ADMINISTRATIVE ORDER
NO. 2014-05-01

IN THE CIRCUIT COURT OF THE
NINTH JUDICIAL CIRCUIT, IN AND
FOR ORANGE COUNTY, FLORIDA

AMENDED ORDER GOVERNING THE JURY SELECTION PLAN, ORANGE COUNTY

WHEREAS, pursuant to Article V, section 2(d) of the Florida Constitution and section 43.26, Florida Statutes, the chief judge of each judicial circuit is charged with the authority and the power to do everything necessary to promote the prompt and efficient administration of justice; and

WHEREAS, pursuant to the chief judge's constitutional and statutory responsibility for administrative supervision of the courts within the circuit and to create and maintain an organization capable of effecting the efficient, prompt, and proper administration of justice for the citizens of this State, the chief judge is required to exercise direction, *see* Fla. R. Jud. Admin. 2.215(b)(2), (b)(3); and

WHEREAS, pursuant to section 40.001, Florida Statutes, the chief judge of each judicial circuit is vested with the overall authority and responsibility for the management, operation, and oversight of the jury system within his or her circuit; and

WHEREAS, pursuant to section 40.225(2), Florida Statutes, the chief judge of the circuit, if performing the duties of juror candidate selection as provided in section 40.02, Florida Statutes, shall submit a plan for the selection of juror candidates to the Chief Justice for approval. The plan must be reapproved whenever required by a change in the law, a change in the technical standards and procedures, or a change in the approved hardware or software used in the automated system of jury venire selection; and

WHEREAS, the Chief Judge of the Ninth Judicial Circuit performs the duties of juror candidate selection as provided in section 40.02, Florida Statutes; and

WHEREAS, it is necessary to implement changes to the jury management vendor, and technical standards and procedures, and in the approved hardware and software used in the automated system of jury venire selection in Orange County to facilitate an improved system consistent with technological advancements; and

WHEREAS, this automated system of jury venire selection for Orange County, as set forth in this Administrative Order, was approved and authorized by the Chief Justice on August 9, 2018, by Supreme Court of Florida Administrative Order No. AOSC18-40, attached as Attachment D.

NOW, THEREFORE, I, Frederick J. Lauten, pursuant to the authority vested in me as Chief Judge of the Ninth Judicial Circuit of Florida under Florida Rule of Judicial Administration 2.215, effective immediately, do order and establish the procedures and method in Orange County for the selection of persons for grand jury and petit jury service:

1. Vendor: The jury software vendor is Jury Systems Incorporated (JSI) which provides the jury management system (JMS) entitled "JURY + Web Generation."

2. Equipment: The JMS SQL database runs on a Cisco USC C220 M4 server. This server is located in the Orange County Courthouse complex which is a secured facility under Court Administration's control. There are both test and production instances of the JMS application and both run under Windows Server 2012 R2 and Microsoft SQL Server 2014.

   a. The main jury database is replicated to a secondary jury server at the Orange County Courthouse complex. This secondary server is an VMWare ESXi virtual server running Windows Server Datacenter Edition 2016. In addition to the replicated JMS database, this server will be configured as a secondary web/application server to execute the JMS application in the event of a failure of the main server.

   b. In the event of a network failure, users in the Orange County Courthouse will execute the application from the replicated JMS database.

   c. Nightly, Court Administration will fully back up the JMS database using Commvault Simpana backup software for Windows servers stored to both local and offsite disks. Backup disks are kept for one month.

d. All hardware and software associated with the jury application will be upgraded on an as needed basis.

3. Method for Selecting Venire:

a. The names shall be taken from:

1. A quarterly updated list of Florida Department of Highway Safety and Motor Vehicle (DHSMV) licensed drivers and identification card holders, 18 years of age or older, who are citizens of the United States, and legal residents of Florida residing in Orange County.

2. Persons filing affidavits pursuant to section 40.011, Florida Statutes.

b. The Clerk of Court for Orange County is designated the official custodian of the DHSMV list provided specifically for venire selection and shall ensure that it is not accessible to anyone other than those directly involved in the selection of venires, as herein provided.

c. The Court Administrator shall cause Orange County petit and grand jury venires to be selected from the DHSMV list programmed into Court Administration's computer network using the method described in Attachment A (Process for Maintaining and Updating Prospective Juror File) and Attachment B (Juror Selection Process) in accordance with directions received from the Chief Judge or the Chief Judge's designee. Court Administration jury personnel may draw the venires and perform any other functions allowed by statute.

4. Administrative Order 2014-05 is vacated and set aside except to the extent that it has been incorporated and/or amended herein. Vacating an Administrative Order that vacates a prior Order does not revive the prior Order.

DONE AND ORDERED at Orlando, Florida, this 14th day of August, 2018. Nunc pro tunc to August 9, 2018.

/s/
Frederick J. Lauten
Chief Judge

Copies provided to:

Clerk of Courts, Orange County
Clerk of Courts, Osceola County
General E-Mail Distribution List
http://www.ninthcircuit.org

Attachment A

Process for Maintaining and Updating Prospective Juror File

1. The Florida Department of Highway Safety and Motor Vehicles (DHSMV) sends an electronic file of licensed drivers and ID card holders to the Florida Association of Court Clerks (FACC). After separating the records into multiple files based on "county of residence" and excluding drivers and ID card holders under 18 years of age, the FACC transmits the appropriate county's file to the Clerk of Court for that county via email.

2. A software utility from Jury Systems Incorporated (JSI) is used to match and merge the new FACC file with existing juror records stored in the Jury Management System (JMS), an application also from JSI. The JMS database includes juror personal data, service history, and excusal status. Jurors who are temporarily or permanently excused are flagged as ineligible but not deleted from the database. The records from the FACC file are compared to the existing juror records in the JMS database using driver's license number. Where no match is found against either driver's license number or social security number, the FACC records are matched to the JMS records using the last name, suffix, first name, and date of birth as a set of matching criteria. Where a match is found, the addresses are compared and, if different, the DHSMV address replaces the JMS address. Records that exist in the FACC file but not in the JMS database, are added to the JMS database. Records that exist in the JMS database but not in the FACC file, are flagged as inactive in the JMS database unless they were created as a result of the filing of an affidavit pursuant to section 40.011, Florida Statutes. Records that are flagged as inactive remain in the JMS database indefinitely but are bypassed by the system during the jury pool selection process.

3. Maintenance is conducted on a regular basis to update juror records in accordance with section 40.022, Florida Statutes, (e.g., identifying convicted felons, deceased persons, and legally incapacitated persons, and processing them according to statute). New maintenance procedures will be developed and employed to comply with other relevant statutes when implemented, assuming that data and/or processes from external agencies are available.

4. Persons filing affidavits pursuant to section 40.011, Florida Statutes are added to the JMS database through an on-line process within two working days.

Attachment B

Juror Selection Process

1. The selection of candidates for weekly petit jury pools is done at least five weeks in advance of the reporting date. For Grand Jury, selection of candidates is done twice a year, three months in advance of the reporting date.

2. Using JMS, the Jury Pool Manager or his/her respective designees, enters the jury pool location and the number of jurors required (minimum of 250 per section 40.02, Florida Statutes) for the service date specified. No other information is supplied by the user. The JMS will select and summon the number of jurors requested. Data associated with the selection of a juror pool (e.g., date, number of jurors requested) is stored for future retrieval and reporting.

3. Prior to invoking the process for randomly selecting jurors, JMS determines the number of jurors previously postponed, deferred, or re-summoned to the service date specified and subtracts this number from the total number requested. The result is the number of jurors that JMS must randomly select from the juror database.

4. The JMS random selection process is then invoked for each jury pool requested. For this process, Jury Systems Incorporated uses a Universal Random Generator system, more fully detailed in Attachment C.

5. Prior to printing and sending summonses, juror addresses are processed by Peregrine Solutions software. Peregrine Solutions performs address verification, a process of checking an address to ensure that it is properly formatted and conforms to address structure standards. If Peregrine Solutions cannot resolve an address, it is left unchanged.

6. A jury summons, specifying the jury service location and date and time to appear, is then produced and mailed to each individual selected.

Attachment C

JURY SYSTEMS
INCORPORATED

JURY+ *Next Generation*

Universal Random Generator

Detailed Design & Functionality

Notice

Techniques and work product contained in this document are considered proprietary to Jury Systems Incorporated. They may not be revealed or released to any party without the express written consent of Jury Systems Incorporated.

This material may not be copied or reproduced in any form without the express written permission of Jury Systems Incorporated.

©October 2006

CONTENTS

1. Introduction 4

2. The Definition of the Universal Random Number Generator 4

3. Development of the Universal Random Number Generator 4

4. JURY+ use of the Universal Random Number Generator 5

5. Logic Specifications for the Universal Random Number Generator 6

6. Validation of the Universal Random Number Generator 7

I. APPENDIX A - Toward a Universal Random Number Generator By George Marsaglia and Arif Saman 8

1. Introduction

This document describes the theory and structure of the random number generator that is used by the JURY+ Jury Management System to perform those jury management business functions that require randomization.

The random number generator employed by the JURY+ software is the "Universal" generator which appeared in an article written by George Marsaglia and Arif Zaman who are part of the "Supercomputer Computations Research Institute and Department of Statistics" at The Florida State University, Tallahassee. Also contributing to the article was Wai Wan Tsang a member of the "Department of Computer Science" at the University of Hong Kong.

The article (titled: "Toward a Universal Random Number Generator" is included in its entirety as an appendix to this document.

2. The Definition of the Universal Random Number Generator

The Universal generator algorithm is a combination of a Fibonacci sequence (with lags of 97 and 33, and operation "subtraction plus one, modulo one") and an "arithmetic sequence" (using subtraction).

It passes ALL of the tests for random number generators and has a period of 2^{144} and is completely portable (gives bit identical results on all machines with at least 24-bit mantissas in the floating point representation).

The Universal random number generator employed by Jury Systems Incorporated in its JURY+ application software is a true, exact implementation of the algorithm defined in "Toward a Universal Random Number Generator" and thus all randomness tests for that process published in statistical literature applies to the JURY+ implementation.

3. Development of the Universal Random Number Generator

In June 2006, the Florida State AOC required that all randomization for purposes of jury selection be accomplished using the Universal Random Number generator described in an article titled "Towards a Universal Random Number Generator" by George Marsaglia. The Universal Generator is a combination generator. It combines two different generators, the first of which takes two user seed values, converts them into 4 seed values and generates a sequence of 97 random numbers. These number become "seed" values for the second random generator which uses them in a combination process to combine the series of random numbers, producing a "Universal" value.

Previously, Jury Systems Incorporated used the "Marsaglia" random number generator which is a feedback shift register (FSR) method to generate uniform random numbers between 0 and 1, inclusive. The method was named for and based upon the idea of George Marsaglia (1965) who developed a coupled random number generator called super duper. Super duper couples a multiplicative-congruential generator with an FSR

generator. This generator was subjected to extensive testing by Rand Laboratories and shown to pass all randomness tests for all sample sizes likely to be encountered in the JURY selection process. This routine is a 2-seed routine, in that each number in a random sequence is provided based on two seed values.

Using the referenced article and a published C-language implementation of the Universal random generator (both of which are included as an appendix to this document), Jury Systems Incorporated created a version of the routine for integration into its JURY+ application for use in Florida and any other site that may desire it. The JSI implementation was done in August of 2006 and is a COBOL version of the routine. A copy of the JSI implementation is also included in an appendix.

4. JURY+ use of the Universal Random Number Generator

Wherever randomness is requisite in the JURY+ application, the Universal generator is employed. Those application functionalities include the following:

- Source List Processing
  When source lists are processed to supply juror names to JURY+, each member of the list is assigned a random number. The list is then sorted by the random number (known as a Juror Identification Number - JID) and the first 'n' records are selected per client requirements.

- Juror Summonsing
  When it is necessary to summon jurors to a specific court and date, the full set of eligible jurors is assigned a random number. The list is sorted and the first 'n' number of jurors are selected.

- Panel Selection
  When requests for juror panels are received at the assembly room, the user initiates a computer program to create a panel of the requested size. The computer program provides a randomly ordered list of jurors available for service at that moment.

  Each juror in the pool is assigned a random number. Once all jurors have thus been assigned a temporary unique number, the list is ordered by that number. Once the panel jurors are selected from this list, the panel jurors are re-randomized and a Case Information Sheet listing them is produced assuring that each juror has an equal opportunity to be the first seated for voir dire.

- Reporting
  Many of the JURY+ reports allow the user to select list of jurors that are ordered randomly.

5. Logic Specifications for the Universal Random Number Generator

The Universal Generator is a combination generator in that it combines two different random generators to provide a random series that passes every randomness test. The principal component of the two has a very long period, about 10^{36}. It is a lagged-Fibonacci generator based on the binary operation x times y on reals x and y.

The Fibonacci generator has an extremely long period and appears to be suitably random based on results of stringent tests that were applied to it. However, there is one test which it fails: the "birthday-spacings test. In order to get a generator that passes all of the stringent tests the first generator is combined with a second generator.

The choice of the second generator is a simple arithmetic sequence for the prime modulus 2^{24} -3 = 16777213.

Detailed information regarding the theory behind the Universal Random number generator is provided in the published article included as appendix C of this document. The article provides a Fortran language version of the algorithm.

Sometime after the original article appeared, the Fortran program was converted into a "C" programming language implementation and published. The "C" version is included as an appendix to this document.

For implementation into JURY+, JSI developed a COBOL language implementation of the Universal generator by duplicating the logic published in the "C" program. The JSI version is also included in an Appendix to this document.

To the greatest extent possible, the variable names used in the JSI version directly correspond to identically named variables in the "C" implementation. This makes the comparison of the two sets of logic much more straight forward.

A review of the "COBOL" version shows that there are two entry points (distinct processes) in the Universal Algorithm. The first entry point is a routine "1000-set-seeds" (this corresponds to the subroutine called "RMARIN" in the "C" version.

The 1000-set-seeds routine implements the "first" random generator in the Universal process. Using two seed values supplied by the user, it creates four seed values and uses them to create a Fibonacci sequence of 97 random numbers. This series of random numbers is used in the creation of the Universal random number.

Additionally this routine creates a representation of a second sequence (initially set to 362436/16777216 and referenced by variables "C", "CD", "CM"). The Fibonacci series and this series are combined (in the random number generation routine below) to create a Universal Random number.

The second routine is 2000-Gen-Rand (this corresponds to the "RANMAR" routine in the "C" program). This routine generates a Universal Random number by combining the two sequences (series) set up in the 1000-set-seeds routine.

First, two entries from the Fibonacci series (referenced with variables "I" and "J") are subtracted from each other, (the first time a Universal random number is requested the two entries referenced are 97 and 33 respectively) giving the basis for our Universal random number.

After the basis calculated, it replaces the Fibonacci number referenced by "I" (in preparation for the next time a universal number is needed). Then, both references ("I" and "J") are decremented. When either of the reference indicators ("I" or "J") reach zero, they are reset to their initial value (97 and 33 respectively). Thus the series of 97 numbers is processed in a circular fashion. (All of this is in preparation for the next request for a Universal random number).

Finally, the next number in the second series (which was initialized in 1000-set-seeds and are represented by variables "C", "CD", and "CM") is computed and combined with the basis random number (via subtraction). The result is returned to the calling program as a "Universal" random number.

6. Validation of the Universal Random Number Generator

As indicated in the "Towards a Universal Random Number", the statistical "randomness" of this routine has been thoroughly tested and documented. It is also clearly explained that an appropriately coded algorithm, regardless of the language it is written in or the computer it is executed on, produces exactly the same "random" sequence when given identical seed values.

Thus, the validation (and thus proof of randomness) becomes one of showing that two different implementation produce the same known results when given appropriate seed values.

The JSI implementation produces the same results as the program on which it was modeled. The "C" version of the program indicates the following test to insure a properly functioning Universal random number generator algorithm:

```
Use IJ = 1802 & KL = 9373 to test the random number generator. The
 subroutine RANMAR should be used to generate 20000 random numbers.
 Then display the next six random numbers generated multiplied by
4096*4096
 If the random number generator is working properly, the random numbers
 should be:
           6533892.0  14220222.0   7275067.0
           6172232.0   8354498.0  10633180.0
```

These are exactly the results produced by calling the JSIRAND1 routine with seed value 1082 and 9373 and viewing the 20001 through 20006th random numbers.

I. APPENDIX A – Toward a Universal Random Number Generator By George Marsaglia and Arif Saman

Statistics & Probability Letters 8 (1990) 35–39
North-Holland

January 1990

TOWARD A UNIVERSAL RANDOM NUMBER GENERATOR

George MARSAGLIA and Arif ZAMAN

Supercomputer Computations Research Institute and Department of Statistics, The Florida State University, Tallahassee, FL 32306, USA

Wai Wan TSANG

Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong

Received December 1987
Revised June 1988

Abstract: This article describes an approach towards a random number generator that passes all of the stringent tests for randomness we have put to it, and that is able to produce exactly the same sequence of uniform random variables in a wide variety of computers, including TRS80, Apple, Macintosh, Commodore, Kaypro, IBM PC, AT, PC and AT clones, Sun, Vax, IBM 360/370, 3090, Amdahl, CDC Cyber and even 205 and ETA supercomputers.

Keywords: Random number generator.

1. Introduction

An essential property of a random number generator is that it produce a satisfactorily random sequence of numbers. Increasingly sophisticated uses have raised questions about the suitability of many of the commonly available generators (see, for example, Marsaglia, 1986) Another shortcoming in many, indeed most, random number generators is they are not able to produce the same sequence of variables in a wide variety of computers. Such a requirement seems essential for an experimental science that lacks standardized equipment for verifying results.

We address these deficiencies here, suggesting a combination generator tailored particularly for reproducibility in all CPU's with at least 16-bit integer arithmetic. The random numbers themselves are reals with 24-bit fractions, uniform on [0, 1). We provide a suggested Fortran implementation of this "universal" generator, together with suggested sample output with which one may verify that a particular computer produces exactly the same bit patterns as the computers enumerated in the abstract. The Fortran code is so straightforware that versions may be readily written for other languages; so far, correspondents have written or confirmed results for Basic, Fortran, Pascal, C, Modula II and Ada versions.

A list of desirable properties for a random number generator might include:

(1) *Randomness*. Provides a sequence of independent uniform random variables suitable for all reasonable applications. In particular, passes all the latest tests for randomness and independence.

(2) *Long period*. Able to produce, without repeating the initial sequence, all of the random variables for the huge samples that current computer speeds make possible.

(3) *Efficiency*. Execution is rapid, with modest memory requirements.

(4) *Repeatability*. Initial conditions (seed values) completely determine the resulting sequence of random variables.

(5) *Portability*. Identical sequences of random variables may be produced in a wide variety of computers, for given starting values.

(6) *Homogeneity*. All subsets of bits of the

0167-7152/90/$3.50 © 1990, Elsevier Science Publishers B.V. (North-Holland)

numbers must be random, from the most- to the least-significant bits.

2. Choice of the method

We seek a generator that has all of these desirable properties. (All? Well, almost all; the generator we propose falls short on *efficiency*, for it is slower than some of the standard, simple, machine-dependent generators. But all of the standard generators fail one or more of the stringent tests for randomness. See Marsaglia, 1986.)

Our choice is a combination generator. It combines two different generators. The principal component of the two has a very long period, about 10^{36}. It is a lagged-Fibonacci generator based on the binary operation $x \cdot y$ on reals x and y defined by

$$x \cdot y = \{\text{if } x \geqslant y \text{ then } x-y, \text{ else } x-y+1\}.$$

Ultimately, we require a sequence of reals on [0, 1): $U_1, U_2, U_3, \ldots$, each with a 24-bit fraction. We choose 24 bits because it is the most common fraction size for single-precision reals and because the operation $x \cdot y$ can be carried out exactly, with no loss of bits, in most computers—those with reals having fractions of 24 or more bits.

This choice allows us to use a lagged-Fibonacci generator, designated $F(r, s, \cdot)$, as the basic component of our universal generator. It provides a sequence of reals by means of the operation $x \cdot y$:

$$x_1, x_2, x_3, \ldots, \quad \text{with } x_n = x_{n-r} \cdot x_{n-s}.$$

The lags r and s are chosen so that the sequence is satisfactorily random and has a very long period. If the initial (seed) values, $x_1, x_2, \ldots, x_r$ are each 24-bit fractions, $x_i = I_i/2^{24}$, then the resulting sequence, generated by $x_n = x_{n-r} \cdot x_{n-s}$, will produce a sequence with period and structure identical to that of the corresponding sequence of integers

$$I_1, I_2, I_3, \ldots, \quad \text{with } I_n = I_{n-r} - I_{n-s} \bmod 2^{24}.$$

For suitable choices of the lags r and s the period of the sequence is $(2^{24}-1) \times 2^{r-1}$. The need to choose r large for long period and randomness must be balanced with the resulting memory costs: a table of the r most recent x values must be stored. We have chosen $r = 97$, $s = 33$. The resulting cost of 97 storage locations for the circular list needed to implement the generator seems reasonable. A few hundred memory locations more or less is no longer the problem it used to be. The period of the resulting generator is $(2^{24}-1) \times 2^{96}$, about 2^{120}, which we boost to 2^{144} by the other part of the combination generator, described below. Methods for establishing periods for lagged-Fibonacci generators are given in Marsaglia and Tsay (1985).

3. The second part of the combination

We now turn to choice of a generator to combine with the $F(97, 33, \cdot)$ chosen above. We are not content with that generator alone, even though it has an extremely long period and appears to be suitably random from the stringent tests we have applied to it. But it fails one of the tests: the birthday-spacings test. A typical version of this test goes as follows: let each of the generated values $x_1, x_2, \ldots$, represent a "birthday" in a "year" of, say, 2^{24} days. Choose, say, $m = 512$ birthdays, $x_1, x_2, \ldots, x_m$. Sort these to get $x_{(1)} \leqslant x_{(2)} \leqslant \cdots \leqslant x_{(m)}$. Form spacings $y_1 = x_{(1)}$, $y_2 = x_{(2)} - x_{(1)}$, $y_3 = x_{(3)} - x_{(2)}, \ldots, y_m = x_{(m)} - x_{(m-1)}$. Sort the spacings, getting $y_{(1)} \leqslant y_{(2)} \leqslant \cdots \leqslant y_{(m)}$. The test statistic is J, the number of duplicate values in the sorted spacings, i.e., initialize $J \leftarrow 0$ then for $i = 2$ to m, put $J \leftarrow J + 1$ if $y_{(i)} = y_{(i-1)}$. The resulting J should have a Poisson distribution with mean $\lambda = m^3/(4n) = m^3/2^{26}$.

Lagged-Fibonacci generators $F(r, s, \cdot)$ fail this test, unless the lag r is more than 500 or the binary operation $\cdot$ is multiplication for odd integers mod 2^k. The count J, the number of duplicate spacings, is only asymptotically Poisson distributed, requiring that n, the length of the year, be large. Applications of the birthday spacings test typically choose n to be 100000 or more—for example, using the leftmost 18 or more bits of the random number to provide a "birthday".

Detailed discussion of the test and test results will appear elsewhere, but here are results of a typical test on four different generators (see Table 1): two lagged-Fibonacci generators using subtrac-

Table 3
A birthday-spacings test for four generators

duplicate spacings	number expected	$F(97, 33, -)$ observed	$F(55, 24, -)$ observed	$F(97, 33, \cdot)$ observed	congruential observed
0	36.79	41	29	41	36
1	36.79	16	14	33	37
2	18.39	18	34	20	20
$\geqslant 3$	8.03	25	23	6	7
chi-square for 3 d.f.		48.1	56.91	1.53	0.29
probability of better fit		1.0000	1.0000	0.432	0.33

tion, a lagged-Fibonacci generator using multiplication on odd integers, and a popular congruential generator, $x_n = 69069x_{n-1}$, all for modulus 2^{32}. The leftmost 25 bits are used to choose one of 512 birthdays. Thus $n = 2^{25}$ and $m = 2^9$, so J should be Poisson distributed with $\lambda = m^3/(4n) = 1$. Of the four, only the $F(97, 33, \cdot)$ and the congruential generator pass. The two lagged-Fibonacci generators using subtraction fail the test. Their duplicate-spacing counts are far from Poisson distributed, and remain so, whatever the choice of seed values, (and for a wide variety of choices of n, m and lags r, s as well).

In order to get a generator that passes *all* the stringent tests we have applied, we have resorted to combining the $F(97, 33, \cdot)$ generator with a second generator. Combining different generators has strong theoretical support (see Marsaglia, 1986).

Our choice of the second generator is a simple arithmetic sequence for the prime modulus $2^{24} - 3 = 16777213$. For an initial integer I, subsequent integers are $I - k$, $I - 2k$, $I - 3k, \ldots,$ mod 16777213. This may be implemented in 24-bit reals, again with no bits lost, by letting the initial value be, say $c = 362436/17666216$, then forming successive 24-bit reals by the operation $c \circ d$, defined as

$$c \circ d = \{\text{if } c \geqslant d \text{ then } c - d, \text{ else } c - d + 16777213/16777216\}.$$

Here d is some convenient 24-bit rational, say $d = 7654321/16777216$. The resulting sequence has period $2^{24} - 3$, and while it is far too regular for use alone, it serves, when combined by means of the $\cdot$ operation with the $F(97, 33, \cdot)$ sequence, to provide a composite sequence that meets all of the criteria mentioned in the introduction, except for efficiency. All of the operations in the combination generator are simple and efficient, and the generation part is quite simple, but the setup procedure, setting the initial 97 x values, is more complicated than the generating procedure. We now turn to details of implementation.

4. Implementation

We have two binary operations, each able to produce exact arithmetic on reals with 24-bit fractions:

$$x \cdot y = \{\text{if } x \geqslant y \text{ then } x - y, \text{ else } x - y + 1\},$$

$$c \circ d = \{\text{if } c \geqslant d \text{ then } c - d, \text{ else } c - d + 16777213/16777216\}.$$

We require computer instructions that will generate two sequences:

$$x_1, x_2, x_3, \ldots, x_{97}, x_{98}, \ldots, \quad \text{with } x_n = x_{n-97} \cdot x_{n-33},$$

$$c_1, c_2, c_3, \ldots, \quad \text{with } c_n = c_{n-1} \circ (7654321/16777216).$$

Then produce the combined sequence

$$U_1, U_2, U_3, \ldots, \quad \text{with } U_n = x_n \cdot c_n.$$

The c sequence requires only one initial value, which we arbitrarily set to $c_1 = 362436/16777216$. The x sequence requires 97 initial (seed) values, each a real of the form $I/16777216$, with $0 \leqslant I \leqslant 16777215$. The main problem in implementing the universal generator is in finding a suitable way to set the 97 initial values, a way that is both random and consistent from one computer to another.

Table 2
Fortran subprograms for initializing and calling UNI

```
      SUBROUTINE RSTART (I, J, K, L)
      REAL U(97)
      COMMON /SET1/ U, C, CD, CM, IP, JP
      DO 2 II = 1,97
      S = 0.
      T = .5
        DO 3 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
3       T = .5*T
2     U(II) = S
      C = 362436./16777216.
      CD = 7654321./167777216.
      CM = 16777213./16777216.
      IP = 97
      JP = 33
      RETURN
      END
```

```
      FUNCTION UNI()
C***  FIRST CALL RSTART (I, J, K, L)
C***  WITH I, J, K, L INTEGERS
C***  FROM 1 TO 168, NOT ALL 1
C***  NOTE: RSTART CHANGES I, J, K, L
C***  SO BE CAREFUL IF YOU REUSE
C***  THEM IN THE CALLING PROGRAM.
      REAL U(97)
      COMMON /SET1/ U, C, CD, CM, IP, JP
      UNI = U(IP) - U(JP)
      IF(UNI.LT.0.) UNI = UNI + 1.
      U(IP) = UNI
      IP = IP - 1
         IF(IP.EQ.0) IP = 97
       JP = JP - 1
      IF(JP.EQ.0) JP = 97
      C = C - CD
      IF(C.LT.0.) C = C + CM
      UNI = UNI - C
      IF(UNI.LT.0) UNI = UNI + 1
      RETURN
      END
```

The $F(97, 33, -\text{ mod } 1)$ generator is quite robust, in that it gives good results even for bad initial values. Nonetheless, we feel that the initial table should itself be filled by means of a good generator, one that need not be fast because it is used only for the setup. Of course, we might ask that the user provide 97 seed values, each with an exact 24-bit fraction, but that seems too great a burden. After considerable experimentation, we recommend the following procedure:

Assign values bit-by-bit to the initial table $U(1), U(2), \ldots, U(97)$ with a random sequence of bits $b_1, b_2, b_3, \ldots$. Thus $U(1) = 0.b_1b_2 \ldots b_{24}$, $U(2) = 0.b_{25}b_{26} \ldots b_{48}$ and so on. The sequence of bits is generated by combining two different generators, each suitable for exact implementation in any computer: one a 3-lag Fibonacci generator using multiplication, the other an ordinary congruential generator for modulus 169.

The two sequences that are combined to produce bits $b_1, b_2, b_3, \ldots$, are:

$y_1, y_2, y_3, y_4, \ldots$,
  with $y_n = y_{n-3} \times y_{n-2} \times y_{n-1}$ mod 179,
$z_1, z_2, z_3, z_4, \ldots$,
  with $z_n = 53 z_{n-1} + 1$ mod 169.

Then b_i in the sequence of bits is formed as the sixth bit of the product $y_i z_i$, using operations which may be carried out in most programming languages: b_i = {if $y_i z_i$ mod 64 < 32 then 0, else 1}.

Choosing the small moduli 179 and 169 ensures that arithmetic will be exact in all computers, after which combining the two generators by multiplication and bit extraction stays within the range of 16-bit integer arithmetic. The result is a sequence of bits that passes extensive tests for randomness, and thus seems well suited for initializing a universal generator.

The user's burden is reduced to providing three seed values for the 3-lag Fibonacci sequence, and one seed value for the congruential sequence $z_n = 53 z_{n-1} + 1$ mod 169. For Fortran implementations (see Table 2) of the universal generator, we recommend that a table $U(1), \ldots, U(97)$ be shared, in (labelled) COMMON, with a setup routine, say RSTART(I, J, K, L) and the function subprogram, UNI( ), that returns the required uniform variate. An alternative approach is to have a single subprogram that includes an entry for the setup procedure, but not all Fortran compilers allow multiple entries to a subprogram. The seed values for

the setup are I, J, K and L. Here I, J, K must be in the range 1 to 178, and not all 1, while L may be any integer from 0 to 168. If (positive) integer values are assigned to I, J, K, L outside the specified ranges, the generator will still be satisfactory, but may not produce exactly the same bit patterns in different computers, because of uncertainties when integer operations involve more than 15 bits.

To use the generator, one must first call RSTART(I, J, K, L) to set up the table in labelled common, then get subsequent uniform random variables by using UNI( ) in an expression as, for example, in X = UNI( ) or Y = 2.*UNI( ) − ALOG(UNI( )), etc.

5. Verifying the universality

We now suggest a short Fortran program for verifying that the universal generator will produce exactly the same 24-bit reals that other computers produce. Conversion to an equivalent Basic, Pascal or other program should be transparent, but those who wish to may get the setup, generating and verification programs for various languages by writing to the authors.

Assume then that you have implemented the UNI routine with its RSTART setup procedure in your computer. Running the short program of Table 3, or an equivalent, should produce the output as shown in Table 4.

If it does, you will almost certainly have a universal random number generator that passes all the standard tests, and all the latest—more stringent—tests for randomness, has an incredibly long period, about 2^{144}, and, for given RSTART values I, J, K, L, produces the same sequence of 24-bit reals as do almost all other commonly-used computers.

Good luck.

Table 3

```
      CALL RSTART(12, 34, 56, 78)
      DO 6 II = 1, 20005
      X = UNI()
 6    IF(II.gt.20000)
        print 21, (MOD(INT(X * 16. * * I), 16), I = 1, 7)
21    FORMAT(8X, 7I3)
      END
```

Table 4

6	3	11	3	0	4	0
13	8	15	11	11	14	0
6	15	0	2	3	11	0
5	14	2	14	4	8	0
7	15	7	10	12	2	0

References

Marsaglia, G. (1986). A current view of random number generators. *Computer Science and Statistics: Proc. 16th Symp. Interface, Atlanta, March 1984* (Elsevier Science Publishers, Amsterdam).

Marsaglia, G. and L.H. Tsay (1985), Matrices and the structure of random number sequences, *Linear Algebra Appl.* **67**, 147–156.

Attachment D

Supreme Court of Florida

No. AOSC18-40

IN RE: JUROR SELECTION PLAN: ORANGE COUNTY

ADMINISTRATIVE ORDER

Section 40.225, Florida Statutes, provides for the selection of jurors to serve within the county by "an automated electronic system." Pursuant to section 40.225(2), the chief judge of the circuit shall submit to the Supreme Court of Florida a plan for the selection of juror candidates. Section 40.225(3), Florida Statutes, charges the Chief Justice of the Supreme Court with the review and approval of the proposed juror selection process, hereinafter referred to as the "juror selection plan."

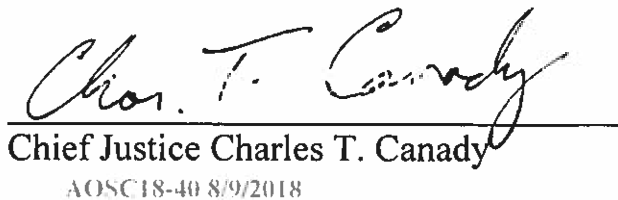
The use of technology in the selection of jurors has been customary within Florida for more than 20 years and the Supreme Court has developed standards necessary to ensure that juror selection plans satisfy statutory, methodological, and due process requirements. The Court has tasked the Office of the State Courts Administrator with evaluating proposed plans for compliance with those standards.

On July 20, 2018, the Chief Judge of the Ninth Circuit submitted the Orange County Juror Pool Selection Plan for review and approval in accordance with section 40.225(2), Florida Statutes. The proposed plan reflects changes to both hardware and software used for juror pool selection in Orange County.

The Office of the State Courts Administrator has completed an extensive review of the proposed Orange County Juror Selection Plan, including an evaluation of statutory, due process, statistical, and mathematical elements associated with selection of jury candidates. The plan meets established requirements for approval.

Accordingly, the attached Orange Juror Selection, received on July 20, 2018, from The Honorable Frederick J. Lauten, Chief Judge of the Ninth Circuit, is hereby approved for use.

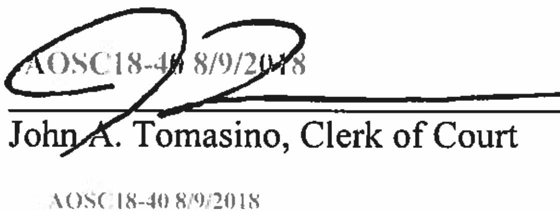
DONE AND ORDERED at Tallahassee, Florida, on August 9, 2018.



Chief Justice Charles T. Canady

AOSC18-40 8/9/2018

ATTEST:



John A. Tomasino, Clerk of Court

AOSC18-40 8/9/2018