Samostudium 02

Praxe

Vít Petřík A4
13. dubna 2020

Přípravné práce

Váš výpočetní (programový) systém nedisponuje vestavěnou matematickou funkcí pro výpočet funkční hodnoty

- $f(x) = e^x$

Proto je nutné tento nedostatek obejít programově. Z matematické teorie plyne, že uvedený funkční předpis je možné nahradit výpočtem hodnoty polynomu zvaného *Taylorův rozvoj* ve tvaru:

- $e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} \ldots \sum_{i=0}^{+\infty} \frac{x^i}{i!}, pro\ x\ \in (+\infty; -\infty)$

Jako podpůrný, pomocný prostředek k vyřešení úlohy sestavte v jazyce C# program, který výše uvedený Taylorův rozvoj implementuje. Spočítejte pomocí vaší implementace postupně funkční hodnoty pro:

- $x = \{4{,}4;\ 1;\ 0;\ -1;\ -4{,}4\}$

Pro každou z uvedených hodnot proměnné xx stanovte potřebný počet kroků výpočtu tak, aby výsledná přesnost vašeho výpočtu byla na 5 platných cifer (číslic). Pro zjištění, zda jste již dosáhli požadované přesnosti použijte jako referenční hodnotu výpočet pomocí vestavěné matematické knihovny MATH.

Zadání

Diskutujte získané výsledky, zejména pak rychlost konvergence. PROČ je nutných právě tolik výpočetních kroků k dosažení požadované přesnosti pro tu kterou hodnotu x , která vám výpočtem vyšla. Případně, pokud si toto někdo z vašich studí matematiky pamatuje, navrhne opatření, aby se rychlost výpočtu zpřesnila, ev. zrychlila.

Pomocné funkce

Výpočet faktoriálu

```
static long Factorial(int number) {
    if (number < 2)
        return 1;
    else
        return number * Factorial(number - 1);
}
```

Navrácení prvních n-čísel

```
static int First_N_Places(int n, double x) {
    while(true)
    {
        if (x >= Math.Pow(10, n))
            x /= 10;
        else if (x < Math.Pow(10, n - 1))
            x *= 10;
        else
            break;
    }
    return (int) x;
}
```

Implementované algoritmy

Verze 1

Nejhloupější algoritmus. Nízká rychlost výpočtu, ale zase jednoduchý pro pochopení. Nevýhodou je výpočet faktoriálu i mocniny pokaždé znovu v každé iteraci *for cyklu.*

Experimentálně jsem zjistil, že v případě záporného *x* je lepší počítat s opačnou hodnotou a jako výsledek vrátit převrácenou hodnotu výsledku. Takto jsem dosáhl větší přesnosti s menším počtem iterací

```
static double Exponential_V1(int n, double x) {
    Boolean minus = false;
    if (x < 0) { x = -x; minus = true; }

    double sum = 1;

    for (int i = 1; i < n; i++)
        sum += Math.Pow(x, i) / Factorial(i);

    if (minus)
        sum = 1 / sum;

    return sum;
}
```

Verze 2

Už trochu chytřejší. Faktoriál se počítá průběžně ve *for cyklu*. Tato implementace je více jak *2x rychlejší* jak verze 1.

```
static double Exponential_V2(int n, double x)
{
    Boolean minus = false;
    if (x < 0) { x = -x; minus = true; }

    double sum = 1;
    long factorial = 1;

    for (int i = 1; i < n; i++)
    {
        factorial *= i;
        sum += Math.Pow(x, i) / factorial;
    }

    if (minus)
        sum = 1 / sum;

    return sum;
}
```

Verze 3

Poskytuje nejvyšší rychlost, které jsem byl schopen dosáhnout. Odstraňuje 2. krok iterace, který za každých podmínek vychází jako *x*. Zároveň odstraňuje použití knihovní funkce *Math.Pow* pomocí postupného počítání mocniny v každém kroku *for cyklu.*

Očekávaným problémem je přetečení datového typu *long* při výpočtu faktoriálu. Tento problém se mi samotnému nepodařilo vyřešit.

```
static double Exponential_V3(int n, double x)

{
    Boolean minus = false;
    if (x < 0) { x = -x; minus = true; }

    double sum = 1 + x;
    long factorial = 1;
    double power = x;
    for (int i = 2; i < n; i++)
    {
        factorial *= i;
        power *= x;
        sum += power / factorial;
    }

    if (minus)
        sum = 1 / sum;

    return sum;
}
```

Rekurzivní algoritmus ze stránky GeeksForGeeks

Pro kompletnost vypracování přikládám i algoritmus ze stránky GeeksForGeeks, který řeší problém mého algoritmu – tedy přetečení datového typu *long* při výpočtu faktoriálu. Rovnice *Taylorovy řady* je přepsaná do rekurzivního tvaru.

```
static double Exponential_stack(int n, double x)
{
    Boolean minus = false;
    if (x < 0) { x = -x; minus = true; }
    double sum = 1;

    for (int i = n - 1; i > 0; --i)
        sum = 1 + x * sum / i;

    if (minus)
        sum = 1 / sum;

    return sum;
}
```

Metodika měření rychlosti exekuce funkcí

Rychlost funkce byla měřena pomocí třídy *Stopwatch*, která, jak už název napovídá realizuje funkci stopek. Měření proběhlo za konstantních podmínek. Jediným spuštěným programem mimo Visual Studio bylo Spotify.

Aby byl získaný čas věrohodný byl změřen čas 10 000 000 exekucí jednoho algoritmu a z výsledku vypočítán průměrný čas jedné exekuce. Chybu měření způsobenou for cyklem zanedbáváme.

```
long testIterations = 10000000;
watch.Restart();
for (long i = 0; i < testIterations; i++)
    Exponential_V3(iteration, x);
watch.Stop();
Console.WriteLine("Doba= " + watch.ElapsedMilliseconds);
```

Výsledky byly zapsány do tabulky.

Výsledky

Exponent 4,4; Počet testů: 10000000; Přesnost na prvních 5 platných cifer		
Algoritmus	**Celková doba (ms)**	**Doba jednoho výpočtu (ns)**
Math.Exp	103	10,3
GeeksForGeeks	1623	162,3
v3	891	89,1
v2	8923	892,3
v1	17728	1772,8
Počet iterací: 17		

Exponent 1; Počet testů: 10000000; Přesnost na prvních 5 platných cifer		
Algoritmus	**Celková doba (ms)**	**Doba jednoho výpočtu (ns)**
Math.Exp	103	10,3
GeeksForGeeks	591	59,1
v3	431	43,1
v2	780	78
v1	2966	296,6
Počet iterací: 8		

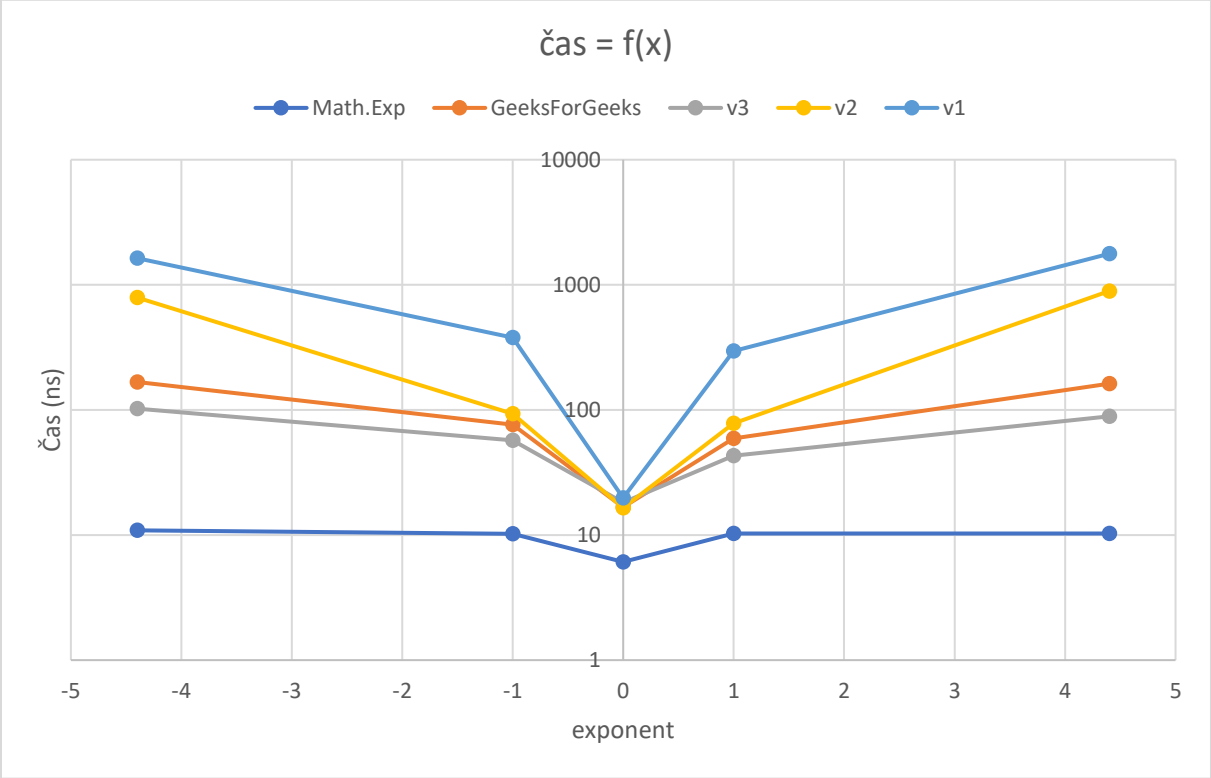
Exponent 0; Počet testů: 10000000; Přesnost na prvních 5 platných cifer		
Algoritmus	**Celková doba (ms)**	**Doba jednoho výpočtu (ns)**
Math.Exp	61	6,1
GeeksForGeeks	167	16,7
v3	182	18,2
v2	165	16,5
v1	198	19,8
Počet iterací: 0		

Exponent -1; Počet testů: 10000000; Přesnost na prvních 5 platných cifer		
Algoritmus	**Celková doba (ms)**	**Doba jednoho výpočtu (ns)**
Math.Exp	102	10,2
GeeksForGeeks	763	76,3
v3	572	57,2
v2	930	93
v1	3787	378,7
Počet iterací: 9		

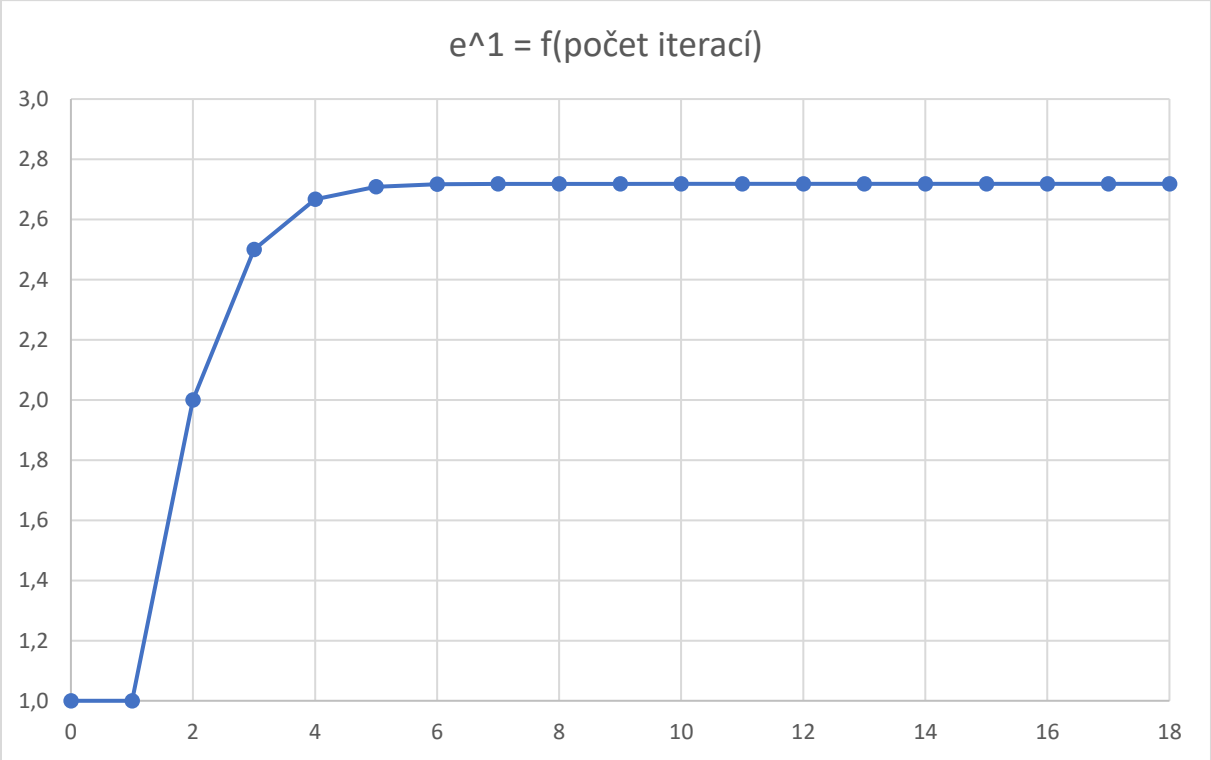
Exponent -4,4; Počet testů: 10000000; Přesnost na prvních 5 platných cifer		
Algoritmus	**Celková doba (ms)**	**Doba jednoho výpočtu (ns)**
Math.Exp	109	10,9
GeeksForGeeks	1672	167,2
v3	1024	102,4
v2	7904	790,4
v1	16340	1634
Počet iterací: 16		

Graf

Pro jednodušší představu jsem z výsledků sestavil graf. *Osa y* je v logaritmickém měřítku. Graf má málo datových bodů, ale lepší jak nic.



Pro znázornění vlivu počtu iterací na algoritmus, sestavil jsem graf závislosti výsledku na počtu iterací



Zhodnocení výsledků

Samozřejmě nejlépe vyšla nativní funkce Math.Exp. Nepovedlo se mi najít samotnou implementaci funkce, ale předpokládám, že je někde na úrovni systému. Překvapilo mě, jak dobře dopadla moje implementace ve verzi 3. Rychlostí překonává rekurzivní algoritmus. Problém ale nastane při nároku na větší přenost, kde už brzy dojde k přeteční datového typu long při počítání faktoriálu.

Verze 1 a 2 ani nestojí za řeč. Jsou pomalé a neposkytují žádnou výhodu oproti verzi 3. Jejich pomalost plyne z nutnosti počítat faktoriál a mocninu při každé iteraci for cyklu od začátku.

Pomyslným vítězem se tedy stává rekurzivní algoritmus ze stránky GeeksForGeeks, který poskytuje vysokou rychlost exekuce, ale i odolnost proti přetečení.

Závěr

Překvapilo mě, jak blízko jsem se svým algoritmem dostal k rychlosti nativní funkce. Rád jsem se naučil práci s třídou Stopwatch, která je pro provádění těchto benchmarků velice užitečná, skoro až perfektní.

Určitě ještě někdy použiji funkci, která z čísla vezme prvních n-číslic a jsem rád, že jsem si ji při této příležitosti implementoval.

Zajímavým poznatkem je narůstající počet potřebných iterací s exponentem. Domnívám se, že to je dáno tím, že čím větší má být výsledek, tím více se projeví chyba při výpočtech. V podstatě se *„zesílí“*.