7. fejezet

Véletlenszám-generálás, időmérés

Haladó Alkalmazott Programozás
ELTE, 2019

Berényi Dániel – Nagy-Egri Máté Ferenc

Véletlenszámok C++-ban

Véletlen számok generálása

C++11-ben egy nagyon modern és jól használható könyvtár került a Standard Library-be, amely több féle generátort és eloszlást ismer.

<random>

Véletlen számok generálása

Generátor motorok (engine-k) :

A generátorok itt szűkebb értelemben azok a rutinok, amelyek egy adott, determinisztikus módszer szerint pszeudo-véletlenszámokat állítanak elő egyenletes eloszlással.

Jellemzőik:

- Sebesség
- Visszatérési „idő” (mennyi szám után kezd ismétlődni a sorozat)
- Hány dimenzióban korrelálatlanok a generált számok

Véletlen számok generálása

Generátor motorok (engine-k) :

A C++11-ben bevezetett engine-k:

- Lineáris kongruencia generátor
- Mersenne Twister
- Késletetett Fibonacci

Az első kettő összehasonlítva:

A Lineáris kongruencia generátor kicsi, gyors, de csak néhány dimenizóban korrelálatlan, a Mersenne Twister nagy, lassabb, de sok dimenzióban korrelálatlan, és nagyon nagy a visszatérési ideje. Fizikai szimulációhoz az utóbbit ajánljuk, grafikai, vagy játék alkalmazásokhoz elég lehet az előbbi.

Véletlen számok generálása

Generátor motorok (engine-k) :

A C++11-ben bevezetett engine-k:

- Lineáris kongruencia generátor
- Mersenne Twister
- Késletetett Fibonacci

Az első kettő összehasonlítva:

A Lineáris kongruencia generátor kicsi, gyors, de csak néhány dimenizóban korrelálatlan, a Mersenne Twister nagy, lassabb, de sok dimenzióban korrelálatlan, és nagyon nagy a visszatérési ideje. Fizikai szimulációhoz az utóbbit ajánljuk, grafikai, vagy játék alkalmazásokhoz elég lehet az előbbi.

Véletlen számok generálása

Néhány előre beállított generátor:

std::minstd_rand0 Lineáris Kongruencia generátorok
std::minstd_rand

std::mt19937 32 és 64 bites Mersenne Twister
std::mt19937_64

std::default_random_engine alapértelmezett engine

Véletlen számok generálása

Generátor motorok (engine-k) :

Közös metódusok:

- .seed() megadhatunk egy értéket, amiből beállítja a generátor belső állapotát. Azonos értékekből azonos számok fognak jönni azonos sorrendben.
- operator() generál egy új számot
- min(), max() a legnagyobb és legkisebb érték, ami a generátorból kijöhet.

Véletlen számok generálása

Ha mindenképpen szeretnénk véletlen szerű seed-et választani, arra van egy std::random_device objektum.

Ez lehet, hogy nem determinisztikus, de kevés szám előállítására van kitalálva, és általában csak a seed-elésre használják.

Ennek is a generátorokhoz hasonló tagfüggvényei vannak.

Véletlen számok generálása

Eloszlások:

A generátorokból egyenletes (uniform) számok jönnek ki, adott tartományban, nekünk azonban másféle tartományok és eloszlások is kellenek.

Véletlen számok generálása

Eloszlások:

C++-ban tudunk választani:

- Egyenletes eloszlású egész és lebegőpontos számok közül tetszőleges intervallumban
- Továbbá Bernoulli, Exponenciális, Poisson, Gamma, Normál, Lognormál, és még sok más eloszlás közül.

Véletlen számok generálása

Példa 10 db normál eloszlású double generálása:

```
#include <iostream>
#include <random>
int main()
{
  std::random_device rd{};
  std::mt19937 gen(rd());
  std::normal_distribution<double> distr(100,20);

  for(int i=0; i<10; ++i)
  {
    std::cout << distr(gen) << '\n’;
  }
}
```

Véletlen számok generálása

Példa 10 db normál eloszlású double generálása:

```
#include <iostream>
#include <random>
int main()
{
  std::random_device rd{};
  std::mt19937 gen(rd());
  std::normal_distribution<double> distr(100,20);

  for(int i=0; i<10; ++i)
  {
    std::cout << distr(gen) << '\n’;
  }
}
```

Itt adjuk meg a seed-et, ez most a random device-ből jön különböző gépen, különböző eredményre vezethet!

Véletlen számok generálása

Példa 10 db normál eloszlású double generálása:

```
#include <iostream>
#include <random>
int main()
{
  //std::random_device rd{};
  std::mt19937 gen(42);
  std::normal_distribution<double> distr(100,20);

  for(int i=0; i<10; ++i)
  {
    std::cout << distr(gen) << '\n’;
  }
}
```

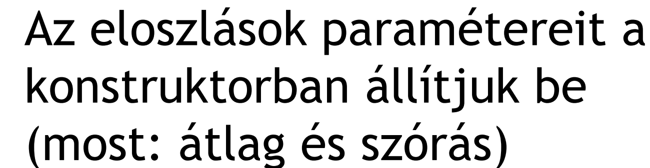
Ha fix értéket írunk be, minden gépen ugyan azokat a számokat kell kapnunk!

Véletlen számok generálása

Példa 10 db normál eloszlású double generálása:

```
#include <iostream>
#include <random>
int main()
{
  //std::random_device rd{};
  std::mt19937 gen(42);
  std::normal_distribution<double> distr(100,20);

  for(int i=0; i<10; ++i)
  {
    std::cout << distr(gen) << '\n’;
  }
}
```



Berényi Dániel – Nagy-Egri Máté Ferenc

Véletlen számok generálása

Wigner GPU Lab

Egy std::vector feltöltése véletlen számokkal:

```
std::vector<double> data(1500);
std::generate(data.begin(),
              data.end(),
              [&]{ return distr(gen); });
```

Időmérés C++-ban

Időmérés

C++11-ben bekerült a <chrono> könyvtár is a Standard Library-be, így időt is lehet, akár nagy pontossággal is mérni.

Ez például akkor hasznos, amikor az ember szeretné össze akarja hasonlítani két implementáció sebességét.

Időmérés

A <chrono> könyvtár alap eleme az óra, ebből több is van:

- std::chrono::system_clock
  - Csak ezt lehet valódi idővé, dátummá konvertálni, de ez bármikor változhat (nem feltétlen monoton növekvő)
- std::chrono::steady_clock
  - Ez garantált, hogy állandóan növekszik
- std::chrono::high_resolution_clock
  - Ez vagy valamelyik az előzőek közül, vagy egy harmadik, pontosabb

Időmérés

Példa a használatra:

```
#include <chrono>
auto t1 = std::chrono::high_resolution_clock::now();
f();
auto t2 = std::chrono::high_resolution_clock::now();

//egész értékként:
long long n =
std::chrono::duration_cast<std::chrono::nanoseconds>(t2-t1).count();

//lebegőpontos számként
double x =
(static_cast<std::chrono::duration<double, std::milli>>(t2-t1)).count();
```

Időmérés

A mért idők értelmezése azonban trükkös feladat...

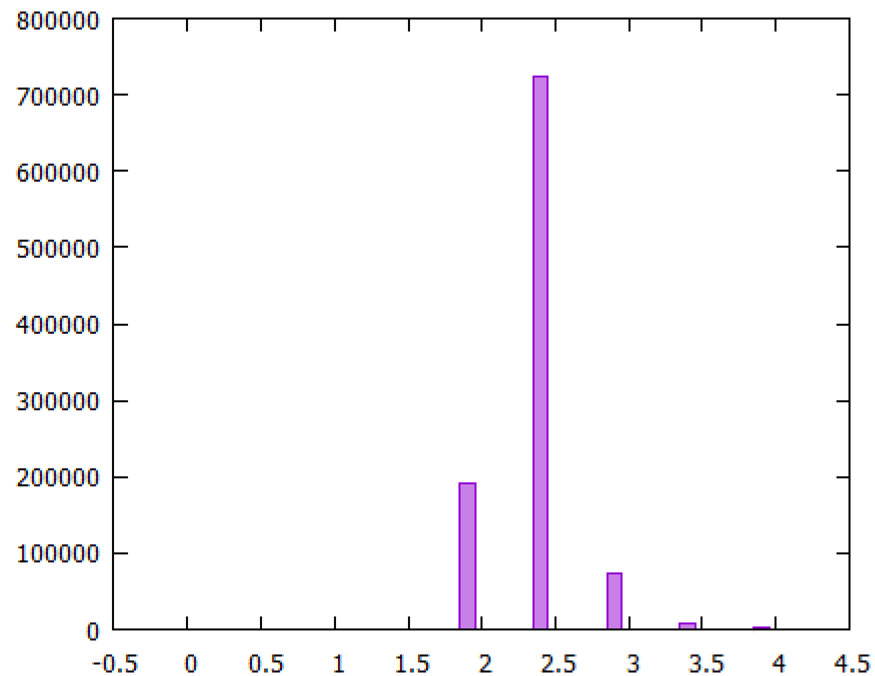
- A rendszer hívások idejei sok tényezőtől függhetnek, például a rendszer terheltségétől
- Az időmérés maga is egy rendszer hívás

- A szálkezelő félre teheti a szálat, de a mért időket nem korrigálja
- A szálkezelő átteheti egyik magról a másikra a szálat
  a mért időkben kitör a káosz

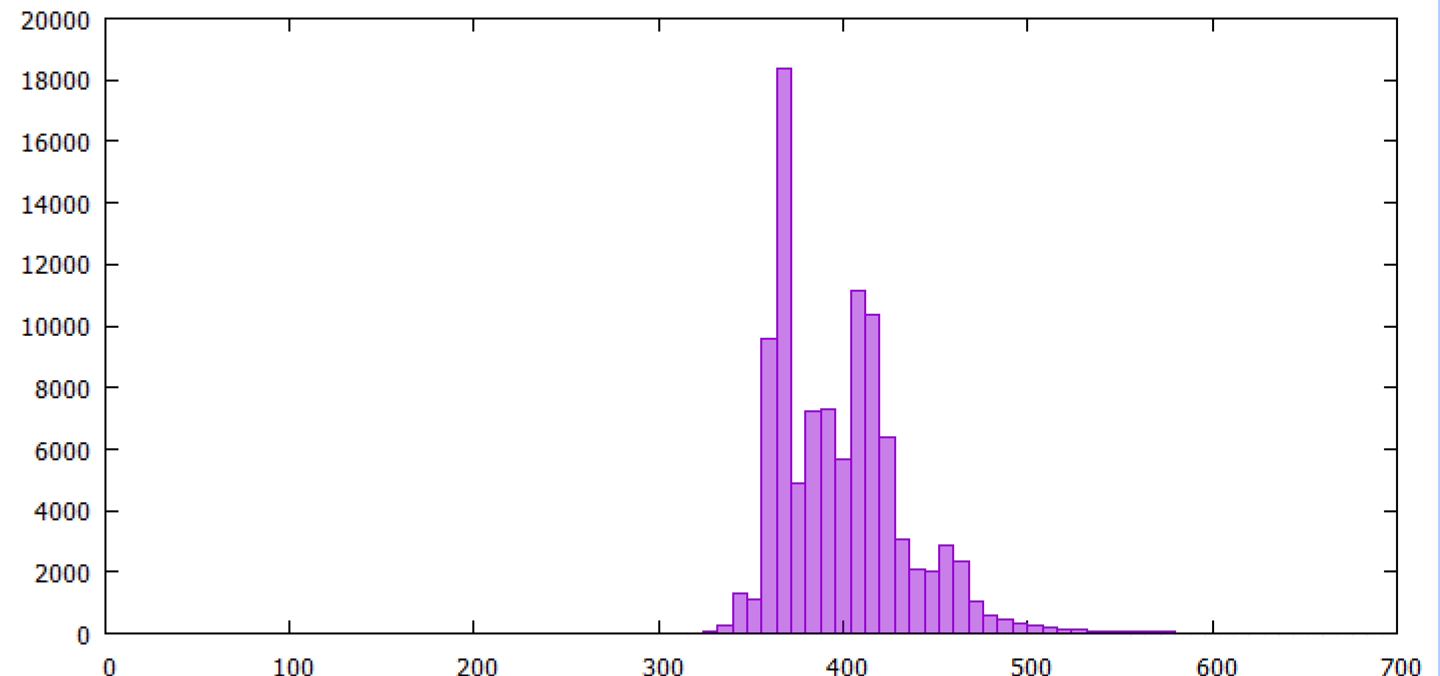
- Maga a hardveres óra is csinálhat fura dolgokat, főleg többmagos rendszereken

Időmérés

A mért idők eloszlása közel sem triviális:



Trigonometrikus összeg (50 tag) [μs]



Memória allokáció (300 kB) [μs]

Időmérés

Tehát:

- Érdemes többször (10x-100x) lemérni a kódot
  (lehetőleg terheletlen rendszeren)

- Ha összehasonlítást akarunk tenni, akkor venni a minimumot:
  - mert a hardveres és szoftveres megszakítások ideális esetben csak növelik a futás időt

- Ezért a minimum robosztusabb, és jobban jellemzi a kód hatékonyságát, mint az átlag.

Időmérés

Wigner GPU Lab

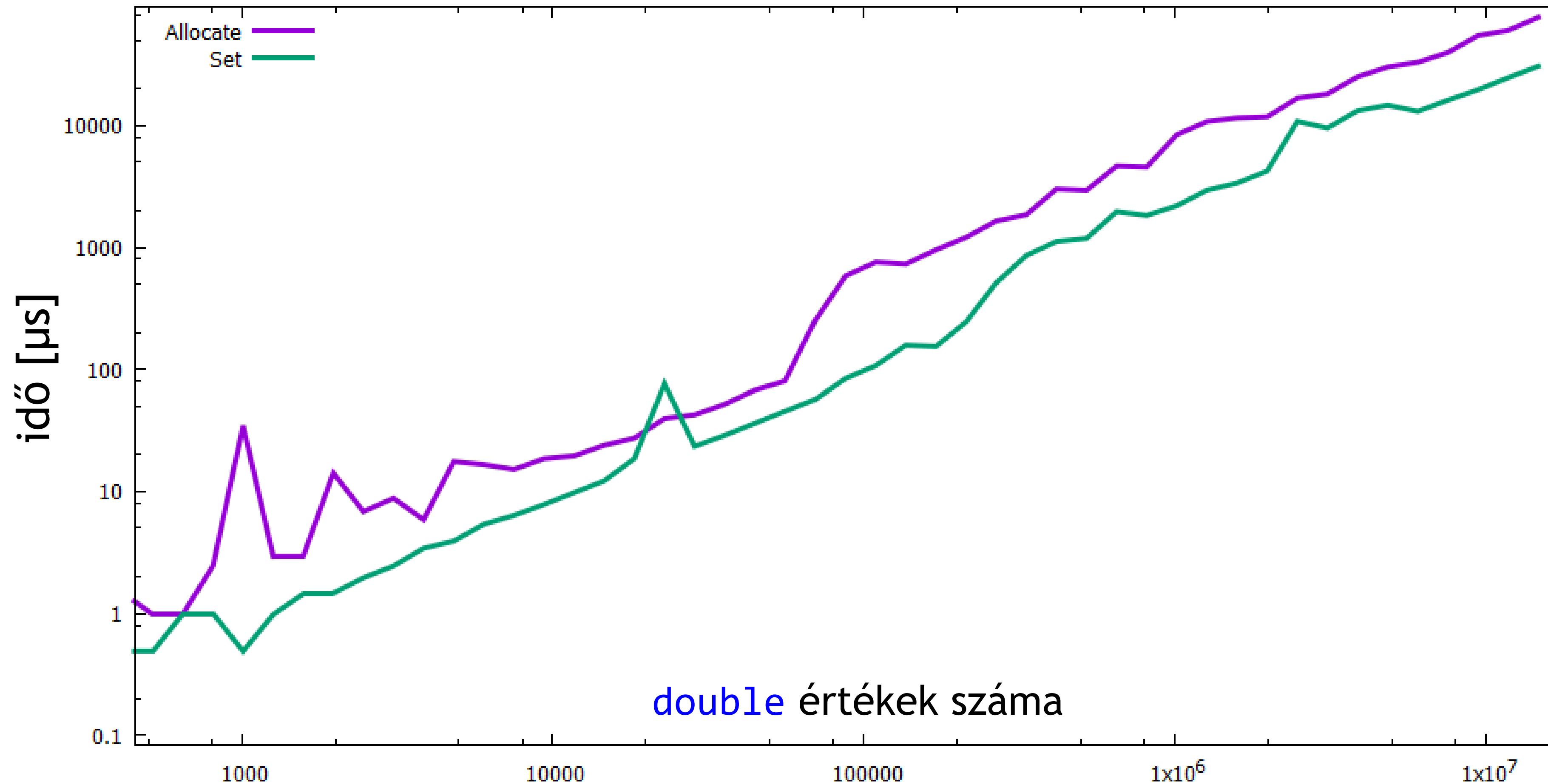
Példakód: Melyik tart tovább?

- Lefoglalni sok `double`-et
- Vagy kinullázni őket?

[time_measurement.cpp]

Időmérés

time_measurement.cpp



Debug vs Release

A legtöbb fordítási esetben megkülönböztetünk két fordítási modellt:

- Debug:

  A fordító extra ellenőrzéseket helyez el a kódban, amivel a memória kezeléssel és címzésekkel kapcsolatos hibákat könnyebben lehet észlelni, és további információkat, hogy a változók értékeit-neveit elérhetővé tegye számunkra.

  Ezek miatt a Debug kód lassabb, de ebben lehet hibákat keresni.

Debug vs Release

A legtöbb fordítási esetben megkülönböztetünk két fordítási modellt:

- Release:

  A fordító minden Debug információt és ellenőrzést kidob, bekapcsolja az optimalizációkat, amik teljesen átrendezhetik a kódot, függvények, változók tűnhetnek el, stb.

  Ezt a kódot nem lehet könnyen debuggolni, illetve sok szimbólum hiányozni fog belőle, viszont sokkal gyorsabb lesz.

Debug vs Release

A legtöbb fordítási esetben megkülönböztetünk két fordítási modellt:

- Release:

  A fordító minden Debug információt és ellenőrzést kidob, bekapcsolja az optimalizációkat, amik teljesen átrendezhetik a kódot, függvények, változók tűnhetnek el, stb.

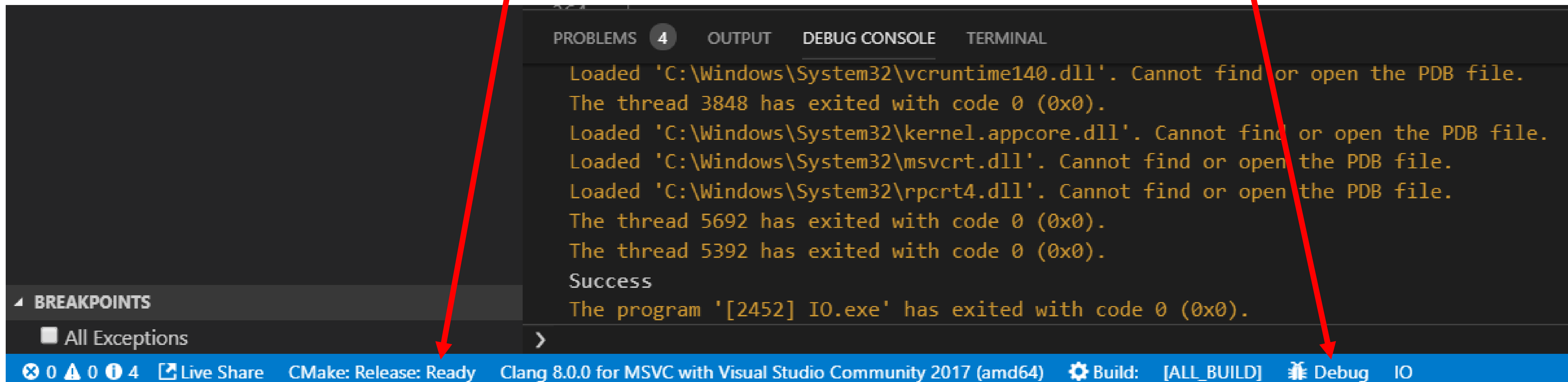
  Ezt a kódot nem lehet könnyen debuggolni, illetve sok szimbólum hiányozni fog belőle, viszont sokkal gyorsabb lesz.

  Ha memóriahibás a programunk, akkor nagyon gyakran előfordul, hogy Debugban látszólag működik, Release-ben meg szétfagy, vagy teljesen értelmetlen dolgokat számol…

Debug vs Release

A két konfiguráció között Visual Studio Codeban ide kattintva válthatunk

Futtatni továbbra is a bogár ikonnal lehet, csak ekkor valószínűleg nem áll meg a break pointoknál…



Házi feladat

Házi feladat:

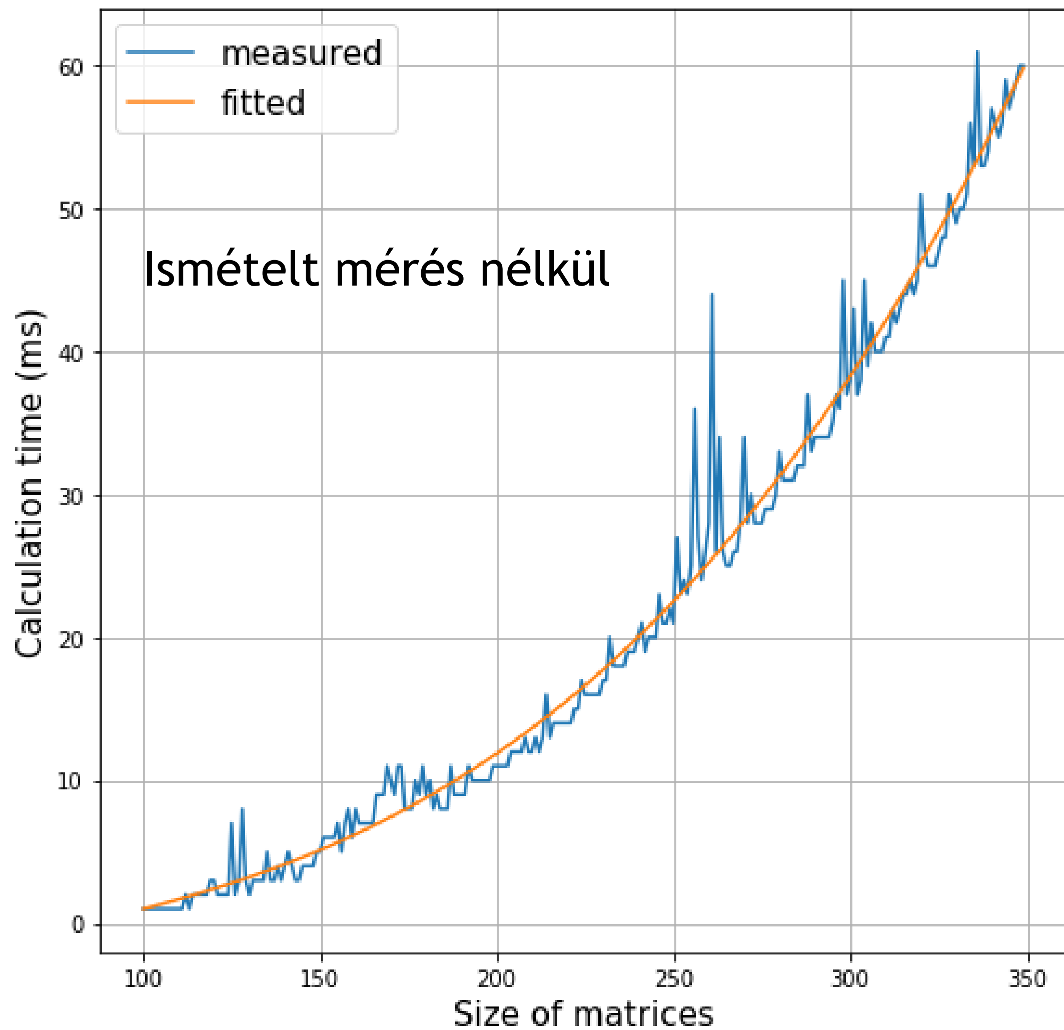
Töltsünk fel a korábban írt mátrix objektumunkból kettőt véletlen számokkal és mérjük le, mennyi idő összeszorozni őket különböző N-ek esetén.

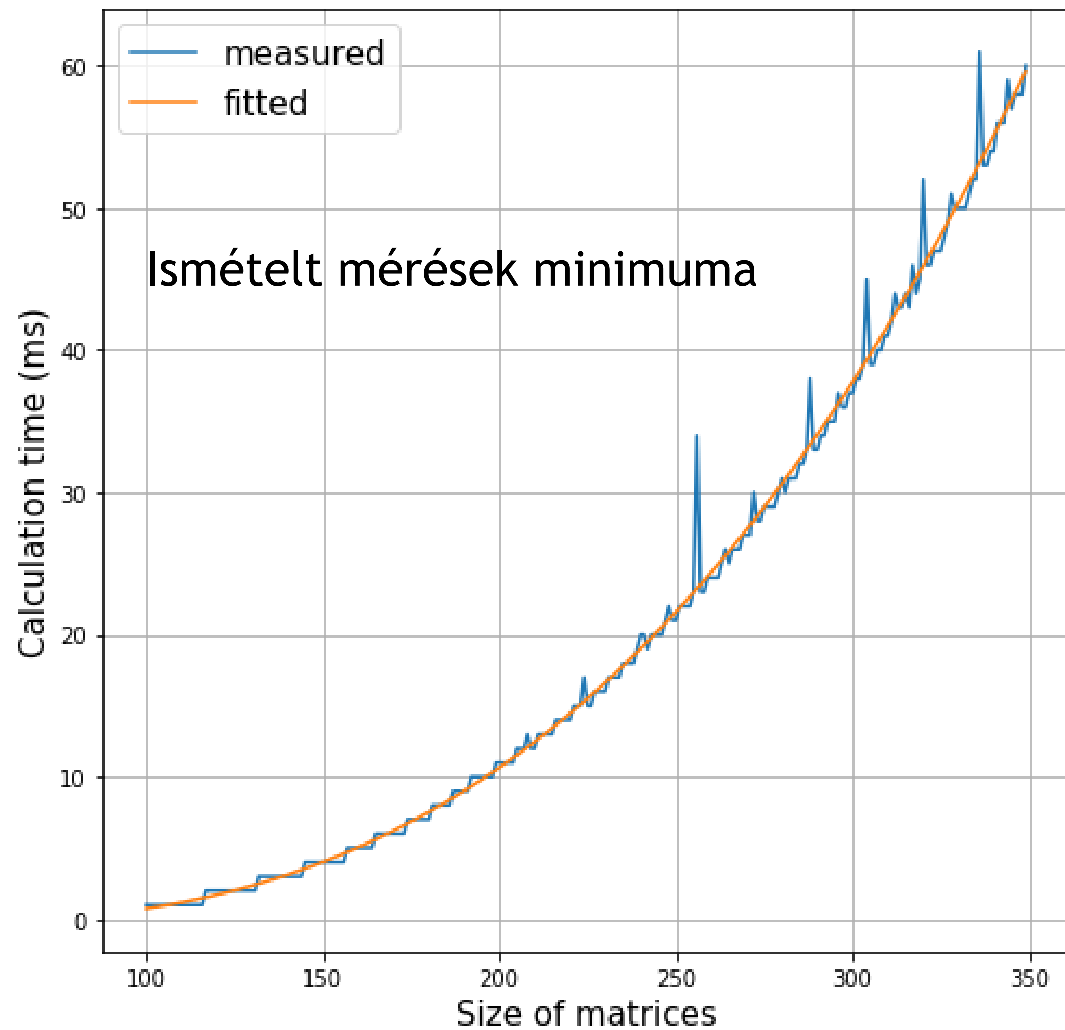
Próbáljuk meg kimérni az N^3-ös skálázást.

Szorgalmi:

Hasonlítsuk össze, hogy a mátrix szorzásnál melyik a gyorsabb: a csak a sort/oszlopot letároló módszer, vagy az, ha a teljes mátrixot allokáljuk.

Minden időmérést Release-ben végezzünk!


measured
fitted
Ismételt mérés nélkül
Calculation time (ms)
Size of matrices
0
10
20
30
40
50
60
100
150
200
250
300
350


measured
fitted
Ismételt mérések minimuma
Calculation time (ms)
Size of matrices
0
10
20
30
40
50
60
100
150
200
250
300
350