GENERÁTOR NÁHODNÝCH ČÍSEL V MATLABU

P.Kahanek

e-mail: R04686@student.osu.cz

Katedra Informatiky a počítačů, PřF OU, 30.dubna 22, 701 03 Ostrava 1

Abstrakt

Generování pseudonáhodných čísel je nezbytnou součástí nejrůznějších simulací. Souběžně s rostoucím výkonem počítačů a potažmo počtem generovaných čísel rostou požadavky na generátory - na vygenerované posloupnosti nesmí být poznat, že ve skutečnosti náhodná není, nesmí se začít opakovat příliš brzy atd.

Nějaký generátor je zabudován v podstatě v každém statistickém programu, programovacím jazyce nebo vývojovém prostředí. Někdy se s těmito generátory pracuje, aniž by se vědělo, jak vlastně fungují a jestli jsou pro tu kterou úlohu "dost dobré".

Cíle tohoto článku jsou tyto:

- vytvořit základní přehled o tom, jaké metody se v této oblasti uplatňují,
- zdokumentovat a otestovat generátor v Matlabu
- představit dostupné alternativy k těmto zabudovaným generátorům.

Tento text vznikl za podpory grantu GAČR 201/05/0284.

Abstract

Pseudorandom number generation is very important in many fields of research, especially for various kinds of simulation. As performance of computers grows fast, there is a possibility of generating more and more numbers. Along with that, requirements laid on pseudorandom number generators grow: the pseudorandom sequence should look "as random as possible", it shouldn't start repeating itself too early and so on.

Today, nearly every statistical tool or development environment implements some pseudorandom generator. However, it is often not known how does the generator work, nor if it is suitable for certain type of application.

Main goals of this paper are as follows:

- to document methods and testing suites for pseudorandom generator testing,
- to introduce the generator in Matlab on theoretical and source code level
- to test the generator in Matlab with appropriate tool

The creation of this paper was supported by grant GACR 201/05/0284.

1 Testování "náhodnosti"

Je samozřejmé, že nemáme-li k dispozici nějaké zařízení fyzikální podstaty (jako např. osudí), je generátor představován deterministickým algoritmem. Výsledná posloupnost je tedy vždy pouze pseudonáhodná a nás zajímá "míra její náhodnosti" potažmo pak "použitelnost" toho kterého generátoru. Na tyto dost vágní pojmy se dnes nahlíží tak, že se požaduje, aby generátor splňoval sadu jistých předpokladů. Nejčastěji to bývají tyto [7][8]:

1) **Dlouhá perioda.** Ukazuje se, že pro generování "dobré" pseudonáhodné posloupnosti je vhodné, aby její délka byla jen zlomkem délky periody generátoru. L'Ecuyer [7] uvádí, že pokud používáme lineární kongruenční generátor, měl by mít periodu alespoň o řád větší, než je druhá mocnina počtu generovaných čísel. Také doporučuje minimálně hodnotu 2^{60} a raději mnohem více, což dnes platí dvojnásob, protože článek je z roku 1994. Příklad generátoru s opravdu dlouhou periodou je CMWC4096 George Marsaglii [11], pro který je uváděna tak trochu šílená hodnota 2^{131086}.
2) **Efektivita.** Generátor by měl být pokud možno rychlý a využívat co nejméně paměti. Tento požadavek se týká hlavně rozsáhlých simulací, při kterých je obvykle zapotřebí mít dostatek systémových prostředků pro jiné úlohy.

3) **Opakovatelnost.** Opět vlastnost důležitá především pro simulační úlohy. Měla by existovat možnost generovat shodné pseudonáhodné posloupnosti, tj. mělo by být možné nastavovat seed hodnotu.
4) **Přenosnost.** Algoritmus generátoru by měl být snadno implementovatelný a fungovat stejně na různých platformách, hardwarových i softwarových. Kupř. algoritmus, který používá typ 64-bitový integer je méně přenosný než ten, který používá integery 32-bitové apod.
5) **"Nevypočitatelnost"** je důležitá pro kryptografy (a také třeba pro kasina). Z vygenerované posloupnosti by nemělo být možné v rozumném čase určit, jaké číslo bude následovat. Splnit tento požadavek je kupodivu velmi obtížné pro člověka; existují algoritmy, které dokáží na základě několika člověkem vymyšlených čísel se slušnou úspěšností předpovědět to příští. L'Ecuyer [8] zavádí pojem **PT-perfektního** (polynomial-time perfect) generátoru: generátor je PT-perfektní, existuje-li konstanta $\varepsilon>0$ taková, že pro libovolné přirozené k neexistuje algoritmus s polynomiální časovou složitostí, který by z vygenerované posloupnosti bitů u_0, ..., u_{i-1} předpověděl u_i s pravděpodobností úspěchu větší než $1/2 + e^{-k\varepsilon}$. Zatím bohužel není dokázána ani existence PT-perfektního generátoru. O mnoha generátorech se nicméně ví, že PT-perfektní nejsou.
6) **Úspěšnost v empirických testech**. Generátor je podroben mnoha testům, které ověřují hypotézu H_0: "vygenerované hodnoty jsou nezávislá náhodná čísla z rovnoměrného rozdělení na intervalu (0,1)". Vzhledem k tomu, že generátory jsou deterministické algoritmy, není možné vytvořit generátor, který by prošel všemi testy. Za "špatné" generátory jsou považovány ty, které neprojdou jednoduchými testy, zatímco "dobré" neprojdou jen testy velmi složitými, které jsou implementačně i výpočetně velmi náročné. Hranice oné složitosti se samozřejmě vzhledem k neustále rostoucí rychlostí počítačů posouvá; testy, které byly před časem náročné, jsou dnes triviální [7][8][9][10].

1.1 Empirické testy generátorů

Testování generátorů náhodných čísel je dnes již velice široká oblast. S rostoucím výkonem počítačů je při simulacích generováno více a více čísel a je proto potřeba vyvíjet lepší a lepší generátory. A aby se o generátoru dalo říct, že je "dobrý", musí procházet testy, jejichž náročnost koresponduje s tím, kolik čísel budeme potřebovat. Dnes to běžně mohou být desítky či stovky miliónů.

Jedny z prvních testů zformuloval D. E. Knuth ve své slavné knize The Art of Computer Programming, vol. 2 [15], čímž zavedl jeden z prvních zásadních standardů v této oblasti, tj. generátory byly považovány za dobré, pokud těmito testy procházely.

Nový standard přinesl program DIEHARD z roku 1995 [10], díky němuž mnoho generátorů odpadlo. Autorem této testové sady je George Marsaglia z Florida State University, tvůrce mnoha generátorů i testů, z nichž mnohé se dnes řadí mezi ty klasické.

Svou testovou sadu definoval také americký NIST (National Institute Of Standards And Technology). Je zaměřena na nepředvídatelnost, tj. testuje vhodnost generátoru pro kryptografické úlohy, viz [12].

Zatím poslední výrazný skok v testování generátorů představuje open source knihovna TESTU01 v ANSI C [9] (Zdrojové kódy i jejich velmi podrobná dokumentace jsou volně ke stažení na adrese http://www.iro.umontreal.ca/~simardr/indexe.html), se kterou pomyslné síto kvality ještě více zhoustlo, a to dost podstatně. Jejími tvůrci jsou P.L'Ecuyer a R.Simard. TESTU01 zahrnuje velmi komplexní řadu testů, a navíc obsahuje několik předdefinovaných sad (batteries), včetně např. sady pseudoDIEHARD, která simuluje Marsagliův program, nebo "NIST test suite", zahrnující testy doporučené NIST. Jednu z předdefinovaných sad - Crush - použili McCullough a Wilson [1] pro testování generátoru v Excelu 2003 a bude použita i zde pro testování matlabovského generátoru.

2 Matlabovský generátor

Do roku 1995 byl v Matlabu zabudován jednoduchý kongruenční generátor [5]

$$x_{i+1} = 16807\, x_i \pmod{2^{31}-1}$$

s periodou $2^{31}-2$. S dnešního hlediska je už zcela nepoužitelný.

Do verze 5 (1995) byl implementován nový algoritmus, založený na práci Marsaglii, [5][6]. Tento nový generátor se de facto skládá ze dvou různých podgenerátorů (reálného a celočíselného), jejichž výstupy kombinuje v jedinou hodnotu pomocí operace xor. Délka periody je v tomto případě naprosto dostačující - 2^{1492}. V Mathworks se také domnívají, že by měla být generována všechna čísla v plovoucí řádové čárce z intervalu [eps/2, 1-eps/2], kde eps = 2^{-52}.

Reálný podgenerátor pracuje s bufferem o dvaatřiceti reálných položkách, přičemž vrací číslo z_i získané podle předpisu

$$z_i = z_{i+20} - z_{i+5} - b,$$

přičemž indexy *i*, *i+20* a *i+5* jsou interpretovány mod 32. Hodnota b je buď 0, nebo ulp ("unit in the last place" neboli 2^{-53}, viz Tabulka 2), podle toho, jestli bylo v minulém kroku spočteno kladné nebo záporné z_i. Kód v C vypadá takto

```
x = z[(i+20)%32] – z[(i+5)%32] - b;      // vypočítáme další číslo
if (x < 0) {                             // pokud vyšlo záporné
      x = x + 1;                         // přičteme 1
      b = ulp;
}
else
      b = 0;
z[i] = x;                                // zapíšeme výsledek do
                                         // bufferu,
i = (i+1)%32;                            // a posuneme se v bufferu
                                         // na další index (mod 32)
```

Než začneme, je samozřejmě nutné buffer nějak naplnit. Jedna hodnota do bufferu se generuje takto: bit po bitu se vyrobí 53-bitové celé číslo *n* a do bufferu se zapíše $n2^{-53}$. Všechny hodnoty v bufferu jsou tedy celočíselné násobky 2^{-53}. Protože se pro vygenerování kýženého pseudonáhodného *x* používá pouze operace sčítání (resp. odčítání), má i ono vygenerované číslo tuto vlastnost, tj. je určitě ve tvaru $n2^{-53}$, kde *n* je nějaké celé číslo.

To znamená, že většina čísel v plovoucí řádové čárce z intervalu [0, 1] nemůže být tímto postupem vygenerována. Tento závěr plyne z binární reprezentace reálných čísel, tj. typů float, resp. double, jak ji definuje IEEE Standard 754 [13].

Float je reprezentován dvaatřiceti bity, double čtyřiašedesáti. Čísla jsou uchovávána ve tvaru

$$(+/-)\ \text{mantisa} * 2^{\text{exponent}},$$

přičemž mantisa je ve tvaru

$$1.\text{fraction}$$

kde "." značí binární tečku, tj . dvojkovou analogii tečky desetinné. Aby mohl exponent nabývat i záporných hodnot, odečítá se od něj tzv. bias. Následující tabulka ukazuje, jak jsou znaménko, fraction a exponent uloženy v bitové reprezentaci.

Tabulka 2: Binární reprezentace typů float a double

	sign	**Exponent**	**fraction**	**bias**
Float	1 [31]	8 [30-23]	23 [22-00]	127
Double	1 [63]	11 [62-52]	52 [51-00]	1023

Dále budeme uvažovat pouze double. Z tabulky vidíme, že pro každý exponent máme k dipozici 2^{52} různých fractions. To znamená, že mezi každými dvěma následujícími mocninami dvojky lze rozlišit 2^{52} různých čísel ve formátu double. Je tedy 2^{52} čísel v intervalu [1/2, 1), 2^{52} čísel v intervalu [1/4, 1/2), 2^{52} čísel v intervalu [1/8, 1/4) atd. Diference mezi nimi jsou v každém takovém intervalu jiné, a sice (velikost intervalu)/2^{52}.

Která z čísel v plovoucí řádové čárce je ale schopen vyrobit reálný podgenerátor?

Tabulka 3: Možná nenulová čísla vygenerovaná reálným podgenerátorem.

53 bitové číslo *n*	**Počet nul v *n* zleva**	**Bin. reprezentace $n*2^{-53}$ podle IEEE 754**	**Fractions na exponent**
0.....01	52	$1,0 * 2^{-53}$ (**ulp**)	1
00.....10 00.....11	51	1,0 1,1 } $* 2^{-52}$	2
00.....100 00.....101 00.....110 00.....111	50	1,00 1,01 1,01 1,01 } $* 2^{-51}$	4
00.....1000 00.....1001 00.....1010 00.....1011 00.....1100 00.....1101 00.....1110 00.....1111	49	1,000 1,001 1,010 1,011 1,100 1,101 1,110 1,111 } $* 2^{-50}$	8
...	...	...	...
2^{51} možností	1	výraz $* 2^{0}$	2^{51}
2^{52} možností	0	výraz $* 2^{-1}$	2^{52}

Z tabulky vidíme, že např. pro exponent *-51* dokáže algoritmus vygenerovat jen čtyři čísla z 2^{52} možných, pro exponent *-50* jen osm čísel atd. Právě proto byl do matlabovského generátoru přidán ještě celočíselný podgenerátor. Funguje následovně

```
randint(j) {
      j = j ^(j<<13);
      j = j ^(j>>17);
      j = j ^(j<<5);
}
return j;
```

Zde operátory << resp. >> znamenají bitový posun o zadaný počet bitů a operátor ^ bitový XOR.

Číslo *j*, které tento generátor vrací, se použije k "roztřesení" fraction: obě hodnoty se zaxorují. Protože *j* může být teoreticky cokoli, může být i výsledek XORu jakýkoli, tj. pro daný exponent bychom měli být schopni vyrobit všechny možné fractions. Odtud už plyne, že by matlabovský generátor měl generovat všechna čísla typu double z intervalu $[2^{-53}, 1-2^{-53}]$. (Na závěr malinkou poznámku: ne že by to nebyla pěkná vlastnost, ale nejsem si tak úplně jist, k čemu je to vlastně dobré.)

Stavový vektor matlabovského generátoru má tedy celkem 35 položek: 32 hodnot je uloženo v bufferu, uchovává se *i* (aktuální pozice v bufferu), *b* (0 nebo *ulp*) a také *j* (stav celočíselného podgenerátoru). Seed hodnota se přiřazuje právě do *j*, implicitně to je 2^{31}.

3 Empirické testy

K testování byla použita sada Crush z knihovny TestU01. Ve verzi 0.5.4 obsahuje 94 testů. Jak již bylo řečeno, sada Crush je dosti náročná. Zatímco oblíbený DIEHARD zabere na relativně moderním počítači (Pentium IV, 2.4GHz, 512MB RAM, Windows XP) několik vteřin, Crush stejný stroj zaměstná na dvě až tři hodiny. Generuje se přitom zhruba 2^{35} pseudonáhodných čísel, pročež je velice

vhodné mít testovaný generátor implementovaný co nejjefektivněji. Podle mých zkušeností je naprostou nezbytností alespoň 512 MB operační paměti.

Výstupy testů jsou klasické p hodnoty (dosažené významnosti testů), přičemž jako neúspěšné se berou ty testy, pro které p padne mimo interval [0.01,0.99] (resp. [0, 0.99]). Jako jasná selhání (tzv. systematické chyby generátoru) se berou testy, pro které se p blíží k 0 nebo k 1 na několik desetinných míst. Meze pro selhání jsou uváděny různě, častými hodnotami jsou např. 10^{-6} a 10^{-10}. Testy, pro které p padne "těsně mimo interval [0.01, 0.99]" (p bude řekněme 0.005 nebo 0.999), jsou chápány jako podezřelé a je vhodné je opakovat s pozměněnými hodnotami parametrů.

Vhodnost generátoru pro velmi specifický účel je samozřejmě nejlepší ověřovat příslušným, velmi specifickým testem, pro nějž platí, že výpočet statistiky koresponduje s řešením dané úlohy. Pokud něco takového dělat nechceme, je nejlepší použít ten generátor, který je nejúspěšnější v nějakých standardních testech, např. právě v sadě Crush.

3.1 Některé testy obsažené v sadě Crush

Omezíme se pouze na ty testy, které nějak souvisejí se selháními matlabovského generátoru. Popis úplně všech implementovaných testů zájemce nalezne v dokumentaci k TESTU01 [9].

Hypotézou H_0 v následujícím výčtu se rozumí "vygenerovaná čísla jsou náhodné veličiny z rovnoměrného rozdělení na (0,1)". Parametry N, n, r znamenají vždy totéž: N počet opakování testu, n velikost výběru pro jeden test, r počet bitů (od nejvyššího), které test nebere v úvahu.

Serial(N,n,r,d,t) [Knuth]
Základní myšlenkou je testovat uniformitu vygenerovaných čísel pomocí testu dobré shody, obecně v t-dimenzionálním prostoru. Tj. interval $[0,1]^t$ se rozdělí na d^t buněk, ve kterých se porovnávají pozorované a očekávané četnosti, přičemž se používá n*t vygenerovaných hodnot (n vektorů, t složek každý). Vektory se nepřekrývají. Tímto testem v zásadě prochází drtivá většina generátorů, dobré i špatné.

SerialOver(N,n,r,d,t) [Marsaglia]
Pracuje velmi podobně jako Serial s tím, že vstupní vektory se překrývají. Je generována posloupnost $u_0,...,u_{n-1}$ a n vektorů je vytvořeno jako $(u_0,...,u_{t-1})$, $(u_1,...,u_t)$, ..., $(u_n,u_0,...,u_{t-2})$

CollisionOver(N,n,r,d,t) [Marsaglia]
Podobné jako Collision, vektory jsou ale generovány s překrytím, stejně jako v SerialOver.

MultinomialBitsOver(N,n,L,s,r,sparse)
Bitová varianta SerialOver (sparse=FALSE) resp. CollisionOver(sparse=TRUE). Nejprve se generuje řetězec bitů tak, že se vezme *s* bitů z každého vygenerovaného čísla, každých L po sobě jdoucích bitů v tomto řetězci (tj. každý podřetězec délky L) pak identifikuje buňku v L-rozměrném prostoru nad tělesem Z_2.

Gap(N,n,r,alpha,beta) [Knuth]
Uvažujme interval $[\alpha, \beta]$, $0 \leq \alpha < \beta \leq 1$. Test pro s = 0,1,... počítá, kolik prvků posloupnosti *s* po sobě jdoucích čísel padne mimo interval $[\alpha, \beta]$. Pak porovnává očekávané a pozorované četnosti pomocí chí-kvadrát testu.

SimpPoker(N,n,r,d,k) [Knuth]
Vygeneruje *n* skupin po *k* integerech z intervalu [0,d-1], pro každou skupinu počítá *s* = počet různých hodnot ve skupině. Pomocí chí-kvadrát testu srovnává očekávané a pozorované četnosti pro různá *s*.

SumCollector(N,n,r,g)
Test generuje náhodná čísla tak dlouho, dokud jejich součet nepřekročí *g*, aby zjistil hodnotu $J = \min\{k \geq 0: u_0 + ... + u_k > g\}$. Postup se opakuje n-krát, tj. máme hodnoty J_1, ..., J_n, jejichž rozdělení se poté srovnává s očekávaným rozdělením, popsaným v článku [14].

SampleProd(N,n,r,t)
Je generováno *t***n* čísel. Test zkoumá rozdělení *n* součinů, které vznikají násobením *t* po sobě jdoucích čísel, resp. srovnává ho s jejich teoretickým rozdělením.

WeightDistrib(N,n,r,k,alpha, beta)
Generuje se *k* čísel u_0, ..., u_{k-1} a počítá se statistika W = počet u_j, které padly do intervalu [α, β). Za platnosti H_0 má W binomické rozdělení s paramerty *k* a p= β-α. Postup se opakuje n-krát a rozdělení získaných W_1, ...,W_n se srovnává s binomickým pomocí chí-kvadrát testu.

3.2 Výsledky sady Crush

Matlabovský generátor příliš nepřesvědčil. Periodu sice má naprosto dostačující, ale systematická chyba přichází v pěti testech. Nic naplat, je už přece jen starý.

```
============== Summary results of Crush ==============

Generator:        Matlab
The following tests gave p-values outside [0.01, 0.99]:
 (eps  means a value < 1.0e-015):

      Test                            p-value
 ----------------------------------------------
  7  MultinomialBitsOver            9.5e-03
 25  SimpPoker (d = 16)              0.9986
 32  Gap                              eps
 34  Gap                              eps
 41  SampleProd                     1 -  2.4e-15
 45  WeightDistrib (r = 0)            eps
 49  SumCollector                     eps
 ----------------------------------------------
 All other tests were passed
```

Budeme-li sadu Crush považovat za nový standard v oblasti posuzování kvality generátorů, nemůžeme matlabovský generátor pro rozsáhlé úlohy doporučit.

Podobně neúspěšný je například generátor Wichmann-Hill [16][17] implementovaný v Excelu 2003, ten selže hned v deseti testech (dodejme, že v jiných než matlabovský generátor). Viz rovněž [1]. Oproti tomu příkladem generátoru, který přes Crush úspěšně prochází, je v poslední době velmi oblíbený Mersenne Twister (též označovaný MT19937) autorů Matsumota a Nishimury [4].

3.3 Jak teď interpretovat výsledky testů?

Pokud bychom se drželi tradice mnoha článků, které zkoumají vlastnosti statistického softwaru, zněla by odpověď jednoduše: NIJAK, generátor neprošel standardní sadou, tečka. Zde ale výsledky rozebereme podrobněji. Matlabovský generátor selhal v následujících testech (je uvedeno i nastavení parametrů):

```
32 Gap N = 1, n = 10^8,   r =  0, Alpha = 0, Beta  =  0.125
34 Gap N = 1, n = 5*10^6, r =  0, Alpha = 0, Beta  = 0.003906
41 SampleProd  N = 1, n = 10^7, r = 0, t = 30
45 WeightDistrib N = 1, n = 2*10^6, r = 0, k = 256, Alpha = 0, Beta =
0.125
49 SumCollector N = 1, n = 2*10^7, r = 0, g = 10
```

Tyto mají jednu společnou vlastnost: všechny nějakým způsobem dělí generovanou posloupnost na podposloupnosti, ze kterých něco spočítají a zkoumají rozdělení výsledků. Proto jsem zkusil aplikovat ještě relativně jednoduché testy uniformity, které jsou právě tímto specifické. Několik takových testů je v TESTU01 zahrnuto.

SampleMean(N,n,r)
Test vygeneruje *n* čísel u_1, ..., u_n a spočítá jejich průměr. Zopakuje to N-krát a pak pomocí srovnává rozdělení N průměrů s jejich přesným teoretickým rozdělením pro n<60, resp. s normálním

rozdělením s parametry (0.5, 1/(12*n*)) pro $n \geq 60$. K porovnání jsou použity testy Kolmogorov-Smirnov, resp. Anderson-Darling.

SumLogs(N,n,r)
Jediný rozdíl oproti SampleMean je ten, že se nepočítá průměr, ale sumy

$$P = -2\sum_{j=1}^{n} \ln(u_j),$$

které mají za platnosti H_0 rozdělení chí-kvadrát s 2n stupni volnosti. Je-li v průběhu výpočtu vygenerována hodnota příliš blízká nule, u_j < DBL_EPSILON/2, provede se přiřazení u_j = DBL_EPSILON/2 (DBL_EPSILON = 2.2204460492503131e-016 je systémová konstanta svázaná s typem double; je to nejmenší z čísel *x*, pro které dá operace 1.0 + *x* výsledek různý od 1.0).

Nakonec došlo i na ten nejtypičtější případ, totiž chí-kvadrát test dobré shody na normální rozdělení N součtů *n* po sobě jdoucích čísel.

Pokud je matlabovský generátor v této oblasti opravdu nedostačující, měly by všechny tři testy dávat velmi podobné výsledky. Podrobněji zde rozebereme chí-kvadrát test.

Tabulka 4: výsledky testů (p-hodnoty) , N=106

N	X^2 na normalitu	Anderson-Darling na SumLogs	Anderson-Darling na SampleMean
30	0.99839	0.05	0.26
50	1.00000	eps	1.50E-11
80	1.00000	eps	1.70E-11
100	1.00000	eps	1.50E-11
200	1.00000	eps	1.50E-05
500	1.00000	1.20E-07	1.10E-03
1000	0.98978	7.40E-03	0.04

Poté, co jsem provedl několik testů s různými nastaveními parametrů, jsem pojal podezření, že se empirická distribuční funkce s rostoucím *n* vzdaluje od distribuční funkce normálního rozdělení. Na obrázcích, na kterých to mělo být vidět, ale nebylo vidět vůbec nic, grafy obou funkcí se v podstatě překrývaly. Přesto chí-kvadrát test dával výsledky naprosto popírající, že by mohlo jít o totéž.

Rozuzlení je pohříchu jednoduché: při zvoleném N (10^5, $5*10^5$, 10^6, $3*10^6$) je porovnávání "od oka" už k ničemu. Potřebujeme nějakou exaktnější míru, abychom mohli posoudit, jak moc se od sebe distribuční funkce odlišují. A když už počítáme chí-kvadrát statistiku, proč ji k tomu nepoužít? N i počet kategorií necháme stejné, měnit budeme jen *n*.

Máme-li všude stejný počet kategorií, můžeme navíc chí-kvadrát statistiku přepočítat na p. Samotné p nám toho ale zase tak moc neříká - výsledky jsou různé, 0.95, 0.20, 0.74, ... jaký je mezi nimi rozdíl? V našem kontextu žádný. U testování generátorů pomocí tak velkého počtu čísel nehraje 0.95, 0.99 ani 0.999 podstatnou roli. Jak píše G.Marsaglia v dokumentaci ke svému DIEHARDu [10], "p happens". Abychom mohli udělat nějaký kategorický závěr, potřebujeme hodnotu velmi blízkou 0 nebo 1 (v tomto případě 1), s přesností na několik desetinných míst. Tohoto faktu využijeme při interpretování výsledků.

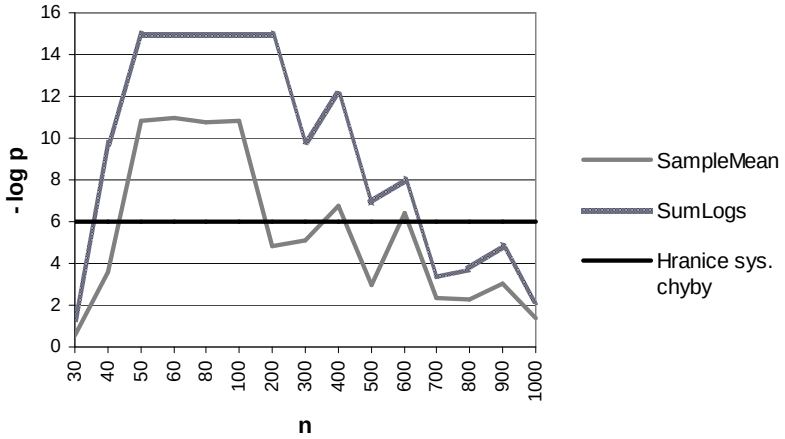
Abychom měli srovnání, provedeme test normality součtů také pro již zmiňované generátory MT19937 a v Excelu 2003 zabudovaný Wichmann-Hill. Budeme přitom sledovat, zda a kdy se generátory dostanou do velkých problémů, tj. kdy p překročí hranici systematické chyby. Tu zvolíme jako 10^{-6} (resp. $1\text{-}10^{-6}$); máme v tom určitou libovůli, různí autoři ji volí různě, např. až 10^{-10}.

Výsledky jsou na Obr. 1. Počítáme s přesností 10^{-15}, graf má tedy lehce "useknutý" vrchol. Je vidět, že se Matlab od distribuční funkce nevzdaluje nadobro, jen si na chvíli "odskočí"; aby se po určité době - poněkud neochotně - vrátil do mezí definovaných centrální limitní větou. Problematický interval pro *n* je zhruba [40, 500].

Obrázek 1: Na kolik desetinných míst se p blíží k 1, N=10^6

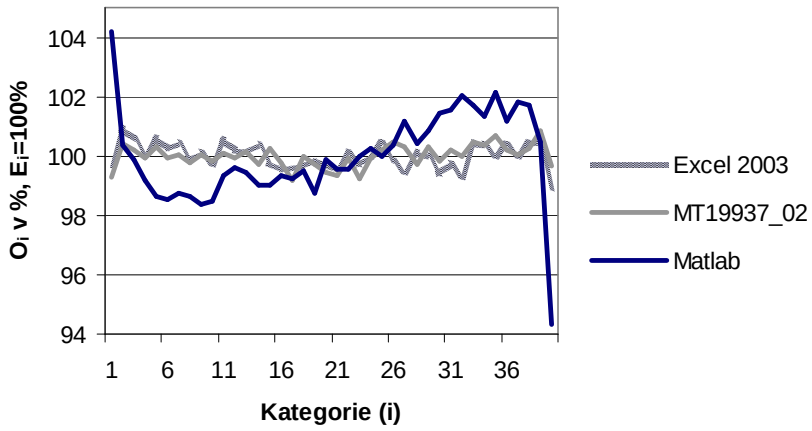
Potíže se projevují přibližně od N=10^6 a s dalším růstem N jsou stále zřetelnější - "kopec" na Obr.1 se "roztahuje do šířky". Výsledky pro SampleMean resp. SumLogs jsou velice podobné.

Obrázek 2: Výsledky SumLogs a SampleMean



Zajímavá je analýza toho, ve kterých kategoriích nastává problém při výpočtu chí-kvadrát statistiky. Vezměme nějaké problematické *n*, třeba 80, a zkoumejme rozdíly O_i-E_i pro jednotlivé kategorie.

Obrázek 3: Analýza četností v kategoriích pro X^2 test



Označme počet kategorií n_k. Kategorie byly voleny tak, aby platilo $E_i = N/n_k$ pro každé relevantní *i*. Za platnosti H_0 by grafem rozdílů O_i-E_i tedy měl být jakýsi "šum" okolo osy x. Namísto toho ale v případě Matlabu dostáváme šum okolo jakéhosi polynomu, z jehož tvaru se dá odhadnout, která čísla z normálního rozdělení jsou generována častěji na úkor jiných. Zajímavé je, že za neúspěch v tomto testu je zodpovědný celočíselný podgenerátor. Pokud jej zakážeme, veškeré problémy s testem dobré shody na normální rozdělení součtů jsou ty tam.

Ať už jsou důvody selhání v tom kterém testu jakékoli, všechny testy dopadly velice podobně. Naše podezření, že má testovaný generátor v této oblasti problémy, se potvrdilo.

4 Závěr

Matlabovský generátor má problém, pokud generujeme N hodnot tak, že každá je výsledkem nějakého výpočtu provedeného nad posloupností délky *n*. Je-li přibližně $N > 10^6$ a $40 < n < 500$, nastává selhání (systematická chyba) v příslušném empirickém testu. Pro rozsáhlé, složité simulace jej proto nelze doporučit.

5 Poděkování

Tento text vznikl za podpory grantu GAČR 201/05/0284. Chtěl bych rovněž touto cestou poděkovat Doc. Ing. Josefu Tvrdíkovi za poskytnutí materiálů, upozornění na řadu chyb a překlepů a užitečné připomínky k organizaci textu.

6 Odkazy

[1] McCullough , B., Wilson, B.: *On the accuracy of statistical procedures in Microsoft Excel 2003*, Computational Statistics & Data Analysis (accepted 21 June 2004)

[2] *Description of improvements in the statistical functions in Excel 2003 and in Excel 2004 for Mac*, Microsoft Knowledge Base, Article ID : 828888

[3] *Description of the RAND function in Excel 2003*, Microsoft Knowledge Base, Article ID : 828795

[4] Matsumoto M., Nishimura T.: *Mersenne Twister: A 623-equidistributed uniform pseudorandom number generator*, ACM Transactions on Modeling and Computer Simulations: Special Issue on Uniform Random Number Generation, 1998

[5] Moler, C.: *Random thoughts (10^{435} years is a very long time)*, MATLAB News & Notes, Fall 1995, 12-13

[6] *What is the random number generator in MATLAB?* The MathWorks Product Support, Solution Number: 1-169M3

[7] L'Ecuyer, P.: *Uniform random number generation*. Annals of Operations Research, 53: 77--120, 1994

[8] L'Ecuyer, P.: *Random number generation*. Draft for a chapter of the forthcoming Handbook of Computational Statistics, J. E. Gentle, W. Haerdle, and Y. Mori, eds., Springer-Verlag, 2004.

[9] L'Ecuyer, P., Simard, R.: *TestU01-0.6.0, Empirical Testing of Random Number Generators (beta version)*, http://www.iro.umontreal.ca/~simardr/indexe.html

[10] Marsaglia, G.: *The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness*, http://stat.fsu.edu/pub/diehard/

[11] Marsaglia, G.: *Good C random number generator*, http://school.anhb.uwa.edu.au/personalpages/kwessen/shared/Marsaglia03.html, 2003

[12] *Random Number Generation and Testing*, An interdivisional project in the Information Technology Laboratory at NIST, http://csrc.nist.gov/rng/

[13] *Floating Point Numbers*, IEEE Standard 754, http://stevehollasch.com/cgindex/coding/ieeefloat.html

[14] Ugrin-Sparac, G.: *Stochastic investigations of pseudo-random number generators.* Computing, 46:53-65, 1991.

[15] Knuth, Donald E.: The Art Of computer Programming, volume 2: Seminumerical Algorithms, Third Edition (Reading, Massachusetts: Addison-Wesley, 1997), ISBN 0-201-89684-2

[16] Wichmann, B.A., Hill, I.D.: Algorithm AS 183: an efficient and portable pseudo-random number generator, Appl. Statist. 31, 188–190, 1982

[17] Wichmann, B.A., Hill, I.D.: Correction: algorithmAS 183 : an efficient and portable pseudo-random number generator. Appl. Statist. 33, 123, 1984
[18] Antoch, J.: *Jak pomocí simulací dokázat nemožné*, Informační Bulletin České Statistické Společnosti, ročník 9., č.1, 1998
[19] Máša, P.: *Zajímavý generátor náhodných čísel*, Informační Bulletin České Statistické Společnosti, ročník 14., č.4, 2003