'(x) = 3x^2 - 1$.

Code can be found at https://github.com/Quillstacks/lecturecode_numericalmethods.git.

Newton by hand. Find the root of $f(x) = x^3 - x = 0$ solving Newton's method geometrically, and then by hand. The roots are at $x = -1, 0, 1$. Let's find the root at $x = 1$ starting from $x_0 = 1.4$. The 1st order Taylor expansion around x_n gives us:

$$f(x) \approx f(x_n) + f'(x_n)(x - x_n),$$
$$f(x_n) \approx f(x) - f'(x_n)(x - x_n)$$

We have $f(x) = x^3 - x = 0$, with $f'(x) = 3x^2 - 1$, so we can rearrange to find the next guess, starting from $x_0 = 1.4$:

$$\begin{aligned}
x_1 &= x_0 - \frac{f(x_0)}{f'(x_0)} \\
&= x_0 - \frac{x_0^3 - x_0}{3x_0^2 - 1} \\
&= 1.4 - \frac{2.744 - 1.4}{5.88 - 1} \\
&= 1.4 - \frac{1.344}{4.88} \approx 1.127 \\
x_2 &= 1.127 - \frac{1.127^3 - 1.127}{3(1.127)^2 - 1} \\
&\approx 1.127 - \frac{0.432}{2.808} \\
&\approx 0.976 \\
x_3 &= 0.976 - \frac{0.976^3 - 0.976}{3(0.976)^2 - 1} \\
&\approx 0.976 - \frac{-0.072}{1.857} \\
&\approx 1.014
\end{aligned}$$

The root at $x = 1$ is found quickly.

Secant by hand. Apply the Secant method to the same problem with $x_0 = 1.2$, $x_1 = 1.4$. Again first geometrically, then by hand.

$$
\begin{aligned}
x_2 &= x_1 - f(x_1) \cdot \frac{x_1 - x_0}{f(x_1) - f(x_0)} \\
&= 1.4 - (0.744) \cdot \frac{1.4 - 1.2}{0.744 - 0.128} \\
&= 1.4 - 0.744 \cdot \frac{0.2}{0.616} \\
&\approx 1.4 - 0.241 \\
&\approx 1.159 \\
x_3 &= 1.159 - f(1.159) \cdot \frac{1.159 - 1.4}{f(1.159) - f(1.4)} \\
&\approx 1.159 - (0.283) \cdot \frac{-0.241}{0.283 - 0.744} \\
&\approx 1.159 - 0.283 \cdot \frac{-0.241}{-0.461} \\
&\approx 1.159 - 0.283 \cdot 0.522 \\
&\approx 1.159 - 0.148 \\
&\approx 1.011
\end{aligned}
$$

GEOMETRICAL FAILURE SEARCH. Find starting points x_0 for different failure modes: One where Newton's method fails due to zero derivative, one where it converges to a non-target root, and one where it oscillates between points.

ENTER THE MACHINE. Let's have a look on the convergence and convergence rates of Grid-Search, Newton and Secant method in Python. Then continue to experiment with differentstarting points, and observe the failure modes in practice. Try to first find the starting points geometrically, then try finding them analytically, before finally moving over to verifying in code.

Self-Reflection and Recap

SELF-REFLECTION Questions to guide your understanding:

- What is the main advantage of using Newton's method over Grid-Search?
- Why is a linear approximation sufficient for finding roots? What does the Taylor expansion tell us about this choice?
- Why is quadratic convergence so powerful? How many iterations would Newton need to go from error 10^{-1} to error 10^{-16}?

- In what situations would you prefer Secant over Newton? Why not in others?

RECAP of Key Concepts:

- Newton-Raphson gives (limited) convergence guarantees near simple roots.
- Taylor expansion provides a systematic way to approximate functions locally.
- You saw how Taylor expansion justifies the linear approximation in Newton's method and explains its quadratic convergence and allows to estimate the error in each iteration.
- In situations where the derivative is difficult to compute, the Secant method approximates the derivative from two points and converges.

TEASER. How can we make sure that we find the global solution, and not just a local one?

LOCAL METHODS FIND LOCAL SOLUTIONS. Newton converges to whatever optimum is nearby, it is also prone to oscillate between points, or diverge if the derivative is zero or near zero. In the next chapter, we will reformulate the root finding into an optimization problem. And from there we'll explore how to go on and handle global optimization when the landscape has many valleys or is otherwise pathologically difficult.

Global Optimization

2026-05-06 · feisty cranberry Hermelin

And then there were many.

The Why

In the previous chapter, we developed Newton's method and gradient descent for finding roots. The methods we learned about are local methods: They converge to whatever solution is nearby. But what if the nearby solution is bad? Or if another solution is much better but farther away?

Time to do away with oversimplifications. In this chapter, we will:

- Learn about Shekel's foxholes, a classic test function for global optimization
- Shortly show, how we can formulate an optimization problem as a root-finding problem.
- Introduce strategies for finding global minima

In machine learning and especially deep learning, understanding multi-dimensional and global optimization is crucial for training large models effectively, and to make them converge, and actually learn something useful. Neural network loss landscapes [8] are highly non-convex, do have flat regions, and contain many local minima, making optimization challenging.

Visualization of neural network loss landscapes Li et al. (2018) shows the complex terrain of neural network optimization. Different initializations lead to different final solutions.

[8] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein. Visualizing the loss landscape of neural nets. 31:6389–6399, 2018. URL https://proceedings.neurips.cc/paper_files/paper/2018/file/a41b3bb3e6b050b6c9067c67f663b915-Paper.pdf

Hands On Experience

SHEKEL'S FOXHOLES is a classic test function for global optimization with controlled multimodality. In 1D, it takes the simple form:

$$f(x) = -\sum_{i=1}^{m} \frac{c_i}{(x - a_i)^2 + r_i} \tag{35}$$

where a_i are the "foxhole" locations, c_i controls the depth of each hole, and r_i controls the width. By adjusting these parameters, we can create a landscape with multiple local minima and one global minimum, making it an ideal playground for testing optimization algorithms. Consider a specific example with three foxholes:

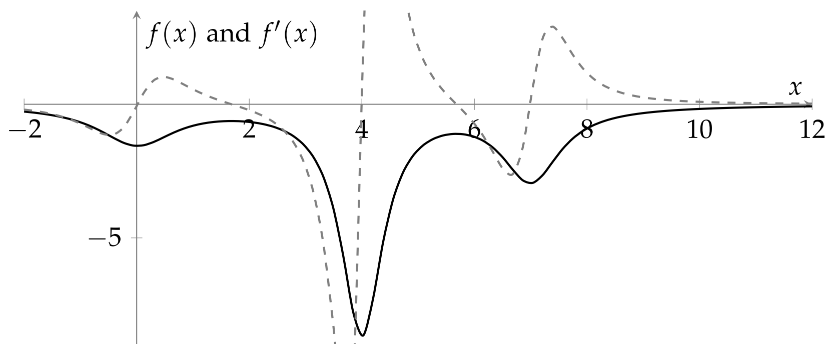

Figure 24: A 1D Shekel's foxholes function (black) with three local minima and its derivative (gray dashed). The function has valleys at $x = 0, 4, 7$, with the deepest (global minimum) at $x = 4$. The derivative (dashed line) crosses zero at each minimum.

$$f(x) = -\frac{1}{(x-0)^2 + 0.7} - \frac{1.7}{(x-4)^2 + 0.2} - \frac{1.1}{(x-7)^2 + 0.4}$$

THE DERIVATIVE, illustrated as the dashed line is the answer to formulating optimization as root finding. The minima of $f(x)$ correspond to the roots of $f'(x)$.

THE GLOBAL MINIMUM is at $x = 4$. The other two minima at $x = 0$ and $x = 7$ are local minima, respectively. This information is of course not available to us when we run an optimization algorithm. We only see the function values and gradients at the points we evaluate. But the knowledge comes in handy for exploring the behavior of optimization methods.

THE SENSITIVITY TO INITIAL CONDITIONS of x_0 is something you already experienced. If we start Newton's method near $x = 0$ or $x = 7$, we'll converge to a local minimum, not the global one at $x = 4$.

LET'S APPLY NEWTON'S METHOD to our 1D Shekel function[9] from different starting points and observe where the method converges. Let's show it for one starting point, try another one yourself, and then we will discuss the general behavior.

[9] J. Shekel. Test functions for multimodal search techniques. 1971

FROM $x_0 = 7.2$: Newton's method converges to the local minimum at $x \approx 7$. To apply Newton's method to optimization, we solve the root finding problem on $f'(x) = 0$, to reformulate the optimization problem. The Newton update becomes:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)} \tag{36}$$

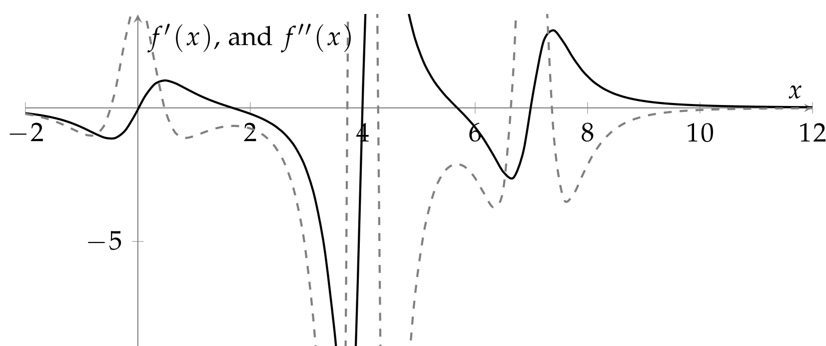

Figure 25: A 1D Shekel's foxholes function's first derivative (black), and second derivative (gray dashed). The function has valleys at $x = 0, 4, 7$, with the deepest (global minimum) at $x = 4$. The first derivative crosses zero at each minimum, and the second derivative is positive at minima (confirming local curvature).

The derivatives simplify to:

$$f'(x) = \frac{2x}{(x^2+0.7)^2} + \frac{3.4(x-4)}{((x-4)^2+0.2)^2} + \frac{2.2(x-7)}{((x-7)^2+0.4)^2}$$

$$\begin{aligned} f'(7.2) &\approx 0.005 + 0.100 + 2.27 \\ &= 2.38 \end{aligned}$$

$$f''(x) = \frac{2(0.7-3x^2)}{(x^2+0.7)^3} + \frac{3.4(0.2-3(x-4)^2)}{((x-4)^2+0.2)^3} + \frac{2.2(0.4-3(x-7)^2)}{((x-7)^2+0.4)^3}$$

$$\begin{aligned} f''(7.2) &\approx -0.002 - 0.091 + 7.23 \\ &= 7.14 \end{aligned}$$

$$\begin{aligned} x_1 &= 7.2 - \frac{2.38}{7.14} \\ &\approx 6.87 \end{aligned}$$

$$\begin{aligned} x_2 &\approx 7.02 \\ x_3 &\approx 7.00 \end{aligned}$$

See how root finding methods can be used for optimization by applying them to the derivative of the function. The derivative $f'(x)$ is zero at extrema, so at this point we find minima or maxima.

LOCAL MINIMA NOTATION: The asterisk (*) is conventionally reserved for global optima. To distinguish local from global minima, consider alternative notation like x° (x-circle) or x_i for the i-th local minimum.

Here, we were unlucky, and the method is attracted to the nearby foxhole at $x^{\circ} \approx 7$.

THE FUNDAMENTAL PROBLEM HOWEVER IS that local optimization only guarantees finding a nearby extrema. Even with convergence guarantees, we would have got lucky starting from $x_0 = 3$, but from $x_0 = -2$ or $x_0 = 8$, we miss the global minima. Enough motivation for the global optimization strategies we'll discuss next.

THE LEARNING OBJECTIVES of this chapter aim at providing you with the abilities to:

- Explain why local methods cannot guarantee finding global minima
- Apply strategies for finding global extrema
- Understand how gradient descent helps to direct optimization towards minima

Global Optimization Strategies

Let us formalize local and global extrema.

LOCAL OPTIMIZATION finds a nearby minimum:

$$\mathbf{x}^* = \arg\min_{\mathbf{x}\in\mathcal{N}(\mathbf{x}_0)} f(\mathbf{x}) \tag{37}$$

where $\mathcal{N}(\mathbf{x}_0)$ is some neighborhood of the starting point.

$\mathcal{N}$ IS A SET of points around $\mathbf{x}_0$. For example, $\mathcal{N}(\mathbf{x}_0) = \{\mathbf{x} : \|\mathbf{x} - \mathbf{x}_0\| < \epsilon\}$ for some radius ϵ.

GLOBAL OPTIMIZATION finds the best minimum:

$$\mathbf{x}^* = \arg\min_{\mathbf{x}\in\Omega} f(\mathbf{x}) \tag{38}$$

where Ω is the entire search domain. It is easy to see how the search domain can be inifinitely large.

CONVEXITY IS THE DIVIDING LINE. For convex problems, any local method will find the global optimum. The difficulty arises in non-convex optimization, where the landscape contains multiple local minima, saddle points, and plateaus.

CONVEX VS. NON-CONVEX: A function f is *convex* if $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)$ for all $\lambda \in [0,1]$. For convex functions, every local minimum is the global minimum—local methods suffice. Non-convex functions (like Shekel's foxholes or neural network loss surfaces) are the hard case.

RANDOM RESTARTS is the simplest global strategy, in its notion it feels like brute-forcing again:

Algorithm 3: Random Restarts

Require: Search domain Ω, local optimizer LocalOptimize, restarts K
1: $\mathbf{x}_{\text{best}} \leftarrow$ None; $f_{\text{best}} \leftarrow +\infty$
2: **for** $k = 1$ to K **do**
3: $\quad\mathbf{x}_0 \leftarrow \text{RandomPoint}(\Omega)$
4: $\quad\mathbf{x}^* \leftarrow \text{LocalOptimize}(\mathbf{x}_0)$
5: $\quad$**if** $f(\mathbf{x}^*) < f_{\text{best}}$ **then**
6: $\quad\quad\mathbf{x}_{\text{best}} \leftarrow \mathbf{x}^*$; $f_{\text{best}} \leftarrow f(\mathbf{x}^*)$
7: $\quad$**end if**
8: **end for**
9: **return** $\mathbf{x}_{\text{best}}$

If the basin of attraction of the global minimum has probability p, then with K restarts:

$$P(\text{find global}) = 1 - (1-p)^K \tag{39}$$

NEURAL NETWORK INITIALIZATION and training multiple networks with different seeds is implicit random restarts. However, this is not practical for large models.

For $p = 0.1$ and $K = 20$: $P = 1 - 0.9^{20} \approx 0.88$. Usually it is not possible to know p in advance. A heuristic to set K without the knowledge of p is to increase K until the best solution stabilizes (no improvement after several restarts). Patience, is a common hyperparameter for this heuristic, which controls how many restarts we wait without improvement before stopping.

BASIN HOPPING explores the landscape by alternating between local optimization and random jumps. This is different from random restarts, which completely resets the search. Basin hopping allows us to escape local minima while still leveraging local optimization to find better solutions.

Algorithm 4: Basin Hopping

Require: Initial point $\mathbf{x}_0$, local optimizer LocalOptimize, jump distribution for δ, iterations K

1: $\mathbf{x}_{\text{best}} \leftarrow \mathbf{x}_0$; $f_{\text{best}} \leftarrow f(\mathbf{x}_0)$
2: $\mathbf{x}_{\text{current}} \leftarrow \mathbf{x}_0$
3: **for** $k = 1$ to K **do**
4: $\quad \mathbf{x}^* \leftarrow \text{LocalOptimize}(\mathbf{x}_{\text{current}})$
5: $\quad$ **if** $f(\mathbf{x}^*) < f_{\text{best}}$ **then**
6: $\quad\quad \mathbf{x}_{\text{best}} \leftarrow \mathbf{x}^*$; $f_{\text{best}} \leftarrow f(\mathbf{x}^*)$
7: $\quad$ **end if**
8: $\quad$ Sample jump: $\delta \sim \mathcal{D}$
9: $\quad \mathbf{x}_{\text{current}} \leftarrow \mathbf{x}^* + \delta$ $\triangleright$ Random jump
10: **end for**
11: **return** $\mathbf{x}_{\text{best}}$

A jump distribution is specified as $\delta \sim \mathcal{D}$, which could be a Gaussian distribution centered at zero, or a uniform distribution over a certain range.

NOISY NEWTON follows the same idea around hopping, by adding Gaussian noise to the Newton step: $\mathbf{x}_{n+1} = ... + \xi_n$.

SIMULATED ANNEALING combines basin hopping with a probabilistic acceptance criterion for accepting point candidates. In the beginning, when hopping into a worse solution, it is accepted with probability $\exp^{-\Delta f/T}$, where T (temperature) decreases over time. Later on, the algorithm becomes more conservative and only accepts candidate points that improve the objective, allowing it to converge to an optimum.

Δf, is the increase in the objective function value when moving from the current solution to a worse candidate solution.

STOCHASTIC METHODS add noise by approximating the loss landscape to escape local optima. Let's approximate the 1D shekels function as a second order function by picking three points on the foxhole curve.

THE TAKE AWAY is that the loss landscape is not static, but is to be engineered and formed. By approximating the loss landscape on mini-batches (different selection of points), we get a noisy estimate of the true loss landscape, which can help us escape local minima. The loss landscape is then of course heavily influenced by the choice

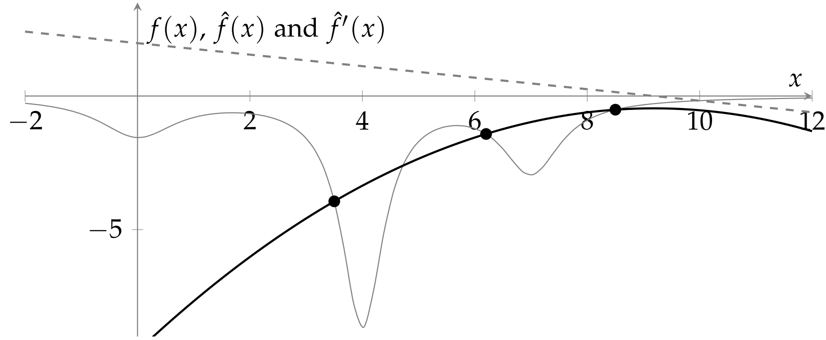

Figure 26: The original Shekel function (gray thin) with a second-order approximation (solid black) fitted through three sample points (dots). The derivative of the approximation is shown as a gray dashed line. The quadratic captures the general trend but smooths out the individual foxholes.

of the loss function itself. That being said, the Newton step is very sensitive to noise, we will see other approaches that deal better with this.

IN DEEP LEARNING, we typically use local methods (SGD, Adam) but hope that: (1) most local minima are roughly equally good, (2) overparameterization makes bad minima rare, (3) mini-batch noise helps escape sharp minima, (4) adaptive learning rates help navigate plateaus.

Newton's Method for finding Minima

IN THE HANDS-ON, we saw that reformulating optimization as root finding allows us to apply Newton's method to find optima:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}. \tag{40}$$

MINIMA, MAXIMA, AND SADDLE POINTS. Newton's method finds points where $f''(x) = 0$, but these can be minima, maxima, or saddle points. The second derivative test distinguishes them:

(a) $f''(x^*) > 0$: local minimum (curve bends upward)

(b) $f''(x^*) < 0$: local maximum (curve bends downward)

(c) $f''(x^*) = 0$: inconclusive (possible saddle point or inflection point)

In order to direct the Newton method towards minima only, we can modify the update as such:

HOW WOULD YOU modify the update to find maxima instead?

$$x_{n+1} = x_n - \frac{f'(x_n)}{|f''(x_n)|}. \tag{41}$$

In this way we ensure that we always move in the direction of decreasing $f(x)$. This forces the step to always move in the direction of the negative gradient (downhill so to speak).

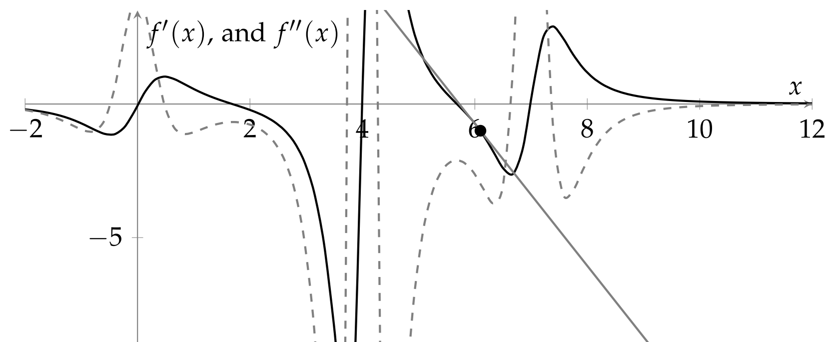

Figure 27: Solid black: the first derivative $f'(x)$ of the 1D Shekel function; gray dashed: the second derivative $f''(x)$. The black marker at $x = 6.1$ is a sample location; the solid gray line is the tangent to $f'(x)$ at that point (slope ≈ -2.66). The sign and magnitude of $f''(x)$ determine the Newton update direction and step size when solving $f'(x) = 0$.

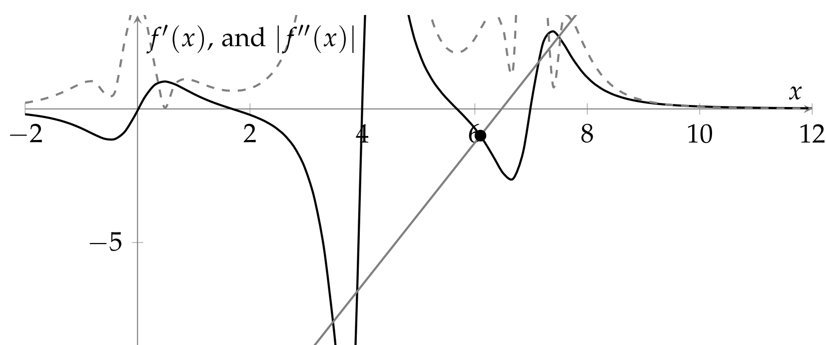

Figure 28: Solid black: the first derivative $f'(x)$; gray dashed: the absolute curvature $|f''(x)|$. Marker at $x = 6.1$ shows the same sample point as above; the solid gray line illustrates replacing $f''(x)$ by $|f''(x)|$ (slope $\approx +2.66$). Using $|f''(x)|$ in the Newton denominator removes the curvature sign, enforcing downhill steps and reducing the chance of stepping toward a local maximum.

Examples & Exercises

Code can be found at `https://github.com/Quillstacks/lecturecode_numericalmethods.git`.

Let's put the strategies to work on concrete numbers. Suppose you are optimizing a function with 5 local minima, and the basin of attraction of the global minimum covers 15% of the search domain ($p = 0.15$).

What is the probability of finding the global minimum with $K = 10$ random restarts?

Hint: Use $P = 1 - (1-p)^K$ and solve for K: $K = \frac{\ln(1-P)}{\ln(1-p)}$.

$$\begin{aligned} P &= 1 - (1 - 0.15)^{10} = 1 - 0.85^{10} \\ &= 1 - 0.1969 = 0.803 \end{aligned}$$

About 80% chance, not too bad, but far from certain. How many restarts do we need for 99%?

$$\begin{aligned} K &= \frac{\ln(1-0.99)}{\ln(1-0.15)} = \frac{\ln(0.01)}{\ln(0.85)} \\ &= \frac{-4.605}{-0.1625} \approx 28.3 \end{aligned}$$

So $K = 29$ restarts. Now compute K for $p = 0.05$ and $p = 0.01$ at the same 99% target.

ON YOUR MACHINE, you will first design your own custom 1D Shekel function. Keep it simple in the beginning, but come back later to make it more complex. Then first familiarize yourself with the local optimization methods you have learned so far, and apply them to your function. Show where Newton's method fails (again). Then explore different x_0 and see how the convergence behavior changes. Think again about the loss landscape and basin boundaries.

BASIN OF ATTRACTION. Revisit the plots of the Shekel function, mark the basins of attraction for each local minimum, and estimate their relative sizes. Which one is the global minimum? How does this relate to the probability of finding it with random restarts?

DIRECTING THE SEARCH TOWARDS MINIMA. Remember the $|f''|$ trick, flips the sign of the curvature in the denominator, forcing the step to always go downhill. Think about what that means geometrically for the tangent line construction. Then try it in code.

ESCAPING LOCAL MINIMA, by global optimization strategies. Deploy random restarts and basin hopping on your custom 1D Shekel function, and compare how many function iterations each method needs to find the global minimum. How prone are they to getting stuck in local minima? How does the choice of the step size distribution affect basin hopping's performance?

NON-CONVEX BASINS. Shekel in general is non-convex, however the basins are, note down a function, where the basins of attraction are not so easy to map out.

CAN YOU THINK OF a smart way to automatically adjust the step size distribution during optimization? What would be a good heuristic to follow? How could you prevent disastrous jumps? Implement it.

NOISE AND LANDSCAPE ENGINEERING. The loss landscape is not static, but is to be engineered and formed. By approximating the loss landscape on mini-batches (different selection of points),

CODE EXERCISE: implement random restarts and basin hopping on the 1D Shekel function. Compare how many function evaluations each method needs to find the global minimum at $x \approx 4$.

FINALLY, LET'S MAP OUT THE BASINS OF ATTRACTION. Consider the 1D Shekel function and assume a local optimizer always converges to the nearest foxhole.2

(a) Sketch (roughly) the basins of attraction for the three foxholes at $x = 0, 4, 7$. Where are the basin boundaries?

(b) If basin hopping uses a jump of $\delta \sim \text{Uniform}(-3, 3)$ starting from the local minimum $x^\circ = 7$, what is the probability that a single jump lands in the basin of the global minimum?

(c) Why might basin hopping find the global minimum faster than random restarts here?

Self-Reflection and Recap

SELF-REFLECTION Questions to guide your understanding:

- Why does local optimization fail on non-convex landscapes like Shekel's foxholes?
- How does the required effort scale withthe size of the global basin when doing random restarts?
- How do basin hopping differently from random restarts?
- What does $f''(x_n)$ tell us about the loss landscape, what does it characterize, and how can we put it to use?
- How does approximating the loss landscape with noisy estimates (mini-batches) help escape local optima?
- Choosing x_0 such that it lies in the basin of a local minimum, now find different ways such that your optimization method can escape it, how effective is what?

RECAP of Key Concepts:

- Local optimization methods (like Newton's method) can get stuck in local minima on non-convex functions.
- Global optimization strategies (random restarts, basin hopping, simulated annealing, stochasticity) are designed to explore the search space more broadly to find the global optimum.
- Modifying optimization methods to use curvature information $f''(x)$ allows to specify the direction of descent more accurately.
- The loss landscape is ours to engineer through the choice of loss function and noise, such as stochastic gradients.

GRADIENT DESCENT IS PRONE TO GETTING STUCK. And so far our answer was merely better than brute-forcing out way out, yet again. In the next chapter, we will learn how to smoothen loss landscape to increase the robustness of our gradient descent approach.

TEASER. What if we now face a high-frequency version of Shekels foxholes. What is the problem that arises?

Numerical Integration

2026-05-04 · sneaky olive Falke

Smooth Operator.
The Why

Melts all your memories and change into gold

In the previous chapter, we explored global optimization strategies to escape local minima. Imagine you are optimizing a function with thousands of tiny, sharp local minima like a high-frequency version of Shekel's Foxholes. A gradient descent algorithm will get stuck instantly in the first microscopic pothole it finds.

The solution for this is one that will feel familiar, but probably did go unnoticed until now: Numerical integration.

$$I = \int_a^b f(x)\,dx \tag{42}$$

This gives us good reasons to understand numerical integration,

- as it allows us to smoothen the loss landscape and make optimization methods more robust.
- as it tends to find more robust optima due to the averaging effect[10] of a region.

[10] N. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. Tang. On large-batch training for deep learning: Generalization gap and sharp minima. 09 2016. doi: 10.48550/arXiv.1609.04836

In deep learning stochastic gradient descent (SGD) is the workhorse optimization algorithm.

It is a Monte Carlo[11,12] method that estimates the gradient of the loss function by averaging over the gradient of a random mini-batch of data points. We are lucky to have numerical integration around, as it would be prohibitive expensive to train models on large-scale datasets.

Monte Carlo is a strategy which basically solves problems by random sampling. Embrace the beauty:

[11] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305, 2012

[12] Y. Gal and Z. Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning, 2016. URL https://arxiv.org/abs/1506.02142

Hands On Experience

We already know how to discretize a continuous function, and how to handle finite precision and systems with sparse information at discrete points.

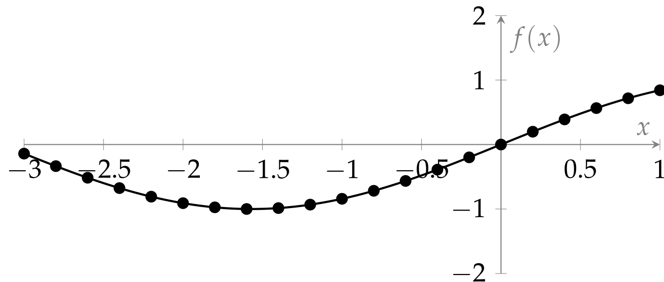

Figure 29: Continuous curve $f(x) = \sin(x)$ and its discretized version (black dots) on $[-3, 1]$ with step size $h = 0.2$.

THE SHEKEL FUNCTION, as a high-frequency function, did pose a challenge for global optimization, yet did not render our general approaches obsolete. We still discretize and treat these functions as we are used to.

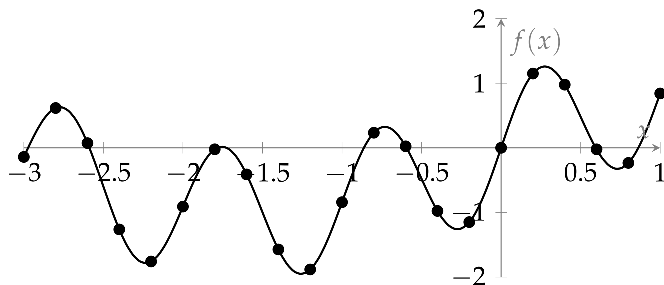

Figure 30: Continuous curve $f(x) = \sin(x) + \sin(2\pi x)$ and its discretized version (black dots) on $[-3, 1]$ with step size $h = 0.2$.

TAKING A CLOSER LOOK AT ANY local minima shows how the high-frequency loss landscape will nudge our optimization algorithm into the nearest local minimum. For example at $x = 0.6$ or $x = -0.4$.

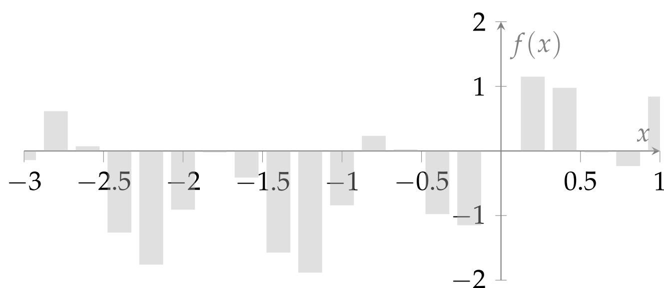

Figure 31: Continuous curve $f(x) = \sin(x) + \sin(2\pi x)$ and its discretized version (black dots) on $[-3, 1]$ with step size $h = 0.2$.

Consider approximation by numerical integration. Let's see what it does at $x = -0.4$, instead of solving directly for:

$$f(-0.4) = -0.97720, \tag{43}$$

let's approximate the value through integration with width $h = 0.2$:

$$\begin{aligned} h \cdot f(-0.4) &\approx \int_{-0.5}^{-0.3} f(x)\,dx \\ &\approx \frac{h}{2}[f(-0.5) + f(-0.3)] \\ &\approx 0.2 \cdot \frac{-1.14973 - 0.97720}{2} \\ &= -0.11285 \end{aligned}$$

$$\begin{aligned} f(-0.4) &\approx \frac{-1.14973 - 0.97720}{2} \\ &= -1.06347 \end{aligned}$$

This is a much better approximation having global optima in mind, than the direct evaluation at $x = -0.4$ which gave us -0.97720. The numerical integration gave us a correction of the local loss landscape towards higher losses at this local minima. The next figure shows the effect of this smoothing on the whole curve.

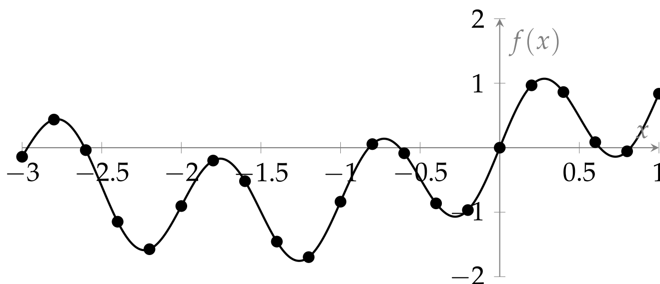

Figure 32: Continuous curve $f(x) = \sin(x) + \sin(2\pi x)$ and its discretized version (black dots) on $[-3, 1]$ with step size $h = 0.2$, and smoothed by numerical integration with interval length h.

The smoothing becomes even more apparent when integrating over larger intervals in general, or in our engineered example when choosing h such that the frequency of the noisy signal gets averaged out.

What if, we would be integrating over an interval which results in destructive inference of the underlying higher frequency sine curve? Think wavelengths.

While you have seen the smoothing effect driven by the interval size, the approximation of the interval so far is a pretty coarse approximation, relying on two points only.

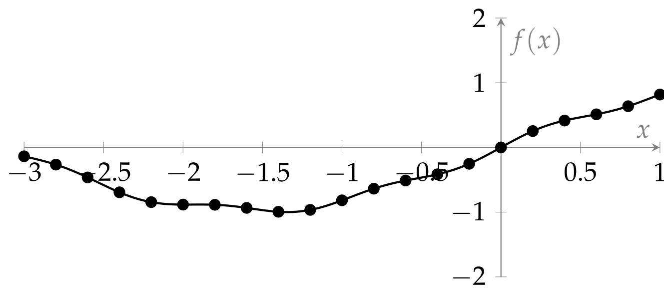

Figure 33: Continuous curve $f(x) = \sin(x) + \sin(2\pi x)$ and its discretized version (black dots) on $[-3, 1]$ with step size $h = 0.2$, smoothed by neighbour averaging with $h = 0.48$ (so $h/2 = 0.24$ gets close to cancel the $\sin(2\pi x)$ component).

THE LEARNING OBJECTIVES of this chapter will provide you with the abilities to:

- Derive and apply the midpoint method, trapezoid and Simpson's rules for numerical integration
- Understand composite rules for improved accuracy
- Know about Lagrange interpolation and how it relates to quadrature rules
- Understand how the curse of dimensionality motivates Monte Carlo
- Derive and apply Monte Carlo integration for high-dimensional problems
- Apply Monte Carlo integration and recognize that batch averaging in ML is numerical integration

Methods of Numerical Intgration

THE MOST TRIVIAL APPROXIMATION OF INTEGRALS is using a single discretized value.

$$I = \int_a^b f(x)\,dx.$$
$$I = \frac{b-a}{n} \cdot f\left(\frac{a+b}{2}\right)$$
$$I = h \cdot f(m)$$

It is easy to see that the height of the rectangle is the value at the midpoint, and the width is h.

We can further refine the approximation of the integral by dividing $[a, b]$ into n subintervals of equal width h, and summing the contributions from each interval. This is also known as Riemann sum approximation, and can be expressed as:

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

where $x_i = a + ih$ for $i = 0, 1, \ldots, n$.

THE MIDPOINT RULE of these composite intervals uses the midpoint, which is the average of the endpoints, of each interval, which often gives better results than left or right endpoint methods. Intuitively, this is because the midpoint approximates the curvature of the function better on the interval.

ON THE INTERVAL $[a, b]$, the midpoint value is $f(m)$. With left endpoint, the height is $f(a)$; with right endpoint, the height is $f(b)$.

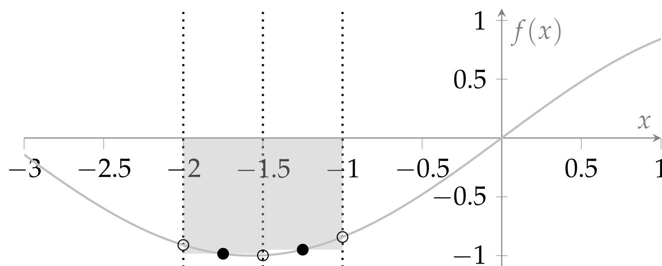

Figure 34: Continuous curve $f(x) = \sin(x)$ and midpoint rule integrals (gray bars) for $a = -2$, $b = -1$, $h = 0.5$. Black dots: midpoints. Circles: endpoints. The bars now touch, illustrating the midpoint rule without a gap.

INCREASING THE NUMBER OF SUBINTERVALS reduces the error.

DOUBLING THE SUB INTERVALS reduced error by $\sim 4\times$. This suggests $O(h^2)$ convergence.

This of course comes at the cost of more function evaluations, and thus more computational cost. So can we do better than approximating midpoints?

TRAPEZOIDS are more expressive geometrically than rectangles, and can capture linear changes in the function between the endpoints.

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left[f(a) + f(b)\right] \tag{44}$$

where $h = b - a$.

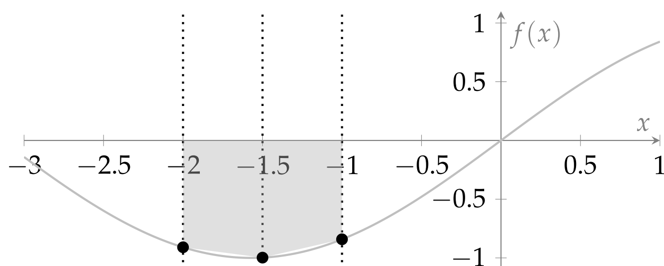

Figure 35: Continuous curve $f(x) = \sin(x)$ and trapezoid rule areas (gray trapezoids) for $a = -2, b = -1$, $h = 0.5$. Black dots: endpoints. The area under the straight lines connecting endpoints illustrates the trapezoid rule approximation.

COMPOSITE TRAPEZOID RULE again applies the trapezoid rule to each subinterval and sums the results.

$$\begin{aligned}
\int_a^b f(x)\,dx &\approx \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)\,dx \\
&\approx \sum_{i=0}^{n-1} \frac{h}{2}\left[f(x_i) + f(x_{i+1})\right] \\
&= \frac{h}{2}\left(f(x_0) + f(x_1)\right) + \frac{h}{2}\left(f(x_1) + f(x_2)\right) + \cdots + \frac{h}{2}\left(f(x_{n-1}) + f(x_n)\right) \\
&= \frac{h}{2}\left[f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(x_n)\right] \\
&= \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right]
\end{aligned}$$

where $x_i = a + ih$ for $i = 0, 1, \ldots, n$.

The function values at the endpoints a and b are each weighted by 1 in the formula, while all interior points are weighted by 2. Intuitively, this is because each interior point contributes to two adjacent trapezoids, while the endpoints only contribute to one trapezoid each. In your mind step through the circled enpoints in the figure above, and see how the interior points get counted twice, once as a right endpoint and once as a left endpoint, while the endpoints only get counted once.

DIVIDING THE INTEGRAL BY h gives us a weighted average of the function values, which is a better approximation to the integral than just using the midpoint or endpoints. Implicitly reflecting local curvature and information about the function's behavior across the interval.

SIMPSON'S RULE can capture curvature and thus often provides a much better approximation than the trapezoid rule. Following the idea of fitting a linear function, we fit a quadratic polynomial through 3 points instead of a line through 2. You will see Simpson's rule is exact for quadratic polynomials.

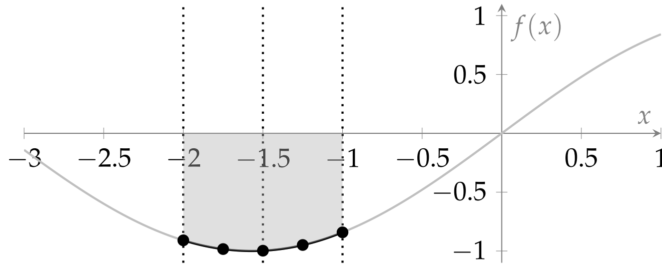

Figure 36: Continuous curve $f(x) = \sin(x)$ and Simpson's rule areas (gray, under the parabolas) for $a = -2$, $b = -1.5$, $h = 0.5$ and $a = -1.5$, $b = -1$, $h = 0.5$. Black dots: endpoints. Black dots: midpoints at $x = -1.75$ and $x = -1.25$. Solid line at $x = -1.5$ separates the two intervals. Each parabola illustrates the quadratic interpolant used by Simpson's rule on its subinterval.

LAGRANGE INTERPOLATION AND QUADRATURE. Lagrange interpolation constructs the unique polynomial of degree $\leq n$ that passes through given nodes $\{(x_j, y_j)\}_{j=0}^{n}$ using the basis

$$L_j(x) = \prod_{\substack{m=0 \\ m \neq j}}^{n} \frac{x - x_m}{x_j - x_m}. \tag{45}$$

The general form of a quadratic polynomial is:

$$f(x) = f(a) \cdot \frac{(x-m)(x-b)}{(a-m)(a-b)} + f(m) \cdot \frac{(x-a)(x-b)}{(m-a)(m-b)} + f(b) \cdot \frac{(x-a)(x-m)}{(b-a)(b-m)}$$

For a symmetric interval, the area under $f(x)$ is best approximated by giving the endpoints weight $h/6$ each and the midpoint weight $2h/3$: $A = C = h/6$, $B = 2h/3$ - we can verify this by integrating the Lagrange basis polynomials over $[a, b]$ and confirming that they yield these weights, but we omit the detailed calculation here. In a symmetric interval, the nodes are equally spaced: $x_0 = a$, $x_1 = m = \frac{a+b}{2}$, $x_2 = b$, so that the midpoint satisfies $m - a = b - m = h$.

TRY WITH SOME POINTS. Start with $x = 0$ and see how the linear factors play out.

Thus,

$$\begin{aligned}\int_a^b f(x)\,dx &\approx \frac{h}{6} f(a) + \frac{2h}{3} f(m) + \frac{h}{6} f(b) \\ &= \frac{h}{6}[f(a) + 4f(m) + f(b)]\end{aligned}$$

COMPOSITE SIMPSON'S RULE again applies Simpson's rule on each pair of subintervals and sums the results (assume n is even and $x_i = a + ih$). Starting from the integral:

$$
\begin{aligned}
\int_a^b f(x)\,dx &\approx \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)\,dx \\
&\approx \sum_{k=0}^{n/2-1} \frac{h}{6}\left[f(x_{2k}) + 4f(x_{2k+1}) + f(x_{2k+2})\right] \\
&= \frac{h}{6}\left[f(x_0) + 4\sum_{k=0}^{n/2-1} f(x_{2k+1}) + 2\sum_{k=1}^{n/2-1} f(x_{2k}) + f(x_n)\right] \\
&= \frac{h}{3}\left[f(x_0) + 4\sum_{\substack{i=1 \\ i \text{ odd}}}^{n-1} f(x_i) + 2\sum_{\substack{i=2 \\ i \text{ even}}}^{n-2} f(x_i) + f(x_n)\right].
\end{aligned}
$$

What happens, when you approximate a linear function with a quadratic?

PRACTICAL ADVICE: Stop at Simpson. For higher accuracy, use composite rules with more subintervals. High-degrees ($n \geq 8$) develop negative weights and become unstable.

RUNGE'S PHENOMENON: High-degree polynomial interpolation oscillates due to overfitting, leading to large errors at the edges of the interval.

Method	**Single Interval Error**	**Accumulated Error**
Left/Right	$O(h^2)$	$O(h)$
Midpoint	$O(h^3)$	$O(h^2)$
Trapezoid	$O(h^3)$	$O(h^2)$
Simpson's 1/3	$O(h^5)$	$O(h^4)$
In General (Gaussian Quadrature)	$O(h^{2n})$	$O(h^{2n-1})$

Table 3: Comparison of quadrature method accuracies. Higher-order methods achieve given accuracy with fewer function evaluations. h is the step size, and n is the number of evaluation points for Gaussian quadrature.

Monte Carlo & the Curse of Dimensionality

FOR d-DIMENSIONAL INTEGRALS, deterministic quadrature with N points per dimension needs N^d total points. There is no way to do this, let alone iterate on it during iterative optimization.

A NEURAL NETWORK WITH $d = 10^6$ or 1 million parameters would need $(N)^{10^6}$ grid points. Even $N = 2$ gives 2^{10^6}, a number with 300,000 digits.

RANDOM SAMPLING allows to estimate the integral without evaluating it on a full grid. The integral can be rewritten as an expectation:

$$
I = \int_\Omega f(\mathbf{x})\,d\mathbf{x} = |\Omega| \cdot \mathbb{E}_{\mathbf{X}\sim\text{Uniform}(\Omega)}[f(\mathbf{X})]. \tag{46}
$$

This identity means the definite integral equals the domain volume times the average value of f under a uniform draw from Ω. Practically this motivates a sampling estimator: draw points uniformly in Ω, compute f at those points, average the results, and multiply by $|\Omega|$ to estimate the integral.

Ω is the domain of integration, and $|\Omega|$ is its length. For a 1D integral over $[a,b]$, we have $|\Omega| = b - a$. For higher dimensions, it's the product of the lengths of each dimension.

MONTE CARLO UNIFORM SAMPLING on $[-2, -1]$.
Let $X \sim \text{Uniform}(-2, -1)$. Then

$$I = \frac{|\Omega|}{N} \sum_{i=1}^{N} f(\mathbf{X}_i), \qquad \mathbf{X}_i \overset{\text{i.i.d.}}{\sim} \text{Uniform}(\Omega). \tag{47}$$

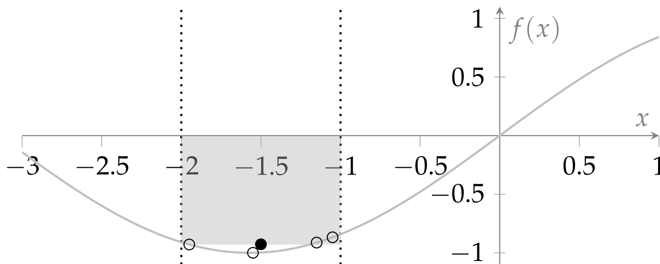

Figure 37: Continuous curve $f(x) = \sin(x)$ and midpoint rule integrals (gray bars) for $a = -2$, $b = -1$, $h = 0.5$. Circles: endpoints. The bars now touch, illustrating the midpoint rule without a gap.

BIAS. Because expectation is linear, this estimator is unbiased, meaning its expected value equals the true integral, for the number of $N \to \infty$:

$$\mathbb{E}[\hat{I}_N] = I. \tag{48}$$

MIDPOINT RULE is a special case of Monte Carlo with $N = 1$ sample at the midpoint.

VARIANCE. The variance, a measure of the spread of the estimator around its mean, or how how large the error of the estimator can be, follows from independence of the samples:

$$\text{Var}(\hat{I}_N) = \frac{|\Omega|^2}{N} \text{Var}\big(f(\mathbf{X})\big) = \frac{|\Omega|^2 \sigma_f^2}{N}, \tag{49}$$

where $\sigma_f^2 = \text{Var}(f(\mathbf{X}))$. Hence the standard error is

$$\text{SE}(\hat{I}_N) = \frac{|\Omega|\, \sigma_f}{\sqrt{N}}. \tag{50}$$

I.I.D, independent and identically distributed, means that all $\mathbf{X}_i$ are not correlated and that they all come from the same underlying distribution.

DIMENSION INDEPENDENT. It is trivial to see, that for large N the sampling error converges with, $O(1/\sqrt{N})$ for Monte Carlo, which makes it attractive in high dimensions. Quadrature rules outperform in terms of their convergence rates, but the convergence effort explodes with the dimension, as we need N^d points to maintain the same accuracy.

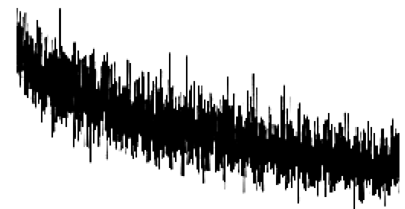

Figure 38: Effect of batch size on gradient noise: smaller batches produce larger variance (showing up in the width of the training loss band) in the gradient estimate (Source: Stanford CS231n Note).

CONVERGENCE IN SGD. Mini-batch gradients used in SGD are Monte Carlo estimates of the true gradient, based on a samples drawn into a batch. Increasing the batch size reduces variance roughly by $1/b$, directly analogous to the $1/N$ variance scaling above. This connection explains why batch size controls the noise–variance trade-off in training.

As a beneficial side effect, computational cost is n/b times cheaper per update step. For $n = 10^6$, and $b = 100$: SGD does 10,000 iterations with the compute GD needs for 1.

Examples & Exercises

LET'S STICK to our example from the methods sections, and calculate the integral of $\sin(x)$ from -2 to -1 using different methods. In order to be able to quantify the error of each method, we need the exact value. So let's compute the exact value of the integral:

$$\begin{aligned}\tilde{I} &= \int_{-2}^{-1} \sin(x)\,dx \\ &= [-\cos(x)]_{-2}^{-1} \\ &= -\cos(-1) + \cos(-2) \\ &\approx -0.9564491424.\end{aligned}$$

MIDPOINT RULE BY HAND. Compute $\int_{-2}^{-1} \sin(x)dx$ using the midpoint rule with $n = 1$ interval. The midpoint is $m = (-2 + (-1))/2 = -1.5$, so the midpoint rule gives us:

$$\begin{aligned}\hat{I} &\approx (b-a) \cdot f(m) \\ &= 1 \cdot \sin(-1.5) \\ &\approx -0.99749.\end{aligned}$$

The error is $|\tilde{I} - \hat{I}| \approx 0.04104$, which is the small difference between the midpoint estimate and the true value.

FURTHER THINGS WORTH TRYING are to calculate the approximation on the endpoints of the interval, comparing the error.

MIDPOINT RULE WITH MORE INTERVALS. Now compute the midpoint rule with $n = 2$ intervals, which means we will have two midpoints at -1.75 and -1.25. This gives us:

$$\begin{aligned}h &= 0.5 \\ \hat{I} &\approx h \cdot [f(-1.75) + f(-1.25)] \\ &= 0.5 \cdot [\sin(-1.75) + \sin(-1.25)] \\ &\approx -0.927228.\end{aligned}$$

The error is $|\tilde{I} - \hat{I}| \approx 0.02922$, which is smaller than the error with $n = 1$ interval, illustrating how composites, or increasing the number of intervals improves the approximation.

TRAPEZOID BY HAND. Use the trapezoid rule to compute $\int_{-2}^{-1} \sin(x)dx$ with $n = 2$ intervals. The endpoints are -2, -1.5, and -1. The trape-

zoid rule gives us:

$$\begin{aligned}
h &= 0.5 \\
T_2 &= \frac{h}{2}[f(-2) + 2f(-1.5) + f(-1)] \\
&= 0.25 \cdot [-0.90930 + 2(-0.99749) + (-0.84147)] \\
&\approx -0.927228.
\end{aligned}$$

The error is the same as the midpoint rule with $n = 2$ intervals, which is not necessarily a coincidence, as both methods are second-order accurate and can yield similar results for certain functions and interval choices.

SIMPSON'S RULE BY HAND. Use Simpson's rule to compute $\int_{-2}^{-1} \sin(x)dx$ with $n = 2$ intervals. The endpoints are -2, -1.5, and -1. Simpson's rule gives us:

$$\begin{aligned}
h &= 0.5 \\
S_2 &= \frac{h}{3}[f(-2) + 4f(-1.5) + f(-1)] \\
&= \frac{0.5}{3} \cdot [-0.90930 + 4(-0.99749) + (-0.84147)] \\
&\approx -0.9564491424.
\end{aligned}$$

The error is $|\tilde{I} - S_2| \approx 0.0000000000$, which is essentially zero, illustrating that Simpson's rule is exact for this particular integral, as $\sin(x)$ can be well approximated by a quadratic polynomial over the interval $[-2, -1]$.

TRY WITH MORE INTERVALS, see how the error decreases and check whether you know your way around composites in Simpson's and Trapezoid rules.

MONTE CARLO BY HAND. Use Monte Carlo estimation to compute $\int_{-2}^{-1} \sin(x)dx$ with $N = 4$ random samples, which is similar to the compute efforts of the trapezoid method. Let's assume the random samples are:

$$\begin{aligned}
X_1 &= -1.95, \quad f(X_1) = \sin(-1.95) \approx -0.92895, \\
X_2 &= -1.55, \quad f(X_2) = \sin(-1.55) \approx -0.99978, \\
X_3 &= -1.15, \quad f(X_3) = \sin(-1.15) \approx -0.91276, \\
X_4 &= -1.05, \quad f(X_4) = \sin(-1.05) \approx -0.86742.
\end{aligned}$$

The Monte Carlo estimate is:

$$\begin{aligned}
\hat{I} &= (b - a) \cdot \frac{1}{N} \sum_{i=1}^{N} f(X_i) \\
&= 1 \cdot \frac{1}{4}(-0.92895 - 0.99978 - 0.91276 - 0.86742) \\
&\approx -0.927228.
\end{aligned}$$

The error is $|\tilde{I} - \hat{I}| \approx 0.02922$, which is the same as the midpoint method.

ALTERNATIVE ERROR ESTIMATION FOR MONTE CARLO. For our concrete example with $N = 4$ and the sample mean of $\bar{f} \approx -0.927228$. Using the unbiased sample variance estimator:

$$Var(f(\mathbf{X})) = \frac{1}{N-1} \sum_{i=1}^{N} (f(X_i) - \bar{f})^2. \tag{51}$$

We obtain the sample variance and standard error of the Monte Carlo estimator as follows:

$$\begin{aligned} \sigma_f^2 &= Var(f(\mathbf{X})) \\ &\approx 0.003036, \\ \sigma_f &\approx 0.05512, \\ \text{SE}(\hat{I}_N) &\approx \frac{|\Omega|\, \sigma_f}{\sqrt{N}} \\ &= \frac{1 \cdot 0.05512}{2} \\ &\approx 0.02756. \end{aligned}$$

What happens if we increase N to 16? The error should decrease by a factor of 2, since the standard error scales as $1/\sqrt{N}$. Briefly elaborate on how the error of the Monte Carlo estimator is dimension independant.

CODE can be found at https://github.com/Quillstacks/lecturecode_numericalmethods.git.

ENTER HIGHER DIMENSIONS ON YOUR MACHINE. Implement Monte Carlo integration for a d-dimensional integral, such as integrating a multivariate Gaussian function over a hypercube. While slowly increasing d, observe how the error behaves for different methods. Even more, observe how grid-based quadrature methods become infeasible in terms of computational cost as d grows. Optionally, see how similar effects arise in optimization by implementing gradient descent and mini-batch stochastic gradient descent on a high-dimensional function. Only watch compute times.

Self-Reflection and Recap

SELF-REFLECTION Questions to guide your understanding:

- How does the error of the midpoint, trapezoid, Simpson's rule and Monte Carlo compare?
- Why has the midpoint rule better accuracy than the left or right endpoint rules?
- How does composite methods improve accuracy, and what is the trade-off?
- Why do deterministic quadrature methods fail in high dimensions?
- In which way, is the midpoint rule a special case of Monte Carlo estimation?
- What is the intuition behind Monte Carlo not exploding in high dimensions?
- When would you use Simpson's rule vs. Monte Carlo?
- How is the mini-batch mean in SGD related to Monte Carlo estimation?
- How does the gradient estimation in SGD help escape local minima?

RECAP of Key Concepts:

- Deterministic quadrature methods (midpoint, trapezoid, Simpson's) approximate integrals using weighted sums of function values at specific points. They are accurate and converge fast for smooth, low-dimensional problems.
- Monte Carlo integration estimates integrals by averaging function values at random samples. It is unbiased and has a convergence rate of $O(1/\sqrt{N})$, making it independent of the dimensionality of the problem.
- In SGD, mini-batch gradients are Monte Carlo estimates of the true gradient.

TEASER. Why do high-order quadrature methods become unstable when approximating low-degree polynomials?

SO FAR STABILITY referred to the sensitivity of the numerical solution to small changes in the input data. But we can also talk about stability in terms of the numerical method itself, and how it behaves

when approximating different types of functions. In the next chapter, we will see other types of stability issues that arise in numerical methods, and how to mitigate them by sophisticated method selection and configuration.

Index

absolute error, 21
analytical, 10
approximation, 12, 58

basin hopping, 51
brute-force method, 33

catastrophic cancellation, 22–24
complexity, 10
complexity analysis, 46
computation, 9
conditioning, 25
consistency, 25
convergence, 25
curse of dimensionality, 64

derivative, 35
discretization, 12, 58

error, 14, 15
error analysis, 24
examples, 14
exercises, 14, 54, 66

fixed-point representation, 18, 21–23
floating-point arithmetic, 17, 24
floating-point representation, 17–19, 23, 24, 26

global optimization, 46, 47, 51
gradient descent, 35
grid search, 14, 24

hands-on, 48

interval, 12, 58
introduction, 9

key concepts, 33

license, 2
linear approximation, 38
linear systems, 10
local information, 35
local minima, 47
local optimization, 51
loss of significance, 18, 22–24

machine epsilon, 18, 20, 23, 24
model error, 25
modeling error, 17, 18, 24
Monte Carlo, 64
Moore's Law, 9
motivation, 9
multimodal, 48

Newton
  optimization, 53
Newton failure modes, 42
Newton method, 34
Newton-Raphson, 38
numerical algorithms, 33
numerical integration, 56
  hands-on, 58
  motivation, 57
numerical method, 33
numerical solution, 33
numerical stability, 17, 18, 23, 24

pseudo-accuracy, 22, 24

quadratic convergence, 41

random restarts, 51
recap, 15, 56, 69
reflection, 15
rounding error, 17, 18, 22–24

Secant method, 42
self-reflection, 33
significant digits, 22, 24
Simpson's rule, 62
simulated annealing, 51
stability, 25
stepsize, 12, 58
stochastic gradient descent, 56

Taylor expansion, 39
truncation error, 17, 18