3. SZÁMÍTÁSOK HIBÁI

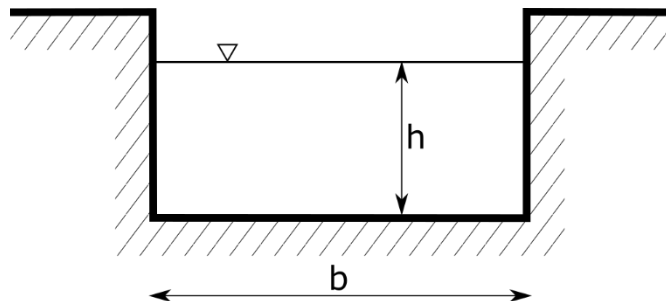
BEVEZETÉS A NUMERIKUS MÓDSZEREKBE

Bizonyos feladatokat, problémákat a hagyományos analitikus matematikai módszerekkel nem, vagy csak nagyon nehezen lehet megoldani. Ezekben az esetekben használhatóak a numerikus módszerek. Az analitikus megoldások egzakt megoldások, ahol a feladatban szereplő változók zárt képletekkel kifejezhetőek. Ilyen például egy másodfokú egyenlet megoldó képlete. A numerikus megoldás csak egy megközelítő numerikus értéke a tényleges megoldásnak. Bár a numerikus megoldás csak egy közelítés, az értéke nagyon pontos is lehet. A legtöbb numerikus módszer iteratív eljárás, ahol fokozatosan közelítjük a megoldást, amíg a kívánt pontosságot el nem érjük. Nézzünk egy építőmérnöki példát, olyan esetre, ami analitikusan nem megoldható!

Hidraulikában gyakori feladat a csatorna méretezés. A nyílt felszínű csatorna alakja, anyaga, esése, szélessége és a vízmagasság függvényében levezethető az elszállított vízhozam nagysága. Nézzük például egy szabadfelszínű, téglalap keresztmetszetű csatorna vízhozamának a képletét!

$$Q = \frac{\sqrt{S}}{n} \cdot \frac{(b \cdot h)^{\frac{5}{3}}}{(b + 2 \cdot h)^{\frac{2}{3}}}$$

ahol Q - vízhozam, n - Manning-féle érdességi együttható, S - esés, b - csatorna szélessége, h - vízmagasság.



Ez egy egzakt képlet Q-ra, ha azonban arra vagyunk kíváncsiak, hogy a mértékadó vízhozam mekkora vízmélységgel képes lefolyni, akkor azt már nem tudjuk explicit módon kifejezni. Korábban erre a feladatra különböző grafikus vagy táblázatos formájú méretezési segédleteket használtak. Ugyan analitikus módon most sem tudjuk előállítani a megoldást, viszont a számítógépek megjelenése óta, numerikus módszereket használva előre megadott pontosságú közelítő értéket tudunk adni a megoldásra. Ez azt jelenti, hogy ha visszahelyettesítjük a megoldást h-ra, akkor nem kapjuk ugyan vissza pontosan a vízhozam (Q) értékét, de nagyon közel leszünk hozzá.

Évszázadok óta vannak kidolgozott numerikus technikák az ilyen feladatok megoldására, de ezeknek az alkalmazása a mai számítógépek megjelenése előtt nagyon bonyolult volt. A kézzel, vagy mechanikus számológéppel végzett számítások nagyon időigényesek és könnyen elszámolhatóak voltak. Ezek a technikák csak a számítástechnika megjelenésével terjedhettek el, mivel ezekkel már nem okoz problémát a sok ismétlődő, bonyolult számítás elvégzése rövid idő alatt.

Egy mérnöki probléma megoldása során először definiálni kell a feladatot, változókat, feltételeket (határértékek, kezdőértékek). Utána fel kell írni a fizikai modellt a problémához, lehet ez a vízhozam képlete, tömegvonzás, Newton törvényei stb. El kell dönteni, hogy megoldható-e a feladat analitikusan vagy csak numerikusan. Lehet-e

esetleg elfogadható egyszerűsítéseket (pl. linearizálás) tenni az analitikus megoldáshoz. Időnként numerikus számítások esetében is egyszerűsítésekkel kell élni, ha túl bonyolult a modell, hogy belátható időn belül megoldható legyen a feladat (lásd például időjárás előrejelzések).

Numerikus számítások alkalmazása esetén is ki kell választani, hogy melyik módszert alkalmazzuk. Minden feladat típushoz sokféle kidolgozott módszer közül választhatunk. A módszerek különbözhetnek pontosságban, számításigényben és a programozás bonyolultságában is. A kiválasztott algoritmust az általunk használt programozási nyelven implementálni is kell. Matlab/Octave alkalmazása esetén a legtöbb esetben nem nekünk kell leprogramozni az algoritmusokat, mivel nagyon sok beépített numerikus módszer található meg bennük, azonban ezeknek a hátterével is célszerű megismerkedni, hogy helyesen tudjuk alkalmazni őket.

A feladat megoldása után le is kell valamilyen módon ellenőriznünk a megoldást. Egy nemlineáris egyenlet esetében ez történhet például visszahelyettesítéssel. Bonyolultabb esetekben, például differenciálegyenletek megoldásakor, a numerikus megoldást összehasonlíthatjuk egy hasonló probléma ismert megoldásával, vagy megoldhatjuk különböző módszerekkel és vizsgálhatjuk ezek eltéréseit.[1]

KEREKÍTÉSI HIBA, LEBEGŐPONTOS SZÁMÁBRÁZOLÁS

Mivel a feladatokat számítógép segítségével fogjuk megoldani, ismernünk kell a számítógépek korlátait is, tudnunk kell, hogy hogyan tárolják a számítógépek a számokat, milyen hibák fakadhatnak a tárolási módból, vagy az algoritmusok megválasztásából. Nézzük meg a következő egyszerű példákat!

Próbáljuk ki Matlab-ban a következőt:

```
> x1 = 0.3, x2 = 0.1+0.1+0.1
```

Egyenlő egymással ez a két szám? Nyilvánvaló, hogy igen. Azért ellenőrizzük le Matlabban is!

```
> x1==x2
```

Válaszul azt kaptuk, hogy 0 (hamis)! Mi lehet az oka? Miért nem tud a Matlab egy ilyen nyilvánvaló, egyszerű példát megoldani? Hogy megértsük vizsgáljuk meg hogy tárolja a számítógép a számokat!

- A számok tárolásának alapértelmezett és leggyakoribb formája a lebegőpontos számábrázolás (double precision floating point = dupla pontosságú lebegőpontos).
- Ez az IEEE szabványosított formátuma (IEEE 754).
- Minden számot bináris formátumban tárolnak 64 bit (0-63) felhasználásával.

Szám ábrázolása:

$$(-1)^s \cdot m \cdot 2^{(e-1023)},$$

[1] Lásd: Amos Gilat, Vish Subramaniam (2011): Numerical Methods, An Introduction with Applications Using MATLAB (SI Version), John Wiley & Sons (Asia)

ahol *s* az előjel bit (1 bit), *m* a mantissza (az értékes jegyek 52 biten) és *e* az exponenciális kitevő (11 biten) (s+m+e=>1+11+52 = 64 bit).

A végtelen sok valós számból nem lehet mindent pontosan reprezentálni, lásd pl. 0.1 a kettes számrendszerben végtelen szakaszos tizedes (vagyis jelen esetben 'kettedes') tört lesz, amiből mi csak az első 52 bitet használjuk, a többit eldobjuk. Ez a kerekítési hiba.

$$\frac{1}{10} = \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^8} + \frac{1}{2^9} + \frac{1}{2^{12}} + \frac{1}{2^{13}} + \frac{1}{2^{16}} + \cdots$$

0.1 in binary → 0.0001100110011… (végtelen szakaszos tört)

A végtelen szakaszos tört háromszori összeadása kerekítési hibát eredményez, így a két érték nem lesz pontosan megegyező a számábrázolás során használt pontosságon belül.

Nézzük meg mekkora a $tg\left(\frac{\pi}{2}\right)$ értéke? A tangens függvény π/2-nél nincs értelmezve, a végtelenhez tart. Mi történik, ha Matlabban szeretnénk ezt kiszámolni? Hibaüzenetet kapunk?

```
> tan(pi/2)
```

Mit ad a Matlab eredményül? $1.6331 \cdot 10^{16}$, ami nem végtelen, csak egy nagyon nagy szám. Miért? Mert a kerekítési hiba miatt $\frac{\pi}{2}$ értékét sem tudjuk pontosan megadni.

Mi is az a kerekítési hiba pontosan?

$$1 = 1 + \delta, if\ \delta < \varepsilon_m,$$

ahol δ a kerekítési hiba, és ε a gépi pontosság vagy gépi epszilon.

A gépi pontosság a legkisebb ábrázolható távolság alapesetben 1 és az azt követő legkisebb szám között. Matlab-ban az **eps** változóval vagy az **eps()** függvénnyel lehet lekérdezni.

```
> eps
```

Alapesetben az értéke $\approx 2 \cdot 10^{-16}$. Adjunk hozzá 10^{-17}-t (ami kevesebb, mint a gépi epszilon) 2-höz!

```
> a = 2
> b = 2 + 2·10^-17 % (gépi epszilonnál kevesebb)
> a == b  % -> 1 (Igaz)
```

Adjunk most hozzá 10^{-14}-t (ami nagyobb, mint a gépi epszilon) két különböző számhoz, az első legyen 1, a második 12345.

```
> a = 1; b = 12345; c = 1e-14;
> a == a+c % 0 (hamis)
> b == b+c % 1 (igaz)
```

Miért lehet az, hogy az egyik esetben sikerült hozzáadni ugyanazt a gépi epszilonnál nagyobb értéket a számhoz, a másik esetben pedig nem? A válasz erre az, hogy a gépi pontosság alapértéke az 1-től való távolság, de értéke a szám nagyságrendjével változik. Nézzük meg!

```
> eps(a) % 2.2204e-16
> eps(b) % 1.8190e-12
> eps(1e20) % 16384
```

Ezt azt jelenti, hogy 10^{20} után tárolható legkisebb szám 16384-gyel nagyobb. Ha 16384/2-t vagy annál kisebb számot adunk hozzá 10^{20}-hoz, akkor a szám értéke nem fog változni, ha ennél nagyobbat, akkor már igen (a kerekítés miatt, ha 16384/2+1-et adunk hozzá, akkor már felfelé fog kerekíteni).

```
> 1e20 == 1e20+16384/2 % a válasz igaz
```

KIOLTÓ HIBA

Nézzünk egy másik tipikus kerekítési hibát, a kioltó hibát, amikor közel azonos nagyságú számokat vonunk ki egymásból. Nézzük meg Matlab-ban a következő példát:

```
> x1 = 10.000000000000004 % 14 db 0 a tizedespont után a 4-es előtt
> y1 = 10.00000000000004 % 13 db 0 a tizedespont után a 4-es előtt
> % vagy: x1 = 4e-15 + 10; y1 = 4e-14 + 10;
> (y1-10)/(x1-10)
```

A várt eredmény: 0.000000000000004 / 0.00000000000004 = 10, a kapott eredmény azonban: 11.5!

Fontos ismerni még a legkisebb és legnagyobb tárolható számok méretét is, mert ennek a túllépése is komoly hibákhoz vezethet!

```
> realmin % 2.2251e-308
> realmax % 1.7977e+308
```

Miért fontos nekünk a numerikus hibák ismerete? Érdemes néhány tanulságos esetet megnézni a numerikus hibák okozta katasztrófák között! Az Öbölháború idejében 1991-ben egy Patriot légvédelmi rakéta nem találta el az iraki Scud rakétát, ami miatt 28-an meghaltak. Az oka numerikus hiba volt. A tizedmásodpercben mért időket a rendszer megszorozta 1/10-del, hogy másodpercet kapjon, mivel ezt nem lehetett pontosan reprezentálni binárisan, fellépett egy kis kerekítési hiba, sokszor egymás után elvégezve a műveletet a hiba halmozódott. 100 órás üzem esetén az eltérés: 0.34 másodperc volt, ami alatt egy Scud rakéta több, mint fél kilométert tesz meg (1.676 m/s sebességgel).

Ugyancsak numerikus hiba okozta 1996-ban az ESA Ariane 5 rakétájának 40 másodperccel a kilövése után történő felrobbanását! („Famous number computing errors” - https://blog.penjee.com/famous-number-computing-errors/, „Disasters due to rounding error” - https://web.ma.utexas.edu/users/arbogast/misc/disasters.html). Érdekes lehet még a következő oldal is: https://www5.in.tum.de/persons/huckle/bugse.html, ahol egy nagyobb gyűjtemény található szoftverhibák okozta problémákról (sok köztük a numerikus számítási probléma).

CSONKÍTÁSI HIBA

A véges számábrázolásból adódó kerekítési hiba hatását már láttuk, és azt is, hogy kerülni kell a közel azonos nagyságú számok kivonását, ha van rá mód.

Egy másik fontos hiba akkor lép fel, amikor az egzakt matematikai kifejezés helyett annak közelítését alkalmazzuk a numerikus számítások során, például: Taylor-soros közelítés, numerikus deriválás, integrálás. Általában akkor lép fel a probléma, amikor egy bonyolult, szimbolikusan nem megoldható problémát, egyszerűbb, számítógéppel könnyebben kezelhető problémával helyettesítjük.

Jó példa erre a Taylor-soros közelítés, ahol minél több tagot veszünk figyelembe, értelemszerűen annál kisebb lesz a csonkítási hiba. Most nézzük meg az e^x közelítését Taylor-sorral 4 taggal: $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!}$

```
> x=1
> f = exp(x) % 2.7183 - tényleges érték
> g = 1 + x + x^2/2 + x^3/6 % 2.6667 - csonkítási hibával terhelt érték
```

A kerekítési és a csonkítási hiba együtt adja a teljes hibát!

kerekítési hiba + csonkítási hiba = teljes hiba

ABSZOLÚT ÉS RELATÍV HIBA

A hibákat többféleképpen csoportosíthatjuk, a pontos értéktől való tényleges eltérést abszolút hibának nevezzük, azonban a valóságban sokszor nem ezzel lehet a legjobban reprezentálni a hiba nagyságát, célszerű bevezetni a relatív hiba fogalmát is. Legyen egy x valós szám közelítése $\tilde{x}$.

Abszolút hibának nevezzük a tényleges eltérést: $\Delta = |x - \tilde{x}|$

A relatív hiba az abszolút hiba osztva x értékével: $\varepsilon = \frac{|x - \tilde{x}|}{|x|}$

Amikor x egy körüli érték, akkor nincs lényeges eltérés a kettő között, azonban amikor x>>1, a relatív hiba jobban tükrözi a hiba jelentőségét. Nézzünk erre egy példát! Legyen két távolság (t_1, t_2), amit becslünk ($\tilde{t_1}, \tilde{t_1}$).

$t_1 = 1000\,m;$ $\tilde{t_1} = 900\,m;$ $\Delta = 100\,m;$ $\varepsilon = 10\,\%;$	$t_2 = 200\,m;$ $\tilde{t_1} = 100\,m;$ $\Delta = 100\,m;$ $\varepsilon = 50\,\%;$

Az abszolút hiba mindkét esetben 100 m, de az első becslés mégis sokkal pontosabb, mint a második és ez a relatív hibák nagyságában is tükröződik (10% ill. 50%).

STABILITÁS, KONDÍCIÓSZÁM

Miután láttuk, hogy a numerikus hibákkal együtt kell élnünk a számítások során, a következő fontos kérdés, hogyan kezeljük őket? Megbízhatunk-e a kapott eredményben? Ehhez még két fogalmat kell megismernünk, az adott probléma érzékenységét/kondícionáltságát és az algoritmus stabilitását.

- Egy matematikai probléma jól kondicionált ha a bemeneti paraméterek kis változására az eredmény is kis mértékben változik.
- Egy algoritmus numerikusan stabil ha bemeneti paraméterek kis változására az eredmény is kis mértékben változik.

A pontosság függ a probléma kondicionáltságától és az algoritmus stabilitásától.

Pontatlanságot okozhat, ha stabil algoritmust alkalmazunk rosszul kondicionált problémára, vagy instabil algoritmust alkalmazunk jól kondicionált problémára.

KONDÍCIÓSZÁM

Oldjuk meg a következő lineáris egyenletrendszert!

$$6x_1 - 2x_2 = 10$$

$$11.5x_1 - 3.85x_2 = 17$$

Az egyenletrendszer mátrixos alakban ($A \cdot x = b$):

$$A = \begin{pmatrix} 6 & -2 \\ 11.5 & -3.85 \end{pmatrix}; \quad b = \begin{pmatrix} 10 \\ 17 \end{pmatrix}$$

Oldjuk meg ezt a feladatot Matlab-ban. A megoldás matematikailag: $x = A^{-1} \cdot b$. Matlab-ban az **inv** parancs számítja egy mátrix inverzét. Oldjuk meg a feladatot!

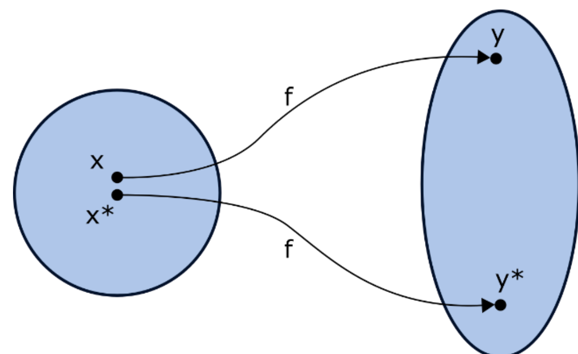
```
> A = [6,-2; 11.5,-3.85]; b = [10; 17];
> x = inv(A)*b % 45.0000; 130.0000
```

Megoldás a 45 és a 130 lett. Módosítsuk egy kicsit a második egyenletben az x_2 együtthatóját -3.85-ről -3.84-re, és oldjuk meg újra a feladatot!

```
> A = [6,-2; 11.5,-3.84]
> x = inv(A)*b % 110.0000; 325.0000
```

Most a megoldás 110 és 325 lett. Csak egy kicsit változtattunk az egyenletrendszeren, mégis óriási lett a változás a végeredményben! Azt vártuk volna, hogy mivel a két bemenet nagyon közel van egymáshoz a megoldások is hasonlóak lesznek. Nem így történt. Mi okozhatja ezt?

Vannak olyan mátrixok, amelyek nagyon érzékenyek a bemenet kis megváltozására. Ezt az érzékenységet lehet mérni a mátrix kondíciószámával. A kondíciószám (κ) teremt kapcsolatot a kimenet relatív hibája és a bemenet relatív hibája között. Minél nagyobb ez a szám, a bemenet kis megváltozása annál nagyobb változást fog okozni a kimenetet illetően.



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

Nézzük meg ennek a mátrixnak a kondíció számát!

```
> cond(A) % 4.6749e+03
```

Eredménye: 4674.9 lett. Minél nagyobb ez a szám, annál bizonytalanabb a megoldás, ugyanis a bemenet kis hibája ugyanennyi szeresére nagyítódik föl a kimenetnél! Erre nagyon oda kell figyelni a mérnöki munkák során, ahol a bemeneti mérések, állandók többnyire csak közelítő értékek, hibákkal terheltek.

PÉLDA INSTABIL ALGORITMUSRA

Nézzük meg a következő másodfokú egyenlet megoldását!

$$x^2 - 100.0001\, x + 0.01 = 0$$

Ennek az egzakt megoldása a következő: $x_1 = 100; x_2 = 0.0001;$ Megoldás felírható a másodfokú megoldóképlettel:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}; \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a};$$

Oldjuk meg Matlab-ban, az eredmények megjelenítését állítsuk át több tizedes jegyre!

```
> format long;
> a = 1; b = -100.0001; c = 0.01;
> D = sqrt(b^2 - 4*a*c) % 99.999899999999997
> x1 = (-b + D)/(2*a) % 100
> x2 = (-b - D)/(2*a) % 1.000000000033197e-04
```

x_2-re nem kaptunk pontos eredményt a kerekítési hiba miatt. Miután *b* negatív, így ebben az esetben a számlálóban két egymáshoz nagyon közeli értéket kellett kivonni egymásból, itt is fellépett a kioltó hiba.

Sok esetben, amikor a matematikai kifejezés két egymással közel megegyező kifejezés különbségét tartalmazza, a feladat átalakítható olyan formába is, ami kevésbé érzékeny a kerekítési hibákra. x_2 esetében ezt megtehetjük, ha megszorozzuk a formulát $(-b + \sqrt{b^2 - 4ac})/(-b + \sqrt{b^2 - 4ac})$-vel:

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \cdot \frac{-b + \sqrt{b^2 - 4ac}}{-b + \sqrt{b^2 - 4ac}} = \frac{2c}{-b + \sqrt{b^2 - 4ac}};$$

Használjuk most ez utóbbi kifejezést a megoldáshoz:

```
> x2m = (2*c)/(-b+D) % 1.000000000000000e-04
```

Most a várt eredményt kaptuk.

PÉLDA STABIL ALGORITMUSRA

Nézzünk egy másik példát az algoritmus megválasztásának fontosságára! Kétféle módon is közelíthetjük az e^{-x} értékét:

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots$$

$$e^{-x} = \frac{1}{e^x} = \frac{1}{1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots}$$

Számoljuk ki a függvény értékét x = 8.3 helyen a fent megadott két módszerrel! Töltsük be az emx.m függvényt, ami két kimenettel megadja a kétféle közelítést n tag esetében, x helyen!

```
> function [f g] = emx(x,n)
>     f = 1; % első közelítés 1 - x + x^2/2 - x^3/6 + ...
>     p = 1; % első közelítés a nevezőre (1 + x + x^2/2 + x^3/6 + ...)
>     for i=1:n
>         s = x^i/factorial(i);
>         f = f +(-1)^i*s;
>         p = p + s;
>         g = 1 / p;
>     end
> end
```

A megoldás pontos értéke: exp(-8.3) = 2.4852e-04

Próbáljuk ki az előbbi függvényt n = 10, 20, 30 esetben! Álljunk vissza a kevesebb számjegy kijelzésére, most elég lesz ez is.

```
> format short
> megoldas = exp(-8.3) % Pontos érték: 2.4852e-04
> [f g] = emx(8.3,10) % 10 tag esetében: f=188.0344,   g=3.1657e-04
> [f g] = emx(8.3,20) % 20 tag esetében: f=0.2833,     g=2.4856e-04
> [f g] = emx(8.3,30) % 30 tag esetében: f=2.5151e-04, g=2.4852e-04
```

Az eredmények:

n=	10	20	30
f	188.0344	0.2833	2.5151e-04
g	3.1657e-04	2.4856e-04	2.4852e-04

Látjuk, hogy a két algoritmus közül a második sokkal hamarabb közelíti meg a pontos értéket, nem mindegy milyen módszerrel oldjuk meg a feladatot!

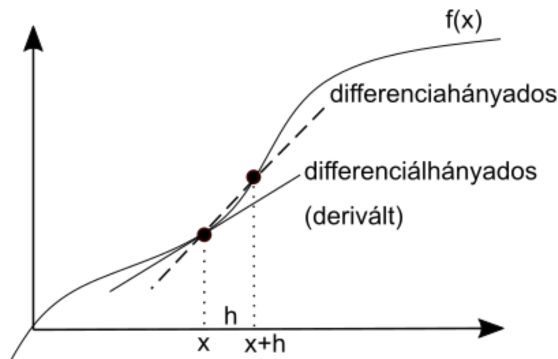
TELJES HIBA[2]

Nézzünk meg egy olyan példát, ahol mind a csonkítási, mind a kerekítési hiba szerephez jut. Nem csak a Taylor soros közelítés tartalmaz csonkítási hibát, hanem a numerikus integrálás is, vagy a derivált differenciahányadossal való közelítése is. Nézzünk ez utóbbira egy példát:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

[2] Kiegészítő anyag

A csonkítási hiba felső korlátja becsülhető. Határozzuk meg a csonkítási hiba felső korlátját az alábbi függvény deriváltjának a differenciahányadossal történő közelítése esetén, x=2 helyen[3]!



$$y = x^3$$

Közelítsük numerikusan a deriváltat a differenciahányadossal!

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} = \frac{(2+h)^3 - 2^3}{h}$$

A csonkítási hibát a Taylor-soros közelítés alapján becsülhetjük:

$$f(x+h) = f(x) + f'(x) \cdot h + \frac{f''(\xi)}{2} \cdot h^2$$

, ahol $x < \xi < x + h$. Ezért átrendezve igaz a következő:

$$\left| \frac{f(x+h) - f(x)}{h} - f'(x) \right| \leq \frac{f''(\xi)}{2} \cdot h$$

A fenti egyenlet bal oldalán a csonkítási hibát találjuk (a közelítés és a tényleges érték eltérése), a jobb oldalon pedig ehhez egy felső korlátot, aminél biztosan kisebb lesz a hiba. Jelöljük H-val ezt a hibát, a felső korlátot pedig ε-nal. Esetünkben pontosan ismerjük az első ($y' = 3\,x^2 = 12$) és a második deriváltat is ($y'' = 6x$), így a csonkítási hibára igaz a következő:

$$H(h) = \left| \frac{(2+h)^3 - 2^3}{h} - 12 \right| \leq \frac{f''(\xi)}{2} \cdot h = \frac{6\,\xi\,h}{2} = 3\,\xi\,h$$

A csonkítási hiba becsült felső korlátja h függvényében maximális ξ esetén ($\xi = x + h$), $x = 2$ helyen:

$$\varepsilon(h) = 3\,(2+h)\,h$$

A fenti képletből látható, amint az várható is volt, hogy minél kisebb h intervallumokra osztjuk fel a függvényt a differenciahányadosok számításához, annál kisebb lesz a csonkítási hiba.

Ábrázoljuk az előző derivált közelítésének becsült felső korlátját és számítsuk ki a tényleges hibát is a lépésköz változásának függvényében! Vegyük fel a következő lépésközöket:

$$h = 1, 10^{-1}, 10^{-2}, \cdots, 10^{-15}$$

```
> format long
> n = 0:-1:-15, h = 10.^n
```

Adjuk meg egysoros függvényként a becsült felső korlátot és a tényleges hibát is! (Figyeljünk a . használatára, mivel most vektorok elemenkénti műveletéről van szó)

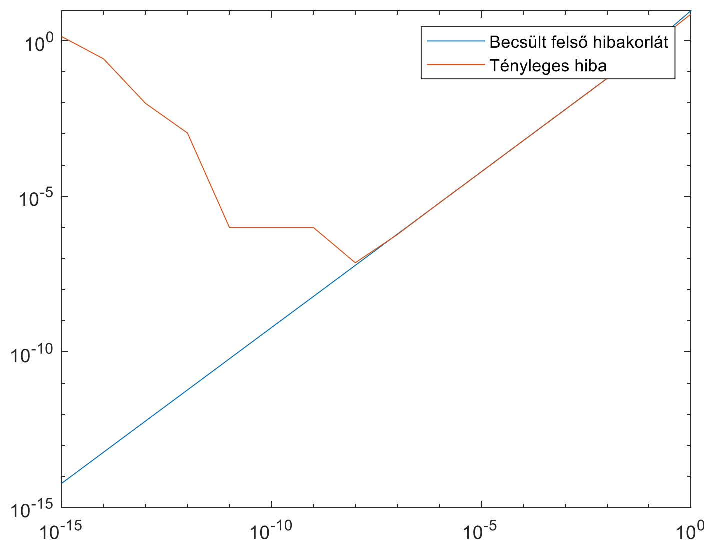
```
> e = @(h) 3*(2 + h).*h; % becsült felső korlát
```

[3] Paláncz Béla numerikus módszerek példatára alapján

```
> H = @(h) abs(((2 + h).^3 - 8)./h - 12) % tényleges hiba
```

Számoljuk ki a két függvény értékét a különböző lépésközökhöz és ábrázoljuk az eredményeket log-log koordináta rendszerben (ezt a **loglog** paranccsal tehetjük meg)!

```
> figure(1)
> loglog(h,e(h)); hold on
> loglog(h,H(h))
> legend('Becsült felső hibakorlát','Tényleges hiba')
```



Mit látunk a fenti ábrán? Ahogy csökken a lépésköz, úgy csökken a csonkítási hiba becsült felső korlátja is és egy darabig a tényleges hiba is ennek megfelelően, azonban egy pont után (valahol 10^{-8} környékén) hirtelen elkezd nőni újra a tényleges hiba. Mi lehet ennek az oka? Ez abból adódik, hogy a teljes hiba a csonkítási és a kerekítési hiba összegeként áll elő. Nagyjából 10^{-8} értékig a csonkítási hiba dominál, utána viszont erőteljesen elkezd nőni a kerekítési hiba. Ennek a jelenségnek igen fontos szerepe van például a differenciálegyenletek numerikus megoldásánál!

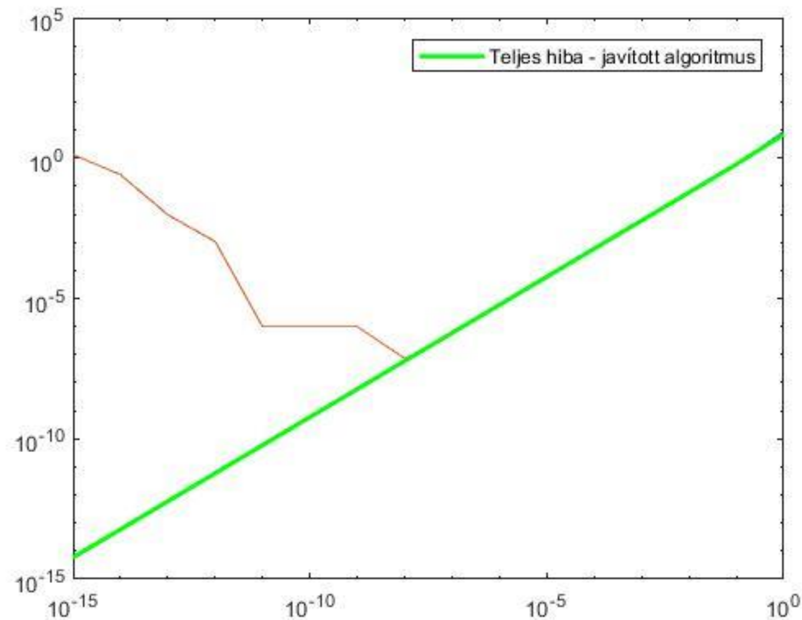
Természetesen hasonlóan a korábban látottaknál itt is lehet olyan algoritmust választani, ami kevésbé érzékeny a kerekítési hibára, hiszen a fő hiba itt is a kioltó hiba, mivel a differenciahányados számlálójában két közel azonos számot vonunk ki egymásból, minél kisebb *h*, annál kisebb a két szám közötti eltérés. Most a differenciahányadost könnyen egyszerűsíthetjük:

$$\frac{(2+h)^3 - 2^3}{h} = 12 + 6\,h + h^2$$

Ebből a csonkítási hiba:

$$H(h) = |12 + 6\,h + h^2 - 12| = |6\,h + h^2|$$

```
> H2 = @(h) abs(6*h + h.^2)
> abr1=loglog(h,H2(h),'g','LineWidth',2)
> legend(abr1,'Teljes hiba - javított algoritmus')
```



A differenciahányadost itt mi egyszerűsítettük manuálisan, természetesen lehetőség van ilyen egyszerűsítésekre Matlab-ban is, ezek azonban már a szimbolikus számítások témakörébe tartoznak, ami a Symbolic Math Toolbox része (Octave-ban ehhez telepíteni kell a symbolic package-t, lásd az 1. gyakorlat Kiegészítés Octave-hoz fejezetét).

Szimbolikus számítások során nem konkrét számokkal számolunk, hanem változókkal. Ehhez először meg kell adni a Matlab számára **syms** paranccsal, hogy melyek lesznek a szimbolikus változóink, ezekkel a szimbolikus változókkal meg kell hívjuk a függvényt, ekkor egyszerűsíthetjük a kifejezést a **simplify** paranccsal. A **simplify** parancs eredménye egy szimbolikus változó lesz, amibe nem lehet konkrét számokat behelyettesíteni, csak, ha visszaalakítjuk hagyományos függvénnyé a kifejezést a **matlabFunction** paranccsal.

```
> syms h
> Hsym = simplify(H(h)) % Hsym = abs(h*(h + 6))
> H2 = matlabFunction(Hsym) % H2 = @(h)abs(h.*(h+6.0))
```

A Workspace-ben is láthatjuk, hogy Hsym értéke szimbolikus. Nézzük meg mi történik, ha megpróbálunk egy konkrét értéket behelyettesíteni!

```
> H2(1e-1) % 0.610000000000000
> Hsym(1e-1)
Subscript indices must either be real positive integers or logicals.

Error in sym/subsref (line 841)
            R_tilde = builtin('subsref',L_tilde,Idx);

Error in gyak3 (line 78)
Hsym(1e-1)
```

A második esetben hibaüzenetet kapunk, mivel szimbolikus kifejezésbe nem lehet behelyettesíteni egy értéket.

A FEJEZETBEN HASZNÁLT ÚJ FÜGGVÉNYEK

eps	-	Gépi epszilon/gépi pontosság nagysága,
realmin	-	Az ábrázolható legkisebb szám (double típus esetében)
realmax	-	Az ábrázolható legnagyobb szám (double típus esetében)
factorial	-	Faktoriális, n!
inv	-	Mátrix inverze
cond	-	Kondíció szám
loglog	-	Ábrázolás logaritmikus skálán (mindkét tengelyen)
syms	-	Szimbolikus változók, kifejezések definiálása
simplify	-	Szimbolikus kifejezések egyszerűsítése
matlabFunction	-	Szimbolikus kifejezések függvénnyé alakítása