Numerical Linear Algebra
Lecture 2

NLA notions

- IEEE system and floating-point numbers
- Stability of polynomial evaluation

IEEE floating-point representation

IEEE 754 (established in 1985 by the Institute of Electrical and Electronics Engineers, Inc.) defines single and double precision standards (as well as other standards).

- During 60 and 70-ies every computer factory produced their own floating point system - not convenient, a lot of trubles.
- The floating-point standard was developed during 80-ties and was used by big IT-factories like Intel and Motorola.
- IEEE standard has 3 important requirements: consistent floating point representation, correct round-off arithmetics, consistent managment of exception situations.

Floating-point representation

- IEEE single precision: sign ("+" 0, "-" 1) 1 bit, exponent 8 bits, mantissa 23 bits = all together = 32 bits:

$$\underbrace{\pm}_{1}\ \underbrace{e_1 e_2 \ldots e_8}_{exponent 8}\ \underbrace{d_0 d_1 \ldots d_{22}}_{mantissa 23}$$

- IEEE dubbel precision: sign ("+" 0, "-" 1) 1 bit, exponent 11 bits, mantissa 52 bits = all together = 64 bits:

$$\underbrace{\pm}_{1}\ \underbrace{e_1 e_2 \ldots e_{11}}_{exponent 11}\ \underbrace{d_0 d_1 \ldots d_{51}}_{mantissa 52}$$

The positive number is normalized if $d_0 \neq 0$.
If $\beta = 2$ then always $d_0 = 1$ and because of that d_0 is not stored.
There exists speciell bitformat for different exceptional situations for example:
[0, -0]
ans = 0000000000000000   8000000000000000

Floating-point numbers

We can represent a floating-point number as:

$$x = \pm \left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \dots + \frac{d_{p-1}}{\beta^{p-1}} \right) \beta^e,$$

where

$$0 \leq d_k \leq \beta - 1, L \leq e \leq U,$$

Here we have:

- β is base
- e exponent
- p precision
- $[L, U]$ is exponent range
- $d_k, k = 0, \dots, p-1$ mantissa (integer)

Most of computers now use binary ($\beta = 2$) arithmetics.

base	p	L	U	
2	24	-126	127	32 bit
2	53	-1022	1023	64 bit

bas	p	L	U	
2	24	-126	127	32 bits
2	53	-1022	1023	64 bits

- The smallest representive normalised number is $2^L \approx 1.17 \cdot 10^{-38}$ in single (see table, $L = -126$) and $\approx 2.2 \cdot 10^{-308}$ in double precision (see table, $L = -1022$).
- The largest representative number has largest exponent and ones in the whole mantissa. In single precision $\approx 3.4 \cdot 10^{38}$, in double precision $\approx 1.8 \cdot 10^{308}$.
- The number which is largest than the largest representative number gives overflow and smallest than smallest positive representative normalised number gives underflow.
- Underflow Level: $UFL = \beta^L$, Overflow Level: $OFL = \beta^{U+1}(1 - \beta^{-t})$. In single precision OFL computes as: $OFL = (2^{128}) \cdot (1 - (2^{-24})) \approx 3.4 \cdot 10^{38}$, in double OFL computes as: $OFL = (2^{1024}) \cdot (1 - (2^{-53})) \approx 1.8 \cdot 10^{308}$.
- Example: Matlab program floatgui.m. Floating-point system: $\beta = 2, p = 3, L = -1, U = 1$. Number of floating point numbers: $2(\beta - 1)\beta^{t-1}(U - L + 1) + 1 = 25$.

bas	p	L	U		exponent bias
2	24	-126	127	32 bits	127
2	53	-1022	1023	64 bits	1023

Quantity of different numbers counts as:

$$2(\beta - 1)\beta^{p-1}(U - L + 1) + 1$$

and is

- $4.2614 \cdot 10^9$ for single precision: $\beta = 2, L = -126, U = 127, p = 24$ in $2(\beta - 1)\beta^{p-1}(U - L + 1) + 1$.
- $1.8429 \cdot 10^{19}$ for double precision : $\beta = 2, L = -1022, U = 1023, p = 53$ in $2(\beta - 1)\beta^{p-1}(U - L + 1) + 1$.

Example

In Matlab:

- $1e-200^2 = 10^{-200^2}$ gives underflow, answer 0.
- $1e200^2$ gives overflow, answer infinity.
- $log(0)$, answer -infinity
- sin(1/0), answer NaN (not a number)
- exp(log(10000)) − log(exp(10000))?

Answer: $\exp(\log(10000)) = 10000$, $\exp(10000) = \mathrm{Inf}$ and $\log(\exp(10000)) = \mathrm{Inf}$ and $\exp(\log(10000)) - \log(\exp(10000)) = -\mathrm{Inf}$

- exp(log(10)) − log(exp(10)) ?

Answer : $\exp(\log(10)) = 10$, $\log(\exp(10)) = 10$ and $\exp(\log(10000)) - \log(\exp(10000)) = 0$.

Exponent bias

sign	exponent 8 bits	mantissa 23 bits

sign	exponent 11 bits	mantissa 52 bits

Table: IEEE representation of floating point number in binary form in single (left table) and double (right table) precisions. For single precision we have following bitformat: 1+ 8 + 23 = 32 bits. For double precision we have following bitformat: 1+11+52 = 64 bits.

- In IEEE 754 floating point numbers, the exponent is biased - the value stored is offset from the actual value by the exponent bias, also called a biased exponent.
- It is done because exponents have to be signed values in order to be able to represent both tiny and huge values.
- To solve this problem the exponent is stored as an unsigned value which is suitable for comparison, and when being interpreted it is converted into an exponent within a signed range by subtracting the bias.
- The purpose of representation of numbers as above in tables is to enable high speed comparisons between floating point numbers using fixed point hardware.

Exponent bias

bas	p	L	U		exponent bias
2	24	-126	127	32 bits	127
2	53	-1022	1023	64 bits	1023

- To calculate the bias for an arbitrarily sized floating point number apply the formula bias $=2^{k-1}-1$, where k is the number of bits in the exponent.
- For a single-precision number, the exponent is stored in the range 1...254 (0 and 255 have special meanings), and is interpreted by subtracting the bias for an 8-bit exponent $(127 = 2^{8-1} - 1)$ to get an exponent value in the range $-126... + 127$.
- For a double-precision number, the exponent is stored in the range 1...2046 (0 and 2047 have special meanings), and is interpreted by subtracting the bias for an 11-bit exponent $(1023 = 2^{11-1} - 1)$ to get an exponent value in the range $-1022.. + 1023$.

Presentation of number in floating-point arithmetics

Example

Represent the number 6 in the binary form (base 2) in single precision. We have:

$$6: \ [1.5] \cdot 2^2 = \underbrace{[1 + 0.5]}_{mantissa} \cdot 2^2$$

Exponent $e = 2$ represents as: $2 + 127 = 129 = 2^7 + 2^0$. Mantissa: we don't represent 1 ,

$$0.5 = \frac{1}{2}x_1 + \frac{1}{4}x_2 + \frac{1}{8}x_3 + \frac{1}{16}x_4 + \ldots = \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 0 + \ldots$$

The rest of mantissa will be zero. Thus, we have obtained following binary representation for 6 in the single precision:

0	10000001	10 0
sign	exponent 8 bits	mantissa 23 bits

Presentation of number in floating-point arithmetics

We have following IEEE-format for numbers:

$$x = \pm \left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + ... + \frac{d_{p-1}}{\beta^{t-1}} \right) \beta^e,$$

Example

$$-3.25: \; -[1.625] \cdot 2^1 = -\underbrace{[1 + 0.625]}_{mantissa} \cdot 2^1$$

Exponent $e = 1$ represents as : $1 + 1023 = 1024 = 2^{10}$. Mantissa: 1 is not presented,

$$0.625 = \frac{1}{2}x_1 + \frac{1}{4}x_2 + \frac{1}{8}x_3 + \frac{1}{16}x_4 + \ldots = \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 0 + \frac{1}{8} \cdot 1 + \ldots$$

$\frac{1}{2} = 0.5 < 0.625$; $\frac{1}{4} = 0.25 : 0.625 - 0.5 = 0.125; 0.25 > 0.125$, because $\frac{1}{4} \cdot 0$, $\frac{1}{8} = 0.125 = 0.125$ and because $\frac{1}{8} \cdot 1$ and stop (the rest of mantissa will be 0).

We obtain for -3.25 :

1	10000000000	1010 0
sign	exponent 11 bits	mantissa 52 bits

Example

-3.25 in binary format:

1	10000000000	1010 0
sign	exponent 11 bits	mantissa 52 bits

-3.25 in hexadecimal (bas 16) form :

c00a000000000000

Bas 16:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
										a	b	c	d	e	f

We divide in 4 bits the binary representation to get:

1100	0000	0000	1010	0000	....	0000

and make coding for first 4 bits: $\boxed{1100} = \text{c}$

$1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 12 = c$

0000 have code 0, and

$\boxed{1010} = \text{a}$

$1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 10 = a$

Example

$-9.28 := -[1.16] \cdot 2^3 = -[1 + 0.16] \cdot 2^3$

$$0.16 = \frac{1}{2}x_1 + \frac{1}{4}x_2 + \frac{1}{8}x_3 + \frac{1}{16}x_4 + \frac{1}{32}x_5 + \ldots =$$
$$= \frac{1}{2} \cdot 0 + \frac{1}{4} \cdot 0 + \frac{1}{8} \cdot 1 + \frac{1}{16} \cdot 0 + \ldots$$

Exponenten 3 writes as: $3 + 1023 = 1026 = 1024 + 2 = 1{\cdot}2^{10} + 1 \cdot 2^1 + 0 \cdot 2^0$

1	10000000010	00101 ....
sign	exponent 11 bits	mantissa 52 bits

$\frac{1}{2} = 0.5 > 0.16$, and thus $\frac{1}{2} \cdot 0$, $\frac{1}{4} = 0.25 : 0.25 > 0.16$, and thus $\frac{1}{4} \cdot 0$, $\frac{1}{8} = 0.125 < 0.16$ and thus $\frac{1}{8} \cdot 1$, $\frac{1}{16} = 0.0625 : 0.16 - 0.125 = 0.035, 0.0625 > 0.035$, and thus $\frac{1}{16} \cdot 0$,
$\frac{1}{32} = 0.0312 : 0.0312 < 0.035$, and thus $\frac{1}{32} \cdot 1$, and so on ...

Example

$-9.28 := -[1.16] \cdot 2^3 = -[1 + 0.16] \cdot 2^3$

In Matlab:

```
q = quantizer('double');
y = num2bin(q,-9.28)
y =
11000000001000101000111101011100001010001111010111000010100011
```

or to see all numbers:

```
y =
1100000000100010100011110101110000101000111101011100001010001111
```

Example

$6.28 = +[1.57] \cdot 2^2 = +[1 + 0.57] \cdot 2^2$

$$0.57 = \frac{1}{2} + 0.07 =$$
$$= 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} + 0 \cdot \frac{1}{8} + 1 \cdot \frac{1}{16} + 0 \cdot \frac{1}{32} + \ldots + \frac{1}{256} + \ldots$$

Exponent 2 represents as: $2 + 1023 = 1025 = 1024 + 1 = 1 \cdot 2^{10} + 1 \cdot 2^0$

0	10000000001	10010 ....10...
sign	exponent 11 bits	mantissa 52 bits

In Matlab:

y = num2bin(q,6.28)

y =

01000000000110010001111010111000010100011110101110000101000111

To see all numbers: y =
0100000000011001000111101011100001010001111010111000010100011111

Example

$1 := [1] \cdot 2^0 = [1 + 0.0] \cdot 2^0$

Mantissa: 0.

Exponent 0 represents as: $0 + 1023 = 1024 - 1 = 1 \cdot 2^{10} - 1 \cdot 2^0 =$
$\underbrace{10000000000}_{11 \;\; bits} - \underbrace{00000000001}_{11 \;\; bits} = \underbrace{01111111111}_{11 \;\; bits}.$

0	01111111111	0000 ....
sign	exponent 11 bits	mantissa 52 bits

Example

$0.5 := [1] \cdot 2^{-1} = [1 + 0.0] \cdot 2^{-1}$

Mantissa: 0.

Exponent -1 represents as: $-1 + 1023 = 1024 - 2 =$
$1 \cdot 2^{10} - 1 \cdot 2^{1} = \underbrace{10000000000}_{11\ \textit{bits}} - \underbrace{00000000010}_{11\ \textit{bits}} = \underbrace{01111111110}_{11\ \textit{bits}}.$

0	01111111110	0000 ....
sign	exponent 11 bits	mantissa 52 bits

Example

$0.1 := [1.6] \cdot 2^{-4} = [1 + 0.6] \cdot 2^{-4}$

Mantissa 0.6:

$$0.6 = \frac{1}{2}x_1 + \frac{1}{4}x_2 + \frac{1}{8}x_3 + \frac{1}{16}x_4 + \frac{1}{32}x_5 + \ldots =$$
$$= \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 0 + \frac{1}{8} \cdot 0 + \frac{1}{16} \cdot 1 + \ldots$$

Mantissa 0.6:
$\frac{1}{2} = 0.5 < 0.6$ *and thus*, $x_{\mathbf{1}} = 1.$; *Next*, $0.6 - 0.5 = 0.1$; $\frac{1}{4} = 0.25 : 0.25 > 0.1$ *and thus* $x_{\mathbf{2}} = 0$; $1/8 = 0.125 > 0.1$ *and thus* $x_{\mathbf{3}} = 0$; $1/16 = 0.0625 < 0.1$ *and thus* $x_{\mathbf{4}} = 1; \ldots$
Exponent -4 lagras som:
$-4 + 1023 = 1019 = 1024 - 5 = 1 \cdot 2^{\mathbf{10}} - (1 \cdot 2^{\mathbf{2}} + 1 \cdot 2^{\mathbf{0}}) = \underbrace{10000000000}_{\mathbf{11}\ bits} - \underbrace{00000000101}_{\mathbf{11}\ bits} = \underbrace{01111111011}_{\mathbf{11}\ bits}.$

0	01111111011	100...
sign	exponent 11 bits	mantissa 52 bits

In Matlab:

```
q = quantizer('double');
y = num2bin(q,0.1)
```

```
y =
0011111110111001100110011001100110011001100110011001100110011010
```

Floating-point numbers: normalized numbers

In base b a normalized number has the form

$$\pm d_0.d_1 d_2 d_3 .... \times b^n$$

Here, $d_0 \neq 0$, $d_0, d_1, ...$ are integers between 0 and $b - 1$.

Example

210.432 in normalized form is:

$$2.10432 \times 10^2$$

-0.00532101 in normalized form is:

$$-5.32101 \times 10^{-3}$$

Any non-zero real number can be normalized.

Normalised decimal number:

$$\pm s_1.s_2s_3s_4... \cdot 10^e,$$

where $s_1, s_2, .., s_1 \neq 0$ denote decimal numbers.
Denormalized numbers in single precision are defined as (f=fraction):

$$\pm 0.f \cdot 2^{-126}.$$

Denormalized numbers in dubbel precision:

$$\pm 0.f \cdot 2^{-1022},$$

Example

$x = 918.082$ in normalised form: $9.18082 \cdot 10^2$.
-0.00574012 in normalised form: $-5.74012 \cdot 10^{-3}$.

Flyttal-räkning: IEEE Standard 754

Floating Point Range

	Binary system, denormalised n.	Binary system, normalised n.
Single p.	$\pm 2^{-149}$ till $(1-2^{-23}) \cdot 2^{-126}$	$\pm 2^{-126}$ till $(2-2^{-23}) \cdot 2^{127}$
Dubble p.	$\pm 2^{-1074}$ till $(1-2^{-52}) \cdot 2^{-1022}$	$\pm 2^{-1022}$ till $(2-2^{-52}) \cdot 2^{1023}$

	Decimal system, denormalized n.
Single p.	$\pm \approx 10^{-44.85}$ till $\approx 10^{38.53}$
Dubble p.	$\pm \approx 10^{-323.3}$ till $\approx 10^{308.3}$

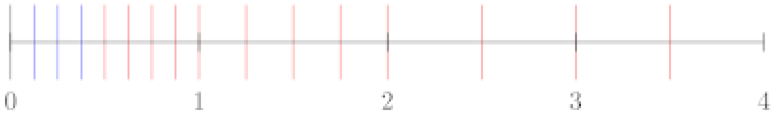


Figure: *Normalised numbers(red), denormalised numbers (blue) (Wikipedia)*

Matlab's program

```
    floatgui.m
```

shows structure of floating point numbers. In this program, the set of positive model floating point numbers is determined by three parameters:

```
  t, emin,  emax
```

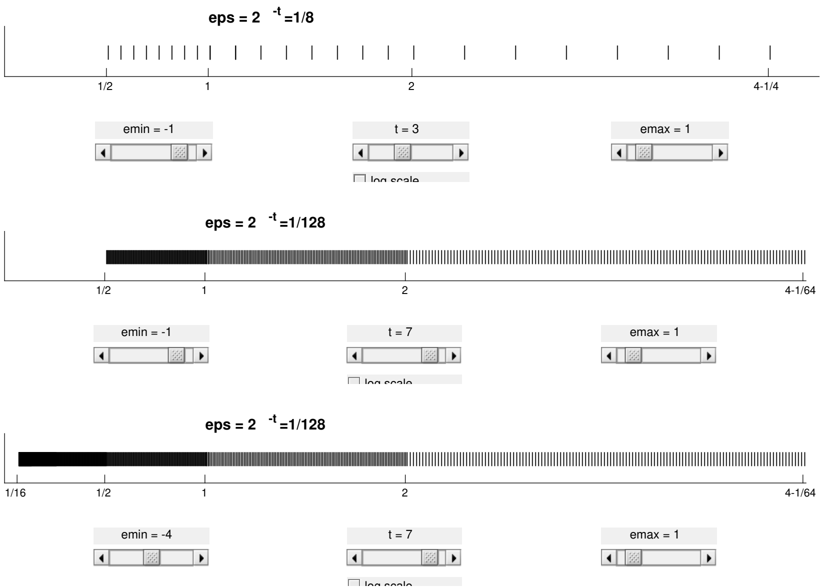
It is the set of rational numbers of the form

```
      x = (1+f)*2^e
```

where

```
  f = (integer)/2^t, 0 <= f < 1, e = integer,
  emin <= e <= emax.
```

Program floatgui.m



eps = 2^{-t}=1/8

emin = -1 t = 3 emax = 1

eps = 2^{-t}=1/128

emin = -1 t = 7 emax = 1

eps = 2^{-t}=1/128

emin = -4 t = 7 emax = 1

Machine epsilon ε_{mach}

ε_{mach} depends on the method which we use in round-off.
There are different definitions for ε_{mach} for precision p:

- In rounding by choping:

$$\varepsilon_{mach} = \beta^{1-p} = \beta^{-t}$$

- In rounding to nearest:

$$\varepsilon_{mach} = \frac{1}{2}\beta^{1-p} = \frac{1}{2}\beta^{-t}$$

In dubbel precision (when $p = 53$ for 64 bits computer) we have $\varepsilon_{mach} = 2^{-p} \approx 1.11 \cdot 10^{-16}$ and in the single precision (when $p = 24$ for 32 bits computer) we have $\varepsilon_{mach} = 2^{-p} \approx 6 \cdot 10^{-8}$.

	chop rounding to 2 digits	to nearest to 2 digits
1.849	1.8	1.8
1.850	1.8	1.9
1.851	1.8	1.9
1.899	1.8	1.9

Machine precision and backward error analysis

If x is real number we define round-off of this number as floating-point number $fl(x)$ (floating).

Example

Assume that we perform computations with 4 numbers.

$$\begin{aligned} fl(\pi) &= fl(3.141592653589...) = 3.142, \\ fl(31415926.53589...) &= 3.142 \cdot 10^7 \end{aligned} \tag{1}$$

in normalized form.

Usually the relative absolute error in round-off procedure (rounding to nearest) is defined as $0.5 \cdot \beta^{1-p}$. This error is called the relative Machine epsilon.

Machine precision and backward error analysis

Relative error in representing of real number $x \neq 0$ within the normalized range of a floating point system is defined as:

$$\left| \frac{fl(x) - x}{x} \right| \leq \varepsilon_{mach}$$

Let us analyze if scalar product

$$\sum_{k=1}^{n} x_k y_k$$

is stable when we use IEEE floating point system.

Machine precision and backward error analysis

Let us take $n = 4$ and $\sigma_k = 1 + \delta_k$, $\pi_k = 1 + \epsilon_k$ (we have σ for summa and π for product when all $|\epsilon_k| < \epsilon_{mach}$ and $|\delta_k| < \epsilon_{mach}$). We have:

- $fl(x_1 y_1) = x_1 y_1 \pi_1$
- $fl(x_1 y_1 + x_2 y_2) = [x_1 y_1 \pi_1 + x_2 y_2 \pi_2]\sigma_2$
- $f(x_1 y_1 + x_2 y_2 + x_3 y_3) = ([x_1 y_1 \pi_1 + x_2 y_2 \pi_2]\sigma_2 + x_3 y_3 \pi_3)\sigma_3$.
- $fl(x_1 y_1 + x_2 y_2 + x_3 y_3 + x_4 y_4) =$ $[([x_1 y_1 \pi_1 + x_2 y_2 \pi_2]\sigma_2 + x_3 y_3 \pi_3)\sigma_3 + x_4 y_4 \pi_4]\sigma_4$.

Then in the common case for all n we have $\sigma_{j:k} = \sigma_j \sigma_{j+1} \dots \sigma_k$, and:

$$fl(x_1 y_1 + x_2 y_2 + ... + x_n y_n) = x_1 y_1 \pi_1 \sigma_{2:n} + x_2 y_2 \pi_2 \sigma_{2:n} + x_3 y_3 \pi_3 \sigma_{3:n} + ... + x_k y_k \pi_k \sigma_{k:n} + ... + x_n y_n \pi_n \sigma_n.$$

Machine precision and backward error analysis

We get

$$fl(\sum_{k=1}^{n} x_k y_k) = x_1 y_1 (1 + n\epsilon_1') + \sum_{k=2}^{n} x_k y_k (1 + (n - k + 2)\epsilon_k')$$

when all $|\epsilon_k'| \leq \epsilon_{mach}$. For example, if we will take $\tilde{x}_k = x_k \sqrt{1 + (n - k + 2)\epsilon_k'}$ and $\tilde{y}_k = y_k \sqrt{1 + (n - k + 2)\epsilon_k'}$ then we have:

$$fl(\sum_{k=1}^{n} x_k y_k) = \sum_{k=1}^{n} \tilde{x}_k \tilde{y}_k$$

When $n\epsilon_{mach}$ is small then computation of scalar product is stable IEEE algorithm.

Cancellation problem in float-point numbers

We should avoid make operations $(+, -, *, /)$ with numbers of very different sizes.
The result of floating point operations might differ from the floating point representation of the result of the operations on real numbers.
This means that $a + (b + c)$ not exactly can be $(a + b) + c$.
Cancellation can happens when we subtract two almost the same numbers.
Example: $1.03678947f - 1.03678935g = 0.00000012t$.
Here, f and g denote error, and vi get new error t after subtracting numbers. We observe that the first error f comes in the 10'th digit $1.03678947 \underbrace{f}_{10 \ digit}$, but in the difference the error has moved to the third digit: $0.00000012t = 1.2 \underbrace{t}_{3 \ digit} e - 7$. This means, that we lost information, the results loses significant digits.
This example shows that if you subtract two numbers that are very close to each other in the sense of small relative difference, then if you use floating point arithmetic you can get wrong answer.

Stability of polynomial evaluation

We will discuss stability of polynomial evaluation by Horner's rule. Let the polynomial is given by

$$p(x) = \sum_{i=0}^{d} c_i x^i = c_0 + c_1 x^1 + \ldots + c_d x^d,$$

where c_i are coefficients of the polynomial, d is its degree.
To compute roots of this polynomial we can use Horner's rule, see alg.below. This rule can be programmed as the following iterative algorithm for every mesh point $x_j \in [x_{left}, x_{right}], j \in 1, 2, \ldots N$, where N is the total number of the discretization points:
Algorithm 1

- Step 0. Set counter $i = d - 1$ and initialize $p_d = c_d$.
- Step 1. Compute $p_i = x_j \cdot p_{i+1} + c_i$
- Step 2. Set $i := i - 1$ and go to step 1. Stop if $i = 0$.

Horner's method: example

Let the polynomial is given by

$$p(x) = \sum_{i=0}^{3} c_i x^i = c_0 + c_1 x^1 + c_2 x^2 + c_3 x^3,$$

then Alghorithm 1 (Horner's method) gives:

$$c_0 + c_1 x^1 + c_2 x^2 + c_3 x^3 = c_0 + x(c_1 + x(c_2 + x c_3)),$$

where iterations of Algorithm 1 are done as follows:

$$\begin{aligned} p &= c_3 \\ p &= c_2 + xp \\ p &= c_1 + xp \\ p &= c_0 + xp \end{aligned}$$

Example

We need evaluate of roots of polynomial together with upper and lower bounds for this solution.
Let us take
$p(x) = (x-9)^9 = x^9 - 81x^8 + 2916x^7 - 61236x^6 + 826686x^5 - 7440174x^4 + 44641044x^3 - 172186884x^2 + 387420489x^1 - 387420489.$
To compute bounds of the solution we insert term with error $1 + (\sigma_{1,2})_i$ for every floating point iteration in Algorithm 1 to get following algorithm:

Algorithm 2

- Step 0. Set counter $i = d - 1$ and initialize $p_d = c_d$.
- Step 1. Compute
  $p_i = (x_j \cdot p_{i+1}(1 + (\sigma_1)_i) + c_i)(1 + (\sigma_2)_i), \quad |(\sigma_1)_i|, |(\sigma_2)_i| \leq \varepsilon$
- Step 2. Set $i := i - 1$ and go to step 1. Stop if $i = 0$.

In the algorithm above ε is the machine epsilon and we define it as the maximum relative representation error $0.5 \cdot \beta^{1-p}$ which is measured in a floating point arithmetic with the base β and with precision $p > 0$. Now the following values of machine epsilon apply to standard floating point formats:

Table 1. *Values of machine epsilon in standard floating point formats. Notation $*$ means that one bit is implicit in precision p. Machine epsilon ε_1 is computed accordingly to Demmel.*

EEE 754 - 2008	description	Base, b	Precision, p	Machine eps.1 $\varepsilon_1 = 0.5 \cdot b^{-(p-1)}$
binary16	half precision	2	11^*	$2^{-11} = 4.88e - 04$
binary32	single precision	2	24^*	$2^{-24} = 5.96e - 08$
binary64	double precision	2	53^*	$2^{-53} = 1.11e - 16$
binary80	extended precision	2	64	$2^{-64} = 5.42e - 20$
binary128	quad. precision	2	113^*	$2^{-113} = 9.63e - 35$
decimal32	single prec. decimal	10	7	5×10^{-7}
decimal64	double prec. decimal	10	16	5×10^{-16}
decimal128	quad. prec. decimal	10	34	5×10^{-34}

Expanding expression for p_i in the algorithm 2 we can get the final value p_0 as

$$
\begin{aligned}
p_0 = & \sum_{i=0}^{d-1} \left((1+(\sigma_2)_i) \prod_{k=0}^{i-1} (1+(\sigma_1)_k)(1+(\sigma_2)_k) \right) c_i x^i \\
& + \left(\prod_{k=0}^{d-1} (1+(\sigma_1)_k)(1+(\sigma_2)_k) \right) c_d x^d
\end{aligned}
\tag{2}
$$

Next, we will write upper and lower bounds for products of $\sigma := \sigma_{1,2}$ provided that $k\varepsilon < 1$:

$$
\begin{aligned}
(1+\sigma_1)\cdot \ldots \cdot (1+\sigma_k) \le (1+\varepsilon)^k \le 1 + k\varepsilon + O(\varepsilon^2), \\
1 - k\varepsilon \le (1-\varepsilon)^k \le (1+\sigma_1)\cdot \ldots \cdot (1+\sigma_k)
\end{aligned}
\tag{3}
$$

Applying estimate above we can get the following approximation

$$
1 - k\varepsilon \le (1+\sigma_1)\cdot \ldots \cdot (1+\sigma_k) \le 1 + k\varepsilon. \tag{4}
$$

Using the estimate above we can rewrite (2) assuming $|\tilde{\sigma}_i| \le 2d\varepsilon$ as

$$
p_0 \approx \sum_{i=0}^{d} (1+\tilde{\sigma}_i) c_i x^i = \sum_{i=0}^{d} \tilde{c}_i x^i \tag{5}
$$

We also can write formula for the computing error in the polynomial:

$$|p_0 - p(x)| = \left|\sum_{i=0}^{d}(1+\tilde{\sigma}_i)c_i x^i - \sum_{i=0}^{d} c_i x^i\right| = \left|\sum_{i=0}^{d} \tilde{\sigma}_i c_i x^i\right| \leq 2d\varepsilon \sum_{i=0}^{d} |c_i x^i| \tag{6}$$

If we will choose $\tilde{\sigma}_i = \varepsilon \cdot \mathrm{sign}(\mathrm{c_i x^i})$ then the error bound above can be attained within the factor $2d$. In this case we can take $\frac{\sum_{i=0}^{d} |c_i x^i|}{|\sum_{i=0}^{d} c_i x^i|}$ as the relative condition number for the case of polynomial evaluation. The following algorithm computes the lower and upper bound bp in the polynomial evaluation at every point x_j on the interval $[p - bp, p + bp]$.
Algorithm 3

- Step 0. Set counter $i = d - 1$ and initialize $p = c_d$, $bp = |c_d|$.
- Step 1. Compute $p = x_j \cdot p + c_i$, $bp = |x_j| \cdot bp + |c_i|$.
- Step 2. Set $i := i - 1$ and go to step 1. Stop if $i = 0$.
- Step 3. Set $bp = 2 \cdot d \cdot \varepsilon \cdot bp$ as error bound at the point $|x_j|$.

Computation of roots of the polynomial

We want to compute roots of $(x-1)^5 = 0$ in Matlab. To do this we need to rewrite it as:

$$(x-1)^5 = x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1$$

We already see that all roots should be $x = 1$. But in Matlab we have:
$r = roots([1 \ \ -5 \ 10 \ \ -10 \ 5 \ \ -1])$
and computed roots are:
$r = 1.0008 + 0.0006i$
$1.0008 - 0.0006i$
$0.9997 + 0.0009i$
$0.9997 - 0.0009i$
0.9990
Error:
$disp(abs(r-1)')$
1.1322e-03 1.1322e-03 1.1326e-03 1.1326e-03 1.1328e-03

Computation of roots of the polynomial

Why ? Let us analyze the problem

$$(x-1)^5 = \varepsilon,$$

Then we can find that

$$x = 1 + \varepsilon^{1/5}$$

If $\varepsilon = 10^{-15}$ then we will have $\varepsilon^{1/5} = 10^{-15/5} = 10^{-3}$. We observe that zeros of polynomial $(x-1)^5$ are very sensitive to the changes in the coefficients. Is it always like that?

For coefficients

```
c = [1  -15  85  -225  274  -120];
```

exact roots are: 1,2,3,4,5:

```
r = roots(c);
fel = sort(r) - (1:5)'
```

And the error is very small now

```
fel = -4.9960e-15 6.6613e-14 -1.5010e-13 9.6811e-14 -8.8818e-16
```

Condition number

We can see the roots r of the polynomial as functions $f(c)$ depending on the coefficients c .

$$r = f(c).$$

When we perturbate coefficients $c + \delta c$, we also perturbate roots $r + \delta r$.
If small changes in data $|\delta c|/|c|$ gives small changes in the output data, or result $|\delta r|/|r|$, then the problem is called well-posed. Otherwise, the problem is ill-posed. Condition number is defined as

$$k = \frac{|\delta r|/|r|}{|\delta c|/|c|} \tag{7}$$

It is not always possible compute this number, but often is possible compute estimate for k from (7):

$$|\delta r|/|r| \leq k|\delta c|/|c|.$$

Stability of the computation of roots of the polynomial: example

Example

How sensitive are perturbations in the roots of the eqution $x^2 + ax + b = 0$, for changes in a and b?

Let's define roots of the equation

$$x^2 + ax + b = 0 \tag{8}$$

by r_1 and r_2, these roots are functions depending on a and b: $r_1(a, b), r_2(a, b)$. Let $r = (r_1, r_2)$ denotes any of root and $r + \delta r$ denotes the perturbated root when we change a with δa and b with δb. Then substituting these perturbations in (8) we get:

$$x^2 + ax + b = (r + \delta r)^2 + (a + \delta a)(r + \delta r) + (b + \delta b) = 0.$$

Example

We can rewrite it as:

$$(r^2 + ar + b) + (\delta r(2r + a) + \delta ar + \delta b) + ((\delta r)^2 + \delta a \delta r) = I_1 + I_2 + I_3 = 0,$$

where

$$\begin{aligned} I_1 &= (r^2 + ar + b) = 0, \\ I_2 &= (\delta r(2r + a) + \delta ar + \delta b) \approx 0, \\ I_3 &= ((\delta r)^2 + \delta a \delta r) \approx 0. \end{aligned} \tag{9}$$

From the second equation of system (9) we get

$$\delta r \approx -\frac{(\delta a \ r + \delta b)}{2r + a}$$

which can be rewritten as

$$|\delta r| \leq \frac{(|\delta a \ r| + |\delta b|)}{|2r + a|} \tag{10}$$

Example

Since r_1 and r_2 are roots of (8) we can write

$$(x - r_1)(x - r_2) = x^2 \underbrace{-(r_1 + r_2)}_{a} x + \underbrace{r_1 r_2}_{b} = x^2 + ax + b$$

since

$$-(r_1 + r_2) = a, \;\; b = r_1 r_2.$$

Adding $2r_1$ to the first of the above equations we can rewrite it as $r_1 - r_2 = 2r_1 + a$, and define the gap as $g := |r_1 - r_2|$.

Example

We rewrite the equation (10) as

$$|\delta r| \leq \frac{|\delta a \ r| + |\delta b|}{|g|} \tag{11}$$

If g i small or if $r_1 \approx r_2$, then $|\delta r|$ is big. Dividing (11) with $|r|$ and compencating with $|a|$ and $|b|$ we get

$$\frac{|\delta r|}{|r|} \leq \frac{1}{|r|} \left(\frac{\frac{|a|}{|a|}|\delta a \ r| + \frac{|b|}{|b|}|\delta b|}{|g|} \right) \leq k \max \left(\frac{|\delta a|}{|a|}, \frac{|\delta b|}{|b|} \right), \tag{12}$$

and condition number is

$$k \approx \frac{|a| + |b/r|}{g}. \tag{13}$$

This is only estimate for the condition number since if we don't know exact roots - we can't use this formula to compute the condition number.

Example

Let us consider the polynomial

$$p(x) = (x-1)(x-1.0001) = x^2 - 2.0001x + 1.0001. \qquad (14)$$

We know from the estimate (13) that $k \approx \frac{|a|+|b/r|}{g}$, where gap is defined as $g := |r_1 - r_2|$. In our case the gap is $|1.0001 - 1| = 10^{-4}$ and the condition number is big:

$$k \approx \frac{|-2.0001| + |1.0001/r|}{|1.0001 - 1|} \approx 3 \cdot 10^4.$$

Suppose that our numerical method gives bad numerical approximations to the roots 1.11 and 0.895. The relative error is around 11%. The perturbated polynom with roots 1.11 och 0.895 is:

$$(x-1.11)(x-0.895) = x^2 - 2.005x + 0.99345$$

and compare it with (14) - we have solved almost right problem. However, our roots are not so good approximations to the exact roots in (14). We have bad approximations since our problem is ill-conditioned.

Question Q1.21 in the text book

Apply Bisection Algorithm (you can find this algorithm in the course book, see question 8.3, Algorithm 8.11) **Bisection**, to find the roots of the polynomial

$$p(x) = (x-1)^3 (x-5)^2 (x-7)^3 = 0$$

for different input intervals $x = [x_{left}, x_{right}]$. Use these algorithms to determine also the roots **of some of your own polynomial**. Note that in the Bisection Algorithm $p(x)$ should be evaluated using Horner's rule. Write your own matlab program. Confirm that changing the input interval for $x = [x_{left}, x_{right}]$ slightly changes the computed root drastically. Modify the algorithm to use the relative condition number for polynomial evaluation given by (8.26) of the course book:

$$cond(p) := \frac{\sum_{i=0}^{d} |c_i x^i|}{\left| \sum_{i=0}^{d} c_i x^i \right|}$$

to stop bisecting when the round-off error in the computed value of $p(x)$ gets so large that its sign cannot be determined. Present your results similarly with results of Fig. 8.2, 8.3 of the course book. Compare your results with results of Fig. 8.2, 8.3 of the course book.

Notes

:

- Study Example 8.3, page 262, of the course book.
- Note, that in Horner's rule a_i denote coefficients of the polynomial

$$p(x) = \sum_{i=0}^{d} a_i x^i,$$

where d is the degree of the polynomial. Thus, you need first compute coefficients of the polynomial $p(x)$. For example, exact coefficients of polynomial $p(x) = (x-9)^9$ are:

$$\begin{aligned} a = [&-387420489; 387420489; -172186884; 44641044; \\ &-7440174; 826686; -61236; 2916; -81; 1]. \end{aligned} \tag{15}$$

Use the Matlab function **coeffs** to compute the coefficients of a polynomial $p(x)$.

Notes

- Compute error bound $bp = \triangle$ as in the algorithm 8.6 of the course book for every point $x \in [x_{left}, x_{right}]$. Below is example of Matlab's function which can be used to compute this error bound:

  ```
  P = a(d); bp = abs(a(d));
  for i = d - 1:(-1):1
  P = x*P + a(i); bp = abs(x)*bp + abs(a(i));
  end
  bp = 2*(d - 1)*eps*bp;
  end
  ```

  Here, eps is the machine epsilon. Machine eps in matlab: $eps = 2.220446049250313e - 16$;