A Primer On Numerical Mathematics

Dr. Achim Kehrein
Dr. Thorsten Camps
William L. Cole

13.08.2019

Contents

Chapter 0

Numerical Mathematics: A Paradigm Shift

> "Because most things don't work the same way on a computer - to a scary extent."

0.1 A "New" Approach to Math.

So far:
Infinite processes (limits, derivatives, integrals, etc.)
$\longmapsto$ Must truncate these processes to make them finite. ("Truncation Error")

Infinitely many numbers, infinite precision:
$\longmapsto$ Computer has only finitely many machine numbers with finite precision ("round-off error")

Error propagation in mathematical recipes: Mathematically equivalent formulas may produce different results on a computer. Which result is more accurate?

Goal of Numerics Course:
Learn to treat each problem individually instead of applying "standard recipes".

Chapter 1

Number Representation in Computers and Calculators

Which numbers does a computer use?

Two types:

- INT - Integer
- FLOAT, DOUBLE - Floating point numbers

In the past, calculators only used FLOATs. Modern Calculators use both INTs and FLOATs.

1.1 Integers

...-1, 0, 1, 2, 3...

Infinitely many in math, but computers can only represent finitely many.

INT_{MIN}, INT_{MIN}+1, ..., -1, 0, 1, ..., INT_{MAX}

Computers use a binary system instead of a decimal system of representation.

Decimal:

$$123 = 1 \cdot 100 + 2 \cdot 10 + 3 \cdot 1$$
$$= 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$$

Binary:

$$_2 1101 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

$$= 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = {}_{10} 13$$

This scheme corresponds to:

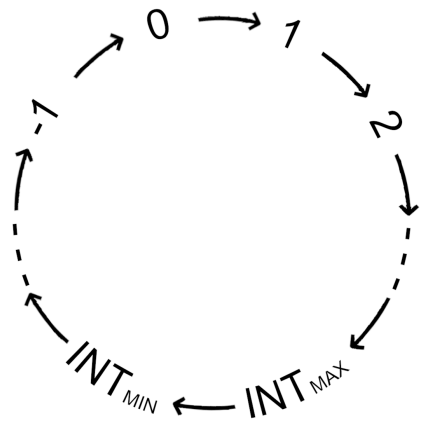


Figure 1.1: In a computer, integers are arranged like a clock.

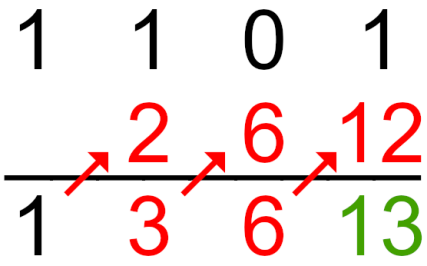
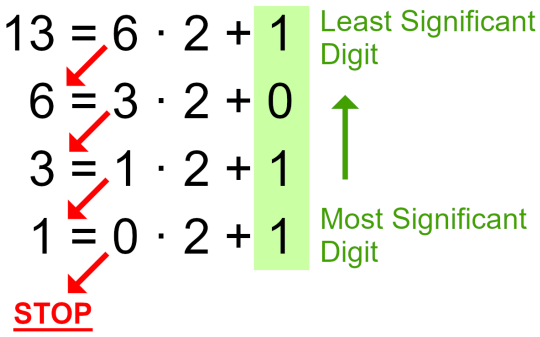


Figure 1.2: One method for binary to decimal conversion is to multiply by 2 and add the next digit.

$$\begin{aligned} & 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ & \quad = (2^2 + 1 \cdot 2^1 + 0) \cdot 2 + 1 \\ & \quad = [(1 \cdot 2 + 1) \cdot 2 + 0)] \cdot 2 + 1 \end{aligned}$$

How do we go in the other direction? How does one obtain the binary representation of a decimal integer? Use division with remainders.



For our calculators
$INT_{MAX} = 2147483647$

$INT_{MIN} = -2137483648$

$2147483647 = 1,073,741,824 \cdot 2 + 1$ ‡

$1073741823 = 536870911 \cdot 2 + 1$

$INT_{MAX} = \underbrace{{}_b111.....11}_{\text{31 digits. 1 bit for the sign}}$

If we leave the range INT_{MAX}, INT_{MIN}, this is called an **overflow error**. *The computer does not notify us about this!* We have to program all of the necessary checks.

As long as the computations do not produce an overflow error, arithmetic with the datatype INT is exact: Addition, subtraction, multiplication, and division with remainders.

We can extend this to rational numbers:

$$Rational\ Number = \frac{Integer}{Non{-}Zero\ Integer}$$

Just store numerator and denominator separately as integers (INTs).
Arithmetic for rational numbers:

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}$$

To avoid unnecessary overflow, cancel the fractions as much and as soon as possible when programming your own routine. For a better interpretation, we can convert $\frac{numerator}{denominator}$ into a decimal representation *at the end* of our computations.

Example: $\frac{13}{12}$

```
1 3          | 1 2
 1 0 0       | 1.0 8 3 3 3
   4 0       |
     4 0     |
       4 0   |
         4   |
```

The decimal representation will either terminate or will eventually become periodic, as in the preceding example.

‡The last digit of this number should be 3, but your calculator will round it.

1.2 Floating Point Numbers

Now let's look at FLOATs. In the sciences and in engineering, we also need very large numbers and very small numbers, i.e close to zero.

Avogadro's Number	6.022×10^{23}
Electrical Charge of an Elimentary Particle	$q_e = 1.602 \times 10^{-19} C$
Planck's Constant	$h = 6.626 \times 10^{-34} m^2 kg/s$

Another number representation is needed. Every real number possesses a decimal representation:

$$\pm d_k d_{k-1} ... d_0 \underbrace{.}_{\text{decimal}} d_{-1} d_{-2} .... \leftarrow \textit{May never stop or become periodic}$$

This is a convergent series:

$$= d_k \cdot 10^k + d_{k-1} \cdot 10^{k-1} + ... + d_0 \cdot 10^0 + d_{-1} \cdot 10^{-1} + d_{-2} \cdot 10^{-2} + ...$$

On a computer we can only use finitely many digits. The number of digits is often called the **precision**. By moving the decimal point, we standardize these numbers:

$$\pm \underbrace{d_b . d_{-1} d_{-2} ... d_{-p+1}}_{\text{significand}} \times 10^{\overset{\leftarrow Exponent}{\overset{\nwarrow}{k}}_{Base}}$$

This is a **floating point number**.

It is **normalized** when there is only one digit, d_0 before the decimal point. In other sources, "normalized" is often shown as

$$\mathbf{0}.d_k d_{k-1} ... d_{-3} * 10^{k+1} + \Delta x$$

For our purposes, a number is normalized when $d_0 \neq 0$, for two main reasons:

- This is how FLOATs are stored on computers and calculators.
- Scientific Notation (i.e. Avogadro's Constant $= 6.022 \times 10^{23}$)

Also, a computer doesn't use the decimal system. It uses **binary**. In binary

- base $= 2$
- digits: 0, 1

In base 2 a normalized floating-point number is

$$\mathbf{1}.b_{-1} b_{-2} ... b_{-p+1} * 2^E$$

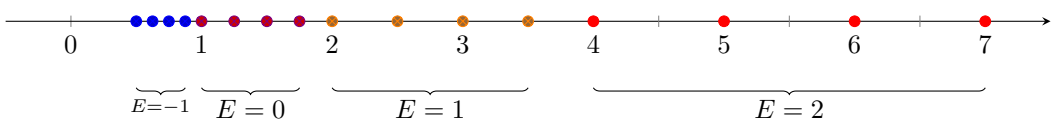
Because the first digit is automatically 1, it is usually not stored in the computer.

1.3 Relative Error

To get a feeling for floating point numbers, let's study a very limited "Toy" number system: $1.b_{-1}b_{-2} \cdot 2^E$

b_{-1}	b_{-2}	E	Decimal Value
0	0	0	$1+0\cdot\frac{1}{2}+0\cdot\frac{1}{4}=1$
0	1	0	$1+0\cdot\frac{1}{2}+1\cdot\frac{1}{4}=1.25$
1	0	0	$1+1\cdot\frac{1}{2}+0\cdot\frac{1}{4}=1.5$
1	1	0	$1+1\cdot\frac{1}{2}+1\cdot\frac{1}{4}=1.75$
0	0	1	$1\cdot 2^1=2$
0	1	1	$1.25\cdot 2^1=2.5$
1	0	1	$1.5\cdot 2^1=3$
1	1	1	$1.75\cdot 2^1=3.5$

1.	b_{-1}	b_{-2}	$E=-1$	$E=0$	$E=1$	$E=2$
	0	0	0.5	1	2	4
	0	1	0.625	1.25	2.5	5
	1	0	0.75	1.5	3	6
	1	1	0.875	1.75	3.5	7



Note that the distance between the numbers depends on their size. Consequentially, the round off error will also be affected by the size of the number. For this reason it's important to use *relative error* instead of absolute error.

Absolute Error	$x + \Delta x$
Relative Error	$x(1+\frac{\Delta x}{x}) = x(1+\delta x)$

Often the relative error is the better way to measure the round off error of floating point numbers.

$$fl(x) = x(1+\delta x) \Leftrightarrow \delta x = \frac{fl(x)-x}{x}$$

1.4 Relative Error Examples

Example 1: Real number, 4 digits, base 10:

$$x = \underbrace{1.23456}_{4digits}$$

$$fl(x) = 1.235$$

$$\delta x = \frac{1.235-1.23456}{1.23456}$$

$$= \frac{0.00044}{1.23456} = 0.000356$$

$$\simeq 0.0004 \leftarrow \text{Only first non-zero digit}$$

For errors, only record record the first non-zero (in most cases). When the first non-zero digit is 1, record the next digit (which can be zero).

Example 2: $y = 123.456 = 1.23456 \times 10^2$

$$fl(y) = 1.235 \times 10^2$$

$$\delta y = \frac{1.235\times10^2-1.23456\times10^2}{1.23456\times10^2}$$

$$= \frac{0.00044\times10^2}{1.23456\times10^2} \leftarrow \textit{Absolute Error}$$

$$\simeq 0.0004 \leftarrow \textit{Relative Error}$$

Notice how the relative error size reflects the number of digits used in the system. If a decimal system uses k digits, then the relative round-off error is bounded by 5×10^{-k}.

1.5 Subnormal Numbers and other special FLOAT values

Note in the number line from earlier that there is a gap at zero. What do we do about this gap? How do we express zero itself as a FLOAT? Not only is the number of digits in the mantissa finite, but the number of digits in the exponent is also finite. In C and other programming languages FLOATs are 32 Bits. 23 of these bits are used for the significand, 7 for the exponent, and 1 for the sign.

In our toy system from earlier, let's use $-2 \leq E \leq 4$

Use the maximal exponent for

INF	infinity (overflow error)
NaN	"not a number"

These two are distinguished by using different mantissae.

Use the minimal exponent for ***subnormal numbers:*** $\mathbf{0}.b_{-1}b_{-2} \times 2^{-2}$

0.00×2^{-2}	0
0.01×2^{-2}	$\frac{1}{8}$
0.10×2^{-2}	$\frac{1}{4}$
0.11×2^{-2}	$\frac{3}{8}$

What is the relative error in our toy system?

$$\tfrac{1}{8} = 2^{-3} = \tfrac{1}{2} \times 2^{-2}$$

Go to seven, which in binary is

$$1.11 \times 2^2$$
$$\tfrac{1}{8} \times 2^2$$

Some example of relative errors in our binary toy system:

$$\tfrac{7-7.4}{7.4} \simeq -0.05$$

$$\tfrac{7-6.6}{6.6} \simeq 0.06$$

$$\tfrac{8-8.9}{8.9} \simeq -0.10$$

Why are subnormal numbers treated differently?

$$\frac{fl(\frac{1}{16})-\frac{1}{16}}{\frac{1}{16}} = \frac{\frac{1}{8}-\frac{1}{16}}{\frac{1}{16}} = \frac{\frac{2}{16}-\frac{1}{16}}{\frac{1}{16}} = \frac{\frac{1}{16}}{\frac{1}{16}} = 1$$

Relative error 2^{-3} does not hold for sub-normal numbers, only normalized numbers. Because of this, it is of the utmost importance to ***avoid dividing by numbers close to zero whenever possible!***

1.6 Adding FLOATs

Now we know the general shape of the machine number system FLOAT, how does arithmetic work with it?

$$\text{Toy System: } \left(\left(\left(\left(4 \oplus \tfrac{1}{4}\right) \oplus \tfrac{1}{4}\right) \oplus \tfrac{1}{4}\right) \oplus \tfrac{1}{4}\right) = 4$$

The sum of two machine numbers may not be a machine number. Addition my create more round- off errors.

$$4 \oplus \left(\tfrac{1}{4} \oplus \left(\tfrac{1}{4} \oplus \left(\tfrac{1}{4} \oplus \tfrac{1}{4}\right)\right)\right)$$

$$= 4 \oplus \left(\tfrac{1}{4} \oplus \left(\tfrac{1}{4} \oplus \tfrac{1}{2}\right)\right)$$

$$= 4 \oplus \left(\tfrac{1}{4} \oplus \tfrac{3}{4}\right) = 4 + 1 = 5$$

If you must add lots of numbers on a computer, add them in order from small to large, to minimize round-off error.

Example: Harmonic Series

$$1 + \tfrac{1}{2} + \tfrac{1}{3} + \tfrac{1}{4} + \tfrac{1}{5} + \tfrac{1}{6} + ...$$

- Divergent Series

- Partial sums "approach" infinity.

We try to recreate the harmonic series in a C program:

```
float sum;
sum = 0.0;
for (i=1; i<= 1 000 000; i++){
    sum += 1/(float) i;
}
```

We repeat this same for-loop, increasing the size of the loop from `i<=1 000 000` to 10,000,000 and 100,000,000. The program returns the following results:

```
1 000 000 summands give sum 14.3573579788
10 000 000 summands give sum 15.4036827087
100 000 000 summands give sum 15.4036827087
```

We see that by adding increasingly smaller summands the computer has already run out of precision before 10,000,000 cycles. If we reverse the order of the for-loop so that the script adds summands in ascending order from smallest to largest we obtain the following results:

```
1 000 000 summands give sum 14.3926515579
10 000 000 summands give sum 16.6860313416
100 000 000 summands give sum 18.8079185486
1 000 000 000 summands give sum 18.8079185486
```

By adding the summands from smallest to largest, it takes the script is able to add smaller numbers, and so obtain a larger partial sum. *When programming, if possible try to add smaller numbers first, for more accurate results.*

$$0.1 = \underbrace{b_{-1}}_{=0}\tfrac{1}{2} + b_{-2}\tfrac{1}{2^2} + b_{-3}\tfrac{1}{2^3} + ...$$

Multiply by 2:

$$0.2 = \underbrace{b_{-1}}_{=0} + b_{-2}\tfrac{1}{2} + b_{-3}\tfrac{1}{2^2} + ...$$

$\times 2$:

$$0.4 = b_{-2} + b_{-3}\tfrac{1}{2} + b_{-4}\tfrac{1}{2^2} + ...$$

$\times 2$:

$$0.8 = b_{-3} + b_{-4}\tfrac{1}{2}$$

A particular C program adds `float 0.0 += 0.1`, with the following results:

$$\begin{aligned}
fl(0.1) &= 0\,.\,100\,000\,001\,5\\
fl(0.2) &= 0\,.\,200\,000\,003\,0\\
fl(0.3) &= 0\,.\,300\,000\,011\,9\\
fl(0.4) &= 0\,.\,400\,000\,006\,0\\
fl(0.5) &= 0\,.\,500\,000\,000\,0\\
fl(0.6) &= 0\,.\,600\,000\,023\,8\\
fl(0.7) &= 0\,.\,700\,000\,047\,7\\
fl(0.8) &= 0\,.\,800\,000\,071\,5\\
fl(0.9) &= 0\,.\,900\,000\,095\,4\\
fl(1.0) &= 0\,.\,200\,000\,119\,2
\end{aligned}$$

b_0	**0**	**.1**← *Double the fractional part*
b_{-1}	0	.2
b_{-2}	0	.4
b_{-3}	0	.8
Discard whole part →	1	.6
b_{-5}	1	.2 ←*Numbers repeat*
b_{-6}	0	.4

The whole part becomes the binary representation, hence:

$$\begin{aligned}
fl({}_{10}0.1) &= \underbrace{{}_{2}0}_{b_0}.000\,110\,011\,001\,1... =_2 0.000\,\overline{1100}\\
&= \underbrace{{}_{2}1.\,1}_{\text{Normalized}}001\ 1001\ 1001... \times 2^{-4}
\end{aligned}$$

FLOATs use 23 fractional bits, so this non-terminating number will be rounded:

$$fl({}_{10}0.1) = 1.1001\ 1001\ 1001\ 1001\ 1001\ 10\cancel{01}^{1} \times 2^{-4}$$

Let's compute the round-off error:

$$\begin{aligned}
\text{absolute error}\rightarrow\quad \Delta x &= 1 \times 2^{-23} \times 2^{-4} - 1.1001\ 1001\ ... \times 2^{-24} \times 2^{-24}\\
&= 2^{-3} \times 2^{-24} -_{10} 0.1 \times 2^{-24}\\
&= ({}_{10}0.125 -_{10} 0.1) \times 2^{-24}\\
&=_{10} 1.49 \times 10^{-9} \simeq 1.5 \times 10^{-9}
\end{aligned}$$

Now let's add two FLOAT values:

$$\begin{array}{rcccccccc}
 & {\scriptstyle 1\,1} & {\scriptstyle 1\,1} & {\scriptstyle 1\,1} & {\scriptstyle 1\,1} & {\scriptstyle 1\,1} & {\scriptstyle 1\,1} & {\scriptstyle 1} & \\
fl(0.1) = & 1. & 1001 & 1001 & 1001 & 1001 & 1001 & 101 & \times 2^{-4} \\
+\, fl(0.1) = & +1. & 1001 & 1001 & 1001 & 1001 & 1001 & 101 & \times 2^{-4} \\
\hline
\text{Not normalized} \rightarrow & 11. & 0011 & 0011 & 0011 & 0011 & 0011 & 010 & \times 2^{-4}
\end{array}$$

$$fl(0.2) = 1.\ 1001\ 1001\ 1001\ 1001\ 1001\ 1010\overset{round-off}{\nearrow} \times 2^{-3}$$

To add an additional $fl(0.1) = 1.100... \times 2^{-4}$ we have to shift the mantissa to adjust the exponent.

$$\begin{array}{rcccccccc}
 & {\scriptstyle 1\,1} & {\scriptstyle 1\,1} & {\scriptstyle 1\,1} & {\scriptstyle 1\,1} & {\scriptstyle 1\,1} & {\scriptstyle 1\,1} & {\scriptstyle 1\,1} & \\
fl(0.2) = & 1. & 1001 & 1001 & 1001 & 1001 & 1001 & 101 & \times 2^{-3} \\
+\, fl(0.1) = & +\,0. & 1100 & 1100 & 1100 & 1100 & 1100 & 1101\overset{1}{\nearrow} & \times 2^{-3} \\
\hline
\text{Normalize} \rightarrow & 10. & 0110 & 0110 & 0110 & 0110 & 0110 & 100 & \times 2^{-3} \\
 & & & & & & & & \\
\Rightarrow & 1. & 0011 & 0011 & 0011 & 0011 & 0011 & 010 & \times 2^{-2}
\end{array}$$

Is this floating point number representative of ${}_{10}0.3$? Let's check!

0	**.3**
0	.6
1	.2
0	.4
0	.8
1	.6 ← *Numbers repeat*
Repeating pattern : 0011 → 1	.2

To add $fl(0.1)$ at this point, we must adjust the exponent by two. There are two options: First, we can shift by one digit twice, which would lead to two cases of rounding up. The second option is to shift by two digits at once, which leads to rounding down. Looking at the results from the computer, we see that the second method is chosen.

$$\begin{array}{rccccccc}
 & {\scriptstyle 1\,1} & {\scriptstyle 1\,1} & {\scriptstyle 1\,1} & {\scriptstyle 1\,1} & {\scriptstyle 1\,1} & {\scriptstyle 1} & \\
1. & 0011 & 0011 & 0011 & 0011 & 0011 & 010 & \times 2^{-2} \\
+\,0. & 0110 & 0110 & 0110 & 0110 & 0110 & 011 & \times 2^{-2} \\
\hline
1. & 1001 & 1001 & 1001 & 1001 & 1001 & 101 & \times 2^{-2} \\
 & & & & & & & = fl(0.4)
\end{array}$$

$$\begin{array}{rcccccccc}
 & {\scriptstyle 1\,1} & {\scriptstyle 1\,1\,1\,1} & {\scriptstyle 1\,1\,1\,1} & {\scriptstyle 1\,1\,1\,1} & {\scriptstyle 1\,1\,1\,1} & {\scriptstyle 1\,1\,1\,1} & {\scriptstyle 1\,1} & \\
fl(0.4) = & 1. & 1001 & 1001 & 1001 & 1001 & 1001 & 101 & \times 2^{-2} \\
+\, fl(0.1) = & +\,0. & 0110 & 0110 & 0110 & 0110 & 0110 & 011 & \times 2^{-2} \\
\hline
\text{Normalize} \rightarrow & 10. & 0000 & 0000 & 0000 & 0000 & 0000 & 000 & \times 2^{-2} \\
 & & & & & & & & \\
fl(0.5) = & 1. & 0000 & 0000 & 0000 & 0000 & 0000 & 000 & \times 2^{-1}
\end{array}$$

Note: In this program $fl(0.5) = 0.5$

Side Remark: When programming FLOAT results, allow for round-off errors. Don't use conditions like

```
result == 0.0
```

Instead use

```
result <= 0.00....1
```

Don't use values that are too small, though.

One more step:

$$\begin{array}{rrllllllll} fl(0.5) = & 1. & 0000 & 0000 & 0000 & 0000 & 0000 & 000 & \times 2^{-1} \\ +\, fl(0.1) = & +\,0. & 0011 & 0011 & 0011 & 0011 & 0011 & \underbrace{010}_{\text{two digits affected}} & \times 2^{-1} \\ \hline & 1. & 0011 & 0011 & 0011 & 0011 & 0011 & 010 & \times 2^{-1} \end{array}$$

Because $fl(0.1)$ needs to be rounded by two digits, there is a larger error in this operation. **Shifting loses significant digits!** Many other math texts ignore this effect.

We may think that the round-off error is gone at 0.5 because it can be represented *exactly* by a machine number, but notice that 1 is also a machine number, and we have a huge round-off error there.

So far we've had

$$fl(0.x) + fl(0.1) = fl(0.x + 1)$$

$$\textit{i.e.} \quad fl(0.4) + fl(0.1) = fl(0.5)$$

Note that:

$$\underbrace{fl(0.9) + fl(0.1)}_{\text{severe round-off error}} \neq fl(1.0)$$

$$\textit{Since} \quad fl(1.0) = 1.0.$$

Where does this happen the first time and why? Notice in the preceding example we do not have a simple case of error propagation. The roundoff error of $fl(0.1)$ does not simply accumulate in the sum.

What errors are produced by arithmetic operations? Sometimes a sum, difference, product, or quotient of machine numbers is not a machine number. So there are round-off errors produced by arithmetic, not just number representation. (Other texts provide rather limited information about this.) Additionally, how do errors propagate if we begin with uncertain numbers (i.e. measurement data).

1.7 Finding Zeros of Functions

How do we compute function values? Due to round-off errors, can we really detect zeros? If a function gets very close to zero, how can we determine whether its value is really different from zero, or just a zero with a round-off error?

Example:

In our toy system with normalized floating-point numbers, of the form $b_0.b_{-1}b_{-2} \times 2^E$ we found:

$$4 \oplus \tfrac{1}{4} = 4$$

Why?

$$fl(4) =_2 1.00 \times 2^2$$
$$fl(\tfrac{1}{4}) =_2 1.00 \times 2^{-2}$$

Add them:

Shift the $fl(\frac{1}{4})$ to adjust the exponent:

$$\begin{array}{rcll} fl_{shifted}(\frac{1}{4}) & = & 0.00\,\cancel{01}^{\text{round down}} & \times 2^2 \\ + fl(4) & = & 1.00 & \times 2^2 \\ \hline & & 1.00 & \times 2^2 \;= fl(4) \end{array}$$

What size of error do we get here?

$$\delta_{sum} = \tfrac{4-4.25}{4.25} = \tfrac{-0.25}{4.25} \simeq -0.06$$

Upper bound for relative error: ${}_20.001 =_{10} 0.125$

Notice, however, that we can make the error arbitrarily large by repeated addition:

$$\Big(\Big(\big(4 \oplus \tfrac{1}{4}\big) \oplus \tfrac{1}{4}\Big) \oplus \tfrac{1}{4}\Big) \oplus \tfrac{1}{4}$$
$$\delta = \frac{4-5}{5} = \frac{-1}{5} = -0.2$$

We have already seen that adding smaller summands first improves the accuracy.

1.8 FLOAT multiplication

Example: Use decimal system with three digits:

$$\begin{aligned}&(1.23 \times 10^4) \cdot (4.56 \times 10^3)\\ &= 5.6088 \times 10^7\\ &\Rightarrow 5.61 \times 10^7\end{aligned}$$

Rounding is necessary, but within the rounding bounds of the number system. For multiplication, the error propagation can be described rather easily:

$$\begin{aligned}&[x(1+\delta x)] \cdot [y(1+\delta y)]\\ &= (x + x \cdot \delta x)(y + y \cdot \delta y)\\ &= xy + xy\delta x + xy\delta y + xy\delta x\delta y\\ &= xy(1 + \delta x + \delta y + \underbrace{\delta x\delta y}_{negligible})\\ &\simeq xy(1 + \delta x + \delta y)\end{aligned}$$

The relative error of the product is (to a very good approximation) the sum of the relative errors of the factors. The relative errors don't explode. They grow slowly.

1.9 FLOAT Division

Example: Use decimal system with three digits:

$$\begin{aligned}\frac{1.23\times10^4}{4.56\times10^3} &= 0.2697 \times 10^1\\ &= 2.697 \times 10^0 \leftarrow \textit{Write exponent so it isn't forgotten!}\\ &\Rightarrow 2.70 \times 10^0\\ \frac{x(1+\delta x)}{y(1+\delta y)} &= \frac{x}{y}\ (1+\delta x)(\underbrace{1 - \delta y + (\delta y)^2 - (\delta y)^3 + -...}_{\text{Geometric series. } \delta y \text{ can be assumed } <1})\end{aligned}$$

Geometric Series:

$$1 + (-\delta y) + (-\delta y)^2 + (-\delta y)^3 + ... = \frac{1}{1-(-\delta y)}$$

$$\text{for } |\delta y| < 1$$

$$\begin{aligned}\frac{x(1+\delta x)}{y(1+\delta y)} &= \frac{x}{y}\ (1 + \delta x - \delta y + \textit{[smaller terms]})\\ &\simeq \frac{x}{y}\ 1 + \delta x - \delta y\end{aligned}$$

As with multiplication, the relative errors in division can add up but can also partially cancel each other out. *Multiplication and Division behave nicely* for FLOATS.

1.10 FLOAT Subtraction

Subtraction is very problematic when values are *almost* equal.

$$4.56 \times 10^3 - 4.55 \times 10^3 = \underbrace{0.01}_{\text{Two significant digits lost!}} \times 10^3$$

Normalize:

$$= \underbrace{1.00}_{\text{These could be 9. We don't know!}} \times 10^1$$

Such results are usually *very* uncertain. *Avoid formulas that lead to differences of almost equal numbers!*

Example 1:

$$\begin{aligned} \ln 5000 - \ln 4999 &\simeq 8.51719391 - 8.516993171 \\ &\simeq 2.0002 \times 10^{-4} \end{aligned}$$

Alternatively:

$$\ln \frac{5000}{4999} \simeq 2.000199947 \times 10^{-4}$$

Example 2:

$$f(t) = \frac{\sqrt{t^2+9}-3}{t^2}$$

This equation can be rewritten:

$$= \frac{\sqrt{t^2+9}-3}{t^2} \cdot \frac{\sqrt{t^2+9}+3}{\sqrt{t^2+9}+3}$$

$$= \frac{t^2+9-9}{t^2(\sqrt{t^2+9}+3)}$$

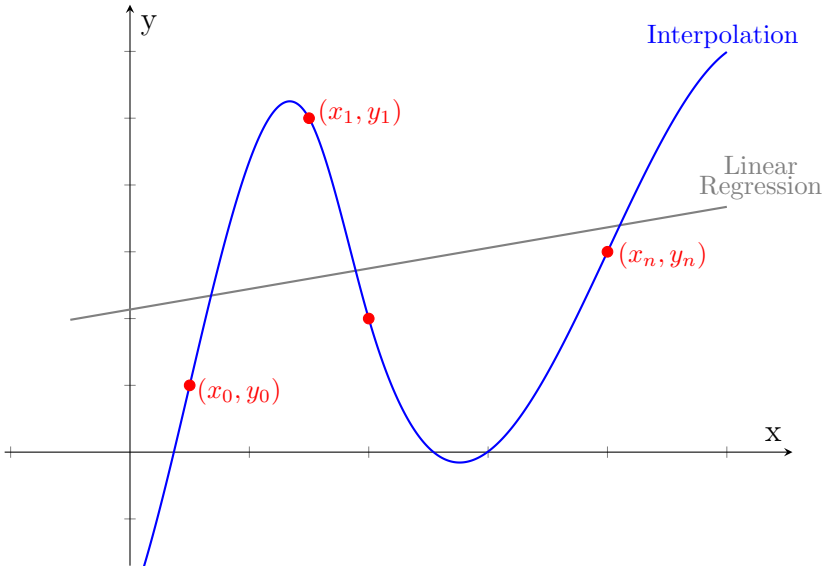
$$= \frac{1}{\sqrt{t^2+9}+3}$$

Computer results:

t	$\frac{\sqrt{t^2+9}-3}{t^2}$	$\frac{1}{\sqrt{t^2+9}+3}$
0.001	1.6667	1.666 6666 2
0.00001	1.6666	

Chapter 2

Interpolation



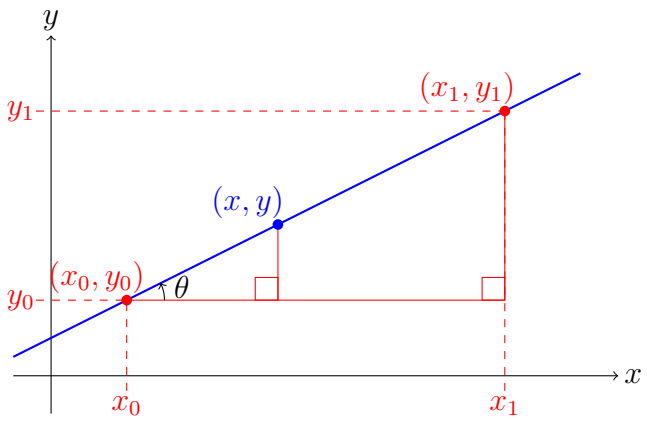
n+1 data points: $(x_0, y_0), (x_1, y_1), ...(x_n, y_n)$. We want an interpolating function, $y = f(x)$ such that $f(x_i) = y_i$ for all $i = 0, 1, ..., n$ Interpolation Conditions.

A totally different approach is the one using the "least squares" method - fit the function as best as possible to the data.

We start with polynomial interpolation, because polynomials are very simple functions with lots of nice properties:

- Can be easily calculated
- Continuous and Differentiable everywhere
- Versatile

Example: Linear interpolation (through two points)



Similar Triangles

$$\frac{y-y_0}{x-x_0} = \underbrace{\frac{y_1-y_0}{x_1-x_0}}_{\text{slope}}$$

$\Leftrightarrow$

Two Point Formula for a line

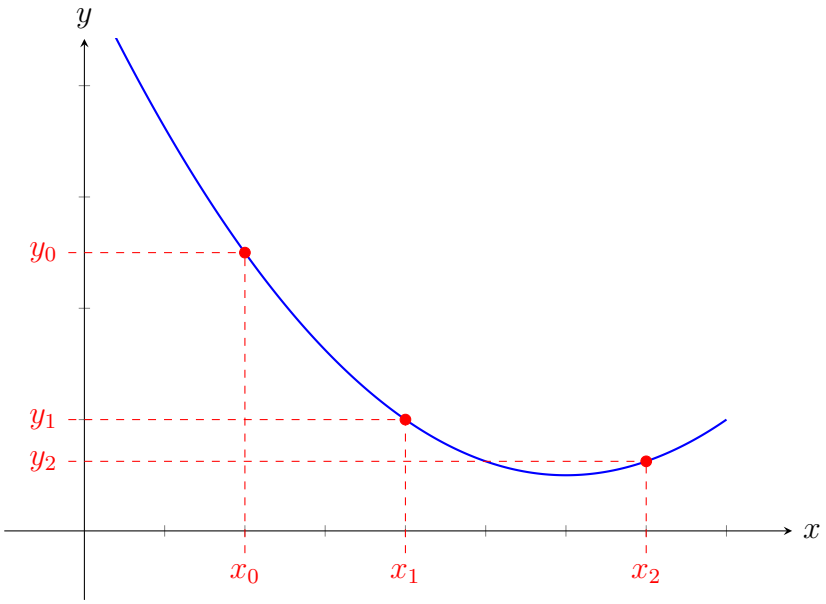
$$y = \frac{y_1-y_0}{x_1-x_0}(x-x_0)+y_0$$

We distinguish between *Interpolation* and*Extrapolation.*

Interpolation: Calculating points *between* smallest and largest x_i values.
Extrapolation: Calculating point *outside* range of x_i values.

2.1 Lagrangian Interpolation

Example: Three Data Points $(x_0, y_0)(x_1, y_1)(x_2, y_2)$



Model: $p(x) = A + Bx + Cx^2$

parameters

How do we compute these parameters? Interpolating conditions lead to:

$$y_0 = p(x_0) = A + Bx_0 + Cx_0^2$$
$$y_1 = p(x_1) = A + Bx_1 + Cx_1^2$$
$$y_2 = p(x_2) = A + Bx_2 + Cx_2^2$$

This is a linear system:

$$\begin{pmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \end{pmatrix} \begin{pmatrix} A \\ B \\ C \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix}$$

A Matrix of the form

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix}$$

is called a **Vandermonde Matrix**.

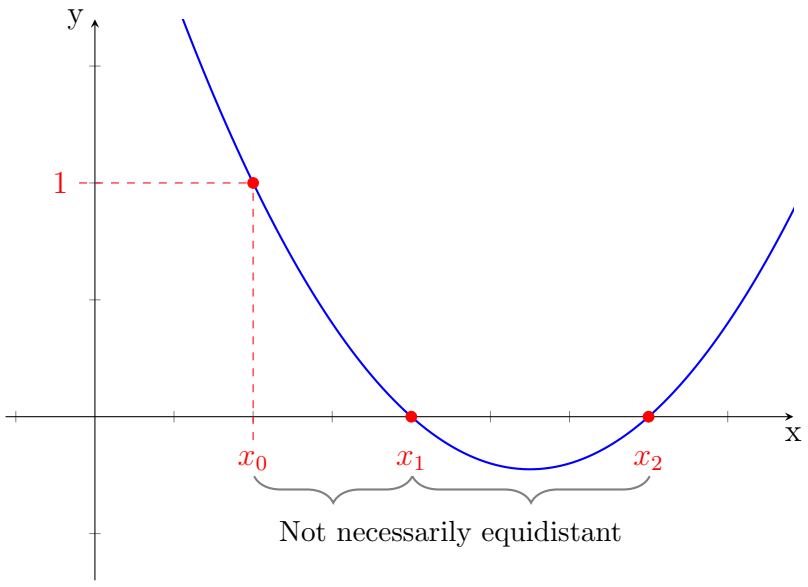
It's determinant is

$$\prod_{i<j}(x_i - x_j)$$

so it will be non-zero when all the x_i's are different. This restriction does not interfere with an interpolation problem, since it also requires different x_i's.

Theorem: For $n+1$ data points $(x_0, y_0), (x_1, y_1), ...(x_n, y_n)$ there is a unique polynomial of degree n interpolating these points.

This polynomial could be computed by Gaussian Elimination, but usually that leads to numerical problems (round-off errors will amplify). *There is a smarter way!*



$$L_0(x) = a(x - x_1)(x - x_2)$$

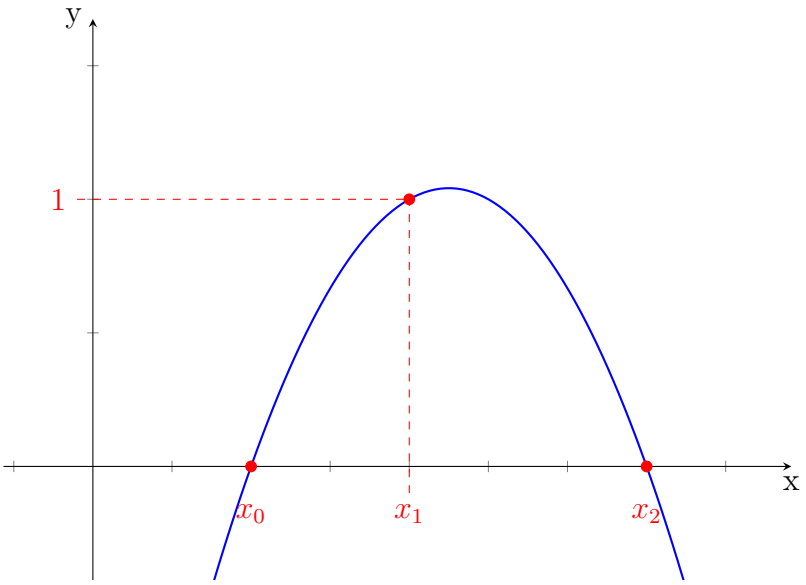
Choose:

$$a = \frac{1}{(x_0 - x_1)(x_0 - x_2)}$$

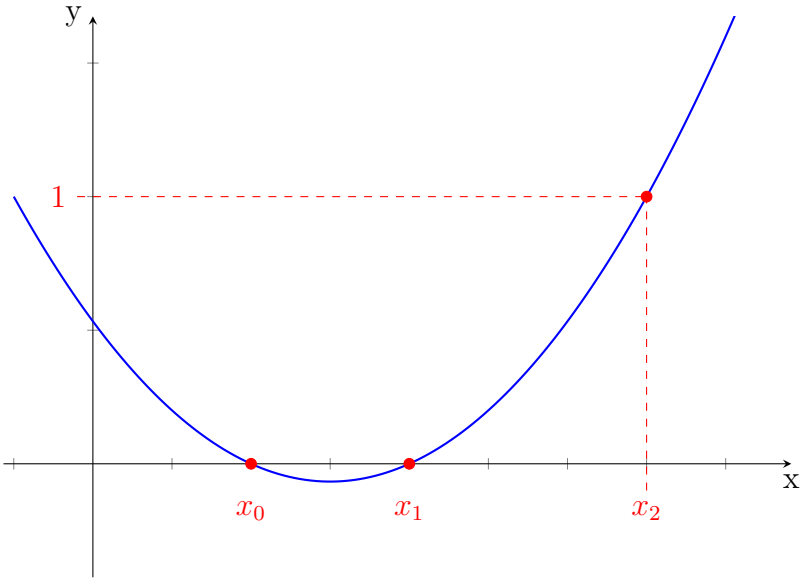
$$\Rightarrow \; L_0(x) = \frac{(x - x_1)(x - x_2)}{x_0 - x_1)(x_0 - x_2)}$$

So:

$$\left.\begin{array}{l} L_0(x_0) = 1 \\ L_0(x_1) = 0 \\ L_0(x_2) = 0 \end{array}\right\} \text{This is the } \textbf{interpolating parablola}$$



$$L_1(x) = \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)}$$



$$L_2(x) = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}$$

Now with (x_0, y_0), (x_1, y_1), and (x_2, y_2), the interpolating polynomial is:

$$p(x) = y_0 \cdot L_0(x) + y_1 \cdot L_1(x) + y_2 \cdot L_2(x)$$

2.2 Newtonian Interpolation

When data points are "added" the Lagrange polynomials have to be computed all over again. The Newton approach is designed in a way that we can use previous calculations when interpolating with additional data points.

Example: Three Coordinates (1,-2), (2,5), and (-1,-4)

Align the coordinates in two separate columns:

x	y
1	-2
2	5
-1	-4

Subtract the first x-coordinate from each subsequent coordinate, and write the difference in the next column to the left.

	x	y
	1	-2
2 - 1 = **1** ←	2	5
-1 - 1 = **-2** ←	-1	-4

In the next column subtract the first value of that column from each subsequent value, just as you did in the first column:

	x	y
	1	-2
1	2	5
-2 - 1 = **-3** ← -2	-1	-4

Repeat this process until there is a left-most column with only one value.

For the y-coordinates, the process is a bit different. Just like with the x-coordinates, subtract the first value in each column from all subsequent values in that column. Then, divide the resulting difference by the difference of the corresponding x-values (i.e. the answers on the left side).

		x	y		
		1	-2		
	1	2	5 → $\frac{5-(-2)}{1}=$ **7**		
-3	-2	-1	-4	$\frac{-4-(-2)}{-2}=$**1**	

Repeat this process until there is a right-most column with only one value:

		x	y		
		1	-2		
	1	2	5	7	
-3	-2	-1	-4	1 → $\frac{1-7}{-3}=$**2**	

How do we turn this into a polynomial?

		x	y		
		1	-2		
	1	2	5	7	
-3	-2	-1	-4	1	2

Beginning with your first y-coordinate, multiply the first value in each y-column as a coefficient in a polynomial of increasing degree, starting with degree 0. These polynomials will be composed of expressions of $(x - x_{n-1})$ Where x_{n-1} are your original x-coordinates. Add these polynomials together for the desired formula:

$$p(x) = -2 \underbrace{\cdot 1}_{\text{degree 0}} + 7\underbrace{(x-1)}_{\text{degree 1}} + 2\underbrace{(x-1)(x-2)}_{\text{degree 2}}$$

$$= -2 + 7x - 7 + 2x^2 - 6x + 4$$

$$= 2x^2 + x - 5$$

What happens if we want to add an additional data-point?

Example: Three Coordinates (1,-2), (2,5), (-1,-4), and (-2, -11):

		x	y		
		1	**-2**		
	1	**2**	**5**	7	
-3	-2	**-1**	**-4**	1	2
		-2	-11		

If we begin building the triangle as we did before, we get the same values we did previously for the first three three coordinates. *The same points in the same order will always result in the same triangle.* Because of this, adding a new coordinate just means adding a new row.

x	y	*Original values*
	1	-2
1	2	5	7
-3	-2	-1	-4	1	2
-1	-4	-3	**-2**	**-11**	3	1	1

Newly calculated values

Repeat the process from the first three coordinates for the new coordinate to obtain fourth row to the triangle. Just as we were able to add the new coordinates to the last row of the triangle, we can append the new calculated values to the end of our previously obtained polynomial:

x	y						
		1	-2				
	1	2	5	7			
	-3	-2	-1	-4	1	2	
-1	-4	-3	-2	-11	3	1	1

$$p(x) = \underbrace{-2 \cdot 1 + 7(x-1) + 2(x-1)(x-2)}_{\text{previously obtained}} + 1 \cdot (x-1)(x-2)(x-(-1))$$

So we see that while the Newton method of interpolation is a bit complicated, it has the nice property of allowing us to add more points of data without having to recalculate our previous values.

2.3 Interpolating Non-Polynomial Functions

Example: Use data points from the function $f(x) = \frac{1}{1+25x^2}$:

x	± 1	± 0.8	± 0.6	± 0.4	± 0.2	0
$f(x)$	0.038	0.058	$\frac{1}{10}$	$\frac{1}{5}$	$\frac{1}{2}$	1

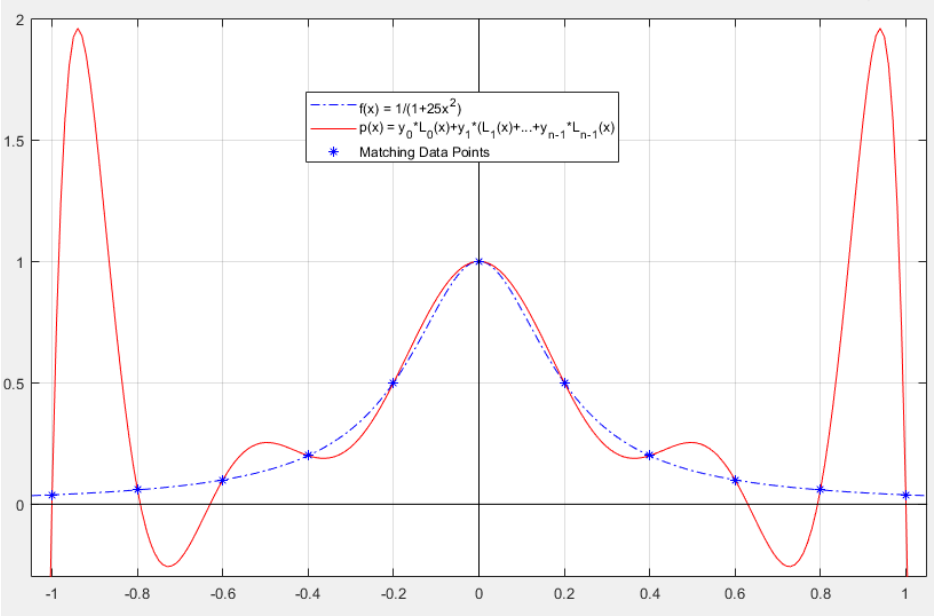


Figure 2.1: We observe high oscillations between the interpolated data points.

What is going on here?

- We try to reconstruct a function that has non-polynomial behavior (horizontal asymptote).
- Equidistant points create extra trouble. *Cluster points at the end points to obtain better results.*

Let's try more midpoints: 0, ± 0.1, ± 0.2... ± 1. Now we have 21 points instead of 11:

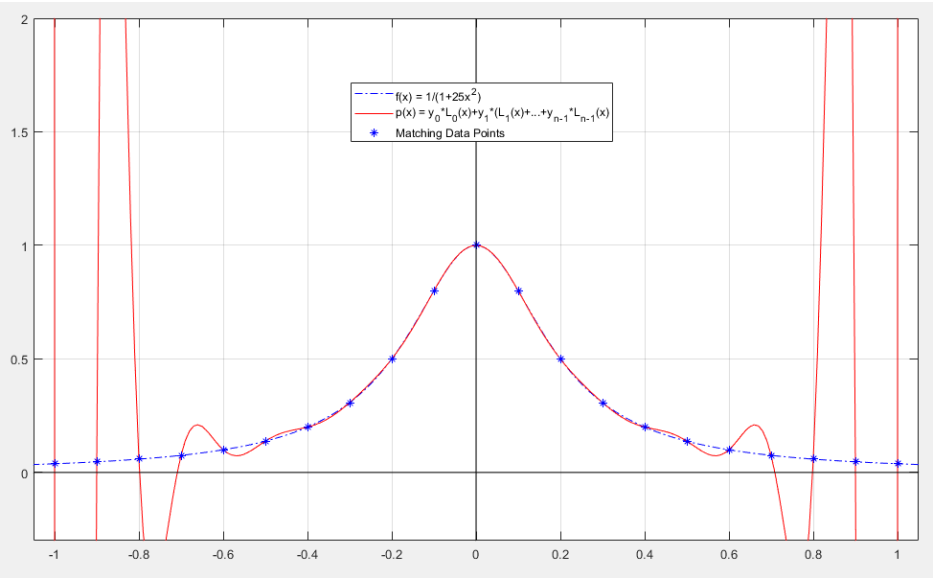
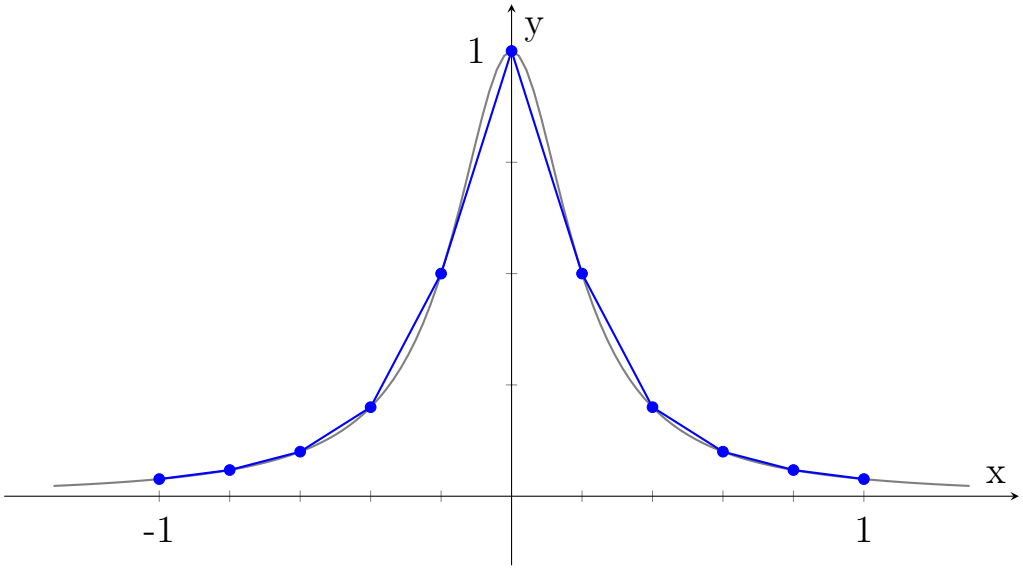


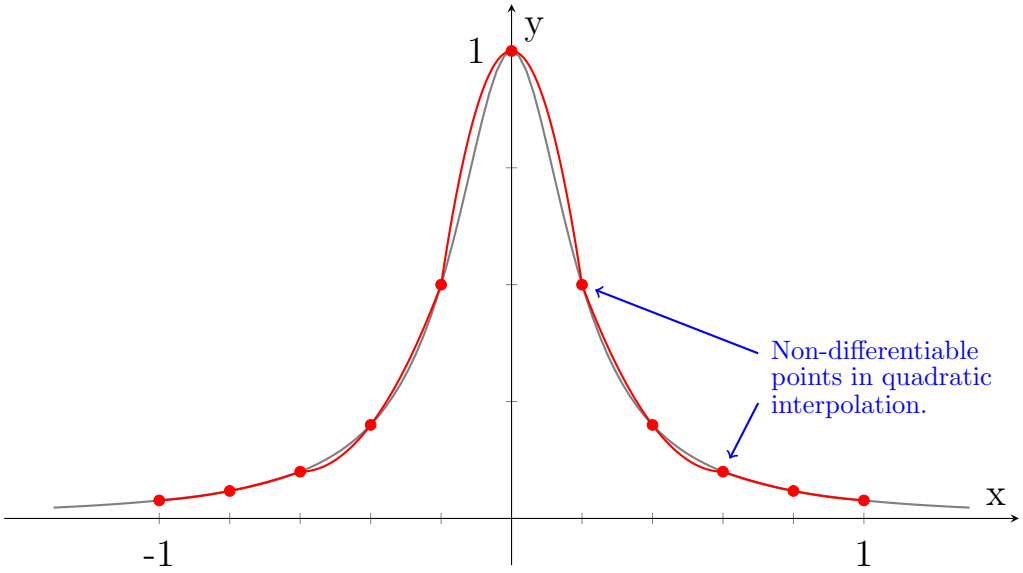
Figure 2.2: Increasing the number of points leads to more oscillations.

2.4 Spline Interpolation

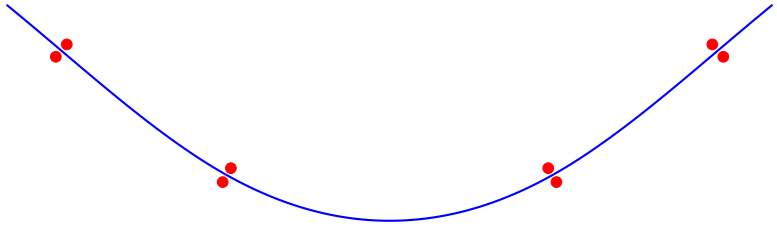
Increasing the number of data points in our interpolating polynomial leads to worse results. What is a practical solution? We could try using piecewise interpolation with small degree polynomials. The simplest case of this would be piecewise linear interpolation.



Piecewise Linear interpolation gets rid of oscillations and is continuous, but usually isn't differentiable at the interpolating points.



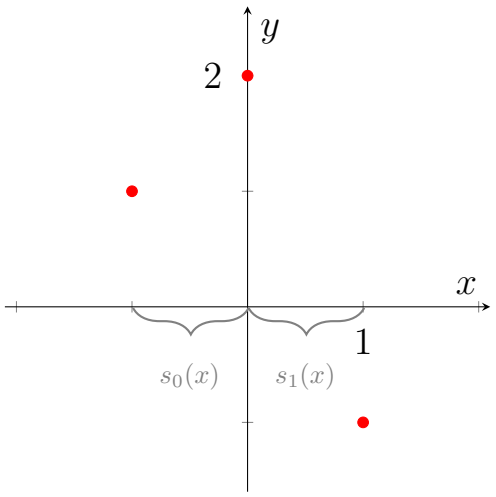
Piecewise Quadratic interpolation has better curvature, much closer to the original function. Unlike piecewise linear interpolation it's differentiable at every other point. Piecewise quadratic interpolation is popular for approximate integration Use function values and integrate the interpolating parabola's. (*Simpson's Rule*).



The term *spline* comes from historic ship building methods, which involved bending long flexible wooden beams into place, using pins. **Spline Interpolation** is a type of piecewise polynomial interpolation, in which the function values, derivatives, curvature, etc. at each of the interpolating points.

The standard method is **Cubic Spline Interpolation**, which uses 3rd order polynomials with matching function values, slope, and curvature.

Example: 3 points: (-1, 1) (0, 2) (1, 0)



The interpolated function will be composed of the following splines:

$$s_0(x) = ax^3 + bx^2 + cx + d \text{ on } [-1,0]$$
$$s_1(x) = ex^3 + fx^2 + gx + h \text{ on } [0,1]$$

We must determine eight parameters:

$$\left.\begin{array}{rcl} s_0(x_0) = y_0 & \Rightarrow & s_0(-1) = 1 \\ s_0(x_1) = y_1 & \Rightarrow & s_0(0) = 2 \\ s_1(x_1) = y_1 & \Rightarrow & s_1(0) = 2 \end{array}\right\} \text{Matching Values}$$
$$s_1(x_2) = y_2 \quad \Rightarrow \quad s_1(1) = -1$$

Matching slope at x_1:

$$s_0{}'(x_1) = s_1{}'(x_1) \quad \Rightarrow \quad s_0{}'(0) = s_1{}'(0)$$

Matching "Curvature":

$$s_0{}''(x_1) = s_1{}''(x_1) \quad \Rightarrow \quad s_0{}''(0) = s_1{}''(0)$$

Two conditions remain. There are several approaches to "fix" these conditions:

$$s_0{}''(x_0) = 0 \qquad s_1{}''(0) = s_1{}''(x_2) = 0$$

This is referred to as **Natural Cubic Splines**. The curvature at the endpoints is zero. This mimics the behavior of the wooden beams in ship building.

With **Clamped Splines** the second derivative of each of the endpoint set to a specified value.

$$s_0{}''(x_0) = [\text{some value}] \qquad s_1{}''(0) = s_1{}''(x_2) = [\text{some value}]$$

Periodic Conditions can also be specified to match the end points of the interpolated function into one that is periodic.

In our example, let's find the natural cubic spline:

$$\begin{gathered} s_0{}'(x) = 3ax^2 + 2bx + c \\ s_0{}''(x) = 6ax + 2b \\ s_1{}'(x) = 3ex^2 + 2fx + g \\ s_1{}''(x) = 6ex + 2fx \end{gathered}$$

Interpolation Conditions:

$$s_0(-1) = a \cdot (-1)^3 + b \cdot (-1)^2 + c \cdot (-1) + d = 1$$

$$\Rightarrow \mathbf{-a + b - c + d = 1}$$

$$s_0(0) = a \cdot 0^3 + b \cdot 0^2 + c \cdot 0 + d = 2$$

$$\Rightarrow \mathbf{d = 2}$$

$$s_1(0) = e \cdot 0^3 + f \cdot 0^2 + g \cdot 0 + h = 2$$

$$\Rightarrow \mathbf{h = 2}$$

$$s_1 = e \cdot 1^3 + f \cdot 1^2 + g \cdot 1 + h = -1$$

$$\Rightarrow \mathbf{e + f + g + h = -1}$$

Matching Slope:

$$s_0{}'(0) = s_1{}'(0)$$
$$\Rightarrow 3a \cdot 0^2 + 2b \cdot 0 + c = 3e \cdot 0^2 + 2f \cdot 0 + g$$
$$\Rightarrow \mathbf{c = g}$$

Matching Curvature:

$$s_0{}''(0) = s_1{}''(0)$$
$$\Rightarrow 6a \cdot 0 + 2b = 6e \cdot 0 + 2f \Rightarrow 2b = 2f \Rightarrow \mathbf{b = f}$$

Natural Conditions:

$$s_0{}''(-1) = 6a \cdot (-1) + 2b = 0$$
$$\Rightarrow \mathbf{-6a + 2b = 0}$$

$$s_0{}''(1) = 6e \cdot 1 + 2f = 0$$
$$\Rightarrow \mathbf{6e + 2f = 0}$$

This leads to a linear system with eight equations for eight unknowns. In this spline set up, this system has a unique solution.

$$\begin{gathered} -a + b - c + d = 1 \\ d = 2 \\ h = 2 \\ e + f + g + h = -1 \\ c = g \\ b = f \\ -6a + 2b = 0 \\ 6e + 2f = 0 \end{gathered}$$

Through Gaussian Elimination we find the solutions:

$$\begin{gathered} a = -\tfrac{4}{5} \\ b = f = -\tfrac{72}{25} \\ c = g = -\tfrac{-27}{25} \\ d = h = 2 \\ e = \tfrac{24}{25} \end{gathered}$$

When we plug these values into our original equations, we get the following:

$$s_0(x) = -\tfrac{4}{5}x^3 - \tfrac{72}{25}x^2 - \tfrac{27}{25}x + 2$$

$$s_1(x) = \tfrac{24}{25}x^3 - \tfrac{72}{25}x^2 - \tfrac{27}{25}x + 2$$