Lecture notes:
Running EGS4 on different architectures

Alex F Bielajew
Institute for National Measurement Standards
National Research Council of Canada
Ottawa, Canada
K1A 0R6

Tel: 613-993-2715
FAX: 613-952-9865
e-mail: `alex@irs.phy.nrc.ca`

National Research Council of Canada Report PIRS-0392

Naeser's Law:
"You can make it foolproof, but you can't make it damnfoolproof.."

1 Introduction

This lecture is a highly personalised description of various issues related to running EGS4 on different machines. The author's direct experience comes from running EGS4 on the following architectures:

- VAX/VMS (*e.g.* 11/780 FPA, μVAX 3600, VAX 8xxx)
- IBM/CMS 3090/180/300/300E (HPO, VF)
- FPS-264 (M64/50)
- IBM PC 386/387 (OS386)
- RISC/UNIX machines (DecStations, SGI's, SunSparcstations, IBM R6000's)

EGS4 has run on a variety of machines from PC/286's to Cray's as well as a variety of architectures, scalar, parallel and vector.

In this lecture I will attempt to address what problems one might expect in getting EGS4 running on a scalar (serial processor) machine. I leave detailed discussion of other architectures to a later date (when the author can get his hands on some).

2 Questions posed by EGS-perts and in-EGS-perts

2.1 What machine should I run EGS4 on?

Usually the response to this question is, "Whatever happens to be available at my institution!". In this case the proper question is, "Can I get EGS4 running on this machine?".

The minimum configuration for EGS code is to have about 500KB of available main memory, about 20 MB of disk storage and a FORTRAN compiler. Increasing both main memory and disk capacities will increase one's ability to handle complex problems. Monte Carlo simulations are generally compute-bound problems, so if economy of funds is important one does not need to invest in the fastest disk storage available. (It is nice to have, however.) Other desirable software/hardware/peopleware capabilities in approximate order of priority are:

- Floating-point processors, (*e.g.* VAX FPA's, 80x87's, Weitek x167's) (almost a waste of time without these)
- Lots of memory. Fixed memory machines (*e.g.* DOS-PC's, Cray's) will not allow your code to run if it is too big. Virtual memory machines can slow down by a factor of 10^4 or more(!) if your job does not fit into memory and it employs disk space as virtual memory.
- Fortran debugging tools
- Graphics capabilities
- *Someone else to run the system for you*

2.2 How fast is fast?

There are many measures of machine speed, MIPS, MFLOPS, Whetstones, Dhrystones, Specmarks, Khornstones, `LINPACK` benchmarks, *etc...* Peak machine speed, peak-MIPS or peak-MFLOPS, (often quoted by manufacturers) or peak-anything is an almost useless measure of a machine's speed. The only possible interpretation of peak speed is, "The speed your application is guaranteed *not* to surpass". The most useful benchmark of a machine's speed is to time your own code running on it. To this end, we distribute a standard timing benchmark code [1] (called `XYZDOS`, part of the UNIX and PC distributions), for comparison of various machines and architectures. The next best estimate can be gleaned from the `LINPACK` benchmarks [2], based on solving a dense set of linear equations of order 100. This benchmark tests not only processing speed but also non-local and distant memory fetches, testing main memory–data cache–CPU bottlenecks, usually the cause of peak speed degradation. Monte Carlo calculations involve a lot of remote addressing and in this respect only are similar to linear algebra on large matrices. However, Monte Carlo codes usually involve iterative loops over large instruction sets and main memory–instruction cache bottlenecks can significantly affect performance. We have noted some anomalies between the comparison of `XYZDOS` and `LINPACK`.

Consider the comparison of an EGS4 simulation based on a realistic problem, the `XYZDOS` benchmark given in Table 1. The most remarkable revelation from the benchmark timing results is that high-end PC's (costing about $2K–$3K) and low-end UNIX workstations (costing about $3K–$6K) have encroached upon mainframe territory in this form of scientific computing.

If you are lucky enough to able to choose machines based upon EGS4 demands, choose the fastest, serial processing machines that return the highest benchmark numbers per unit of money. Note that apart from system maintenance, the number of machines should not be a strong factor in the determination. In the near future, network operating systems will be able to make good use of either many slower machines or a few fast ones. If you are unable to run your code or the `XYZDOS` benchmark, base your decision regarding machine speed by consulting the above list and seeing which architecture matches most closely, especially with respect to the size of the instruction and data caches. Failing this, use the `LINPACK`, Specmark, or MIPS timing benchmarks, but keep in mind that it may not be accurate. (The author has noted that for one architecture, the `LINPACK` benchmark and the `XYZDOS` benchmark differed by a factor of 5!)

2.3 Single or double precision?

For most modern architectures, it does not cost very much extra to run applications in double rather than single precision with the possible exception of the VAX. This feature is true for their entire product line with double precision overheads ranging from 20%–100%! Some architectures do not even support single precision floating-point numbers (*e.g.* FPS serial processors) and other architectures, while supporting single-precision arithmetic, do so with some associated overhead (*e.g.* IBM R6000's)!

The author has found significant differences in comparing calculated results of stopping powers (significant to the fifth digit) from VAX (CISC architectures) and IBM (CISC architectures) single precision simulations. (VAX single precision floating point numbers have slightly more precision than their IBM counterparts). Differences in energy deposition accumulation (good to about 3 digits) have never been observed.

Ray tracing of particles is done better in double precision. Unless the geometry of curved surfaces is coded very carefully, round-off and truncation of floating-point numbers can cause difficulty. The rule of thumb is "If you can afford it, run in double precision".

2.4 What language should I program in?

EGS4 is written in MORTRAN3, a FORTRAN pre-compiler that can convert the EGS4

MACHINE	O/S	FORTRAN	RATIO
Normalization			
VAX 11/780 FPA	VMS 3.7	FORT-11 3.7	$\equiv 1$
PC-based			
Compaq 20 MHz 386/387	OS386 1.0	Lahey F77L-EM/32 2.0	1.1
ACAD 25 MHz 386/387	OS386 2.1.04	Lahey F77L-EM/32 3.01	1.6
Compaq 20 MHz 386/Weitek 1167	OS386 1.0	Lahey F77L-EM/32 2.0	3.0
ACAD 25 MHz 486	OS386 2.1.04	Lahey F77L-EM/32 3.01	3.7
Micronics 33 MHz 386/Weitek 3167	OS386 1.9.16	Lahey F77L-EM/32 2.01	5.0
Dell 33 MHz 486E	OS386 2.1.04	Lahey F77L-EM/32 3.01	5.1
Unix Workstations			
Sun 3/60	SunOS 4.0	SunFortran 1.2 -f68881	0.90
HP-9000/370/Turbo SRX	HP-UX 6.5	HP Fortran V46.24(20)	1.9
SunSparcstation1	SunOS 4.0.3c	SunFortran 1.2 -O3	4.7
SunSparcstation1+	SunOS 4.1.1	SunFortran 1.3 -fast -O4 -Bstatic	6.4
Decstation 3100	U-32 3.1	V1.0 -align	7.3
Okidata Vistra 800 (40 MHz i860)	AT&T System V	Portland Group PGF77 -O0	7.5
Decstation 5000/120	Ultrix 4.2a	V3.1	10
SG Personal IRIS 4D/25	IRIX 3.2.1	3.2 F77 1.31 -static	10
IBM R6000 Model 320	AIX 3.01	xlf 1.01 -Q -O -qnorndsngl	11
SunSparcstation2	SunOS 4.1.1	SunFortran 1.3 -fast -O4 -Bstatic	11
SunSparcstation IPX	SunOS 4.1.1	SunFortran 1.3 -fast -O4 -Bstatic	12
IBM R6000 Model 530	AIX 3.01	xlf 1.01 -Q -O -qnorndsngl	14
SG Personal IRIS 4D/35	IRIX 3.3.2	F77 3.3.2 -O4 -static	20
SunSparcstation 10/31	SunOS 4.1.3	SunFortran 1.3.1 -O4 -Bstatic	21
HP-9000/720	HP-UX 8.0: pre-IC2	HP Fortran 77 V8.01 -K -O	24
SGI Indigo (100 MHz R4000)	IRIX 4.0.5F	F77 3.10 -O3 -mips2 -static -sopt	31
HP-9000/730	HP-UX 8.07	HP Fortran 77 V8.07 -K -O	32
HP-9000/735	HP-UX 9.01	HP Fortran 77 V8.07 -K -O	53
Non-Unix Workstations			
VAXstation 31/30	VMS 5.1	FORT-11 5.3	2.1
VAXstation 31/38	VMS 5.3	FORT-11 5.3	3.1
VAXstation 31/76	VMS 5.4	FORTRAN V5.8-155	6.5
DEC AXP 3000/500	OpenVMS Alpha 1.0	DEC Fortran 6.0	42
Unix Minis/Servers			
Sun 4/330	SunOS 4.0.3	SunFortran 1.2	6.2
SG 280S (1 CPU - 25 MHz)	IRIX 3.2.1	3.2 F77 1.31 -static	13
SG 280S (1 CPU - 25 MHz)	IRIX 4.0.1	3.4.1 F77 -O3 -static	13
SG 280S (1 CPU - 33 MHz)	IRIX 4.0.1	3.4.1 F77 -O3 -static	17
Non-Unix Minis/Servers			
μVax II	VMS 5.2	FORT-11 5.2-33	0.72
VAX 11/780 FPA	VMS 4.5	FORT-11 5.3	1.1
μVax 3300	VMS 5.3	FORT-11 5.3	1.6
μVax 3600	VMS 5.4	FORTRAN V5.8-155	2.1
μVax 3600	VMS 4.7	FORT-11 4.7a	2.4
μVax 3800	VMS 5.1	FORT-11 5.3	2.8
μVax 3800	VMS 5.4	FORTRAN V5.8-155	3.2
VAX 4000/600	VMS 5.5	FORTRAN 5.8	31
Mainframes			
VAX 8200	VMS 5.3	FORT-11 5.3	0.77
Vax 6330 (1 CPU)	VMS 5.2	FORT-11 5.2-33	3.4
IBM 3090/3090E	VM/HPO 4.2 CMS	VSFORT 2.0	24/25
Supercomputers			
Fujitsu VP2200/10 (scalar mode)	UXP/M V10L10	Fujitsu FORTRAN77 EX V12	58
Exotica			
Meiko ($1\times$T800 20 MHz)	MMVCS 1.4	Meiko F77 1.100	2.3
BBN TC2000 ($1\times$MC88000 20MHz)	UNIX 4.3BSD + pSOS^{+m}	Fortran-88000 3.45 -OLM -Jnone -gg	8.6
SkyBolt (i860 40 MHz)	Host	SKYvec F77	10
KSR1 ($1\times$KSR1 processor 20MHz)	KSR OS 1.0.6	KSR Fortran f77 -O2 -inline_from *.il	11

Table 1: Monte Carlo timing comparisons for `XYZDOS`

MORTRAN3 source code into FORTRAN ANSI standard 77 or 66 code. There is no vacillation regarding MORTRAN3. There are those who **HATE** it and those who **LOVE** it. If you prefer you can write all your code in FORTRAN although it will take you longer and it will be difficult to make your code readable. To compensate for all the mistakes you make you will have direct access to FORTRAN debugging tools. Experienced MORTRAN3 programmers write more readable, more compact code than is possible in FORTRAN. On the downside, it is more difficult to debug and you use debugging tools indirectly since references to FORTRAN statements are only indirectly related to the MORTRAN source.

To give you an example: The NRCC statistical analysis code takes about 100 lines of MORTRAN3 source (including comments) which translates into about 600 lines of FORTRAN (with no comments). This code was conceived, written and debugged in one working day. It is **IMPOSSIBLE** to conceive, write and debug 600 lines of FORTRAN code in one day.

It is also possible to write usercodes in any language (*e.g.* C, C++) and communicate with EGS using standard interface routines. The future may have EGS being distributed in standard C with a MORTRAN-like pre-compiler to include all the programming power inherent in MORTRAN.

2.5 Should I use non-standard FORTRAN?

Avoid non-standard FORTRAN coding *like the plague*. Although all compilers have FORTRAN-extensions that make coding faster and more readable, you will duplicate your efforts when you switch to a different machine. There are differences you can not avoid. They are:

- System timing calls like CPU time used, total elapsed time, time and date routines. Unfortunately, these are not standardised like mathematics subroutines. For example, VAX/VMS machines can obtain the date via the call
  `CALL DATE(DATEN)`
  where `DATEN` is a `CHARACTER*9` variable while IBM/CMS machines obtain the date via
  `CALL DATE(DAOFWK,DATEN)`
  where `DAOFWK` is a `CHARACTER*8` variable (day of the week) and `DATEN` is a `CHARACTER*8` variable. Note that in this case, unless you knew to make the change, the compiler would not complain (most FORTRAN's do not check subroutine interfaces) and you would get unexpected results.
- I/O operations are not standardised. For example, to open a file with a specified name in VAX/VMS one issues the call
  `OPEN(...NAME='filename'...)`
  and on IBM/CMS
  `OPEN(...FILE='filename'...)`.
  There are a number of non-standard qualifiers for both `OPEN`, `CLOSE`, `READ` and `WRITE` statements.
- Be aware of the differences in machine precision. For example, a PRESTA single precision convergence routine can work to 1 part in 10^7 on the VAX but only 5 parts in 10^7 on an IBM owing to differences in floating-point binary representations.
- Pseudo random number generation usually depends upon machine architecture. This topic is dealt with in detail in the next section.

If you use different machines it is easy to write macros that can covert your code to run on any machine and design user codes with this general application in mind.

The UNIX FORTRAN world has made generous concessions to VAX/VMS FORTRAN. Most UNIX-based FORTRAN compilers allow many of the VAX/VMS FORTRAN extensions (which are seductive in their simplicity and power). However, other conversions are not available (yet). Yet, the portability of standard ANSI code to other architectures should be motivation enough for adoption of standard coding practice.

3 Random number generators, state-of-the-art *ca.* 1988

The pseudo random number generator is the "soul" of a Monte Carlo calculation. It is what generates the pseudo-random nature of Monte Carlo simulations thereby imitating the true stochastic nature of particle interactions. Consequently, much mathematical study has been devoted to RNG's [3, 4]. The paper by Marsaglia [3] is the seminal work on the topic, allowing one to determine when one had a "good" RNG, while Knuth's book [4] reviews the field and supplies a set of good RNG's.

The operative phrase to be used is, "Use Extreme Caution."

- **DON'T FIDDLE** with RNG's unless you thoroughly understand the underlying mathematics and have the ability to test your new RNG completely.
- **DON'T TRUST** RNG's that come bundled with standard mathematical packages. Scientists have wasted **YEARS** of their life trying to understand bugs in their Monte Carlo code that were really due to bad RNG's.
- **DO USE** the RNG's that are distributed with the EGS4 code. They have been thoroughly tested. Employing "in-line" RNG's *à la* EGS4 will also save you about 20% CPU time. (Some modern compilers will automatically make subroutines "in-line" at high levels of compiler optimisation.) The overhead costs of function and subroutine calls is surprisingly large (about 20–100 machine cycles).

The gathering of random numbers into planes is a well known artefact of the type of RNG that EGS4 employs by default. Marsaglia's paper [3] describes how random numbers fall into $(n-1)$-hyperplanes when seeding n-dimensional hypercubes (for $n > 2$). An example of a bad RNG with this characteristic is shown in fig. 1. This RNG was actually distributed with standard mathematical packages provided with mainframe and mini-computers. However, this artefact can be seen only under special circumstances. If the view angle is rotated by only 10 degrees as in fig. 2, the artefact is not apparent. Good RNG's maximise the number of planes giving the illusion of randomness. The one distributed with EGS4 code gives over 1000 planes in a 3D cube.

3.1 Linear congruential RNG's

The standard method of random number generation is the linear congruential random number generator (LCRNG). It has the form:

$$X_n = (aX_{n-1} + b)_{\mathrm{mod}\ 2^k}, \tag{1}$$

producing a series of pseudo random integers, $X_1, X_2, X_3...$ starting from a "seed" X_0. The factor a is the constant multiplier, b is also a constant and k is usually the integer word size of the computer. If b is an odd number, the sequence length of this RNG is 2^k. Standard EGS4 sets $b = 0$. This version is called a multiplicative congruential (MC) RNG with a sequence length of 2^{k-2}. The constant multiplier a is a *magic* number. **DO NOT** change it unless you know what you are doing. There are guidelines for choosing potentially good candidates, but every potential multiplier has to be tested "experimentally". For $k = 32$, EGS4 employs

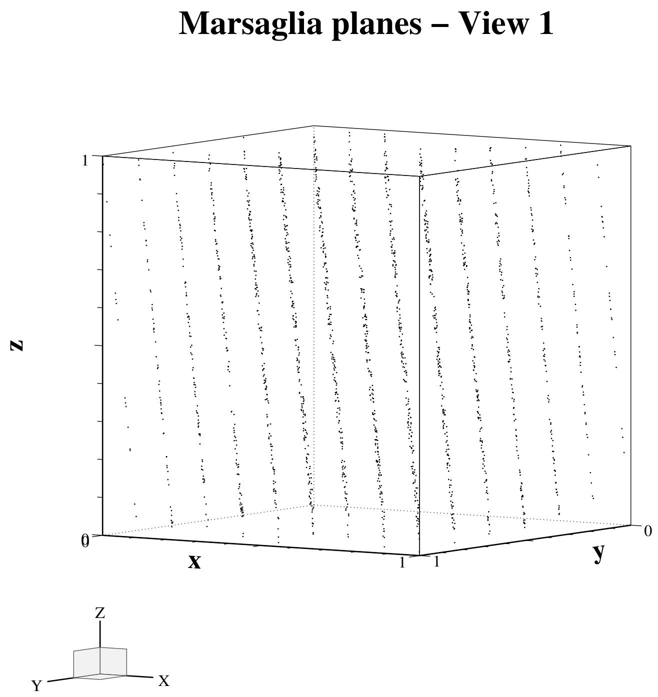

Figure 1: An example of the Marsaglia artefact produced with a catastrophically bad RNG. In this case a 3D unit cube was seeded and the data gathers into 15 planes.

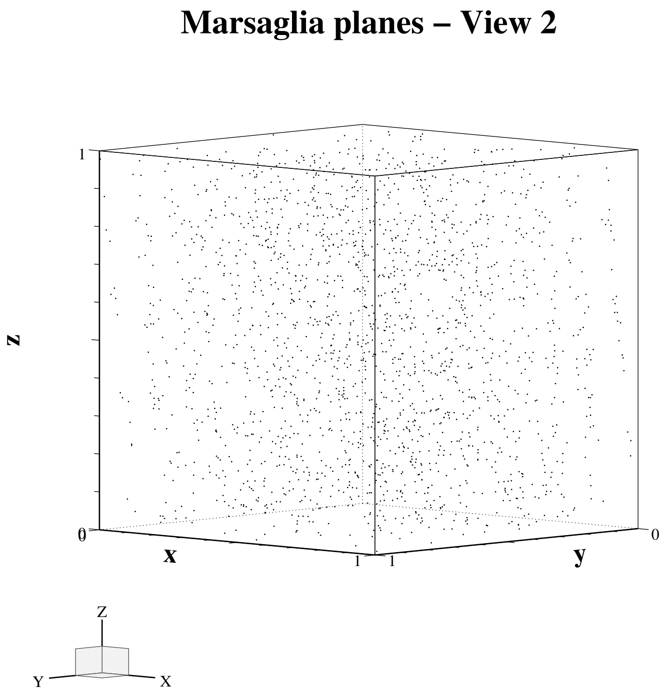

Figure 2: This is the same data as the previous figure only with a 10 degree difference in view angle.

a = 663608941. Knuth [4] claims that a = 69069 (much easier to remember) is best. A particularly bad one is a = 65539 which was employed to produce the data in figs. 1 and 2.

LCRNG's "use up" every integer represented by the computer in a well-defined order and then start over. MCRNG's "use up" half of the odd integers in sequence. Therefore, it is unwise to use up more than, say $1/10^{\mathrm{th}}$, of the sequence. For example, the last half of the sequence will be anticorrelated with the first half. If one requires more than 10^8 random numbers for a simulation (not an uncommon requirement) then one is well advised to employ long sequence random number generators described in the next sections.

Actually, a given Monte Carlo history may consume thousands of random numbers. It is important, if the RNG cycles, that no history starts with the same random number as a previous one. Rejection techniques, which "discard" random numbers, could "synchronise" a recycled RNG. It is difficult and expensive to test for this and much safer to employ a long sequence RNG.

3.2 Generic 32 and 64-bit RNG's

The following MORTRAN3 code will produce a 2^{30} (about 10^9) sequence of random numbers on any 32-bit computer with two's-complement integer circuitry (*i.e.* hard-wired integer arithmetic operations) allowing integer overflows to occur. (You may have to suppress certain classes of arithmetic exceptions. For example, on VAX/VMS one employs the `/CHECK=NOOVERFLOW` in the FORTRAN compilation command.)

```
"Initialisation:"
IXX=9876543421; "Default RNG seed, declare IXX in common block RANDOM"

"Iteration:"
IXX=IXX*663608941; "Common RANDOM must be declared where used"
RNG=0.5+IXX*0.23283064E-09;  "0.23283064E-09 IS 2**(-32)"
```

This RNG produces the following output on a VAX:

```
  IXX,OLD =    987654321      IXX,NEW =  -1879502499      RNG =   6.2394232E-02
  IXX,OLD =  -1879502499      IXX,NEW =     89393817      RNG =   0.5208136
  IXX,OLD =     89393817      IXX,NEW =   1262367013      RNG =   0.7939177
  IXX,OLD =   1262367013      IXX,NEW =  -1401290047      RNG =   0.1737368
  IXX,OLD =  -1401290047      IXX,NEW =    698583597      RNG =   0.6626517
  IXX,OLD =    698583597      IXX,NEW =  -2109875415      RNG =   8.7563396E-03
  IXX,OLD =  -2109875415      IXX,NEW =    217636469      RNG =   0.5506724
  IXX,OLD =    217636469      IXX,NEW =  -1336857135      RNG =   0.1887387
  IXX,OLD =  -1336857135      IXX,NEW =  -1956069379      RNG =   4.4567108E-02
  IXX,OLD =  -1956069379      IXX,NEW =   2070947001      RNG =   0.9821799
  IXX,OLD =   2070947001      IXX,NEW =    930943173      RNG =   0.7167521
  IXX,OLD =    930943173      IXX,NEW =   1414265313      RNG =   0.8292843
  IXX,OLD =   1414265313      IXX,NEW =  -1359510835      RNG =   0.1834642
  IXX,OLD =  -1359510835      IXX,NEW =   1756985161      RNG =   0.9090800
  IXX,OLD =   1756985161      IXX,NEW =   1050548245      RNG =   0.7445998
  IXX,OLD =   1050548245      IXX,NEW =    952334065      RNG =   0.7217326
```

```
IXX,OLD =    952334065     IXX,NEW =   -434396515     RNG =   0.3988592
IXX,OLD =   -434396515     IXX,NEW =   1816224473     RNG =   0.9228727
IXX,OLD =   1816224473     IXX,NEW =   1276396645     RNG =   0.7971843
IXX,OLD =   1276396645     IXX,NEW =    110212353     RNG =   0.5256608
IXX,OLD =    110212353     IXX,NEW =     -7475347     RNG =   0.4982595
IXX,OLD =     -7475347     IXX,NEW =   1595438953     RNG =   0.8714671
IXX,OLD =   1595438953     IXX,NEW =   2026694069     RNG =   0.9718765
IXX,OLD =   2026694069     IXX,NEW =    908182545     RNG =   0.7114527
IXX,OLD =    908182545     IXX,NEW =   -389793475     RNG =   0.4092441
IXX,OLD =   -389793475     IXX,NEW =    382223609     RNG =   0.5889934
IXX,OLD =    382223609     IXX,NEW =   1262111749     RNG =   0.7938583
IXX,OLD =   1262111749     IXX,NEW =    931116065     RNG =   0.7167923
IXX,OLD =    931116065     IXX,NEW =  -1665052659     RNG =   0.1123247
IXX,OLD =  -1665052659     IXX,NEW =   1944279945     RNG =   0.9526880
IXX,OLD =   1944279945     IXX,NEW =   -895236267     RNG =   0.2915616
IXX,OLD =   -895236267     IXX,NEW =    191403313     RNG =   0.5445645
IXX,OLD =    191403313     IXX,NEW =    803537885     RNG =   0.6870883
IXX,OLD =    803537885     IXX,NEW =   -912962791     RNG =   0.2874343
IXX,OLD =   -912962791     IXX,NEW =   2004199333     RNG =   0.9666390
IXX,OLD =   2004199333     IXX,NEW =   1566209857     RNG =   0.8646617
IXX,OLD =   1566209857     IXX,NEW =   -927193939     RNG =   0.2841209
IXX,OLD =   -927193939     IXX,NEW =    775070633     RNG =   0.6804602
IXX,OLD =    775070633     IXX,NEW =   2022784245     RNG =   0.9709662
IXX,OLD =   2022784245     IXX,NEW =    782527057     RNG =   0.6821963
IXX,OLD =    782527057     IXX,NEW =   1439728253     RNG =   0.8352129
IXX,OLD =   1439728253     IXX,NEW =   1154020665     RNG =   0.7686914
IXX,OLD =   1154020665     IXX,NEW =   1302124357     RNG =   0.8031745
IXX,OLD =   1302124357     IXX,NEW =    578190945     RNG =   0.6346206
IXX,OLD =    578190945     IXX,NEW =   -797381299     RNG =   0.3143452
IXX,OLD =   -797381299     IXX,NEW =    263114697     RNG =   0.5612612
IXX,OLD =    263114697     IXX,NEW =   1448330901     RNG =   0.8372158
IXX,OLD =   1448330901     IXX,NEW =   2062664561     RNG =   0.9802516
IXX,OLD =   2062664561     IXX,NEW =   -592890595     RNG =   0.3619569
IXX,OLD =   -592890595     IXX,NEW =   1544799065     RNG =   0.8596766
```

The employment of generic RNG's allows one to synchronise histories on different machines (insofar as differences in floating point precision allow this to happen), which may be useful for debugging purposes.

Some processors mock-up integer arithmetic using floating-point processors. These machines require special code. An example is given later for the discussion of the FPS-264.

If 64-bit two's-complement integer arithmetic were available, the following formula would serve as a generic 64-bit RNG:

```
IXX_64 = 6364136223846793005*IXX_64;
RNG_64 = 0.5 + IXX_64*5.421010862E-20; "5.421010862E-20 IS 2**(-64)"
```

This RNG produces a sequence length of 2^{62} (about 4.6×10^{18}). This sequence length should be long enough for any practical problem. This RNG requires machine dependent coding since `INTEGER*8` arithmetic is not standard FORTRAN. An example for the VAX is given following.

3.3 VAX RNG's

VAX employs the generic 32-bit RNG. The 64-bit version described above takes the form [5]:

```
"Declarations:"
COMMON/RANDM1/LNGXA1;
COMMON/RANDM2/LNGXB1;
COMMON/RANDM3/XAMSK;
COMMON/RANDM4/FFOUR,FTWO;
REAL*8 LNGXA1,LNGXB1,ZERO8;
INTEGER*4 LNGXA2(2),FFOUR(3),LNGXB2(2),WKREA1,WKREA2,WKREA3,XAMSK(3),IXX,JXX;
INTEGER*2 LNGXA4(4),FTWO(4),LNGXB4(4),XAMSK1,XAMSK2,XAMSK3;
EQUIVALENCE (LNGXA1,LNGXA2),(LNGXA2,LNGXA4);
EQUIVALENCE (LNGXB1,LNGXB2),(LNGXB2,LNGXB4);
EQUIVALENCE (LNGXB4(1),WKREA1);
EQUIVALENCE (LNGXB4(2),WKREA2);
EQUIVALENCE (LNGXB4(3),WKREA3);
EQUIVALENCE (XAMSK(1),XAMSK1);
EQUIVALENCE (XAMSK(2),XAMSK2);
EQUIVALENCE (XAMSK(3),XAMSK3);
EQUIVALENCE (XAMSK(1),ZERO8);
EQUIVALENCE (IXX,LNGXA2(1));
EQUIVALENCE (JXX,LNGXA2(2));

"Initialise the constant multiplier:"
DATA FFOUR/Z00007F2D,Z00004C95,Z0000F42D/;
DATA FTWO/Z7F2D,Z4C95,ZF42D,Z5851/;

"Initialise the seed:"
DATA IXX,JXX/987654321,987654321/;

"Select a new random number:"
ZERO8=0.0D0;
XAMSK(3)=0;
WKREA3=0;
XAMSK1=LNGXA4(1);
XAMSK2=LNGXA4(2);
XAMSK3=LNGXA4(3);
WKREA1=FFOUR(1)*XAMSK(1);
WKREA2=WKREA2+FFOUR(1)*XAMSK(2);
WKREA3=WKREA3+FFOUR(1)*XAMSK(3);
LNGXB4(4)=LNGXB4(4)+FTWO(1)*LNGXA4(4);
WKREA2=WKREA2+FFOUR(2)*XAMSK(1);
WKREA3=WKREA3+FFOUR(2)*XAMSK(2);
```

```
LNGXB4(4)=LNGXB4(4)+FTWO(2)*LNGXA4(3);
WKREA3=WKREA3+FFOUR(3)*XAMSK(1);
LNGXB4(4)=LNGXB4(4)+FTWO(3)*LNGXA4(2);
LNGXB4(4)=LNGXB4(4)+FTWO(4)*LNGXA4(1);
LNGXA1=LNGXB1;
RNG=0.5+0.23283064E-09*LNGXB2(2);
```

One can see that much calculation is required to simulate 64-bit integer arithmetic and handle overflow properly. Use of this RNG slows down a typical simulation by approximately 20%. As a rule of thumb, we used to employ this RNG for any simulation using more than about 10^6 histories. However, the lagged-Fibonacci RNG (described below) has supplanted the 64-bit RNG for all long-sequence calculations.

3.4 IBM RNG's

The 32-bit IBM RNG supplied with EGS4 makes elegant use of IBM's machine instructions for doing unnormalised floating point arithmetic. In effect, the following code prenormalises a floating point double precision number and scrambles the low-order 32-bits via integer random number generation.

```
"Declarations:"
INTEGER IXX; "IXX should reside in a common block"
INTEGER JX(2);REAL*8 DRN; "These are defined in the local subroutine"
EQUIVALENCE(JX(1),DRN);

"Initialisation:"
DATA JX(1)/Z46000000/;
IXX=987654321;

"Iteration:"
IXX=IXX*663608941;JX(2)=IXX;RNG=DRN+0.0D0;
```

This method employs four memory fetches, one integer multiplication, one double precision addition (to normalise the floating point number) and three stores. The IBM could as easily use the 32-bit generic method that requires four memory fetches, one integer multiplication, one integer-to-floating-point conversion, one floating point multiplication, one floating point addition and two stores. Clearly, the IBM special coding saves one numerical data type conversion and a floating point multiplication and is clearly more economical.

3.5 FPS-264 RNG

The FPS-264 is an example of a machine that does not have integer arithmetic. It is simply a double precision number cruncher. It does not even support single precision floating point operations. The FPS-264 FORTRAN handles integer operations like floating point operations within the 53-bit mantissa and the high order bits "spilled" like normal integer operations. (This is not documented in the FPS manuals!) The following RNG simulates 48-bit arithmetic employing the 48-bit multiplier times 32. This fills the 53-bit mantissa with the low order bits of the integer multiplication. The 53-bit integer is then normalised to fit in the range $0 <$`RNG`< 1 by multiplication by 2^{-53}. The random integer is then shifted so that the low order 48 bits of the integer multiplication are properly aligned.

```
"Declarations:"
INTEGER IXX; "Store in common. FPS treats as a REAL*8"

"Initialisation:"
IXX=987654321;

"Iteration:"
IXX=IXX*997353120; "This is Knuth's 48-bit multiplier times 32"
RNG=0.5+IXX*0.1110223024625157E-15; "0.1110223024625157E-15 is 2**(-53)"
IXX=IXX/32; "Shift to the right by 5 bits to align properly"
```

4 A universal random number generator, state-of-the-art since 1988

A new "universal" lagged-Fibonacci pseudo random number generator has been developed by Marsaglia, Zaman and Tsang [6, 7]. It will provide *identical* sequences on all machines that support single-precision real numbers with 24-bit fractional parts. The sequence length is 2^{144} (about 2×10^{43}), long enough for any practical calculation. The MORTRAN3 source code is:

```
"Declarations:"
REAL*4 U(97),C,CD,CM;INTEGER IXX,JXX;

"Initialisation:"
IF((IXX.LE.0).OR.(IXX.GT.31328)) IXX=1802; "SETS MARSAGLIA DEFAULT"
IF((JXX.LE.0).OR.(JXX.GT.30081)) JXX=9373; "SETS MARSAGLIA DEFAULT"
I = MOD(IXX/177,177) + 2;
J = MOD(IXX,    177) + 2;
K = MOD(JXX/169,178) + 1;
L = MOD(JXX,    169)    ;
DO II=1,97[
    S=0.0;T=0.5;
    DO JJ=1,24[
        M=MOD(MOD(I*J,179)*K,179);
        I=J;J=K;K=M;L=MOD(53*L+1,169);
        IF(MOD(L*M,64).GE.32) S=S+T;
        T=0.5*T;
        ]
    U(II)=S;
    ]
C  =   362436./16777216.;
CD =  7654321./16777216.;
CM = 16777213./16777216.;
IXX = 97;JXX = 33;

"Iteration:"
RNG=U(IXX)-U(JXX); IF(RNG.LT.0.) RNG=RNG+1.; U(IXX) = RNG;
```

```
IXX=IXX-1; IF(IXX.EQ.0) IXX=97;
JXX=JXX-1; IF(JXX.EQ.0) JXX=97;
C=C-CD; IF(C.LT.0.) C=C+CM;
RNG=RNG-C; IF(RNG.LT.0.) RNG=RNG+1.;
```

The salient features to note are:

The declarations define 100 real single-precision numbers and 2 integers. The "state" of this RNG is defined as the state of *all* these variables. Therefore, if you wish to restart a Monte Carlo run at some point you have to have stored the entire state of the RNG, not just one or two integer seeds. This is a minor price to pay for such a powerful RNG.

The initialisation code is best put into a separate subroutine. This initialisation is performed only once. It specifies the initial state of the RNG, all 102 numbers, based upon the two integer seeds, `IXX` and `JXX`. If a Monte Carlo run is restarted, there is no need to repeat the initialisation. `IXX` and `JXX` are restricted to the ranges $1 \leq \mathtt{IXX} \leq 31328$ and $1 \leq \mathtt{JXX} \leq 30081$. Unique pairs of `IXX` and `JXX` produce *independent* random number sequences. Thus, it is conceivable that one could run independent Monte Carlo runs on $31,328 \times 30,081 = 942,377,568$ computers and combine results at the end with the guarantee that each computer's result is independent of any other's. Besides the intriguing scenario of employing every computer in the world to perform one's application this feature could be employed in parallel or distributed computing environments or enable a joint Monte Carlo project to be carried out among 2 or more institutions, without the danger that simulations have been duplicated.

Note that the iteration loop involves no multiplications. (Modern architectures multiply and add equally fast, usually in one clock cycle.) Each iteration requires 3–6 real additions or subtractions, 2 integer subtractions, 1–3 assignments. Moreover, there is a great deal of independence among these instructions allowing compilers to make advantage of "pipelining" techniques for increased speed. On a typical computer this RNG takes only about 50% longer than the fastest MCRNG. Clearly, lagged-Fibonacci RNG's represent the future of random number generation.

5 Doing it in a UNIX environment

To obtain EGS for use in a UNIX environment, follow the instructions given in the lecture, "How to manage the EGS4 system".

6 Doing it on a PC

This section describes how you would get EGS4 started on a PC/386/387/486 with a DOS operating system. Much of the bootstrapping procedure is analogous for other machines.

1. Purchase a 32-bit compiler and 32-bit addressing operating system. We employ[1] :

[1] Sue Walker has informed me that she has installed and compiled EGS4 codes using Microsoft 32-bit Powerstation Fortran although timing benchmarks are not yet available.

FORTRAN and LINKER:
Lahey Fortran F77L-EM/32 Version 2.0
P.O. Box 6091, Incline Village, NV 89450-6091, U.S.A.
(702)831-2500

Operating system:
Lahey/Phar Lap
(comes with Lahey Fortran)

Note that this software requires at least 2MB (4 or more is better) of memory and DOS Version 3.0 or higher. The batch procedures distributed with PC/EGS4 refer to the above FORTRAN, LINKER and OS. If you have other ones you will just have to change the references to the compiler and linker in the various batch procedures. These are clearly indicated.

2. Obtain the latest PC/EGS4 by writing to:

   Sue Walker
   Lanzl Institute
   3876 Bridge Way North
   Suite 300
   Seattle, Washington 98103-7951
   U.S.A. Tel: 206 545 1141
   Fax: 206 545 1347

   A nominal fee will be charged for covering floppy disks and postage. The distribution will arrive on $3 \times 5\frac{1}{4}$" 1.2MB floppies.
3. Once you have received your floppies, fill out the form to register with W. R. Nelson that you are now an EGS4 user and to receive a **free** copy of the EGS4 manual. Ask for a MORTRAN3 manual as well.
4. Install the EGS4 system on your hard disk by copying `INSTALL.BAT` from Diskette #1 to the root directory where you want EGS4 located. Then type:
   `INSTALL`
   and follow directions. This installation procedure places the files in an organised directory structure. The recommended structure is:
   ... root directory for EGS4
   `...\EGS4` (home of the EGS4 system, batch procedures)
   `...\EGS4\APPENDIX` (text versions of SLAC-265 appendices)
   `...\EGS4\BENCHMRK` (Timing benchmark codes from NRCC)
   `...\EGS4\EXAMPLES` (User code examples from SLAC)
   `...\EGS4\TUTOR` (Tutorial codes from NRCC)
   `...\MORTRAN3` (home of the MORTRAN3 pre-compiler, batch procedures)
   `...\PEGS4` (home of PEGS4, batch procedures)
   `...\PEGS4\DAT` (output directory for data file output from PEGS4)

5. FORTRAN compile the `...\EGS4\MORTRAN3\MORTRAN3.FOR` source code employing the batch command procedure `\EGS4\MORTRAN3\MAKEMOR3.BAT`.
6. Bootstrap MORTRAN3 by running `...\EGS4\MORTRAN3\RAWTOHEX.BAT`. The produces a

Hexadecimal data file, `...\EGS4\MORTRAN3\MORTRAN3.DAT`, that looks like:

```
v....2C USER F77 11JUN85
00002884000013D9000013D9000000000001B6F10001B6F20001B6F1000179E2000179E300000005
00000000000000000000021100000000000000000000000000000003E800000000000000000000000005
00000001000000000000 1E3200000000000000000000000000000000000000000000000100000000
00000000000000040000065900000001000028830000000000000218000000500000019400000000
.
.
.
```

Delete the first line of non-hex data, saving the file as `...\EGS4\MORTRAN3\MORTRAN3.DAT`.

7. Using the batch procedure `...\EGS4\EGS4BCOM.BAT`, attempt to MORTRAN/FORTRAN/LINK some of the `...\EGS4\TUTOR\TUTORn.MOR` codes.
8. Build the PEGS4 execute module utilising the batch procedure `...\EGS4\MORTRAN3\MOR3BCOM.BAT`. This batch procedure may be employed to MORTRAN compile any MORTRAN source code. You may find it preferable to employ MORTRAN for non-EGS4 use.
9. Using the `...\EGS4\PEGS4\*.INP`'s, attempt to create data sets needed for the `...\EGS4\TUTOR\TUTORn` codes.
10. Run the `...\EGS4\TUTOR\TUTORn` codes using the `...\EGS4\EGS4BRUN.BAT` batch procedure.
11. Create your own user codes using `...\EGS4\TUTOR\TUTORn.MOR`, `...\EGS4\EXAMPLES\*.MOR`, or `...\EGS4\BENCHMRK\XYZDOS.MOR` as starting points.

References

[1] A.F. Bielajew and D.W.O. Rogers, *A standard timing benchmark for EGS4 Monte Carlo Calculations*, Medical Physics **19** 303 – 304 (1992).

[2] J.J. Dongarra, *Performance of various computers using standard linear equations software in a Fortran environment*, Comp. Arch. News **16** 47 (1988).

[3] G. Marsaglia, *Random numbers fall mainly in the planes*, Nat. Acad. Sci. **61** 25 – 28 (1968).

[4] D.E. Knuth, *The art of computer programming, Vol. II*, (Addison Wesley, Reading Mass.) (1981).

[5] A.F. Bielajew and D.W.O. Rogers, *RNG64—a 64-bit random number generator for use on a VAX computer*, National Research Council of Canada Report PIRS-0049 (1986).

[6] G. Marsaglia, A. Zaman and W.W. Tsang, *Toward a Universal Random Number Generator*, Statistics and Probability Letters **8** 35 – 39 (1990).

[7] G. Marsaglia and A. Zaman, *A New Class of Random Number Generators*, Annals of Applied Probability **1** 462 – 480 (1991).