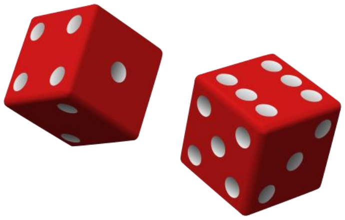
RANDOM_ROM Manual
HP-41 Module



Introduction and Credits.

Welcome to the Random ROM ("RANROM" for short), yet another HP-41 little adventure around the whimsical realm of aleatory confines, where we should have some fun learning about sequences of random numbers and how to determine the goodness of those mystifying little buggers, for the lack of a better term ;-).

As the name implies this module gathers a collection of routines and utilities about Random Numbers on the HP-41 platform. This includes a few MCODE pseudo-random number generators (p-RNG) from diverse sources like the (never released!) Toulouse Math ROM; Journal contributions (eventually coalescing into the the SandMath module); as well as a ported version of the Voyager implementation. As a second category, other MCODE and RPN-based p-RNGs are also included, both from the PPC/Datafile archives and Jean-Marc Baillard's collection.

Besides the expected set of utilities and small routines on the random topic, an attempt has been made to have a self-contained group of routines and programs to make the content as complete as possible - within the space constraints imposed by the ROM format. To that effect, a couple of UPL programs are included to evaluate the different p-RNGs in terms of their randomicity, normality, etc. Due credit is given to the original programmers in the respective sections of this manual.

Special thanks are due to Valentín Albillo who suggested numerous enhancements and additional subjects to include, such as Gaussian-distributed p-RNGs and others. He also provided critical feedback on several sections and steered the development towards a wider range of subjects that have no doubt shaped up the module for the better. Make sure you don't miss the adaptation of his brilliant "Mandelbrot Set Area Estimation" from his HP Collection.

On the programming side, thanks to Mark Power and Håkan Thörgren for their classic contributions in Datafile and PPC. Credit is also due to "Mike (Stgt)" for porting the HP's M-code featured in the Voyager series (HP-11C / HP-15C). And lastly, thanks to the programmers of the Toulouse Math ROM and the CCD Module for their seminal work on the subject.

Caveat Lector: Not being an expert on the field I have used this project as a learning vehicle myself, hoping that the discovery path and final result can also be interesting to other people as well. Whether it has worked or not I can say that at least I tried, but of course you are free to do your very own ;-)

Dependencies.

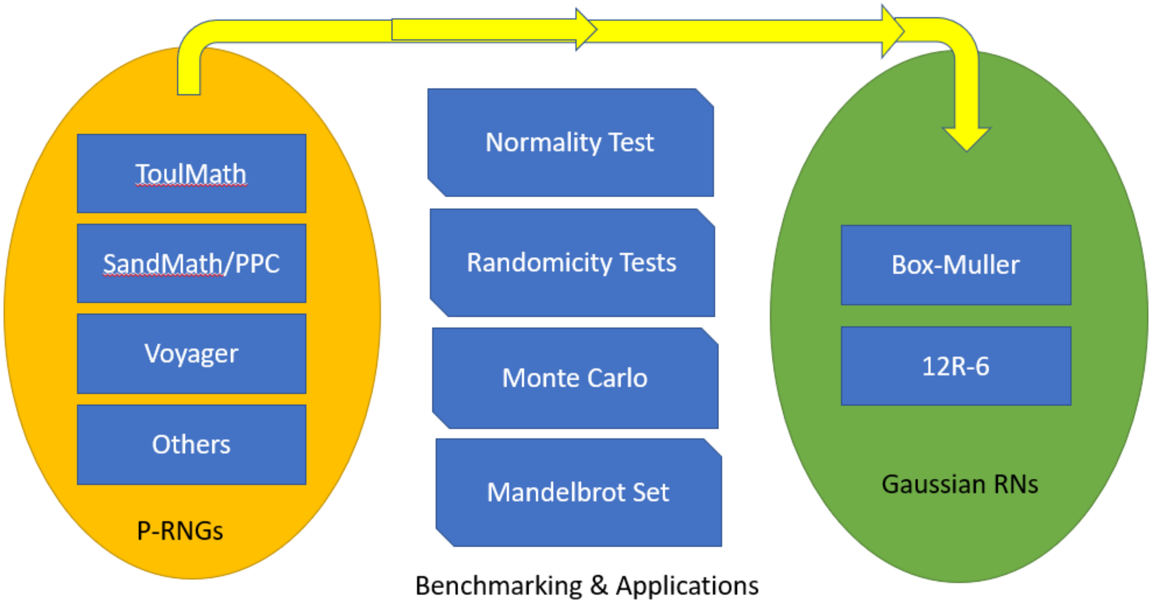
This ROM is designed for the HP-41CX O/S, obviously housed in Q-RAM-capable hardware devices like Clonix/NoVRAM, MLDL_2k and others. As a general rule n*o additional software dependency exists*, so it will also run on any CX-equivalent system such as the SY-41CL (with a TIME module) and the DM-41X. The exception to this rule is the Mandelbrot Set Area, which needs the 41Z Module - and in turn the Library#4 as well.

Without further ado, here is a list of the functions in the Main FAT table.

XROM#	Function	Description	Author
06.00	**-RANDOM 1C**	*Section header*	*n/a*
06.01	**SEEDT**	Sets initial Seed / Manual or Time-based	*Håkan Thörgren*
09.02	**RNDM**	RNG Sequence from SEEDT	*Håkan Thörgren*
06.03	**1RAN**	Time-based initial Seed	*JM Baillard*
06.04	**RAN00**	RNG from seed in R00	*Mark Power*
06.05	**RAN20**	RNG from seed in R20	*Mark Power*
06.06	**RANR _ _**	RNG from seed in prompt register	*Power-Martin*
06.07	**RCLSD**	Recalls Voyager Seed	*HP Co./ Mike (Stgt)*
06.08	**RN**	RN from SEED	*HP Co./ Mike (Stgt)*
06.09	**SEED**	Sets initial Seed	*HP Co./ Mike (Stgt)*
06.10	**RAND**	RNG from SEED	*Toulouse Math ROM*
06.11	**RANDXY**	RNG formatted by XY	*Toulouse Math ROM*
06.12	**STORAND**	Stores new initial seed	*Toulouse Math ROM*
06.13	**-MONTECARLO**	*Section header*	*n/a*
06.14	**AINT**	Append integer X	*Frits Ferwerda*
06.15	**IROUND**	Integer Round	*Ángel Martin*
06.16	**“MCE**	MC-based calculation for e	*Valentín Albillo*
06.17	**“MCPI**	MC-base calculation for p	*Albillo-Martin*
06.18	**“MCLN2**	MC-base calculation for Ln2	*Albert Chan*
06.19	**“MCLN2+**	Driver for MCLN2	*ÁM*
06.20	**“MBA**	Mandelbrot Set Area Estimation	*Albillo-Martin*
06.21	**“MCITG+**	Driver for MCITG	*Ángel Martin*
06.22	**“MCITG**	MC Integration functions one variable	*ÁM*
06.23	**“FX**	Example function got MCITG – 1 variable	*ÁM*
06.24	**“MCITG3**	MC Multiple Integration (up to thee vars)	*Ángel Martin*
06.25	**“FXY**	Example function for MCITG2 – 2 variables	*ÁM*
06.26	**“FXYZ**	Example function for MCITG – 3 variables	*ÁM*
06.27	**“MCITGN**	MC Multiple Integration – n variables	*Greg McClure*
06.28	**-RANDOMNESS**	*Section header*	*n/a*
06.29	**BXMR**	Gaussian RNG using Box Muller	*Ángel Martin*
06.30	**ERF**	Error function	*Baillard-Martin*
06.31	**MREV**	Mantissa Digit Reversal	*Ángel Martin*
06.32	**RANG**	Gaussian RNG using “12R-6”	*Ángel Martin*
06.33	**ΣDGT**	Mantissa Digit Sum	*Ángel Martin*
06.34	**“EVAL**	Evaluation of p-RNGs	*L. H. Gilbert*
06.35	**“RNG1**	p-RNG Sequence 1	*JM Baillard*
06.36	**“RGN2**	p-RNG Sequence 2	*JM Baillard*
06.37	**“RNG3**	p-RNG sequence 3	*JM Baillard*
06.38	**“RNG4**	p-RNG Sequence 4	*JM Baillard*
06.39	**“STRAT**	Stratified Random Sampling	*ravi – MoHP forum*
06.40	**“BENCH**	Benchmarking Gaussian RNGs	*Ángel Martin*
06.41	**“TRANG**	Testing Gaussian RNGs	*Ángel Martin*
06.42	**“12R-6**	“12x minus 6” Method	*Ángel Martin*
06.43	**“BX-MR**	Box Muller Method	*Ángel Martin*

XROM#	Function	Description	Author
06.44	**“INDEX**	Deviation Index from Normal distribution	*Ángel Martin*
06.45	**“TSTRNG**	Testing Integer RNG’s	*Charles T. Tart*
06.46	**“CHI**	Chi-Square Test	*AUG 1982, 10 pgs.*
06.47	**“UC1**	Ulam’s Conjecture – V1	*Robert G. Wilson*
06.48	**“UC2**	Ulam’s Conjecture – V2	*Ward Edwards*
06.49	**“UC3**	Ulam’s Conjecture – V3	*Gerhard Kruse*

The contents in a nutshell:



SandMath Random Numbers

The first set of RNG functions is taken from the SandMath, which used versions of Håkan Thörngren's p-RNGs published in the PPC Calculator Journal. The functions are:

Function	Description	Input	Output
SEEDT	Sets initial seed	Value in X	Loaded in buffer
RNDM	Randon Mumber	Current RN in buffer	Next RN in the sequence

SEEDT takes the fractional part of the value in X as initial seed for the RN's sequence. If the value in X is zero then the function will use the actual Time & Date information to generate the seed, assuming of course that the Time Module is present (not a problem when using a HP-41CX).

The value is saved in the first register (right above the header) of **Buffer #9,** thus it's compatible with the OSX, SandMath and CCD versions of the same functionality.

On the other hand, **RNDM** generates the next random number in the sequence, which obeys to the following LCG rule (albeit using 0 for the final term, 'mod 0"):

$$r(k+1) = FRC [r(k) * 9{,}821 + 0.211327]$$

well-known to PPC members, since it was also used in the RN routine – albeit using a regular data register instead of a buffer for the actual storage of the sequence terms.

Variations on a theme.-

The formula above is good because it already provides the RNs in canonical form. i.e. their values are between 0 and 1 (this one excluded). It however shows a flagrant weakness in that the last three decimal digits are always zero. You can check this using the PPC ROM, the CCD Module or previous versions of the SandMath itself (*), as the three use the same LCG expression (Linear Congruent Generator).

In the RANROM I have used a different approach, simply overwriting the last three digits with the first three of the mantissa *in reversed order*. Now, you may argue (and probably will be right) that this isn't an orthodox way to tackle the problem but intuitively *it's got to be better* than the three-zero case, thus the choice was clear. (**)

<u>Examples:</u> using pi as initial seed calculate the first six RANs:

You'd key: PI, SEEDT, RNDM, RNDM, RNDM, RNDM, RNDM and RNDM

To obtain:

```
0.792782030,   0.123641032,
0.489903098,   0.549656095,
0.383831038,   0.815954052,
```

(*) The SandMath has since been patched with the same variation to remove the zeros.
(**) We'll make a more formal comparison between both cases later on, using the program "EVAL"

Toulouse Math ROM Random Numbers

The second set is a very interesting one, as it'll be determined by the comparisons made for the "benchmarking" sections later on. It's also interesting because the source was never released to the public - at least to my knowledge, pls. send feedback if you know better?

Function	Description	Input	Output
STORAND	Sets initial seed	Value in X	Loaded in buffer
RAND	Randon Number	Current RN in buffer	Next RN in the sequence
RANDXY	RN between [a, b]	Values in X,Y	Integer RN in [a,b]
RCLRAND	Gets current RN from buffer	Current RN in buffer	Current RN in X

Here's a short description for the functions in this group:

STORAND is used to store the initial seed in the buffer. It uses **Buffer #3,** storing the seed value in digits <9:0> *of the header register*. This conflicts with the system standards used across the board and may cause havoc if you use other buffer utilities. BFCAT in particular will partially overwrite the RN value, as it uses the buffer header digits <2:0>; so better not to use it to be safe.

RAND calculates the next RN in the sequence and stores its value as new seed

RCLRAND recalls the current seed – not changing its value in the buffer.

RANDXY provides a boundary for the resulting random number, which must be withing that interval.

This implementation is also different (*"C'est la France, vive la différence!" :-)* in that it uses a digit level algorithm (see chapter on this later) instead of an LCG approach to generate the sequence of RNs. There's a byte table in the code with values used to generate the RN digits according to the algorithm, still not sure how exactly but working on it as we speak.

Same Example using pi as initial seed:

```
0.379167113 ; 0.754274858;
0.260151981 ; 0.904492763;
0.798064282 ; 0.739234419;
```

And if now we want a RN between, say 12 and 17:

12, ENTER^, 17, **RANDXY** => `13.00000000`

This last function comes very handy for your Lottozahl needs (a.k.a "Lotería Primitiva") – assuming your faith in science goes that far of course ;-)

Voyager Random Numbers

The RANROM module includes a version of the Voyager implementation of random numbers capability, taken from the VORANOGE-2 ROM, prepared by *mike-stgt* and published here:

https://forum.hp41.org/viewtopic.php?f=13&t=428#p1279

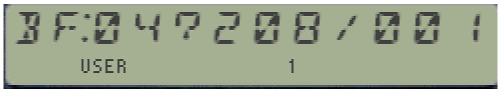
The use in the RANDOM module is covered under the Q Public license, see here for details. A copy of the license document is included as attachment to this manual.

The functions included are shown below:

Function	Description	Input	Output
SEED	Sets initial seed	Value in X	Loaded in buffer
RN	Randon Mumber	Current RN in buffer	Next RN in the sequence
RCLSD	Gets current RN from buffer	None	Current RN in X

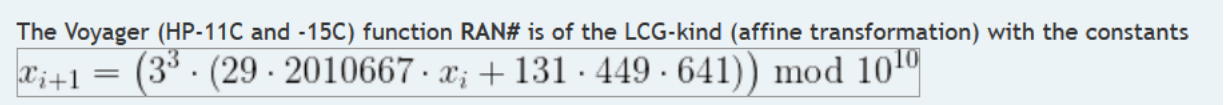
This set operates very much like the previous two cases for the SandMath and Toulouse Math ROM implementations. The values are stored in a buffer (with **id# = 4**) and like the Toulouse Math case – and contrary to the HP-41 default standards – the values are store in the buffer header itself, specifically in digits <9:0> (i.e. 10-digit mantissa format). This criterion was probably used by HP to save RAM memory, which was more at a premium in the 11C and 15C than in the HP-41C.

In addition to the conflict mentioned in the previous case with **BFCAT** (overwriting of the current RN), here it gets more interesting because it will show this buffer as non-committed, i.e. unclaimed by any module and thus will be erased by the OS next time the calculator does a power off/on cycle.



```
3F:04?208/00 1
     USER          1
```

With that out of the way, the actual RNG scheme is given by the expression below:



The Voyager (HP-11C and -15C) function RAN# is of the LCG-kind (affine transformation) with the constants

$$x_{i+1} = \left(3^3 \cdot (29 \cdot 2010667 \cdot x_i + 131 \cdot 449 \cdot 641)\right) \bmod 10^{10}$$

Which is another case of a Linear Congruent Generator (LCG) as defined here:
https://en.wikipedia.org/wiki/Random_number_generation

As far as the functions are concerned, we have a repeat (although chronologically speaking they came earlier) of the buffer-based design, with an initial seed function and a RAN instance for the elements of the sequence. **RCLSD** is a handy twist that can be used to know the current term in the sequence without generating the next one (for instance to make sure the seed is the desired one).

Same example again, with pi as initial seed:

```
0.898387113  ; 0.2343048558;
0.0573819 81 ; 0.1021112763;
0.778064282  ; 0.5144044 19;
```

Digit-Level Random Number Generator

The functions below were written by Mark Power, an old hand in MCODE and active DataFile member, with many remarkable feats under his belt – such as the MCODE Debugger and PLAY ROMs.

See *http://www.hp41.org/LibView.cfm?Command=View&ItemID=1471*
and *http://www.hp41.org/LibView.cfm?Command=View&ItemID=1472*

Function	Description	Input	Output
RAN00	RAN from seed in R00	Value in R00	Next RN in the sequence
RAN20	RAN from seed in R20	Value in R20	Next RN in the sequence
RANR _ _	RAN from sed in prompt reg	Value in Rnn	Next RN in the sequence

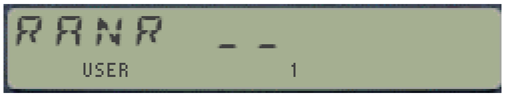
These functions were published in DataFile V6N8 p9. They expect the seed (or previous RN) stored in the corresponding data register. The result is placed in X and stored in the same register upon completion.

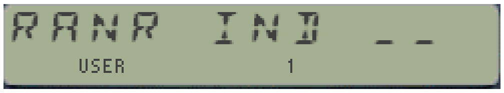
Examples. Storing pi in R00 we'll key:

PI, STO 00, RND00, RND00, RND00, RND00, RND00, RND00

To get: 0.3508848841; 0.5088048817; 0.1878607713;
0.8078081068; 0.4735445592; 0.0362417745;

RANR is a prompting function, accepting also INDirect registers – but not stack registers, sorry.


RANR _ _
USER 1


RANR IND _ _
USER 1

In manual mode just enter the register number where you seed is stored. In a running program it will take it from the X-register instead, so it'll be RANR IND X so to speak.

Time-Based Seed Generator

This function was written by Jean-Marc Baillard, see: *http://hp41programs.yolasite.com/alea.php*

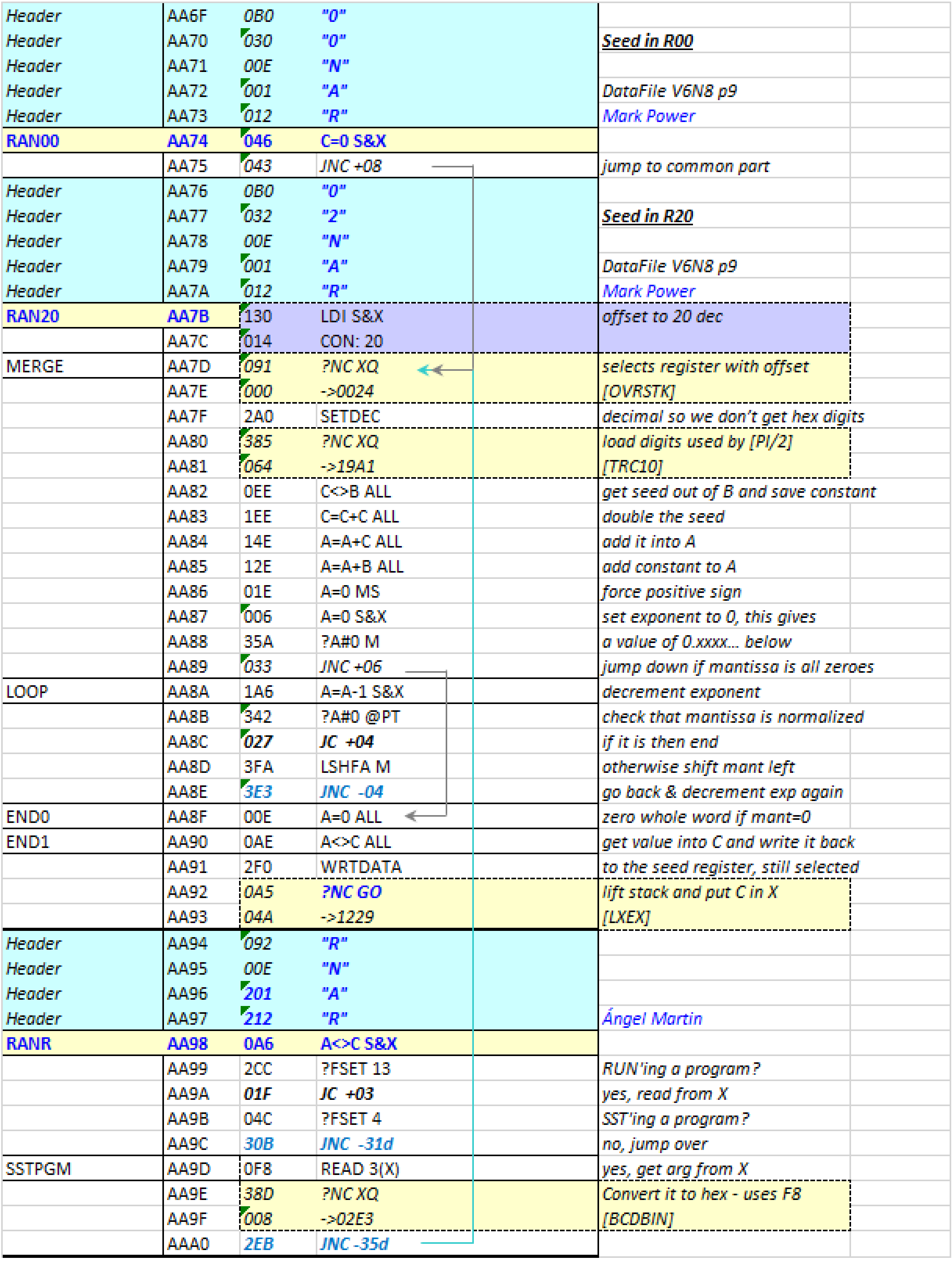
Use it to generate a time-based seed that can be used to generate a sequence of RN's by any of the methods reviewed so far, (perfect complement for RAN00/RAN20 indeed) and a few more still to cover in the manual.

Example:

XEQ "1RAN" => 0.3658253354

Note: don't use 1RAN to generate a sequence of RNs; doing so will generate skewed results due to the fairly similar "seeds" used – all taken from the internal Timer registers, which can be very similar in cases of TURBO running programs!

MCODE listing for RAN00, RAN20, and RANR.



Header	AA6F	0B0	"0"	
Header	AA70	030	"0"	Seed in R00
Header	AA71	00E	"N"	
Header	AA72	001	"A"	DataFile V6N8 p9
Header	AA73	012	"R"	Mark Power
RAN00	AA74	046	C=0 S&X	
	AA75	043	JNC +08	jump to common part
Header	AA76	0B0	"0"	
Header	AA77	032	"2"	Seed in R20
Header	AA78	00E	"N"	
Header	AA79	001	"A"	DataFile V6N8 p9
Header	AA7A	012	"R"	Mark Power
RAN20	AA7B	130	LDI S&X	offset to 20 dec
	AA7C	014	CON: 20	
MERGE	AA7D	091	?NC XQ	selects register with offset
	AA7E	000	->0024	[OVRSTK]
	AA7F	2A0	SETDEC	decimal so we don't get hex digits
	AA80	385	?NC XQ	load digits used by [PI/2]
	AA81	064	->19A1	[TRC10]
	AA82	0EE	C<>B ALL	get seed out of B and save constant
	AA83	1EE	C=C+C ALL	double the seed
	AA84	14E	A=A+C ALL	add it into A
	AA85	12E	A=A+B ALL	add constant to A
	AA86	01E	A=0 MS	force positive sign
	AA87	006	A=0 S&X	set exponent to 0, this gives
	AA88	35A	?A#0 M	a value of 0.xxxx... below
	AA89	033	JNC +06	jump down if mantissa is all zeroes
LOOP	AA8A	1A6	A=A-1 S&X	decrement exponent
	AA8B	342	?A#0 @PT	check that mantissa is normalized
	AA8C	027	JC +04	if it is then end
	AA8D	3FA	LSHFA M	otherwise shift mant left
	AA8E	3E3	JNC -04	go back & decrement exp again
END0	AA8F	00E	A=0 ALL	zero whole word if mant=0
END1	AA90	0AE	A<>C ALL	get value into C and write it back
	AA91	2F0	WRTDATA	to the seed register, still selected
	AA92	0A5	?NC GO	lift stack and put C in X
	AA93	04A	->1229	[LXEX]
Header	AA94	092	"R"	
Header	AA95	00E	"N"	
Header	AA96	201	"A"	
Header	AA97	212	"R"	Ángel Martin
RANR	AA98	0A6	A<>C S&X	
	AA99	2CC	?FSET 13	RUN'ing a program?
	AA9A	01F	JC +03	yes, read from X
	AA9B	04C	?FSET 4	SST'ing a program?
	AA9C	30B	JNC -31d	no, jump over
SSTPGM	AA9D	0F8	READ 3(X)	yes, get arg from X
	AA9E	38D	?NC XQ	Convert it to hex - uses F8
	AA9F	008	->02E3	[BCDBIN]
	AAA0	2EB	JNC -35d	

Other Random Number Generators for the HP-41

This section is taken from Jean-Marc Baillard web pages, see:
http://hp41programs.yolasite.com/alea.php

Overview

Several pseudo random number generators are listed on this web page:

"RNG1" "RNG2" "RNG3" work on every HP-41.
"RNG4" & the M-Code routine **1RAN** require a Time-Module.
Finally, the last program is an attempt to play (win?) the lottery...

Program #1 (35 bytes / SIZE 001)

A well-known RNG is given by the formula: Xn+1 = FRC (9821 Xn + 0.211327) which provides 1 million random numbers.

The following program gives 1,000,000,000 random numbers r (0 <= r < 1). The formula used is:
Xn+1 = FRC (98 Xn + 0.236067977)

The coefficient 98 = 43,046,721 may be replaced by a where a = 1 (mod 20)
and the value in line 0.236067977 may be replaced by b where b*109 is not divisible by 2 or 5.

```
01  LBL "RNG1"          14  *
02  9                   15  FRC
03  ENTER^              16  *
04  ENTER^              17  FRC
05  R^                  18  *
06  *                   19  FRC
07  FRC                 20  *
08  *                   21  FRC
09  FRC                 22  5
10  *                   23  SQRT
11  FRC                 24  +
12  *                   25  FRC
13  FRC                 26  END
```

STACK	INPUTS	OUTPUTS
X	xn	xn+1

Example:

0.2 XEQ "RNG1" yields 0.436067977
R/S 0.779021394 ... etc ...

Program #2 (26 bytes / SIZE 001)

"**RNG2**" provides 9,999,999,996 random numbers with the formula: Xn+1 = (1059 Xn) MOD p where p = 9,999,999,967 is the greatest prime < 1010

Xn are integers between 0 and p (exclusive) which are then divided by p to be reduced to a number between 0 and 1.

This routine works well because the MOD function gives exact results even when the operands are greater than 1010.

Actually, the exponent E59 in line 3 may be replaced by any integer m provided m is relatively prime to p-1 = 2*3*11*457*331543, but I don't know what the best choice is.

Unlike "RNG1" and other routines based upon the same type of formulae, the least significant digits don't go through any cycle of ten, one hundred and so on.

Register R00 is used to store the different xn integers.

```
01  LBL "RNG2"
 02  RCL 00
 03   E59
 04  *
 05  10
 06  10^X
 07  33
 08  -
 09  MOD
 10  STO 00
 11  LASTX
 12  /
 13  END
```

STACK	INPUTS	OUTPUTS
X	/	0 < r < 1

Example:

1 STO 00	XEQ "RNG2" gives	0.3129146797	R00 = 3129146787 = mod (1059, p)
	R/S	0.6904570204	R00 = 6904570181 ... etc ...

These ideas may be used to create your own RNG.

Actually if p is a prime, (Z/pZ-{0} ; *) is a group and if a is an integer, the number of distinct elements in the subset { 1 ; a ; a2 ; .... ; ak ; .... } (mod p) divides p-1

If p-1 is the smallest positive integer q such that aq = 1 (mod p) , then the sequence a ; a2 ; .... ; ak ; .... ; ap-1 (mod p) is a permutation of 1 ; 2 ; ...... ; p-1

In particular, if p = 2p' + 1 where p' is also a prime, and if ap' is not equal to 1 (mod p) then a satisfies the required property.

For instance, p = 7,841,296,787 = 2*3,920,648,393 + 1
7,841,296,787 and 3,920,648,393 are primes and -1024 = 4,851,307,369 (mod p)

satisfies (-1024)p' = -1 therefore the routine below gives 7,841,296,786 random integers
{ E24, *, CHS, 7841296787, MOD }

Program #3 (17 bytes / SIZE 001)

The following algorithm is given by Clifford Pickover in "Keys to Infinity" (John Wiley & Sons) ISBN 0-471-11857-5

```
01 LBL "RNG3"
02 LN
03  E2
04 *
05 1
06 MOD
07 END
```

STACK	INPUTS	OUTPUTS
X	xn	xn+1

Example:

0.1 XEQ "RNG3" produces 0.74149070
R/S 0.09073404 ... etc ...

Program #4 (25 bytes / SIZE 000)

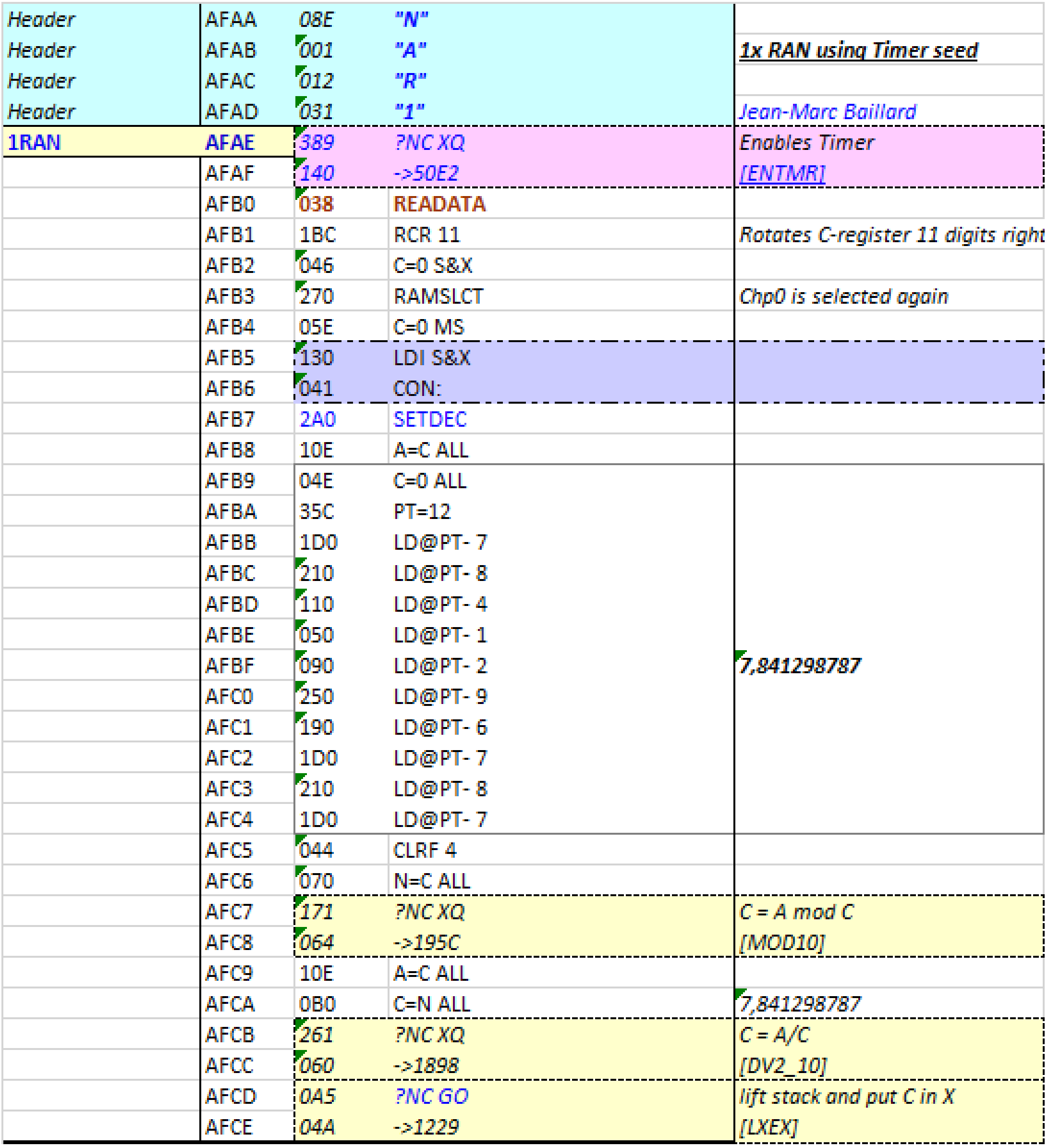
```
01 LBL "RNG4"
02 DATE
03 TIME
04 +
05  E49
06 *
07 PI
08 MOD
09 LN1+X
10 R-D
11 FRC
12 END
```

STACK	INPUTS	OUTPUTS
X	/	0 < r < 1

No examples can be provided since the result depends on the instant you press R/S

M-Code Routine

This M-Code routine uses the TIME module - or an HP41-CX



Header	AFAA	*08E*	**"N"**	
Header	AFAB	*001*	**"A"**	***1x RAN using Timer seed***
Header	AFAC	*012*	**"R"**	
Header	AFAD	*031*	**"1"**	*Jean-Marc Baillard*
1RAN	**AFAE**	*389*	*?NC XQ*	*Enables Timer*
	AFAF	*140*	*->50E2*	*[ENTMR]*
	AFB0	**038**	**READATA**	
	AFB1	1BC	RCR 11	*Rotates C-register 11 digits right*
	AFB2	046	C=0 S&X	
	AFB3	270	RAMSLCT	*Chp0 is selected again*
	AFB4	05E	C=0 MS	
	AFB5	130	LDI S&X	
	AFB6	041	CON:	
	AFB7	2A0	SETDEC	
	AFB8	10E	A=C ALL	
	AFB9	04E	C=0 ALL	
	AFBA	35C	PT=12	
	AFBB	1D0	LD@PT- 7	
	AFBC	210	LD@PT- 8	
	AFBD	110	LD@PT- 4	
	AFBE	050	LD@PT- 1	
	AFBF	090	LD@PT- 2	***7,841298787***
	AFC0	250	LD@PT- 9	
	AFC1	190	LD@PT- 6	
	AFC2	1D0	LD@PT- 7	
	AFC3	210	LD@PT- 8	
	AFC4	1D0	LD@PT- 7	
	AFC5	044	CLRF 4	
	AFC6	070	N=C ALL	
	AFC7	*171*	*?NC XQ*	*C = A mod C*
	AFC8	*064*	*->195C*	*[MOD10]*
	AFC9	10E	A=C ALL	
	AFCA	0B0	C=N ALL	*7,841298787*
	AFCB	*261*	*?NC XQ*	*C = A/C*
	AFCC	*060*	*->1898*	*[DV2_10]*
	AFCD	*0A5*	*?NC GO*	*lift stack and put C in X*
	AFCE	*04A*	*->1229*	*[LXEX]*

Monte Carlo Methods

This chapter covers the application of Monte Carlo method to do diverse calculations, like integrals of functions of several variables, Mandelbrot set Area estimation, and approximations for numerical constants, such as Ln2, pi and e. If there's something they all have in common it's their long execution time (ger ready to use V41 in TURBO mode) and the relative small accuracy of the results – unless many more iterations are done, which worsens the execution time, Be that as it may, this is an appropriate subject for the random topic so ready or not, here it comes.

1. Monte Carlo Integration.

The RANROM includes specific routines for the cases of functions of one, two, and three variables; as well as a general case for n-variables written by Greg McClure

Function	Description	Input/Output	Author
"MCITG+	Driver for MCITG	Prompts for params	*Ángel Martin*
"MCITG	MC Integration – One var.	FNAME in ALPHA	*Ángel Martin*
"FX"	Example function	n/a	*Á. Martin*
"MCITG3	MC Integration, up to 3 vars	Prompts for params	*Ángel Martin*
"FXY"	Example with 2-vars	n/a	*Á. Martin*
"FXYZ"	Example with 3-vars	n/a	*Á. Martin*
"MCITGN	MC Integration, n-vars	See below.	*Greg McClure*

Let's see a brief description for them, starting from the top.-

MCITG and **MCITG+** deal with integration of functions of a single variable (such as the example provided "FX"). It's the simplest case but also the most likely to be used thus the dedicated routine for it. The driver function does all the parameter prompting for the user, and then calls the main subroutine – which can also be run separately provided that the user sets all those parameters manually prior to the call.

The basic formula involves a repeat application of RNs {Xi}, at a very large scale:

$$\langle F^N \rangle = (b-a)\frac{1}{N}\sum_{i=0}^{N-1} f(X_i).$$

The program uses the SandMath-type p-RNG, i.e. functions **SEEDT** an **RNDM**.

So let's integrate *f(x) = x^2 . e^x* between [0,1] using different number of iterations to see how that influences the result.

```
XEQ "MCITG+"        a^b=?
0, ENTER^, 1, R/S   N=?
100, R/S            FNAME?           ; (ALPHA is turned ON)
"FX", R/S           0.6976706602     ; quite clearly we need more points…
```

If flag 10 is set the program will display a countdown showing the current iteration, decreasing to "1"before presenting the result.

Let's repeat the integration using 1000 iterations as follows:

1000 XEQ B => 0.716453287

The correct result is shown below:

$$\int_0^1 x^2 e^x \, dx = e - 2 \approx 0.718281828459045$$

MCITG3 can handle functions of up to three variables, thus the number of variables is also an input parameter to be entered – as well as the integration limits for each dimension and the number of iterations to run. The program will prompt for the input date so no need to set them up in advance. You can use **MCITG3** for functions of a single variable, although **MCITG** will be easier and slightly faster.

Let's use **MCITG3** to calculate an approximation of the provided functions **FXY** and **FXYZ**, between the intervals [0,1] in each dimension.

f(x,y) = x^2 + y^2

$$\int_0^1 \int_0^1 (x^2 + y^2) \, dy \, dx = \frac{2}{3} \approx 0.666667$$

XEQ "MCITG3" => DIM=? (1,2,3)
2, R/S => a?b(1)=?
0, ENTER^, 1, R/S => a?b(2)=?
0, ENTER^, 1, R/S => FNAME? ; (ALPHA is turned ON)
"FXY", R/S => #POINTS=?
1000, R/S => 0.732052382 , not quite good, try w/ more runs:

R/S (or XEQ B) => #POINTS=?
10,000 R/S => 0.680701267, , getting better…

R/S => #POINTS=?
100,000 R/S => 0.666765071

Very long execution times with slow "convergence" (for the lack of a better word), but surprising nonetheless… at least good to have as "the last resort" when everything else fails!

Going now for the 3D example:

f(x,y,z) = x^2 + y^2 + z^2

$$\int_0^1 \int_0^1 \int_0^1 (x^2 + y^2 + z^2)\, dz\, dy\, dx = 1$$

```
XEQ "MCITG3"        =>   DIM=? (1,2,3)
3, R/S              =>   a^b (1)=?
0, ENTER^, 1, R/S   =>   a^b (2)=?
0, ENTER^, 1, R/S   =>   a^b (3)=?
0, ENTER^, 1, R/S   =>   FNAME?
"FXYZ", R/S         =>   #POINTS=?
1000, R/S           =>   0.970860503 , not quite good, try w/ more runs:

R/S                 =>   #POINTS=?
10000, R/S          =>   0.964964521  - it didn't get better!
```

This is an unexpected result and *may* be related to the lackluster quality of the p-RNG. Further testing should be done using other pairs of { SEED, RAN } functions instead to see if that gets things back to the "logical" path, i.e. "the more number of points, the better accuracy".

Program listings.

```
01*LBL "MCITG+"
02*LBL A
03 "a^b=?"
04 PROMPT
05 "N=?"
06 PROMPT
07 "FNAME?"
08 AON
09 PROMPT
10 AOFF
11*LBL "MCITG"
12*LBL B
13 ASTO 00
14 STO 01
15 STO 05
16 RDN
17 X<>Y
18 STO 02
19 -
20 STO 03
21 0
22 STO 04
23 SEEDT
24*LBL 00
25 RNDM
26 RCL 03
27 ABS
28 *
29 RCL 02
30 +
31 XEQ IND 00
32 ST+ 04
33 FS? 10
34 VIEW 01
35 DSE 01
36 GTO 00
37 CLD
38*LBL C
39 RCL 04
40 RCL 05
41 /
42 RCL 03
43 *
44 RTN
45 GTO B
46*LBL "FX"
47 X^2
48 LASTX
49 E^X
50 *
51 END
```

```
01*LBL "MCITG3"
02*LBL A
03  E
04 STO 02
05 3
06 "DIM=?(1,2,3)"
07 PROMPT
08 X>Y?
09 GTO A
10 ST+ X
11 4
12 +
13  E3
14 /
15 5
16 +
17 STO 00
18*LBL 00
19 0
20 ENTER^
21  E
22 "a^b("
23 RCL 00
24 4
25 -
26 2
27 /
28 INT
29  E
30 +
31 AINT
32 RDN
33 "`)=?"
34 PROMPT
35 X<>Y
36 ST- Y
37 X<>Y
38 ST* 02
39 ABS
40 STO IND 00
41 ISG 00
42 X<>Y
43 STO IND 00
44 ISG 00
45 GTO 00
46 "FNAME?"
47 AON
48 STOP
49 AOFF
50 ASTO 01
51*LBL B
52 "#POINTS=?"
53 PROMPT
54 STO \
55 X<> 01
56 STO M
57 CLX
58 SEEDT
59 STO 03
60*LBL 03
61 RCL 00
62 FRC
63 5
64 +
65 STO 00
66*LBL 01
67 RNDM
68 RCL IND 00
69 *
70 ISG 00
71 RCL IND 00
72 +
73 ISG 00
74 GTO 01
75 XEQ IND M
76 ST+ 03
77 VIEW N
78 DSE N
79 GTO 03
80 RCL 03
81 RCL 02
82 *
83 RCL 01
84 /
85 RCL M
86 STO 01
87 RDN
88 "MCIT="
89 ARCL X
90 PROMPT
91 GTO B
92*LBL "FXYZ"
93 XEQ 05
94 X<>Y
95 X^2
96 +
97 RTN
98*LBL "FXY"
99*LBL 05
100 X^2
101 X<>Y
102 X^2
103 +
104 END
```

Note. The Central Limit Theorem establishes that the error in the calculation is proportional to 1/sqr(N), with N being the number of points used. This explains the poor accuracy results for the reduced sets used in the previous example. In practice we should be using N>= 1,000,000 for a decent approximation.

Finally, **MCITGN** is the general-case for functions of N-variables. Unfortunately there's no driver section (ran out of room in the ROM!), therefore all the parameters must be manually entered (yes, this can be onerous…) in the expected registers prior to calling the routine, as follows:

- ORDER SHOULD BE IN R00
- NUMBER OF ITERATIONS (N) SHOULD BE IN R01
- POINTER TO RANDOM REGISTERS WILL BE SAVED IN R02 FOR USERS
- NAME OF USER FUNCTION SHOULD BE IN ALPHA, IT WILL BE SAVED IN R03
- INTEGRAL SUM WILL BE SAVED IN R04 (AS WILL BE FINAL RESULT)
- LIMITS IN R05-R2N+4
- RANDOM VALUES FOR USER FUNCTION IN R2N+5 TO R3N+4

MCITGN was written by Greg McClure and posted on the MoHP Forum here: https://www.hpmuseum.org/forum/thread-6311.html?highlight=montecarlo

Let's see an example of a quintuple integral next.

f(x,y,z,u,v) = sqrt (6 - x^2 - y^2 - z^2 - u^2 - v^2)

```
; EXAMPLE OF QUINTUPLE INTEGRAL OF SQRT(6-X*X-Y*Y-Z*Z-U*U-V*V)
; X FROM 0 TO 0.7
; Y FROM 0 TO 0.8
; Z FROM 0 TO 0.9
; U FROM 0 TO 1.0
; V FROM 0 TO 1.1

; ALPHA = "5DINT"
; REGISTER 00 = 5
; REGISTER 01 = N (10, 100, AND 1000 USED FOR RUNS BELOW)
; REGISTERS 05, 07, 09, 11, 13 = 0
; REGISTER 06 = 0.7
; REGISTER 08 = 0.8
; REGISTER 10 = 0.9
; REGISTER 12 = 1.0
; REGISTER 14 = 1.1
```

```
01  LBL "5DINT"
02  6
03  RCL 15
04  X^2
05  RCL 16
06  X^2
07  RCL 17
08  X^2
09  +
10  +
    -
11  RCL 18
12  X^2
13  RCL 19
14  X^2
15  +
16  -
17  SQRT
18  END
```

```
; 9 RUNS FOR N=10:   1.150 1.229 1.193 1.179 1.193 1.194 1.204 1.189 1.174
; 6 RUNS FOR N=100:  1.192 1.190 1.179 1.193 1.187 1.192
; 4 RUNS FOR N=1000: 1.186 1.192 1.192 1.190
```

Commented Program Listing.

```
01 LBL "MCINT"
02 CF 21        ; AVIEW will show countdown.
03 CLX
04 STO 04       ; clear sum
05 SEEDT        ; Seed - randomize
06 ASTO 03      ; save user function name
07 RCL 01       ; save count in O(7)
08 STO O
09 RCL 00       ; get dimension
10 ST+ X        ; double for # of regs for limits
11 E3
12 /
13 E
14 +            ; convert to ISG value
15 4.004        ; bump to point to first LL
16 +
17 STO M        ; save in M(5)
18 RCL 00       ; create pointer to random regs.
19 E3
20 /
21 RCL 00
22 ST+ X
23 +
24 +
25 STO N        ; save in N(6)
26 STO 02       ; for user
; PRODUCE RANDOM VALUES
27 LBL 00
28 RCL IND M    ; LL(N) in X
29 ENTER^       ; LL(N) in X and Y
30 ISG M
31 RCL IND M    ; UL(N) IN X, LL(N) IN Y AND Z
32 X<>Y
33 -            ; UL(N) - LL(N) in X, LL(N) in Y
34 RNDM         ; RAND
35 *
36 +       ; rand value between LL(N)  UL(N) in X
37 STO IND N
38 ISG M        ; continue
39 STO X
40 ISG N        ; bump random reg. pointer
41 GTO 00       ; continue until done
42 RCL 00
43 ST- N        ; reset limits and values counters
44 ST- M
45 ST- M
; CALL USER FUNCTION, SUM IN R04
46 XEQ IND 03
47 ST+ 04
48 VIEW O
49 DSE O
50 GTO 00
; FINAL RESULT. ; CALC MULTIPLIER
51 1                    ; init multiplier
52  LBL 01
53  RCL IND M     ; LL(N) IN X
54  ISG M
55  RCL IND M   ; UL(N) in X, LL(N) in Y
56  X<>Y
57  -                   ; UL(N) - LL(N) in X
58  *                   ; new multiplier
59  ISG M
60  GTO 01
; MULTIPLY RESULT, DIVIDE BY NUMBER OF
POINTS
61  ST* 04
62  RCL 02
63  ST/ 04
; RESTORE ALPHA NAME, DISPLAY RESULT
64  CLA
65  ARCL 03
66  RCL 04
67  CLD
68  END
```

2. *Approximating Math Constants – Monte Carlo method*

This section uses a variation of the Monte Carlo strategy to evaluate both pi and e. It's not, however, based in circle relationships derived from randomly throwing needles or shooting at targets, but on probability theory instead. It was explained by Valentín himself in his *HP Challenge VA511 - 2020-03-14 - SRC 006 Pi Day 2020 Special.pdf*

Quoting directly from that article:

"It's quite simple, actually. My recent program is this:

```
1  DESTROY ALL @ RANDOMIZE 1 @ FOR K=1 TO 5 @ N=10^K @ S=0
2  FOR I=1 TO N @ IF NOT MOD(IROUND(RND/RND),2) THEN S=S+1
3  NEXT I @ P=S/N @ STD @ DISP N, @ FIX 3 @ DISP 5-P*4 @ NEXT K
```

which is computing the probability that the closest integer to A/B is even, where A and B are uniformly distributed random numbers in [0,1), as produced by the RND keyword. Each time the rounded value is even (i.e., it's 0 modulo 2) the number of favorable outcomes (S) is incremented by one (see line 2). After N tries have been sampled, the probability P for the even case will be the number of favorables outcomes (S) divided by the number of tries (N), thus we have the estimated probability P = S/N.
*But I know from theory that in the limit, for N -> Infinity, the exact probability P = (5-Pi)/4, so isolating Pi we have Pi = 5-P*4, which is displayed by the program in line 3 above."*

Note that he goes on to include yet another possible approach, which results in an even shorter BASIC program. Here's the explanation:

"Now, my earlier program, the one-liner, namely:

```
10 INPUT K @ N=0 @ FOR I=1 TO K @ N=N-MOD(IROUND(RND/RND),2) @ NEXT I @ DISP 1-4*N/K
```

is computing the probability that the closest integer to A/B is odd, where A and B are uniformly distributed random numbers in [0,1), as produced by the RND keyword. Each time the rounded value is odd (i.e., isn't 0 modulo 2) the number of favorable outcomes (N) is decremented by one, and after K tries have been sampled, the probability for the odd case will be the number of favorable outcomes (-N) divided by the number of tries (K), thus we have the estimated probability P = -N/K.

*As the probability of the rounded division being either even or odd is 1 (certainty), the probability for the odd case is 1 minus the probability for the even case, thus it's P = 1-(5-Pi)/4 = (Pi-1)/4, so isolating Pi we have Pi = 1+4*P = 1+4*(-N/K) = 1-4*N/K, which is then displayed by the one-line program."*

I chose to use the first approach in this module, partially because it also requires the IROUND function, and I was intrigued by it. I ended up writing a short MCODE utility for that purpose, which facilitates the porting of the BASIC code to HP-41 FOCAL, shown in next page.

With regard to the e calculation, the source has also been Valentín's *HP Challenge VA030 - Short Sweet Math Challenge 25 San Valentin Special - Weird Math.pdf.* In that thread there's one section (the first "concoction") about calculating a "weird limit" that can be used for the calculation of e (making the sum--to-exceed s=1).

"The limit average count for the sum of a series of [0,1) uniformly distributed random numbers to exceed 1 is exactly e = 2.71828182845904523536+, the base of the natural logarithms, which is pretty "weird" and can be considered an analog of Buffon's Needle experiment to estimate the value of Pi. Here we don't throw needles on a grid but merrily add up random numbers keeping count and we get e instead."

"This is the general formula to numerically compute the theoretically exact value and my simple 1-line, 53-byte HP-71B program to instantly compute them given the sum to exceed: "

$$f(x) = \sum_{k=0}^{[x]} (-1)^k \frac{(x-k)^k}{k!} e^{x-k}$$

```
1 DESTROY ALL @ INPUT X @ S=0 @ FOR K=0 TO IP(X) @ S=S+(K-X)^K/FACT(K)*EXP(X-K) @
NEXT K @ DISP S
```

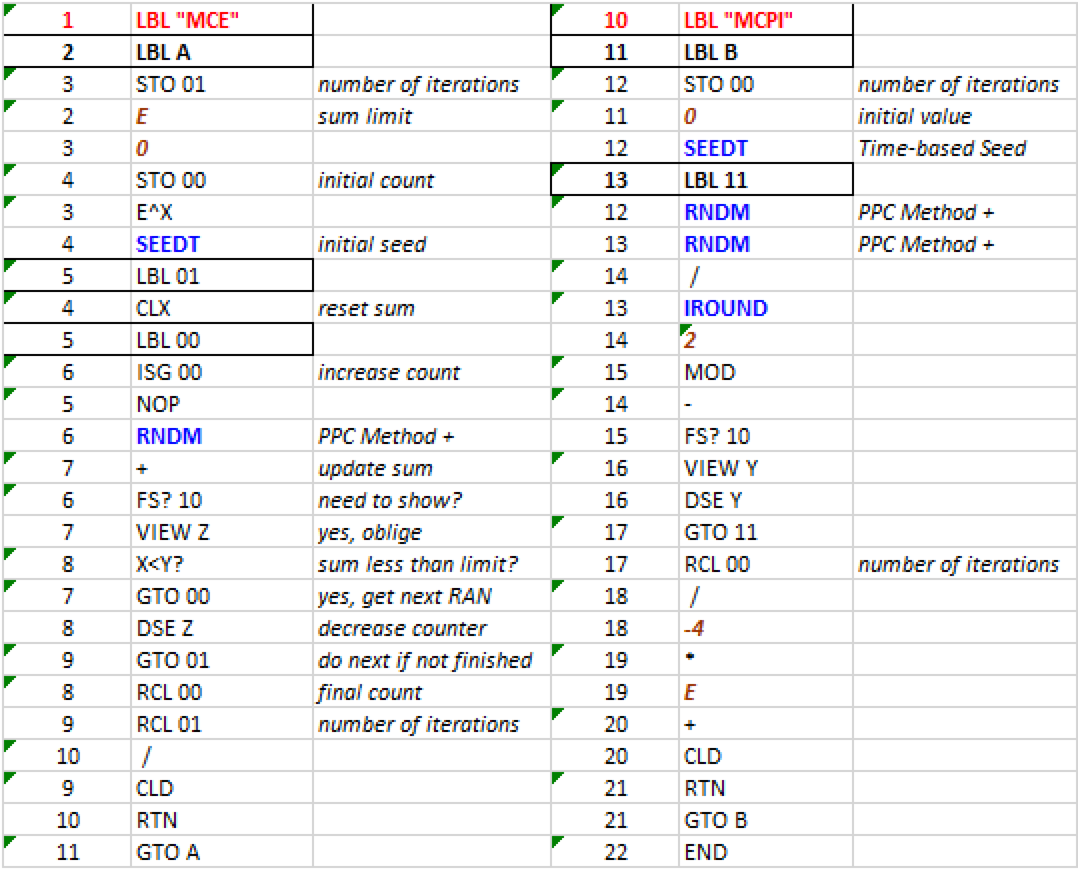
For the porting we'll certainly need the new **IROUND** utility and obviously capable random number capabilities, which shouldn't be much of a problem using the SandMath's functions **SEEDT** and **RNDM**. E'll use a time-generated initial seed (input zero for SEEDT), and RNDM will do the work using the well-known RNG recurrence:

$$r(k+1) = FRC [r(k) * 9{,}821 + 0.211327]$$

A few results are given in the table below:

Iterations	MCE	MCPI
10	2.800000000	3.000000000
100	2.850000000	3.120000000
1,000	2.705000000	3.136000000
10,000	2.717400000	3.131600000
100,000	2.717760000	3.149320000
1,000,000		

As you can see from the table results above both routines require a very large number of iterations to get to a reasonably accurate result, which of course was expected as "it 'comes with the territory" when resorting to this type of approaches. See below for the actual program code.



1	LBL "MCE"		10	LBL "MCPI"	
2	LBL A		11	LBL B	
3	STO 01	*number of iterations*	12	STO 00	*number of iterations*
2	E	*sum limit*	11	0	*initial value*
3	0		12	SEEDT	*Time-based Seed*
4	STO 00	*initial count*	13	LBL 11	
3	E^X		12	RNDM	*PPC Method +*
4	SEEDT	*initial seed*	13	RNDM	*PPC Method +*
5	LBL 01		14	/	
4	CLX	*reset sum*	13	IROUND	
5	LBL 00		14	2	
6	ISG 00	*increase count*	15	MOD	
5	NOP		14	-	
6	RNDM	*PPC Method +*	15	FS? 10	
7	+	*update sum*	16	VIEW Y	
6	FS? 10	*need to show?*	16	DSE Y	
7	VIEW Z	*yes, oblige*	17	GTO 11	
8	X<Y?	*sum less than limit?*	17	RCL 00	*number of iterations*
7	GTO 00	*yes, get next RAN*	18	/	
8	DSE Z	*decrease counter*	18	-4	
9	GTO 01	*do next if not finished*	19	*	
8	RCL 00	*final count*	19	E	
9	RCL 01	*number of iterations*	20	+	
10	/		20	CLD	
9	CLD		21	RTN	
10	RTN		21	GTO B	
11	GTO A		22	END	

Note:- The poor-man version of **IROUND** would consist of setting FIX 0 before the LBL 11 loop, and adding an INT instruction after the division of both random numbers (i.e. replacing IROUND with INT). That's almost equivalent but doesn't handle the EVEN condition for the result, i.e. IROUND(5.5)=5 whereas INT(4.5) in FIX 0 is equal to 4 instead. Not a show-stopper though, considering how unlikely it is to find such an occurrence amongst the hundreds of random points used by the routine.

One more for the road.

And as the adagio goes "*there's never two without three*", so let's add a third constant to this section – namely Ln 2 – another proud member of the irrational family.

The following is taken from this Albert Chan's *post in the MoHP forum.*

He uses another condition from probability, not so well-known:

LN(2) = 2 * probability of integer part of RND/RND is odd

```
10 INPUT K @ N=0 @ FOR I=1 TO K @ N=N+MOD(IP(RND/RND),2) @ NEXT I @ DISP 2*N/K @ GOTO 10
```

Moreover, we can improve LN(2) estimate by scaling RND/RND:

```
> 10 DEF FNL(K) @ N=0
> 20 FOR L1=1 TO K @ N=N+MOD(IP(10*RND/RND),2) @ NEXT L1
> 30 FNL=N/(5*K)+1501/2520 @ END DEF
```

See below the HP-41 version of the code. The driver program just adds the prompts for the initial seed and the number of points, then fall into the main subroutine – that expects N in X and the seed in Y registers.

1	LBL "MCLN2+"		20	RNDM
2	0		21	/
3	"SEED=?"		22	E1
4	PROMPT		23	*
5	STO 01		24	INT
6	"N=?"		25	2
7	PROMPT		26	MOD
8	GTO 01		27	+
9	LBL "MCLN2"		28	DSE Y
10	0		29	GTO 00
11	STO 01		30	RCL 00
12	RDN		31	5
13	LBL 01		32	*
14	STO 00		33	/
15	RCL 01		34	1501
16	SEEDT		35	ENTER^
17	CLX		36	2520
18	LBL 00		37	/
19	RNDM		38	+
			39	END

Examples:

PI , 500 , XEQ "MCLN2" => 0.69683492 1

3. *Mandelbrot set Area estimation*

Saving the best for last, here is a brilliant example of RN's utilization provided by Valentín Albillo's excellent articles on the estimation of the Mandelbrot set area on the HP-42 and Free42 (see here: *HP Article VA040a - Boldly Going - Mandelbrot Set Area (42S).pdf)*

Quoting sections or copying parts of that article is bound to do the reader and the article itself a huge disservice, so you're encouraged to read the original – included in this manual in its entirety. Thanks to Valentín for graciously granting permission to do so.

Porting it to the HP-41 platform was relatively straight-forward, once the function set was enhanced to deal with the required utilities. Obviously the HP-41 has its own limitations compared to the HP-42S and more so to Free42, however it does a good-enough job aided by the **41Z_Complex Number Module**, needed for the complex math functions required by the program.

Here's the program listing on the HP-41 w/ the 41Z Module.

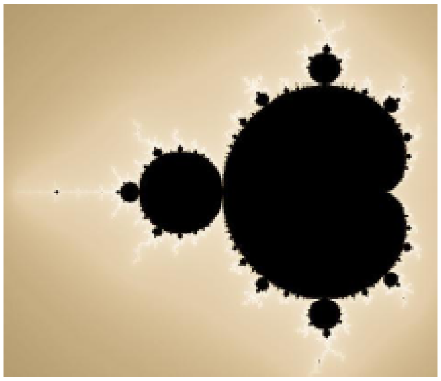
```
01*LBL "MBA"
02 2.5
03 STO 06
04 2
05 STO 07
06 1.2
07 STO 08
08 .25
09 STO 09
10 1
11 SEEDT
12 "POINTS=?"
13 PROMPT
14 STO 04
15 STO 00
16 256
17 "#ITERS=?"
18 PROMPT
19 STO 05
20 CLX
21 STO 02
22 "EVERY=?"
23 PROMPT
24 STO 03
25 CF 21
26 "WORKING..."
27 AVIEW
28 CF 00
29 X#0?
30 SF 00
31*LBL 00
32 RCL 05
33 STO 01
34 FS? 00
35 XEQ 03
36 RNDM
37 RCL 06
38 *
39 RCL 07
40 -
41 RNDM
42 RCL 08
43 *
44 X<>Y
45 ZRPL^
46 ZSIGN
47 ZENTER^
48 RCL 07
49 -
50 Z-
51 ZMOD
52 RCL 09
53 *
54 Z<>W
55 ZMOD
56 X<>Y
57 RDN
58 X<Y?
59 GTO 02
60 SIGN
61 ZRUP
62 RCL Z
63 -
64 ZMOD
65 RCL 09
66 X>Y?
67 GTO 02
68 ZRUP
69 ZRPL^
70*LBL 01
71 Z^2
72 Z+
73 ZMOD
74 RCL 07
75 X<=Y?
76 GTO 04
77 ZRDN
78 LASTZ
79 DSE 01
80 GTO 01
81*LBL 04
82 ISG 02
83*LBL 02
84 VIEW 00
85 DSE 00
86 GTO 00
87*LBL 03
88 RCL 00
89 RCL 03
90 MOD
92 X#0?
93 RTN
94 CLA
95 RCL 04
96 RCL 00
97 -
98 X=0?
99 RTN
100 AINT
101 "`->"
102 RCL 02
103 AINT
104 PROMPT
105 RCL Y
106 /
107 6
108 *
109 "AREA="
110 ARCL X
111 AVIEW
112 END
```

Boldly Going - Mandelbrot Set Area

Welcome to a new article in my *"Boldly Going"* series, this time starring the **Mandelbrot** set and the difficult task of computing an accurate estimation of its area. The task is fraught with difficulties and it's been attacked with really powerful hardware (think 4 GPUs), complex software and extremely long computation times (think 35 days) but all that work has produced only about 8-9 correct digits. Here I'll attempt the feat using just my trusty HP calculators, many orders of magnitude slower and less capable but nevertheless I'll manage to get about 5-6 correct digits in much shorter times.

Introduction

The Mandelbrot set (**M** for short) is the most well-known fractal of all, an amazing mathematical object which mystified everyone since its discovery by B. Mandelbrot ca. 1975 and subsequent popularization in the August 1985 issue of *Scientific American*. There is an incredible amount of readily available literature dealing with all aspects of **M** from the very basic to the most advanced so I'll refer the reader to it and won't discuss them here.



M has a fractal boundary which encloses a finite area whose precise value is still an open question, and an estimation of it is what this article is all about. To wit, there are several ways to try and estimate the area, including[1]:

- the *Monte Carlo* approach, where a large number of random points are generated within some enclosing box, and a tally is kept of how many belong to **M**, which is then used to compute the estimation.
- the *pixel-counting* approach, where finer and finer grids are averaged to tally the number of grid points belonging to **M**.
- the *theoretical* approach, where a large number of terms of an exact formula converging (extremely slowly) to the area of **M** are evaluated and added up to get an estimate.

The *Monte Carlo* approach has some advantages (such as not being prone to potential aliasing problems as may happen with equally-spaced grids) and disadvantages, the main one being that as is typical of standard *Monte Carlo* approaches, to get one more correct digit (i.e., increasing the resolution 10x) the number of generated pixels would need to be increased 100x, which would result in approximately 100x the running time. It also requires a very good, non-biased random number generator with a large cycle (at least several billions long).

The *pixel-counting* approach has been widely used. For example, back in 2012 R. Munafo launched an 8-day run to calculate almost 17 trillion pixels (at 2.4 million px/sec) to get an estimated area of *1.506591856* with an estimated error of *0.0000000256*.

Later, T. Förstemann used some powerful hardware (*Intel Core i7 2600K* CPU, 2x GPU *Radeon HD 5970* for a total of 4 GPUs with 1600 stream processors each, 350W under load) and software (*Mathematica 8.0.4.0* under *Windows 7*, *ATI* driver *Catalyst 11.2* with *AMD Stream SDK 2.3* and installation of a C-compiler [*Visual Studio 2011*] for *Mathematica*) running for 35 days straight with a grid size of *2,097,152* for a total of *87,960,930,222,520* calculated pixels (at more than 29 million px/sec and depths starting at *8,589,934,592* iterations) to get an estimated area/error of *1.50659188849* and *0.0000000028*, ten times better than Munafo's.

[1] Other methods include the *μ-atom method*, used by J. Hill to get a lower bound which is close to the pixel counting methods. He included the area of all components up to period 16 (*main cardioid* is *P1*, *main disk* is *P2*), and all of period 16 but one, and got an area of ***1.506303622***, which differs from Förstemann's by ~ *0.0002883 (0.019%)*.

Finally, the *theoretical* approach uses *Laurent Series*, in particular a specific one introduced by Ewing and Schober, which allows computing the area of **M** by evaluating an infinite series of the form:

$$\boldsymbol{M}_{area} = \pi \left(1 - \sum_{n=0}^{\infty} n.b_n{}^2\right)$$

where b_n are the coefficients of the Laurent series, the first ones being $b_0 = -1/2$, $b_1 = 1/8$, $b_2 = -1/4$, $b_3 = 15/128$, $b_4 = 0$, $b_5 = -47/1024$, etc. For a finite number of terms this formula always gives an upper bound of the area but despite its mathematical elegance it is absolutely unsuitable to compute the area as it converges incredibly *slowly*, with an estimated $6.4.10^{11}$ terms needed to get just *one* correct digit and more than 10^{118} terms to get *two* !

Matter of fact, Ewing *et al* used *500,000 terms* ($b_{500000} \sim 5.5221313 \cdot 10^{-8}$) in 1990 to get an estimated area of *1.72* and later in 2014 Bittner *et al* used *5,000,000* terms (whose b_n coefficients took 3 months to compute, $b_{5000000} \sim 8.0532 \cdot 10^{-11}$) and got an estimation of *1.68288*.

To complicate the matter even further, this theoretical approach seems to converge to a value between *1.60* and *1.70* while the empirical approaches (*Monte Carlo* and pixel counting) give estimates around *1.50659*. This might be due to the fact that the *boundary* of **M** has *Hausdorff* dimension *2* and thus *might* have positive (i.e., non-zero) area, which would account for the discrepancy as none of the empirical approaches can ever generate and calculate points or pixels exactly belonging to **M**'s boundary, so their potential contribution to the area would never be included in the computation. As of 2020, this is still in the realm of speculation but nevertheless it seems quite plausible[1].

Boldly going ...

As stated in the *Introduction* above, the purpose of this article is to use nothing but my trusty HP calculators (whether in physical or virtual form) to try and compute an estimation as accurate as possible (say 5-6 correct digits) for **M**'s area in reasonable times: less than half an hour for a virtual calc, a day or two at most for a physical one), which is no mean feat.

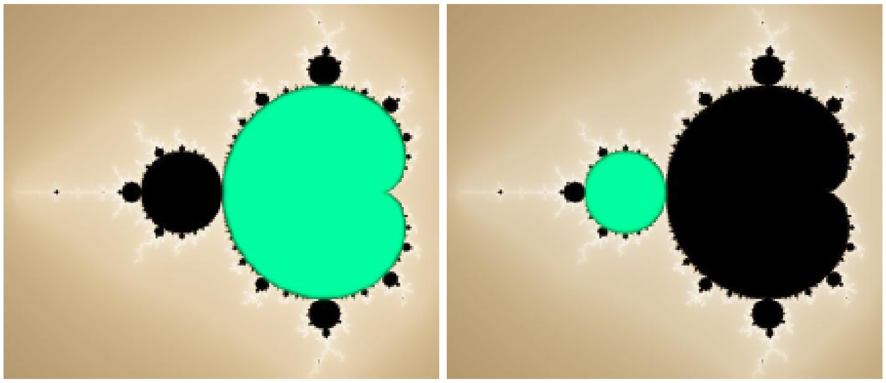
In view of the above described hardware, software and computation time requirements, it's clear that accomplishing my goal will require a good algorithm and pretty optimized code. As this is an informal Article, not a formal research paper, I'll adopt a *Machiavellian* approach *("The Ends Justify the Means")* and I'll mix sound mathematical optimizations with more informal heuristics as required.

To begin with, I'll use a *Monte Carlo* approach, generating a suitably large number *N* of random points within a rectangular box which completely encloses **M**, and counting how many actually belong to **M**. The sought-for area will then be proportional to the count. To make the task manageable I'll use the following optimizations:

- Each point *(x,y)* will be generated as a random complex number *z* within a rectangular box enclosing **M.** Actually, the leftmost extreme of **M** is at $x = -2$, the righmost extreme is at $x = 0.471185334933396+$, the topmost extreme is at $y = 1.122757063632597+$ and the downmost extreme is at $y = -1.122757063632597+$.
- As **M** is *symmetric*, I only need to compute the area of the top half and the total area of **M** will then be twice this value. This means that I can use a smaller rectangular box with *x* ranging from *-2* to *0.5* and with *y* ranging from *0* to *1.2* and I'll generate all random complex points *z* within that box.
- Each randomly generated complex *z* has to be tested for inclusion in **M**, which is done via the usual *escape time* algorithm: start with $z_0 = (0,0)$ and $c = z$, then iteratively compute $z_{n+1} = z_n^2 + c$ until either the absolute value of $z_n \geq 2$, in which case *z* escapes to infinity and so definitely does *not* belong to **M**, or else a max. number of iterations is reached and *z* is considered to belong to **M** and the count is increased by 1.

[1] D. Allingham (see *References*) wrote: *"B. Mandelbrot himself conjetures that the boundary of the set may have Hausdorff dimension 2, which would imply that it actually contributes to the area."*

- As computing whether every *z* belongs to **M** is a very time-consuming iterative process (which will reach the maximum number of iterations if *z* actually belongs to **M**) we can try and avoid it altogether for those *z* which we can easily ascertain in advance as belonging to **M** without performing any iterations. That's the case for those *z* either in the *main cardioid (below left)* or in the largest circular bud *(main disk, below right)*:



- The main cardioid's area is *3π/8 = 1.178097+* (about *78.20%* of the total area), while the main disk has an area of *π/16 = 0.196350+*, (another *13.03%)* and their combined total is *7π/16 = 1.374447+*, which already accounts for *91.23%* of the total area of **M** so we need to compute just the remaining *8.77%*, thus the expensive iterative process will be executed in full less than *9%* of the time, a considerable savings.

- To wit, if we can *quickly* check whether a given *z* belongs or not to the main cardioid or the main disk we'll save lots of running time and as it happens, indeed we actually *can*, using just a few steps for the *RPN* version or just 2 lines of code for the *BASIC* version.

- As for those points not belonging to either the main cardioid or the main disk, checking whether they belong to some other minor disks or cardioids quickly becomes more expensive and complicated than performing the *K* iterations, which will proceed faster if *K* is relatively small, say *256* iterations max.

  However, this will adversely affect the accuracy because there will be points which do not escape to infinity in *256* iterations but would if performing *512* iterations, say, and the same would happen with a bigger *K*, there will always be points (i.e.: those sufficiently close to the boundary) which will require more iterations than any limit we might specify in advance and so those points would be miscounted as belonging to **M** while actually they don't. Nevertheless, there will be fewer of them as *K* grows bigger, which will help increase the accuracy but negatively impact the running time.

- I'll attempt to alleviate this dilemma by calculating a large number *N* of random points but using a relatively low maximum number of iterations, say *K = 256*, which will speed the computation as desired. To increase the accuracy, I'll apply afterwards a *correction factor* to the resulting area, which will be heuristically computed like this: we'll choose a suitably smaller number of random points $N_2 << N$ and we'll obtain the count of the points belonging to **M** using first *K = 256*, then *K = 1024* iterations. The resulting correction factor would then be:

  $$f_{corr} = count_{1024} \,/\, count_{256}$$

  Simple as it is, this non-rigorous, heuristic approach works quite nicely and will allow us to use a relatively low number of max. iterations without actually compromising the obtained accuracy too much.

- In short, my algorithm will rely on: *(a)* rigorous math (statistically-sound *Monte Carlo* method, tight box, symmetry, main cardioid and disk detection, etc.), *(b)* nonrigorous heuristics (the *correction factor*) and last but not least *(c)* a little *luck*. When dealing with random numbers you always need a little luck, as the sequence *7,7,7, ...* has the same probability as any other more random-looking sequence. In practice this means that the results might be *worse* than average or *better* than average and the latter case is the lucky part.

Program Listing for the HP42S[1]

```
01  LBL "AM"
     2.5
    STO 06
     2
05  STO 07
     1.2
    STO 08
     0.25
    STO 09
10   1
    SEED
    "Points?"
    PROMPT
    STO 04
15  STO 00
     256
    "Iters?"
    PROMPT
    STO 05
20  CLX
    STO 02
    "Every?"
    PROMPT
    STO 03
25  RECT
26  CF 21
    "Working…"
    AVIEW
    CF 00
30  X≠0?
    SF 00
    LBL 00
    RCL 05
    STO 01
35  FS? 00
    XEQ 03
    RAN
    RCLx 06
    RCL- 07
40  RAN
    RCLx 08
    COMPLEX
    ENTER
    ENTER
45  SIGN
    RCL- 07
    RCLx ST L
    ABS
    RCLx 09
50  X<>Y
51  ABS
    X<Y?
    GTO 04
    SIGN
55  RCL+ ST Z
    ABS
    RCL 09
    X>Y?
    GTO 04
60  R↑
    RCL 07
    RCL ST Y
    LBL 01
    X↑2
65  RCL+ ST Z
    ABS
    X≥Y?
    GTO 02
    X<> ST L
70  DSE 01
    GTO 01
    LBL 04
    ISG 02
    LBL 02
75  DSE 00
76  GTO 00
    LBL 03
    RCL 00
    RCL 03
80  MOD
    X≠0?
    RTN
    CLA
    RCL 04
85  RCL- 00
    X=0?
    RTN
    AIP
    ├"→"
90  RCL 02
    AIP
    RCL÷ ST Y
     6
     x
95  ├"␊Area~"
    ARCL ST X
    AVIEW
98  END
```

Uses:

- *98 steps (199 bytes)*
- *flags 00, 21*
- *labels 00-04*
- *registers 00-09*
- *sets RECT mode*
- *any angular mode*

Registers:

00:	*N-loop index*
01:	*K-loop index*
02:	*M (count)*
03:	*every P*
04:	*N (# points)*
05:	*K (# iterations)*
06:	*2.5*
07:	*2*
08:	*1.2*
09:	*0.25*

Program details

Steps 01-31: main entry point: initialization[2] and prompting input from the user. *{ 31 steps }*
Steps 32-36: start of the main loop. *{ 5 steps }*
Steps 37-44: generation of a random point within the box, plus 2 copies on the stack. *{ 8 steps }*
Steps 45-53: checking whether the point belongs to the main cardioid (thus, to **M**). *{ 9 steps }*
Steps 54-59: checking whether the point belongs to the main disk (thus, to **M**). *{ 6 steps }*
Steps 60-71: checking whether the point belongs elsewhere in **M** (iterations). *{ 12 steps }*
Steps 72-73: if the point does indeed belong to **M**, increment the count. *{ 2 steps }*
Steps 74-76: decrement the number of points yet to generate/check and loop until no more left. *{ 3 steps }*
Steps 77-98: output routine, displays either the intermediate results and/or the final result. *{ 22 steps }*

[1] To enter text lines use the ALPHA menu; ├ is the ***Append*** character and ␊ is the ***Line Feed*** character, which can be found at the end of the second row of the PUNC submenu of the ALPHA menu.

[2] The initialization part stores four small constants in storage registers R_{06}-R_{09} because of speed considerations. Simply having the constants as program lines and performing the relevant arithmetic operations takes two program steps each and is much slower than using recall arithmetic, which just takes a single step and is faster as well. As these operations are part of the main loop, every speed gain is essential when being repeated many thousands of times.
Also, to save a register and a program step the constant *2* is stored just in R_{07}, then used at 3 different locations in the program, but the very first use at *step 39* depends on the enclosing box *x*-range being from *-2* to *0.5*. If using a different box *x*-range this constant might change and would need to be stored in its own register, say R_{10}, the other instances remaining unaltered.

Usage Instructions

The program accepts the number *N* of points to generate, the maximum number of iterations *K*, and whether you want to display intermediate results every *P* points or just the final estimation for the area.

The program doesn't automatically compute/apply any *correction factor*, that's left at the discretion of the user to decide whether and how to compute it since there's no optimal approach valid for all *N* and *K*, there's plenty of leeway. Of course, the program will greatly assist in computing it, as we'll see in the main run below.

To compute an estimation of the area of **M** proceed as follows:

XEQ	"AM"	→ *Points?*	*{ asks for the number of points to generate, N }*
N	R/S	→ *Iters?*	*{ asks for the max.num. of iterations*[1]*, K. Default=256, just press* R/S *}*
K	R/S	→ *Every?*	*{ asks if you want to display intermediate results every P points*[2]*;* *if you don't and just want the final result, simply press* R/S *}*
P	R/S	→ $\text{Point}_P \rightarrow \text{Count}_P$ $\text{Area} \sim \text{Area}_P$	*{ the intermediate tally of points generated and resulting counts }* *{ the intermediate estimations of the area }*
		…	
		→ $\text{Point}_N \rightarrow \text{Count}_N$ $\text{Area} \sim \text{Area}_N$	*{ the final tally of points generated and resulting count }* *{ the final estimation of the area }*

Further Considerations

To choose the number of points *N* and max. iterations *K*, we'll take into account the following considerations:

- Both the correctness of the estimated area and the running time depend on *N* and *K*, the larger the better as far as the estimated area is concerned but the longer the running time will be. Also, whether you're using a physical *HP42S/DM42* or a virtual *HP42S* and its underlying *OS* (*iOS*, *Android*, *Windows*, *Mac*, *Linux*, other) and hardware, all of it will greatly influence the choice of calculation parameters.

  Generally speaking, a physical original *HP42S* will be the slowest by far, and this will limit the running times allowable without depleting the batteries, probably 1-2 days at most. The *DM42* is ~100x faster and can use an *USB* power source, so it can run the program for much longer. Some experimentation will be required, starting at a low value of *N*, *K* (say *N = 1,000* and *K = 256*) and noting the running time. Then it's possible to select how big *N* and *K* should be, as the time will be proportional to both.

- On the other hand, a virtual *HP42S* will be orders of magnitude faster. For instance, using *Free42*[3] *BCD* on an *Android* mid-range *Samsung* tablet (as done below) will generate and check about *1,000* points per second at *256* max. iterations per point. This means I can use *N = 500,000* points and *K = 256* max. iterations, say, and get the result in less than 10 min. Using a faster version of *Free42* and/or a faster emulator/*OS*/ hardware combination can easily get results even 10x or 100x faster.

- Increasing the number of iterations *K* will always *reduce* the estimated area because performing more iterations weeds out points that never escaped to infinity when using *K* iterations, and thus were included in the count, but actually *did* escape when using more iterations and so weren't included now.

- However, increasing the number of points *N* while leaving *K* fixed results in estimated areas which overshoot/undershoot the area, slowly converging to the correct value of the area *for that number of iterations*, $\mathbf{M_K}$, *not* to the correct area of **M**, which would be the value for *infinite* iterations.

- This can be remedied by using a *correction factor*, which uses $K_{i,j}$ to extrapolate K_∞ as we'll see below.

[1] The number of iterations doesn't need to be a power of *2 (256, 512, …)*, it can be any positive integer (say *1,000*, *687*, …)

[2] If you enter a positive integer value *P*, the intermediate results will be displayed every *P* points as well as the final result once all *N* points have been generated. *P* doesn't need to divide evenly into *N*, the final result will be displayed regardless. If *P* is *0* no intermediate results will be shown, which will mean faster execution but you won't be able to monitor progress.

[3] *Free42* is a fantastic *free* simulation of the *HP42S* created by ***Thomas Okken*** for many operating systems (*Windows*, *Mac OS*, *Android*, *iOS*, *Linux*, etc.) which also runs at the heart of ***SwissMicros*** physical *DM42* calculator. It runs many hundred times faster than a physical *HP42S* and features vastly increased available *RAM*, 34-digit *BCD* precision and much more.

Sample runs

Let's see several examples. We'll asume FIX 05 display mode for all results that follow.

Example 1

For starters, let's estimate **M**'s area using *N = 10,000* points and *K = 256* iterations, showing just the final result.

```
XEQ     "AM"   →  Points?
10000   R/S    →  Iters?             { we'll use 256 iters. which is the default so just press R/S }
        R/S    →  Every?             { we just want the final result so just press R/S }
        R/S    →  10000 →2572        { the final tally: 10,000 points generated, 2,572 landed in M }
                  Area ~ 1.54320     { the estimated area of M, just two correct digits, err=2.43%, 11" }
```

Example 2

Let's improve the estimation using *N = 10,000* points and *K = 512* iterations, showing results every *2,000* points.

```
XEQ     "AM"   →  Points?
10000   R/S    →  Iters?
  512   R/S    →  Every?
 2000   R/S    →  2000 →511      Area~1.53300    { the first intermediate result }
               →  4000 →1041     Area~1.56150    { the 2nd intermediate result }
               →  6000 →1561     Area~1.56100    { the 3rd intermediate result }
               →  8000 →2053     Area~1.53975    { the 4th intermediate result }
               → 10000 →2560     Area~1.53600    { final result, still 2 correct digits but err=1.95%, 19"}
```

The Ultimate Run

Now for the real McCoy. Taking the above considerations into account and as I'll be using a virtual *HP42S* (*Free42 BCD* for *Android*) running on a mid-range *Samsung* tablet, I'll use half a million points and a low *256* iterations for speed but I'll also compute and apply a *correction factor* to try and increase the precision. I'll compute this correction factor first, using 5x fewer points than the main run but 4x more iterations, as follows:

$$f_{corr} = Area_{100000,1024} \,/\, Area_{100000,256}$$

where $Area_{N,K}$ means computing the area using *N* points and *K* iterations. Let's proceed to compute f_{corr} :

```
XEQ      "AM"   →  Points?                             { we'll use 5x less points, just 100,000 }
100000   R/S    →  Iters?                              { we'll use first 1,024 iterations }
  1024   R/S    →  Every?                              { we won't be monitoring progress }
         R/S    → 100000 →25312    Area~1.51872        { the value of Area100000,1024    [ 5'45"] }
                                   STO 10              { we store it for later use }

XEQ      "AM"   →  Points?                             { as above, still just 100,000 }
100000   R/S    →  Iters?                              { now we'll use 256 iterations, so just press R/S }
         R/S    →  Every?                              { we won't be monitoring progress either }
         R/S    → 100000 →25501    Area~1.53006        { the value of Area100000,256    [ 1'58"] }
                                   STO÷ 10             { R10 now contains the c. factor ~ 0.99258853 }
```

Now it's time for the the main computation, to which we'll afterwards apply the just calculated (and stored) *correction factor*. This will take less than 10 min. in all and we'll monitor progress ...

```
XEQ  "AM"    →  Points?                          { we'll use the full 500,000 points }
500000  R/S  →  Iters?                           { we'll use 256 iterations, so just press R/S }
        R/S  →  Every?                           { we'll monitor progress every 100,000 points }
100000  R/S  →100000 →25501     Area~1.53006     { the first intermediate result   [ 1'58"] }
                    ...              ...
             →500000 →126486    Area~1.51783     { the main result, which in itself has err ~ 0.75%
                                                   before applying the correction factor [ 9'47"] }
```

Finally, let's apply to the just computed area in the display the *correction factor* previously computed and stored:

```
RCLx 10  →  1.50658      { more precisely, 1.50658_263   vs.   Förstemann's 1.50659_188 }
```

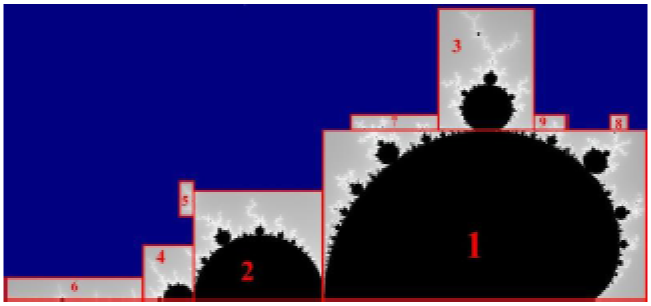
which is my final computed estimation for the area of **M** and it's correct to **6** digits within less than one *ulp* (unit in the last place). It differs from Förstemann's *88-trillion-pixels-calculated-at-8.6-billion-iterations-per-pixel* result by just ~ *0.00000925*, an error of ~ *0.000614%*.

He got an estimated area accurate to **9** correct digits (within possibly a couple *ulps* or three) in 35 days at great expense (both the costly hardware *and* the 35-day electricity bill), while I got **6** correct digits in less than 20 min. (actually *17'30"* = *9'47"* for the main computation plus *5'45"* + *1'58"* for the *correction factor* computation) at negligible expense, so point made. Not bad, isn't it ?

Where to go now

As this is an informal article and the point has already been made, we could really call it a day and move on. But if we were willing to, there's a number of further techniques to consider in order to improve the accuracy and/or reduce the computation times. For instance, among other possibilities:

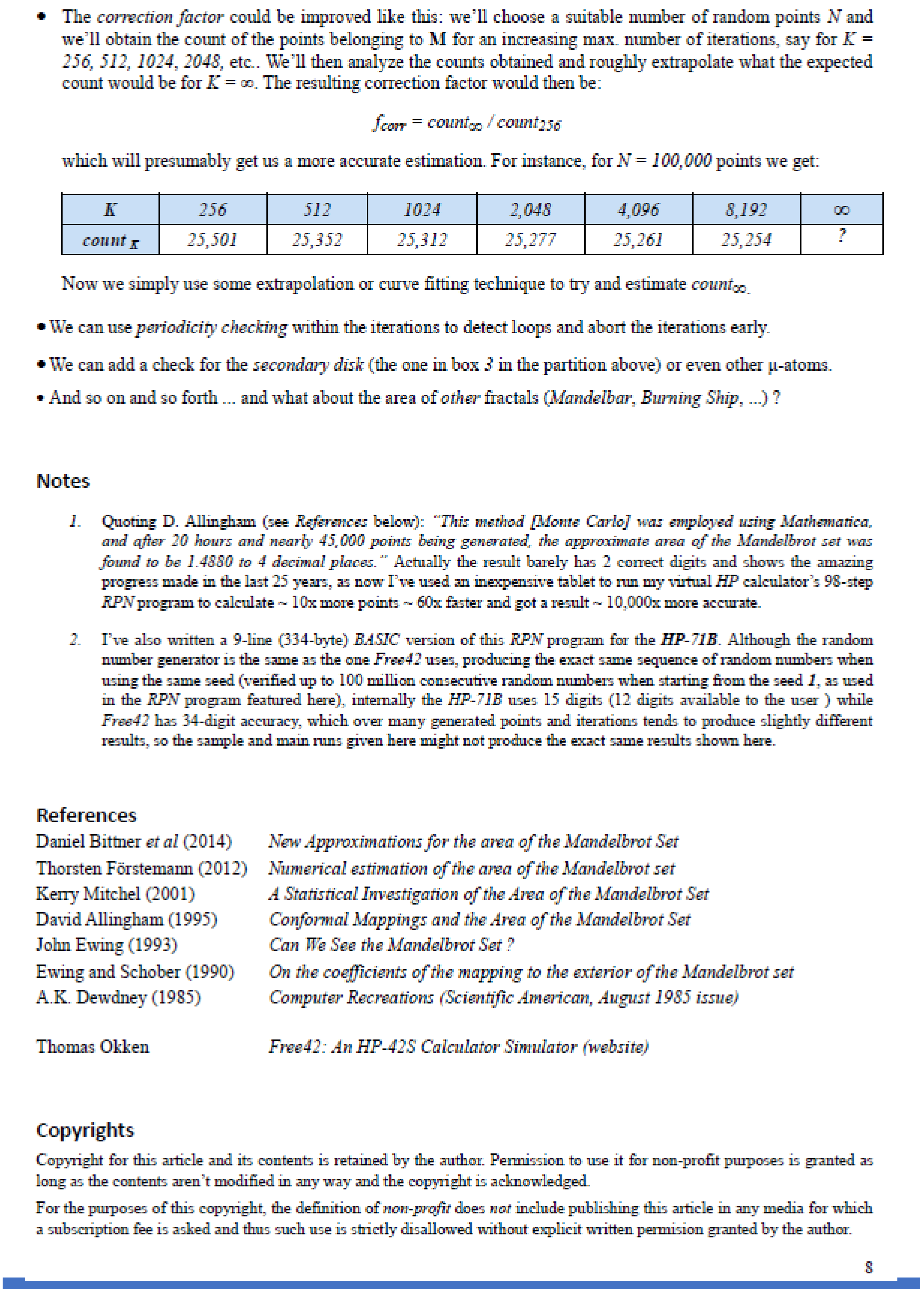
- We can avoid wasting time generating and checking random points in *blank* areas (~*75%* of the enclosing box used here) where no part of **M** is, by subdividing **M** into a number of rectangular boxes (9 in the sample partition below) and then computing the total count as the sum of the counts in each individual box.



It is important to distribute the total number of points **N** among the boxes proportionally to the area of each box so that the density of points is the same. Otherwise we would be adding areas computed with different precisions and this is wasteful as the resulting sum will be no more accurate than the least accurate area. To implement this, the program must be converted into a *subprogram* with no prompting and no output, which accepts the dimension of each box and the number of points N_i to use and returns the count to a main program which first inputs the number of points *N* and max. iterations *K* from the user and then calls the subprogram with the coordinates and the N_i for each box, then adds up the returned counts and computes and outputs the total area. There's no overhead and large blank areas are thus avoided.

Also, the process is faster for each box because some time-consuming checks are avoided altogether:

- Box *1* only needs to check if points belong to the main *cardioid*, but forfeits the check for the disk.
- Box *2* only needs to check if points belong to the main *disk*, but forfeits the check for the cardioid.
- all remaining boxes forfeit **both** checks, which significantly speeds the process.



- The *correction factor* could be improved like this: we'll choose a suitable number of random points *N* and we'll obtain the count of the points belonging to **M** for an increasing max. number of iterations, say for *K = 256, 512, 1024, 2048*, etc.. We'll then analyze the counts obtained and roughly extrapolate what the expected count would be for *K* = ∞. The resulting correction factor would then be:

$$f_{corr} = count_{\infty} / count_{256}$$

which will presumably get us a more accurate estimation. For instance, for *N = 100,000* points we get:

K	*256*	*512*	*1024*	*2,048*	*4,096*	*8,192*	∞
count $_K$	*25,501*	*25,352*	*25,312*	*25,277*	*25,261*	*25,254*	*?*

Now we simply use some extrapolation or curve fitting technique to try and estimate *count$_\infty$*.

- We can use *periodicity checking* within the iterations to detect loops and abort the iterations early.
- We can add a check for the *secondary disk* (the one in box *3* in the partition above) or even other µ-atoms.
- And so on and so forth ... and what about the area of *other* fractals (*Mandelbar*, *Burning Ship*, ...) ?

Notes

1. Quoting D. Allingham (see *References* below): *"This method [Monte Carlo] was employed using Mathematica, and after 20 hours and nearly 45,000 points being generated, the approximate area of the Mandelbrot set was found to be 1.4880 to 4 decimal places."* Actually the result barely has 2 correct digits and shows the amazing progress made in the last 25 years, as now I've used an inexpensive tablet to run my virtual *HP* calculator's 98-step *RPN* program to calculate ~ 10x more points ~ 60x faster and got a result ~ 10,000x more accurate.

2. I've also written a 9-line (334-byte) *BASIC* version of this *RPN* program for the ***HP-71B***. Although the random number generator is the same as the one *Free42* uses, producing the exact same sequence of random numbers when using the same seed (verified up to 100 million consecutive random numbers when starting from the seed ***1***, as used in the *RPN* program featured here), internally the *HP-71B* uses 15 digits (12 digits available to the user) while *Free42* has 34-digit accuracy, which over many generated points and iterations tends to produce slightly different results, so the sample and main runs given here might not produce the exact same results shown here.

References

Daniel Bittner *et al* (2014) *New Approximations for the area of the Mandelbrot Set*
Thorsten Förstemann (2012) *Numerical estimation of the area of the Mandelbrot set*
Kerry Mitchel (2001) *A Statistical Investigation of the Area of the Mandelbrot Set*
David Allingham (1995) *Conformal Mappings and the Area of the Mandelbrot Set*
John Ewing (1993) *Can We See the Mandelbrot Set ?*
Ewing and Schober (1990) *On the coefficients of the mapping to the exterior of the Mandelbrot set*
A.K. Dewdney (1985) *Computer Recreations (Scientific American, August 1985 issue)*

Thomas Okken *Free42: An HP-42S Calculator Simulator (website)*

Copyrights

Gaussian Distribution of Random Numbers

Let's move to a different chapter of the module, dealing with Gaussian- (or Normal-) distributed random numbers. As a way of introduction, Normal distribution of random numbers play an important role in numerous science fields and therefore must be covered.

The module includes two techniques to obtain gaussian random numbers, both using output from the SandMath/PPC p-RNG described earlier in the manual. The techniques are the Box-Muller and the "12R minus six" methods, described below.

Function	Description	Input	Output
BXMR	Gaussian RN (Box Muller)	SEED/RN in buffer	Normal RN in X
RANG	Gaussian RN (12R-6)	SEED/RN in buffer	Normal RN in X

See: https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform
and: https://mathworld.wolfram.com/Box-MullerTransformation.html

Note that both functions *generate GRNs "Nr" with "Standard" Normal Distribution*, i.e. with mean 0 and variance 1. This can be "moved" to GRNs "Xr" of any Normal Distribution with mean μ and variance σ using the relationship: Nr = (Xr - μ) / σ ; and thus: Xr = μ + σ.Nr

The *Box-Muller method* uses two uniformly distributed RN's { x1, x2 } generated via SEEDT, RNDM to calculate two GRNs { z1, z2 } using the following transform:

$$z_1 = \sqrt{-2\ln x_1}\ \cos(2\pi x_2)$$
$$z_2 = \sqrt{-2\ln x_1}\ \sin(2\pi x_2).$$

Whereas the *"12R-6" method* uses the sum of twelve random numbers generated via SEEDT, RNDM p-RNG, and subtracts the value 6 from the result to obtain the GRN. Simpler formula but more demanding requiring 6x more RNs.

z = (x1 + x2 + x3 + … + x12) - 6

Both functions are implemented in MCODE, and the execution times are very similar,

Example. Using pi as initial seed, generate 6 GRNs for each method:

PI, XEQ "SEEDT" => 3.141592654

BXMR	RANG
-1.971258998	1.041297606
0.0659360161	-0.091203312
0.425290149	-0.382308886
-1.432610257	-1.659429274
-0.100011293	-0.276399468
-0.553735152	0.267634817

You should be getting the exact same results as the Time seed was not used.

Digging deeper: which p-RGN to use?

The MCODE functions are "fixed" to the SandMath/PPC p-RNG but their FOCAL counterparts can use any of the three sets of p-RNGs, controlled by the user flags as follows:

Function	Description	Input	Output
BX-MR	Gaussian RN (Box Muller)	SEED/RN in buffer	Normal RN in X
12R-6	Gaussian RN (12R-6)	SEED/RN in buffer	Normal RN in X

p-RNG used:

Flag Set	p-RNG	Conditions
UF 00	SandMath/PPC	None
UF 01	Toulouse Math	UF 00 Clear
UF 02	Voyager 11C/15C	UF 00 and UF 01 clear

Note that the interrogation follows the flag number order, thus if UF 00 is set that will prevail over the status of the other two because it's the first one checked.

Repeating last example for the other two p-RNG we obtain: (don't forget to initialize the corresponding seed with PI, using the corresponding functions: STORAND and SEED instead of SEEDT

For Voyager p-RNG – SF 02 For Toulouse Math p-RNG - SF 01

```
      BX-MR           12R-6              BX-MR            12R-6
   0.460685398    -0.563762097      -1.392177264    -0.391049942
    1.430844326    0.732050058      -0.926716527     0.973123557
  -0.064030634    -1.105056443      -0.670127802     0.096738080
    1.068030738    2.561598080       1.375597030    -1.050317093
   0.823396897    -0.605297092       0.729656991     1.411568919
   0.685792644     1.510668919      -0.724121302    -0.211919406
```

And how to make heads or tails of all these GRNs?

Armed with these two additional routines we can do comparative testing, pitting the methods against each other and choosing which p-RNG to use on every case. The routines "TRANG (Test RANG) and "BENCH are provided to that end, as described next. A third program "INDEX is also provided to evaluate the goodness of each combination of method and p-RNG – by establishing a comparison with the Normal Probability Function P(x).

Normality Tests	Description	Input	
"BENCH	Driver for "TRANG" + "INDEX"	Sample Size, p-RNG, Method	
"INDEX	Calculate Normality Index	Quantile data in {R00-R05}, R10	
"TRANG	Test one Method/p-RNG combo	Sample Size & User Flag 0-3	

Here's how these work.-

TRANG. Besides selecting the appropriate p-RNG user flag, "TRANG uses UF 03 to choose which method to employ: UF 03 Set = Bix-Muller, and UF 03 Clear = 12R-6 method.

Once those four flags are properly set you would execute "TRANG to calculate the 6 quantiles reflecting the GRN probability distribution amongst them. The width of the quantiles is determined by the sample size: w = N/6

A final message in ALPHA like the one below shows the results for the first four quantiles – (not six, need to make it fit in ALPHA), the most important ones since usually the two remaining ones are always 100% (notice no decimal digits is provided for this cas to save some real state).

For example, for a sample size of 50 GRNs and setting UF 02 (Voyager p-RNG) and also setting UF 03 (Box-Muller method) we'll type:

```
XEQ "TRANG"   =>    N=?
50, R/S       =>    WAIT. . .
                    64.00, 98.00, 100, 100
```

Is that good or bad? Well, here's where function INDEX comes to the scene. It expects the quantile GRN probability data already stored (either by TRANG or manually by the user), and compares those percentages with the theoretical cumulative probability corresponding to a normal distribution, i.e.

$$F(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} \exp\left(-\frac{(t-\mu)^2}{2\sigma^2} \right) dt.$$

With x being the five abscissas of the five quantiles, and μ,σ the mean and standard deviation of the distribution.

The module includes the Error Function erf(x) to calculate that integral by the following expression:

$$\Pr[X \le L] = \frac{1}{2} + \frac{1}{2}\,\mathrm{erf}\,\frac{L-\mu}{\sqrt{2}\sigma}$$

The last step is a direct comparison between the theoretical and actual results, i.e. the "Normality Index". For this example:

```
XEQ "INDEX"   =>    RUNNING. . .   =>   0.333450091
```

Calculated as:

$$Index = \Sigma\ [sqrt\ (Q(n)^\wedge 2 - P(n)^\wedge 2],\ \ n = 1,2\ldots,6$$

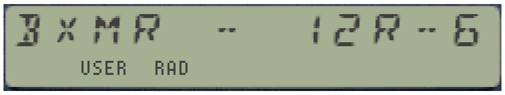
A perfect set result would have index = 0, so the lower the better.

All together now: Driver program

For maximum convenience the driver program "BENCH" does the leg work with the User flag setting for you, navigating the different options with ALPHA prompts, see next.

Let's find the normality index of a set of 100 gaussian random numbers generated using the 12R-6 method with the Toulouse Math p-RNG. We'd type:

XEQ "BENCH" presents the first menu-driven screen with the name of the two methods:

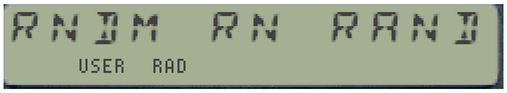


BXMR -- 12R-6

USER RAD

choose [B] or [D]

XEQ [D], which presents the second selection screen with the name of the three p-RNG choices:



RNDM RN RAND

USER RAD

choose [A], [C] or [E]

XEQ [E] for the Toulouse Math, => N=?
100, R/S => WAIT. . .

Set UF 10 to see a countdown with the current GRN being calculated. When completed we'll see the "RUNNING. . ." message and then the ALPHA screen with the GRN percentages for each quantile:

; 68.00; 95.00; 100; 100 ; 100; 100

pressing R/S => 0.274599834

Meaningful comparisons should be made between sets with equal sample sizes, or else the normality will be totally skewed of course.

And the winner is…

The table below summarizes the results like those obtained in the example above, (sample size = 100), repeated for all possible combinations. The initial seed is always 1.

Method / p-RNG	**RNDM**	**RN**	**RAND**
Box-Muller	0.322565407	0.298459560	0.282842176
12R-6	0.350934321	0.292661454	0.274599834

The Toulouse Math ROM takes the gold, and that's using either of the two methods. Second place is for the Voyager 11C/15C p-RNG, and the bronze medal is for the SandMath/PPC contender.

Testing Random Number Generators

There's quite an abundance of relevant literature on this subject available on the internet. If you google the title of this chapter, the hit list is long, and the entries are quite informative, ranging from very pragmatical to heavy on the theoretical side. See for instance this one by Dan Biebighauser.

For the RANROM I decided to take the historic perspective, including two old programs on the direct subject plus a more recent one about a related topic published in the MoHP forum.

Program	Description	Author / Source	
"EVAL"	Evaluation of p-RNGs	*L. H. Gilbert / UPL #10240*	
"RNGTST"	Randomicity Test Paper	Charles T. Tart / Paper	
"CHI"	Chi-square test		
"STRAT"	Stratified Random Samplimg	Rawi / MoHP Article	

1. Evaluation of p-RNGs (by L.H. Gilbert, UPL# 10240)

The original program has been modified slightly to allow for a more convenient handling of the p-RNG selection, which obviously cannot be entered in the main body of the program when this is in ROM. Besides I've replaced the data entry section with a new, menu-driven one. No other changes were made. Quoting from the original UPL document. -

The program executes a user-provided random number generator, and checks the output in the following ways:

- *the mean and standard deviation of a sequence are computed*
- *The correlation between Xi andX(i+1) is calculated*
- *A running check is made to determine when the generator enters a closed cycle (if ever)*
- *A histogram is plotted, and*
- *A Chi-square test of uniformity is calculated.*

The program requires one memory module. A printer is recommended.

Application, Equations, Variables.

The random number generator produces a sequence of (uniform) numbers, X = { xi }, i = 1,..., N.

- *For each xi, x2i is computed and matched. If they're equal, a closed cycle of numbers has been detected.*
- *A histogram of 20 "bins" on the interval 0 – 1 is constructed, and a count made of where each xi falls.*
- *The extended count per bin, N/20, is employed in a Chi-Square test:*

$$\chi\text{^}2 = \Sigma\ [(fi - fe)\text{^}2 / fe \quad ; i= 1,2...,20$$

with fi = expected count, and fe = N/20

At the 5% level of significance the χ^2 should be greater that 30 in order toconclude that the N random numbers are not uniformly distributed.

Operating limits and warnings.

The program is designed to evaluate a generator of uniform random numbers, on the interval [0, 1 [

Using all the program fcacilities, each number takes about 3.6 seconds to generate and evaluate; about 3.0 seconds with the printer switched off.

The RNG must be program under a general label in memory, it will be called by the main program.

Example.

*A sequence of N=100 numbers was generated by the formula: Xi+1 = FRC [9821 * xi + 0.211327] The initial seed was 0.159753, and the formula has been programmed as:*

```
01   LBL “RNG0”
02   9821
03   *
04   ,211327
05   +
06   FRC
07   RTN
```

Starting the program now:

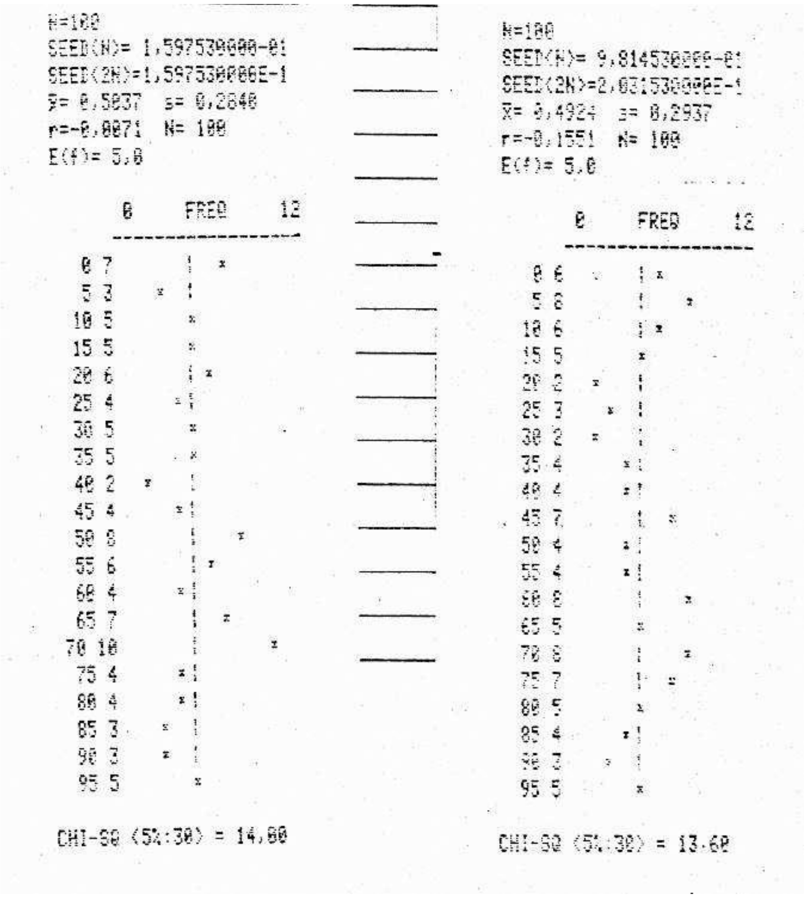
XEQ “EVAL” – sets SIZE 36 and resets all flags.

The initial set of questions prepare the flags for the program execution automatically. You just need to enter the initial letter of the choice you want, and then press R/S. A shortcut is also enabled so if no letter is pressed the default option is the first choice.

STOP/CONT?	*R/S*	*- Clears/Sets UF 00*
CLRG/ACCUM?	*R/S*	*- Clear /Sets UF 01*
XN=2N/OMIT?	*R/S*	*- Clear/Sets UF 02*
HISTO/OMIT?	*"H”, R/S*	*- Clear/Sets UF 03*
ALL/DUP?	*R/S*	*- Clear Sets UF 04*
RNG PRGM?	*“RNG0”, R/S*	
SEED=?	*0.159753, R/S*	
N=?	*100 , R/S*	

The execution starts, showing a countdown as the sample is being processed. Finally, the HP-41 beeps and halts at end. Pressing R/S again starts the output of result – ending with the value of χ^2 in the display.

Here are the frequency counts on each bin for the first (left) and second (right) 100 digits:



```
N=100
SEED(N)= 1,597530000-01
SEED(2N)=1,597530000E-1
X̄= 0,5037  s= 0,2840
r=-0,0071  N= 100
E(f)= 5,0

        0    FREQ    12
       ------------------
  0 7            !  x
  5 3         x  !
 10 5            x
 15 5            x
 20 6            ! x
 25 4           x!
 30 5            x
 35 5            x
 40 2        x   !
 45 4           x!
 50 8            !     x
 55 6            ! x
 60 4           x!
 65 7            !  x
 70 10           !        x
 75 4           x!
 80 4           x!
 85 3          x !
 90 3          x !
 95 5            x

CHI-SQ (5%:30) = 14,80
```

```
N=100
SEED(N)= 9,814530000-01
SEED(2N)=2,031530000E-1
X̄= 0,4924  s= 0,2937
r=-0,1551  N= 100
E(f)= 5,0

        0    FREQ    12
       ------------------
  0 6            ! x
  5 8            !    x
 10 6            ! x
 15 5            x
 20 2        x   !
 25 3         x  !
 30 2        x   !
 35 4           x!
 40 4           x!
 45 7            !  x
 50 4           x!
 55 4           x!
 60 8            !    x
 65 5            x
 70 8            !    x
 75 7            !  x
 80 5            x
 85 4           x!
 90 3          x !
 95 5            x

CHI-SQ (5%:30) = 13.60
```

- *If F00 is clear the execution halts after N numbers so that the printer can be switched off. If it is set no halt occurs, the printer is always on.*

- *If F01 is clear each sequence of N numbers is treated separately,. If it's set the data is accumulated for each successive sequence.*

- *If F02 is clear the program, will check for duplicate numbers. If set there won't be checked 0 faster running time.*

- *If F03 is clear the histogram is printed. If set, there's no print of histogram.*

- *If F04 is clear the program does a complete evaluation. If it's set only a check for duplicate numbers is done.*

*When a duplicate number is detected the program halts showing a message that identifies the element in the sequence causing the event:: "*CYCLE AT nn*",* Let's now run **EVAL** for our three p-RNGs to – finally! – establish a comparison and therefore determine their relative ranking. For that we need to write three trivial FOCAL routines for EVAL to call, as follows:

```
01  LBL "RPPC"          01  LBL "RTOU"          01  LBL "RVYG"
02  RNDM                02  RAND                02  RN
03  RTN                 03  RTN                 03  RTN
```

Using a sequence length of N=1,000 and the same seed = 0.123456789 for all of them.

The results are shown below:

```
XROM "EVAL"
STOP/CONT?              S RUN
CLRG/ACCUM?             A RUN
XN=2N/OMIT?             RUN
HISTO/OMIT?             O RUN
ALL/DUP?                RUN
RNG  PRGM?              "RPPC" RUN
SEED=?                  0.123456789 RUN
N=?                     1,000.000000 RUN
```

```
N=1,000.
SEED<N>= 1.234567890-01
SEED<2N>=1.234567890E-1

µ= 0.4954 s= 0.2791
r=-0.0088 N= 1,000.
E(f)= 50.0
 44. 47. 50. 56. 54. 39. 56. 45. 49. 63. 69. 50. 45. 52. 51. 51. 44. 50. 42. 43.

CHI-SQ<5%:30> = 19.80
```

For the second group we use LBL B - there's no need to repeat all data entry:

```
XEQ C

RNG  PRGM?              "RTOU" RUN
SEED=?                  0.123456789 RUN
N=?                     1,000.000000 RUN
```

```
µ = 0.5152 s= 0.2884
r=-0.0009 N= 1,000.
E(f)= 50.0
43. 53. 56. 45. 36. 49. 40. 51. 41. 59. 51. 44. 59. 56. 58. 52. 51. 56. 44. 56.

CHI-SQ<5%:30> = 18.20
```

And going for the third group now:

```
XEQ C

RNG PRGM?            "RVYG" RUN
SEED=?               0 .123456789 RUN
N=?                  1,000.00 RUN

N=1,000.
SEED<N>= 1.234567890-01
SEED<2N>=1.234567890E-1

µ = 0.4918 s= 0.2837
r=-0.0304 N= 1,000.
E(f)= 50.0
43. 46. 54. 57. 44. 50. 47. 56. 51. 56. 49. 66. 45. 44. 51. 41. 60. 51. 47. 42.

CHI-SQ<5%:30> = 16.44
```

Alright then, according to the results above the better p-RNG is the Voyager' style.

Uniformity Test	SandMath/PPC	Topulouse Math	Voyager 11C/15C
χ^2	19.8	18.20	16.44

This is an interesting result, because despite being the better of the three in *uniformity,* if you recall it however did not hold the winner place for the *normality* of the gaussian random numbers generated from it; so not quite a slam dunk! If anything, this demonstrates that there are different considerations to the quality of "randomness".

Going the extra mile, we can widen the comparisons by also testing some of the other p-RNGs available in the module, like Mark Power's from DataFile and JM Baillard's from his web site. Doing so we obtain the following table of results:

Uniformity Test	RAN00	RNG1	RNG2
χ^2			

Program listing.

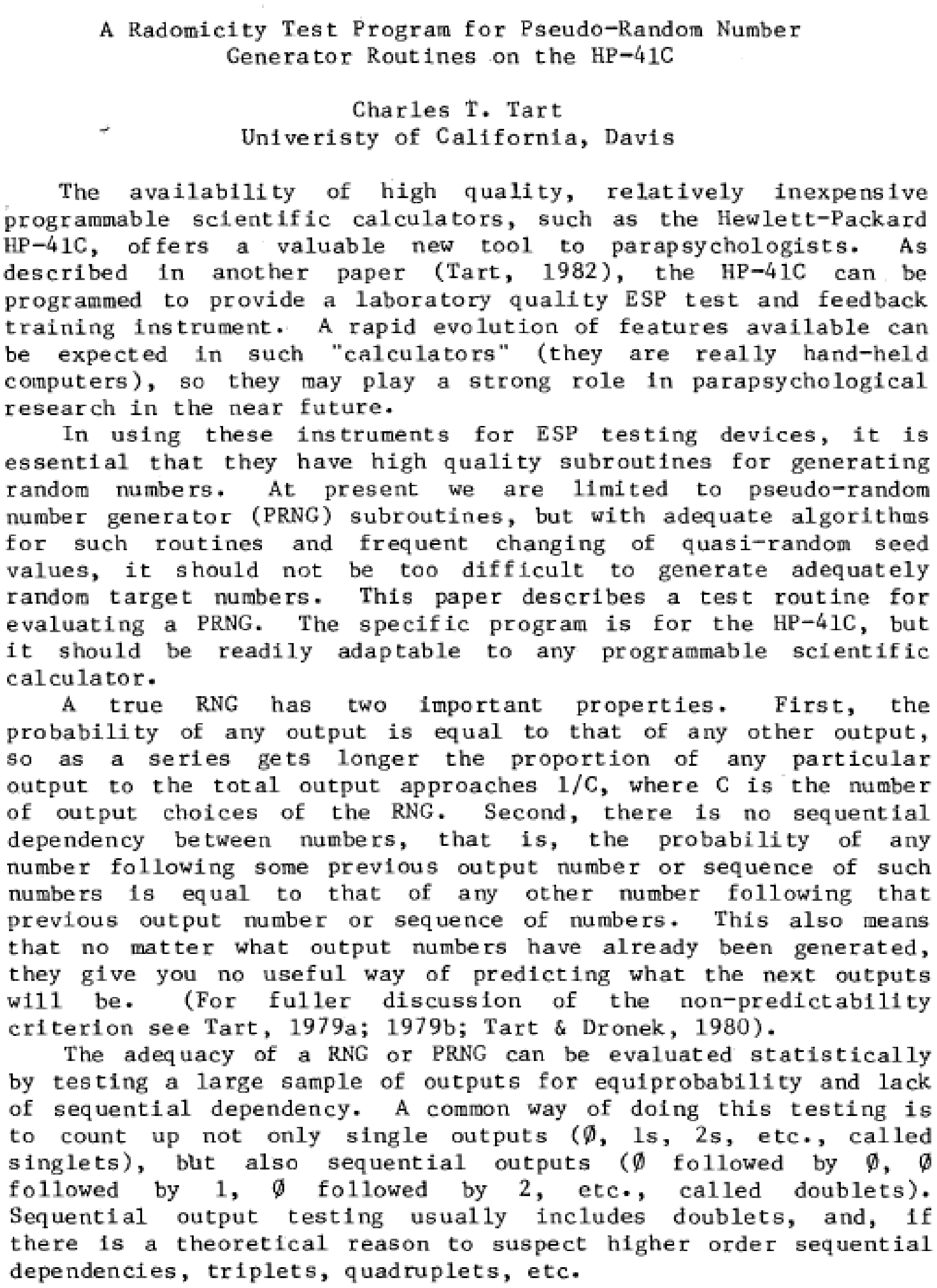
```
01*LBL "EVAL"
02 SIZE?
03 37
04 X>Y?
05 PSIZE
06 CLX
07 X<>F
08 AON
09 "STOP/CONT?"
10 PROMPT
11 ATOX
12 83
13 X#Y?
14 SF 00
15
"CLRG/ACCUM?"
16 PROMPT
17 ATOX
18 67
19 X#Y?
20 SF 01
21
"XN=2N/OMIT?"
22 PROMPT
23 ATOX
24 88
25 X#Y?
26 SF 02
27 "HISTO/OMIT?"
28 PROMPT
29 ATOX
30 72
31 X#Y?
32 SF 03
33 "ALL/DUP?"
34 PROMPT
35 ATOX
36 65
37 X#Y?
38 SF 04
39 AOFF
40 ΣREG 09
41*LBL C
42 CLRG
43 "RNG PRGM?"
44 AON
45 PROMPT
46 AOFF
47 ASTO 36
48 "SEED=?"
49 PROMPT
50 X=0?
51 1RAN
52 STO 00
53 STO 01
54 FS? 04
55 GTO 03
56 "N=?"
57 PROMPT
58 STO 02
59*LBL 01
60 RCL 00
61 RCL 01
62 RCL 02
63 RCL 36
64 FC? 01
65 CLRG
66 STO 36
67 RDN
68 FIX 0
69 "N="
70 ARCL X
71 ACA
72 PRBUF
73 STO 02
74 STO 03
75 RDN
76 STO 01
77 RDN
78 STO 00
79 SCI 9
80 "SEED<N>="
81 ACA
82 ACX
83 PRBUF
84 "SEED<2N>="
85 ARCL 01
86 ACA
87 PRBUF
88 16
89 STO 15
90 CF 21
91 FIX 0
92*LBL 02
93 VIEW 03
94 RCL 00
95 RCL 00
96 XEQ IND 36
97 STO 00
98 s+
99 CLX
100 .05
101 /
102 INT
103 RCL 15
104 +
105  E
106 ST+ IND Y
107 FC? 02
108 XEQ 04
109 DSE 03
110 GTO 02
111 BEEP
112 FC? 00
113 STOP
114 FC? 55
115 PRBUF
116 SF 21
117 XEQ 05
118 XEQ 06
119 FC? 00
120 STOP
121 ADV
122 ADV
123 GTO 01
124*LBL 03
125  E
126 ST+ 03
127 VIEW 03
128 RCL 00
129 XEQ IND 36
130 STO 00
131 XEQ 04
132 GTO 03
133*LBL 04
134 RCL 01
135 XEQ IND 36
136 XEQ IND 36
137 STO 01
138 RCL 00
139 X#Y?
140 RTN
141 TONE 9
142 "CYCLE AT "
143 ARCL 03
144 PROMPT
145 RTN
146*LBL 05
147 FIX 4
148 2
149 ACCHR
150 61
151 ACCHR
152 MEAN
153 ACX
154 2
155 SKPCHR
156 115
157 ACCHR
158 61
159 ACCHR
160 SDEV
161 ACX
162 PRBUF
163 114
164 ACCHR
165 61
166 ACCHR
167 SDEV
168 *
169 RCL 09
170 RCL 11
171 *
172 RCL 14
173 /
174 RCL 13
175 -
176 CHS
177 RCL 14
178  E
```

```
179 -
180 /
181 X<>Y
182 /
183 ACX
184 " N= "
185 FIX 0
186 ARCL 14
187 ACA
188 PRBUF
189 RTN
190*LBL 06
191 CLX
192 STO 08
193 RCL 15
194 RCL 15
195 19
196 +
197  E3
198 /
199 +
200 STO 03
201 69
202 ACCHR
203 40
204 ACCHR
205 102
206 ACCHR
207 41
208 ACCHR
209 61
210 ACCHR
211 RCL 14
212 20
213 /
214 FIX 1
215 ACX
216 FIX 0
217 RND
218 STO 07
219 STO 05
220 PRBUF
221 FS? 03
222 GTO 08
223 RCL 14
224 2
225 /
226 SQRT
227 RND
228 ST+ 05
229 -
230 X<=0?
231 0
232 STO 04
233 ADV
234 ACX
235 5
236 SKPCHR
237 "FREQ"
238 ACA
239 4
240 SKPCHR
241 RCL 05
242 ACX
243 ADV
244 "------"
245 ASTO L
246 ARCL L
247 ARCL L
248 ACA
249 ADV
250 126
251 RCL 05
252 RCL 04
253 -
254 /
255 RCL 07
256 RCL 04
257 -
258 *
259 INT
260  E3
261 /
262 126
263 +
264 STO 06
265*LBL 07
266 RCL 03
267 INT
268 RCL 15
269 -
270 5
271 *
272 CLA
273 ARCL X
274 ACA
275 RCL IND 03
276 RCL 07
277 -
278 X^2
279 ST+ 08
280 RCL IND 03
281 ACX
282 RCL 04
283 RCL 05
284 RCL 06
285 STKPLOT
286 ISG 03
287 GTO 07
288*LBL 09
289 ADV
290 RCL 07
291 ST/ 08
292 FIX 2
293 "CHI-SQ"
294 "`<5%:30> ="
295 ACA
296 RCL 08
297 ACX
298 PRBUF
299 ADV
300 RTN
301*LBL 08
302 RCL IND 03
303 ACX
304 RCL 07
305 -
306 X^2
307 ST+ 08
308 ISG 03
309 GTO 08
310 GTO 09
311 END
```

Note that the initial seed is stored in R00 when first entered by the user, and that R00 is used to store all the random numbers as they're being generated in the sequence. This works well for data-register based p-RNGs but for buffer-based p-RNGs we may need to initialize them manually entering the seed, using the corresponding SEED function for each case. This is only needed if we want the results to be comparable across different p-RNG, of course.

2. *Randomicity Testing. (by Charles T. Tart)*

This program is designed for INTEGER random numbers and thus cannot be used for our cases- nevertheless it's got intrinsic value that warrant including in the RANROM. Here's the complete contents of the paper, available here.



A Radomicity Test Program for Pseudo-Random Number Generator Routines on the HP-41C

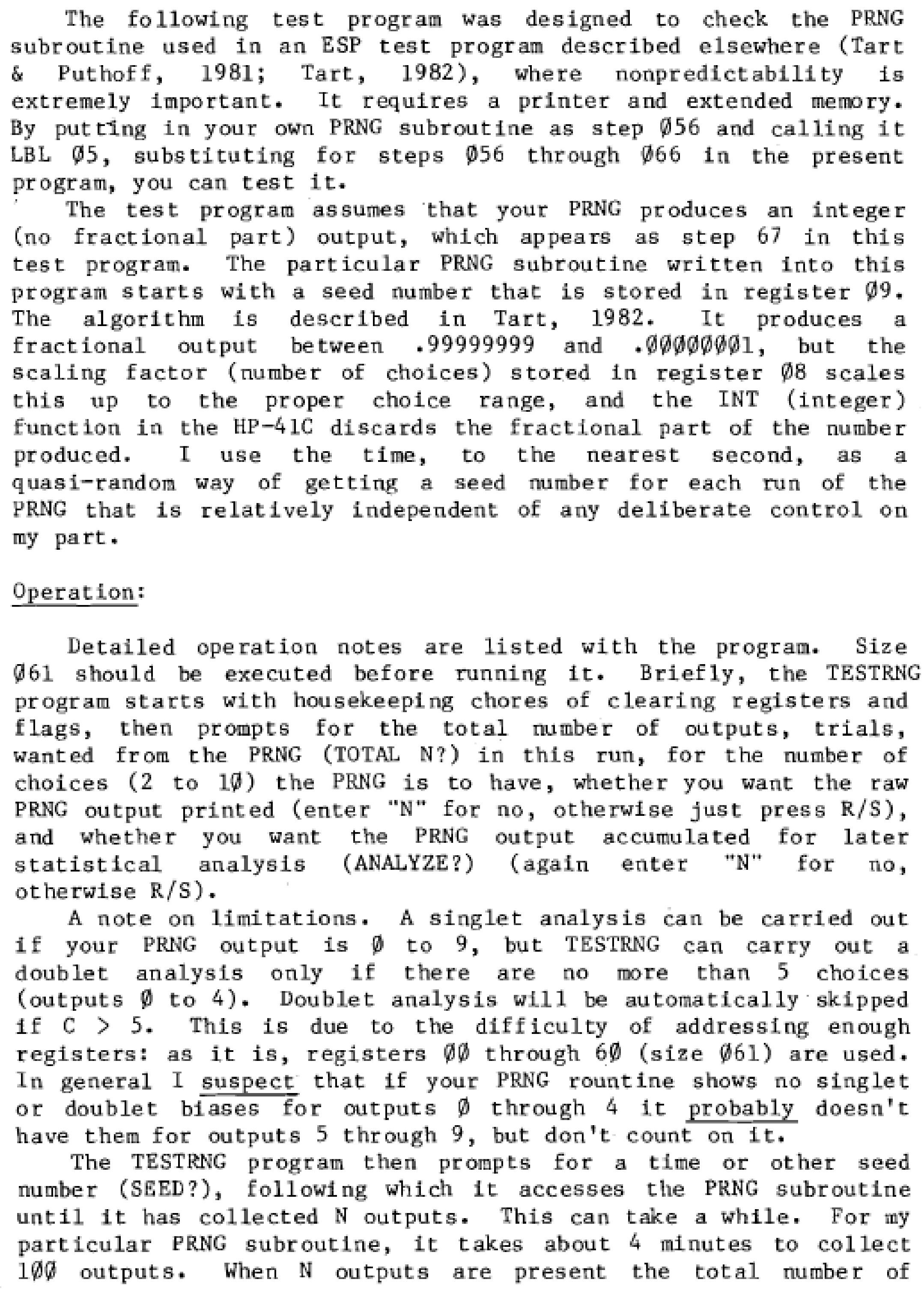
Charles T. Tart
Univeristy of California, Davis

The availability of high quality, relatively inexpensive programmable scientific calculators, such as the Hewlett-Packard HP-41C, offers a valuable new tool to parapsychologists. As described in another paper (Tart, 1982), the HP-41C can be programmed to provide a laboratory quality ESP test and feedback training instrument. A rapid evolution of features available can be expected in such "calculators" (they are really hand-held computers), so they may play a strong role in parapsychological research in the near future.

In using these instruments for ESP testing devices, it is essential that they have high quality subroutines for generating random numbers. At present we are limited to pseudo-random number generator (PRNG) subroutines, but with adequate algorithms for such routines and frequent changing of quasi-random seed values, it should not be too difficult to generate adequately random target numbers. This paper describes a test routine for evaluating a PRNG. The specific program is for the HP-41C, but it should be readily adaptable to any programmable scientific calculator.

A true RNG has two important properties. First, the probability of any output is equal to that of any other output, so as a series gets longer the proportion of any particular output to the total output approaches 1/C, where C is the number of output choices of the RNG. Second, there is no sequential dependency between numbers, that is, the probability of any number following some previous output number or sequence of such numbers is equal to that of any other number following that previous output number or sequence of numbers. This also means that no matter what output numbers have already been generated, they give you no useful way of predicting what the next outputs will be. (For fuller discussion of the non-predictability criterion see Tart, 1979a; 1979b; Tart & Dronek, 1980).

The adequacy of a RNG or PRNG can be evaluated statistically by testing a large sample of outputs for equiprobability and lack of sequential dependency. A common way of doing this testing is to count up not only single outputs (Ø, 1s, 2s, etc., called singlets), but also sequential outputs (Ø followed by Ø, Ø followed by 1, Ø followed by 2, etc., called doublets). Sequential output testing usually includes doublets, and, if there is a theoretical reason to suspect higher order sequential dependencies, triplets, quadruplets, etc.



The following test program was designed to check the PRNG subroutine used in an ESP test program described elsewhere (Tart & Puthoff, 1981; Tart, 1982), where nonpredictability is extremely important. It requires a printer and extended memory. By putting in your own PRNG subroutine as step Ø56 and calling it LBL Ø5, substituting for steps Ø56 through Ø66 in the present program, you can test it.

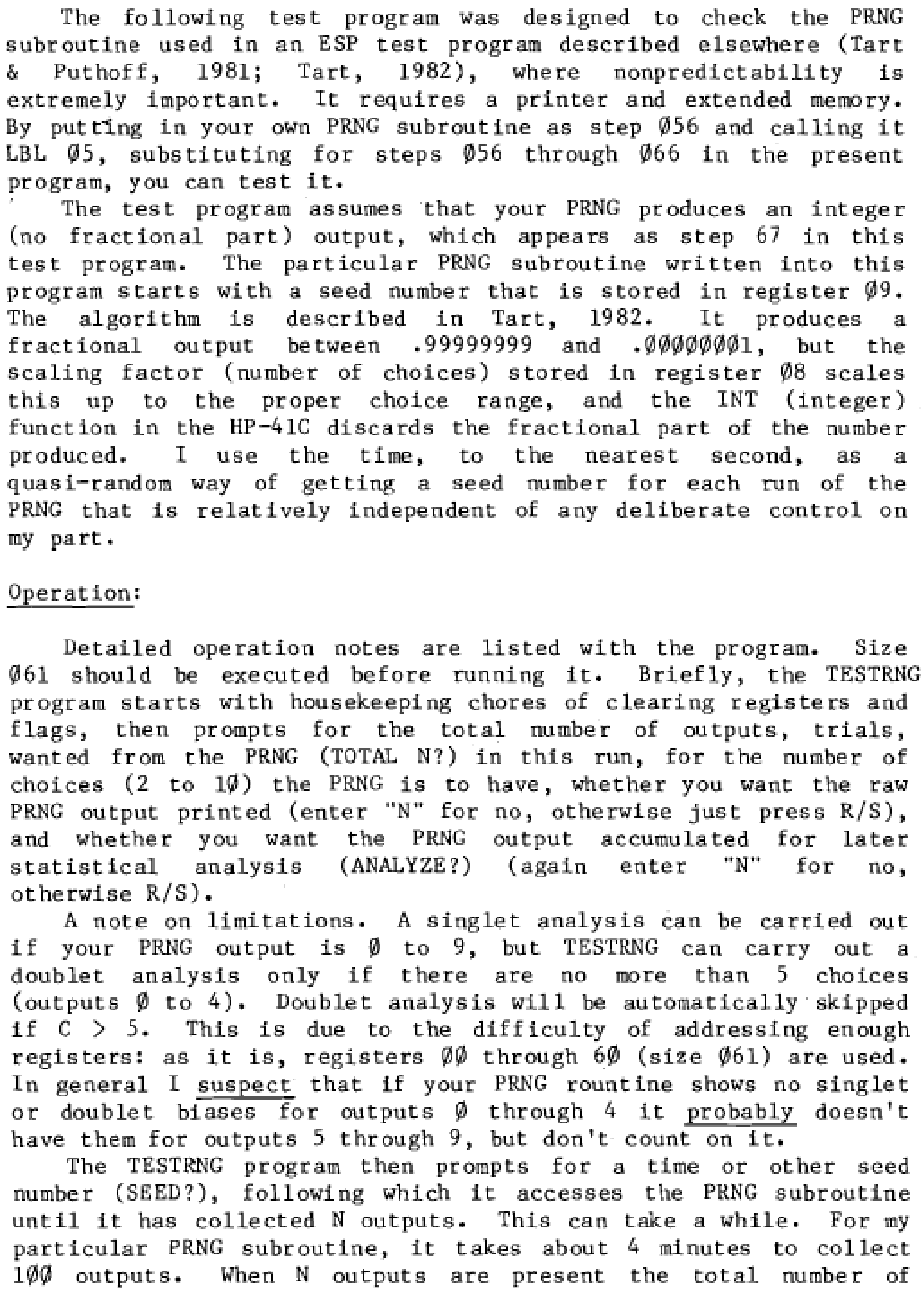
The test program assumes that your PRNG produces an integer (no fractional part) output, which appears as step 67 in this test program. The particular PRNG subroutine written into this program starts with a seed number that is stored in register Ø9. The algorithm is described in Tart, 1982. It produces a fractional output between .99999999 and .ØØØØØØØ1, but the scaling factor (number of choices) stored in register Ø8 scales this up to the proper choice range, and the INT (integer) function in the HP-41C discards the fractional part of the number produced. I use the time, to the nearest second, as a quasi-random way of getting a seed number for each run of the PRNG that is relatively independent of any deliberate control on my part.

Operation:

Detailed operation notes are listed with the program. Size Ø61 should be executed before running it. Briefly, the TESTRNG program starts with housekeeping chores of clearing registers and flags, then prompts for the total number of outputs, trials, wanted from the PRNG (TOTAL N?) in this run, for the number of choices (2 to 1Ø) the PRNG is to have, whether you want the raw PRNG output printed (enter "N" for no, otherwise just press R/S), and whether you want the PRNG output accumulated for later statistical analysis (ANALYZE?) (again enter "N" for no, otherwise R/S).

A note on limitations. A singlet analysis can be carried out if your PRNG output is Ø to 9, but TESTRNG can carry out a doublet analysis only if there are no more than 5 choices (outputs Ø to 4). Doublet analysis will be automatically skipped if C > 5. This is due to the difficulty of addressing enough registers: as it is, registers ØØ through 6Ø (size Ø61) are used. In general I suspect that if your PRNG rountine shows no singlet or doublet biases for outputs Ø through 4 it probably doesn't have them for outputs 5 through 9, but don't count on it.

The TESTRNG program then prompts for a time or other seed number (SEED?), following which it accesses the PRNG subroutine until it has collected N outputs. This can take a while. For my particular PRNG subroutine, it takes about 4 minutes to collect 1ØØ outputs. When N outputs are present the total number of



The following test program was designed to check the PRNG subroutine used in an ESP test program described elsewhere (Tart & Puthoff, 1981; Tart, 1982), where nonpredictability is extremely important. It requires a printer and extended memory. By putting in your own PRNG subroutine as step Ø56 and calling it LBL Ø5, substituting for steps Ø56 through Ø66 in the present program, you can test it.

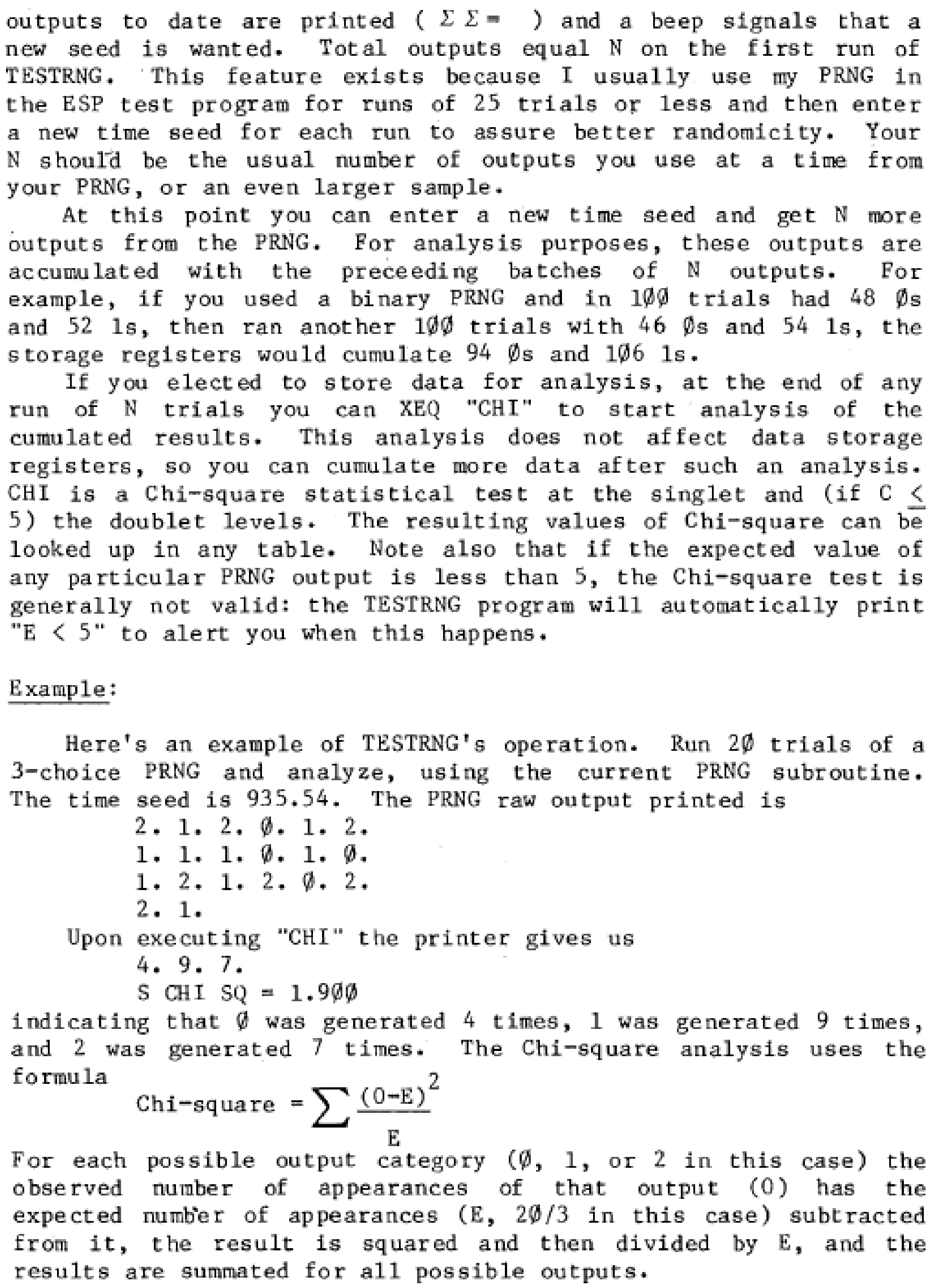
The test program assumes that your PRNG produces an integer (no fractional part) output, which appears as step 67 in this test program. The particular PRNG subroutine written into this program starts with a seed number that is stored in register Ø9. The algorithm is described in Tart, 1982. It produces a fractional output between .99999999 and .ØØØØØØØ1, but the scaling factor (number of choices) stored in register Ø8 scales this up to the proper choice range, and the INT (integer) function in the HP-41C discards the fractional part of the number produced. I use the time, to the nearest second, as a quasi-random way of getting a seed number for each run of the PRNG that is relatively independent of any deliberate control on my part.

Operation:

Detailed operation notes are listed with the program. Size Ø61 should be executed before running it. Briefly, the TESTRNG program starts with housekeeping chores of clearing registers and flags, then prompts for the total number of outputs, trials, wanted from the PRNG (TOTAL N?) in this run, for the number of choices (2 to 1Ø) the PRNG is to have, whether you want the raw PRNG output printed (enter "N" for no, otherwise just press R/S), and whether you want the PRNG output accumulated for later statistical analysis (ANALYZE?) (again enter "N" for no, otherwise R/S).

A note on limitations. A singlet analysis can be carried out if your PRNG output is Ø to 9, but TESTRNG can carry out a doublet analysis only if there are no more than 5 choices (outputs Ø to 4). Doublet analysis will be automatically skipped if C > 5. This is due to the difficulty of addressing enough registers: as it is, registers ØØ through 6Ø (size Ø61) are used. In general I suspect that if your PRNG rountine shows no singlet or doublet biases for outputs Ø through 4 it probably doesn't have them for outputs 5 through 9, but don't count on it.

The TESTRNG program then prompts for a time or other seed number (SEED?), following which it accesses the PRNG subroutine until it has collected N outputs. This can take a while. For my particular PRNG subroutine, it takes about 4 minutes to collect 1ØØ outputs. When N outputs are present the total number of



outputs to date are printed (ΣΣ=) and a beep signals that a new seed is wanted. Total outputs equal N on the first run of TESTRNG. This feature exists because I usually use my PRNG in the ESP test program for runs of 25 trials or less and then enter a new time seed for each run to assure better randomicity. Your N should be the usual number of outputs you use at a time from your PRNG, or an even larger sample.

At this point you can enter a new time seed and get N more outputs from the PRNG. For analysis purposes, these outputs are accumulated with the preceeding batches of N outputs. For example, if you used a binary PRNG and in 1ØØ trials had 48 Øs and 52 1s, then ran another 1ØØ trials with 46 Øs and 54 1s, the storage registers would cumulate 94 Øs and 1Ø6 1s.

If you elected to store data for analysis, at the end of any run of N trials you can XEQ "CHI" to start analysis of the cumulated results. This analysis does not affect data storage registers, so you can cumulate more data after such an analysis. CHI is a Chi-square statistical test at the singlet and (if C $\leq$ 5) the doublet levels. The resulting values of Chi-square can be looked up in any table. Note also that if the expected value of any particular PRNG output is less than 5, the Chi-square test is generally not valid: the TESTRNG program will automatically print "E < 5" to alert you when this happens.

Example:

Here's an example of TESTRNG's operation. Run 2Ø trials of a 3-choice PRNG and analyze, using the current PRNG subroutine. The time seed is 935.54. The PRNG raw output printed is

```
2. 1. 2. Ø. 1. 2.
1. 1. 1. Ø. 1. Ø.
1. 2. 1. 2. Ø. 2.
2. 1.
```

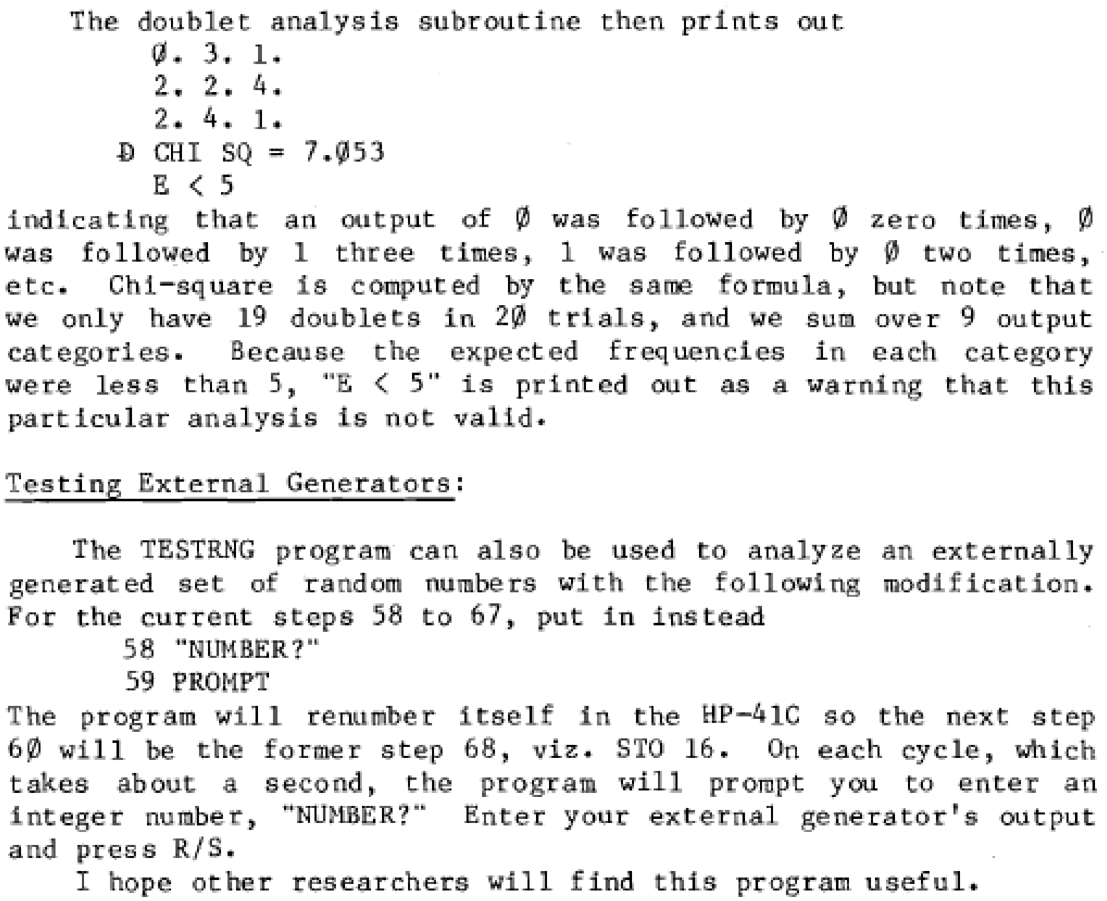
Upon executing "CHI" the printer gives us

```
4. 9. 7.
S CHI SQ = 1.9ØØ
```

indicating that Ø was generated 4 times, 1 was generated 9 times, and 2 was generated 7 times. The Chi-square analysis uses the formula

$$\text{Chi-square} = \sum \frac{(O-E)^2}{E}$$

For each possible output category (Ø, 1, or 2 in this case) the observed number of appearances of that output (O) has the expected number of appearances (E, 2Ø/3 in this case) subtracted from it, the result is squared and then divided by E, and the results are summated for all possible outputs.



The doublet analysis subroutine then prints out

```
    Ø. 3. 1.
    2. 2. 4.
    2. 4. 1.
  Đ CHI SQ = 7.Ø53
    E < 5
```

indicating that an output of Ø was followed by Ø zero times, Ø was followed by 1 three times, 1 was followed by Ø two times, etc. Chi-square is computed by the same formula, but note that we only have 19 doublets in 2Ø trials, and we sum over 9 output categories. Because the expected frequencies in each category were less than 5, "E < 5" is printed out as a warning that this particular analysis is not valid.

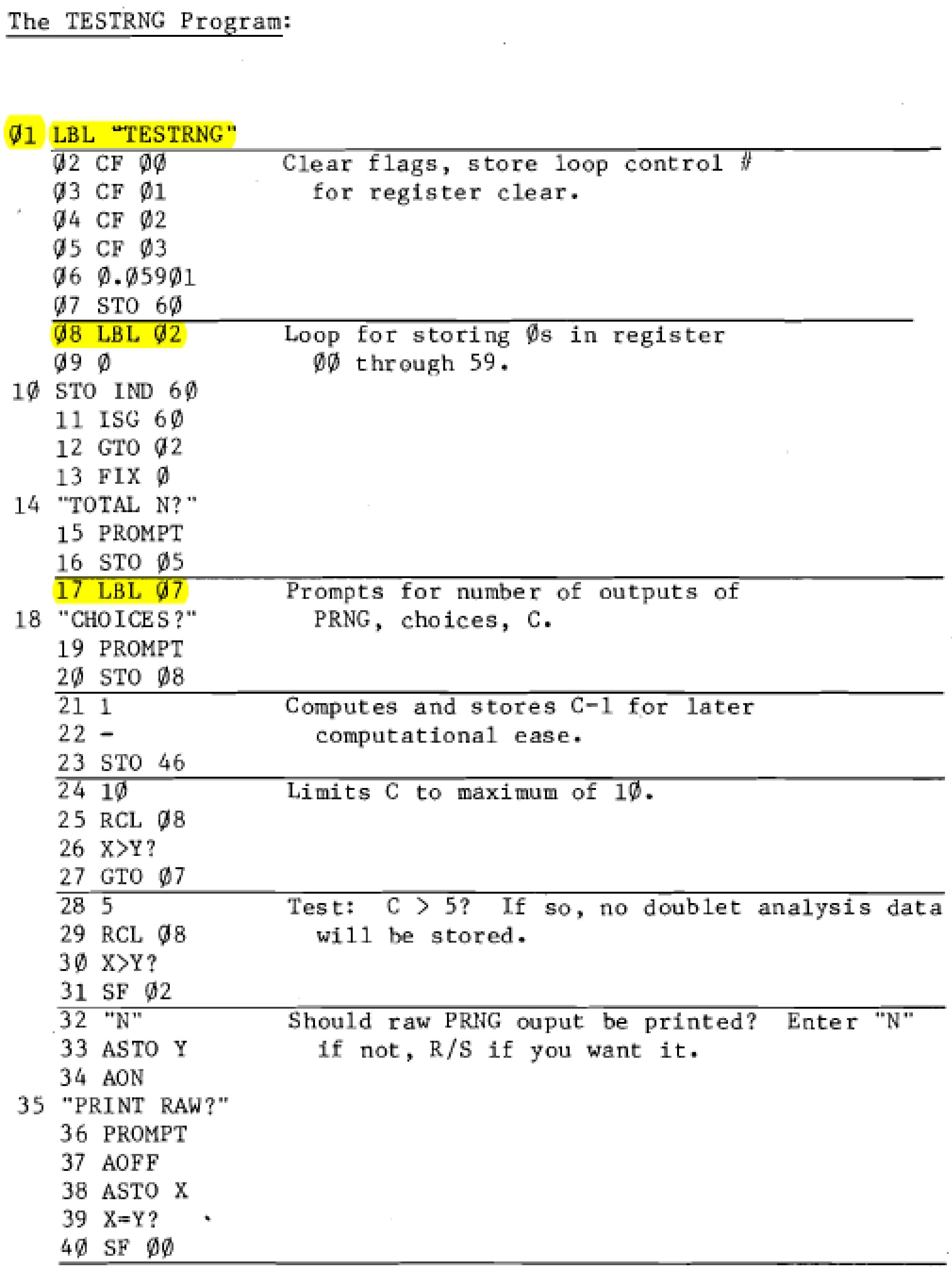
Testing External Generators:

The TESTRNG program can also be used to analyze an externally generated set of random numbers with the following modification. For the current steps 58 to 67, put in instead

```
58 "NUMBER?"
59 PROMPT
```

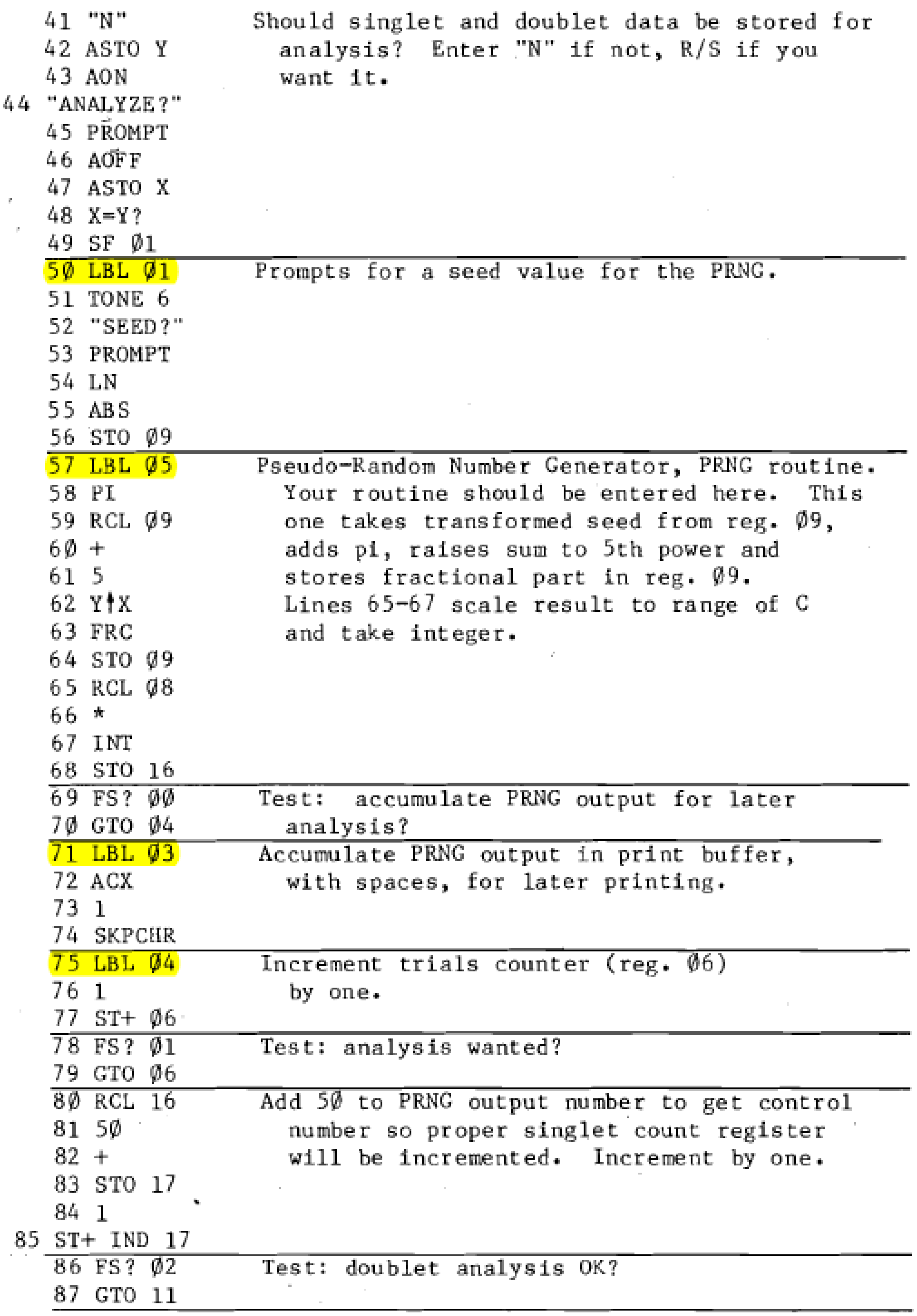
The program will renumber itself in the HP-41C so the next step 6Ø will be the former step 68, viz. STO 16. On each cycle, which takes about a second, the program will prompt you to enter an integer number, "NUMBER?" Enter your external generator's output and press R/S.

I hope other researchers will find this program useful.

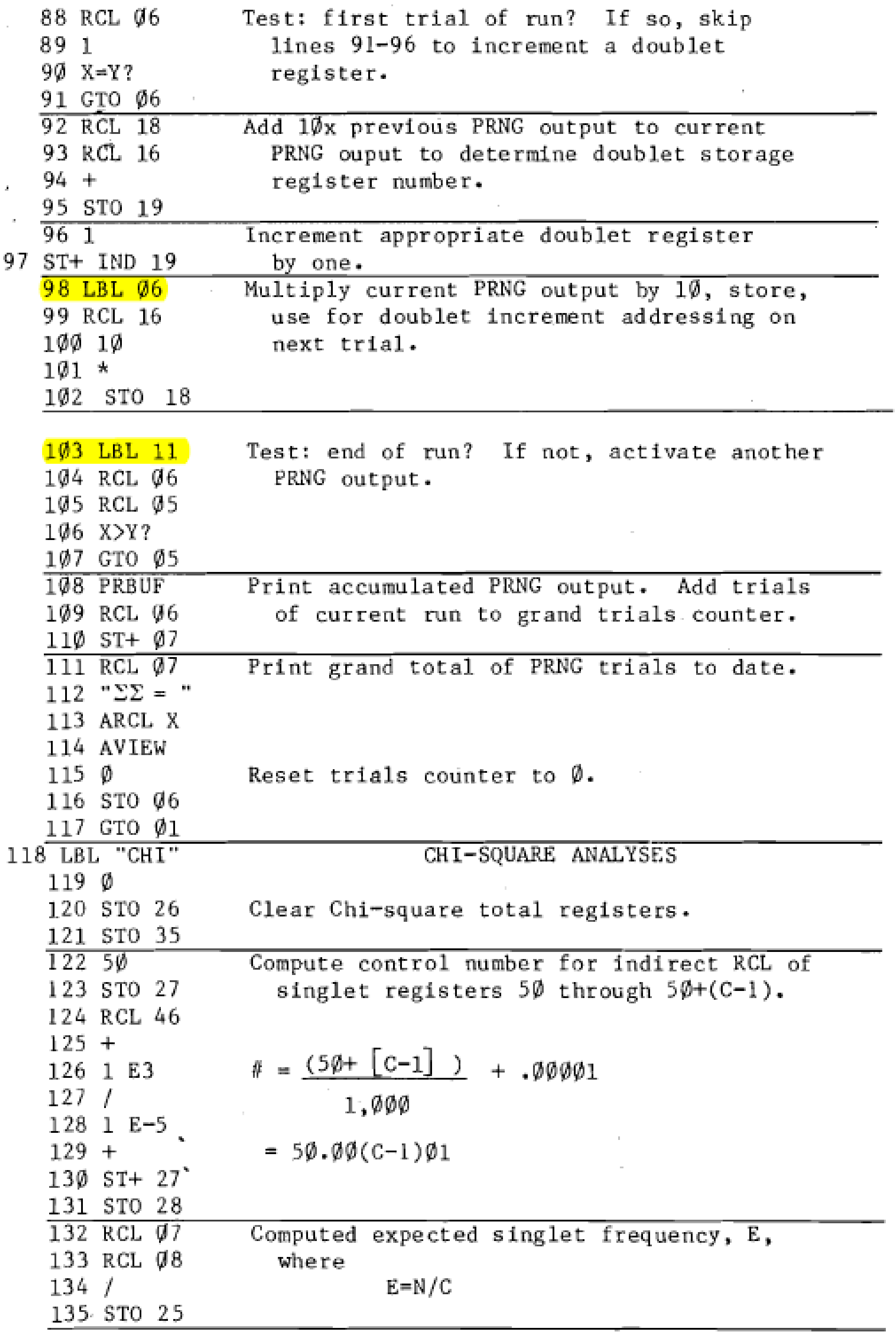


The TESTRNG Program:

```
Ø1 LBL "TESTRNG"
   Ø2 CF ØØ          Clear flags, store loop control #
   Ø3 CF Ø1            for register clear.
   Ø4 CF Ø2
   Ø5 CF Ø3
   Ø6 Ø.Ø59Ø1
   Ø7 STO 6Ø
   Ø8 LBL Ø2         Loop for storing Øs in register
   Ø9 Ø                ØØ through 59.
1Ø STO IND 6Ø
   11 ISG 6Ø
   12 GTO Ø2
   13 FIX Ø
14 "TOTAL N?"
   15 PROMPT
   16 STO Ø5
   17 LBL Ø7         Prompts for number of outputs of
18 "CHOICES?"          PRNG, choices, C.
   19 PROMPT
   2Ø STO Ø8
   21 1              Computes and stores C-1 for later
   22 -                computational ease.
   23 STO 46
   24 1Ø             Limits C to maximum of 1Ø.
   25 RCL Ø8
   26 X>Y?
   27 GTO Ø7
   28 5              Test:  C > 5?  If so, no doublet analysis data
   29 RCL Ø8           will be stored.
   3Ø X>Y?
   31 SF Ø2
   32 "N"            Should raw PRNG ouput be printed?  Enter "N"
   33 ASTO Y           if not, R/S if you want it.
   34 AON
35 "PRINT RAW?"
   36 PROMPT
   37 AOFF
   38 ASTO X
   39 X=Y?
   4Ø SF ØØ
```

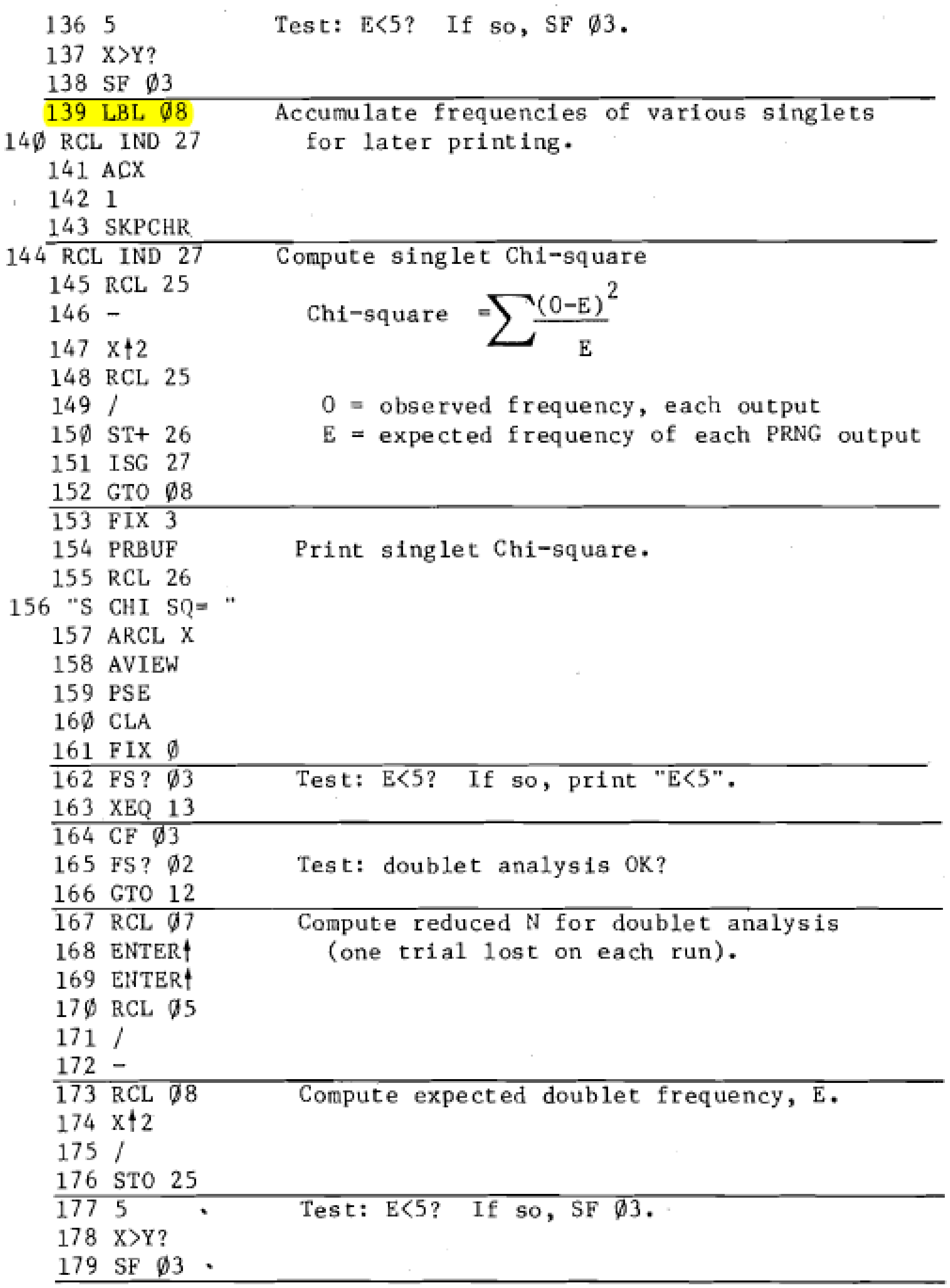


```
 41 "N"           Should singlet and doublet data be stored for
 42 ASTO Y          analysis?  Enter "N" if not, R/S if you
 43 AON             want it.
44 "ANALYZE?"
 45 PROMPT
 46 AOFF
 47 ASTO X
 48 X=Y?
 49 SF Ø1
 5Ø LBL Ø1        Prompts for a seed value for the PRNG.
 51 TONE 6
 52 "SEED?"
 53 PROMPT
 54 LN
 55 ABS
 56 STO Ø9
 57 LBL Ø5        Pseudo-Random Number Generator, PRNG routine.
 58 PI              Your routine should be entered here.  This
 59 RCL Ø9          one takes transformed seed from reg. Ø9,
 6Ø +                adds pi, raises sum to 5th power and
 61 5               stores fractional part in reg. Ø9.
 62 Y↑X             Lines 65-67 scale result to range of C
 63 FRC             and take integer.
 64 STO Ø9
 65 RCL Ø8
 66 *
 67 INT
 68 STO 16
 69 FS? ØØ        Test:  accumulate PRNG output for later
 7Ø GTO Ø4          analysis?
 71 LBL Ø3        Accumulate PRNG output in print buffer,
 72 ACX             with spaces, for later printing.
 73 1
 74 SKPCHR
 75 LBL Ø4        Increment trials counter (reg. Ø6)
 76 1               by one.
 77 ST+ Ø6
 78 FS? Ø1        Test: analysis wanted?
 79 GTO Ø6
 8Ø RCL 16        Add 5Ø to PRNG output number to get control
 81 5Ø              number so proper singlet count register
 82 +               will be incremented.  Increment by one.
 83 STO 17
 84 1
85 ST+ IND 17
 86 FS? Ø2        Test: doublet analysis OK?
 87 GTO 11
```



```
   88 RCL Ø6        Test: first trial of run?  If so, skip
   89 1               lines 91-96 to increment a doublet
   9Ø X=Y?            register.
   91 GTO Ø6
   ----------------------------------------------------------
   92 RCL 18        Add 1Øx previous PRNG output to current
   93 RCL 16          PRNG ouput to determine doublet storage
   94 +               register number.
   95 STO 19
   ----------------------------------------------------------
   96 1             Increment appropriate doublet register
97 ST+ IND 19         by one.
   ----------------------------------------------------------
   98 LBL Ø6        Multiply current PRNG output by 1Ø, store,
   99 RCL 16          use for doublet increment addressing on
   1ØØ 1Ø             next trial.
   1Ø1 *
   1Ø2  STO 18
   ----------------------------------------------------------

   1Ø3 LBL 11       Test: end of run?  If not, activate another
   1Ø4 RCL Ø6         PRNG output.
   1Ø5 RCL Ø5
   1Ø6 X>Y?
   1Ø7 GTO Ø5
   ----------------------------------------------------------
   1Ø8 PRBUF        Print accumulated PRNG output.  Add trials
   1Ø9 RCL Ø6         of current run to grand trials counter.
   11Ø ST+ Ø7
   ----------------------------------------------------------
   111 RCL Ø7       Print grand total of PRNG trials to date.
   112 "ΣΣ = "
   113 ARCL X
   114 AVIEW
   115 Ø            Reset trials counter to Ø.
   116 STO Ø6
   117 GTO Ø1
   ----------------------------------------------------------
 118 LBL "CHI"                    CHI-SQUARE ANALYSES
   119 Ø
   120 STO 26       Clear Chi-square total registers.
   121 STO 35
   ----------------------------------------------------------
   122 5Ø           Compute control number for indirect RCL of
   123 STO 27         singlet registers 5Ø through 5Ø+(C-1).
   124 RCL 46
   125 +
   126 1 E3         # = (5Ø+ [C-1] )  + .ØØØØ1
   127 /                 ------------
   128 1 E-5                1,ØØØ
   129 +
   13Ø ST+ 27         = 5Ø.ØØ(C-1)Ø1
   131 STO 28
   ----------------------------------------------------------
   132 RCL Ø7       Computed expected singlet frequency, E,
   133 RCL Ø8         where
   134 /                        E=N/C
   135 STO 25
   ----------------------------------------------------------
```



```
  136 5             Test: E<5?  If so, SF Ø3.
  137 X>Y?
  138 SF Ø3
  139 LBL Ø8        Accumulate frequencies of various singlets
140 RCL IND 27        for later printing.
  141 ACX
  142 1
  143 SKPCHR
144 RCL IND 27      Compute singlet Chi-square
  145 RCL 25                                  2
  146 -               Chi-square  =∑(0-E)
  147 X↑2                                E
  148 RCL 25
  149 /               0 = observed frequency, each output
  150 ST+ 26          E = expected frequency of each PRNG output
  151 ISG 27
  152 GTO Ø8
  153 FIX 3
  154 PRBUF           Print singlet Chi-square.
  155 RCL 26
156 "S CHI SQ= "
  157 ARCL X
  158 AVIEW
  159 PSE
  16Ø CLA
  161 FIX Ø
  162 FS? Ø3          Test: E<5?  If so, print "E<5".
  163 XEQ 13
  164 CF Ø3
  165 FS? Ø2          Test: doublet analysis OK?
  166 GTO 12
  167 RCL Ø7          Compute reduced N for doublet analysis
  168 ENTER↑            (one trial lost on each run).
  169 ENTER↑
  17Ø RCL Ø5
  171 /
  172 -
  173 RCL Ø8          Compute expected doublet frequency, E.
  174 X↑2
  175 /
  176 STO 25
  177 5               Test: E<5?  If so, SF Ø3.
  178 X>Y?
  179 SF Ø3
```

3. *Stratified Random Sampling (by rawi, MoHP Articles)*

see: *https://www.hpmuseum.org/forum/thread-15709.html*

The following is taken verbatim from the MoHP post.

This is my first program for the HP 41 since decades. So it may be that there are numerous ways to improve the code. I have tested it on a DM 41X and a HP 41CL.

What it does:

The program helps to analyze stratified random samples.
You can either put in the weights (i.e. the shares of the strata in universe) and the standard deviations within the strata or you can put in a density function and the weights of the strata and the standard deviations within the strata are computed.

For the allocation of the sample on the strata you can put in numbers or let the program compute sample sizes for proportional or optimal allocation.

Standard deviation of total mean is computed.

Needs command "FINTG" from the SandMath Module (or INTEG from the Advantage Pac).

Use of registers:

R00 - Final result
R01 - Name of density function global label (Alpha)
R02 - Number of strata (maximum: 6)
R03 - Sample size
R04 - Lower limit of first stratum

Stratum	*1*	*2*	*3*	*4*	*5*	*6*
Upper limit	*R05*	*R06*	*R07*	*R08*	*R09*	*R10*
Weights	*R11*	*R12*	*R13*	*R14*	*R15*	*R16*
Std. dev.	*R17*	*R18*	*R19*	*R20*	*R21*	*R22*

RegisterS R23-R30 are used for computations.

The function FINTG from SandMath module needs another 32 unused program registers.
Usage instructions: See program listing.

Example:

You want to analyze a stratified sample with 4 strata and optimum allocation of sample. The variable has a standard normal distribution. Strata limits are -1, 0, 1. Total sample size is n=1000.
For convenience you take as lower limit of standard normal distribution -7.5 and as upper limit 7.5.

First type in routine for normal distribution under a global label:

```
01 LBL "NV"
02 X^2
03 2
04 /
05 CHS
06 E^X
07 2
08 PI
09 *
10 SQRT
11 /
12 RTN
```

XEQ "STRAT"

```
-> SMPL SZ?        1000 ,  R/S (sample size input)
-> N STRATA?       4 ,  R/S (input of number of strata)
-> 0:I  1:DF?      1 , R/S (we use the densitiy function, so we type 1)
-> NM DF?          "NV" , R/S (input of label of program with density function)
-> LL  S1?         7.5 CHS ,  R/S (lower limit of first stratum is -7.5)
-> UL  S1?         1 CHS ,  R/S  (upper limit of first stratum is -1)
-> UL  S2?         0 ,  R/S
-> UL  S3?         1 ,  R/S
-> UL  S4?         7.5 ,  R/S
```

After about 15 minutes (HP 41) / 20 seconds (DM 41X with USB cable):

```
->        0:I  1:P  2:O?      2 ,  R/S (we want optimum allocation, so we type 2)
->        N1:212              (sample size in stratum 1)
R/S ->    N2  =  288          (sample size in stratum 2)
R/S ->    N3  =  288
R/S ->    N5  =  212
R/S ->    0.0106              (std deviation of stratified sample mean)
```

Weights and standard deviations can be seen in registers, e.g. weight in stratum 1:

```
RCL 11 ->   0.1587
```

Program listing.

```
01  LBL STRAT
02  "SMPL SZ?"
03  PROMPT          Input of sample size
04  STO 03
05  "N STRATA?"
06  PROMPT          Input of number of strata (maximum: 6)
07  1 E3
08  /
09  1
10  +
11  STO 02
12  0
13  STO 00
14  STO 27
15  "0=I 1=DF?"        Input whether weights and std. dev. of strata are individually typed
16  PROMPT          (0) or whether they are computed by given density function (1)
17  4
18  +
19  XEQ IND X
20  "0=I 1=P 2=O?"      Input whether allocation of sample on strata should be individually
21  PROMPT          given (0), proportional to weights (1) or optimal (2)
22  STO 30
23  FIX 0
24  10
25  STO 28
26  16
27  STO 29
28  LBL 08        Determination of sample size in strata
29  1
30  ST+ 28
31  ST+ 29
32  RCL IND 28
33  RCL IND 29
34  XEQ IND 30
35  RCL IND 28
36  RCL IND 29
37  *
38  x^2
39  x<>y
40  /
41  ST+ 00
42  ISG 02
43  GTO 08
```

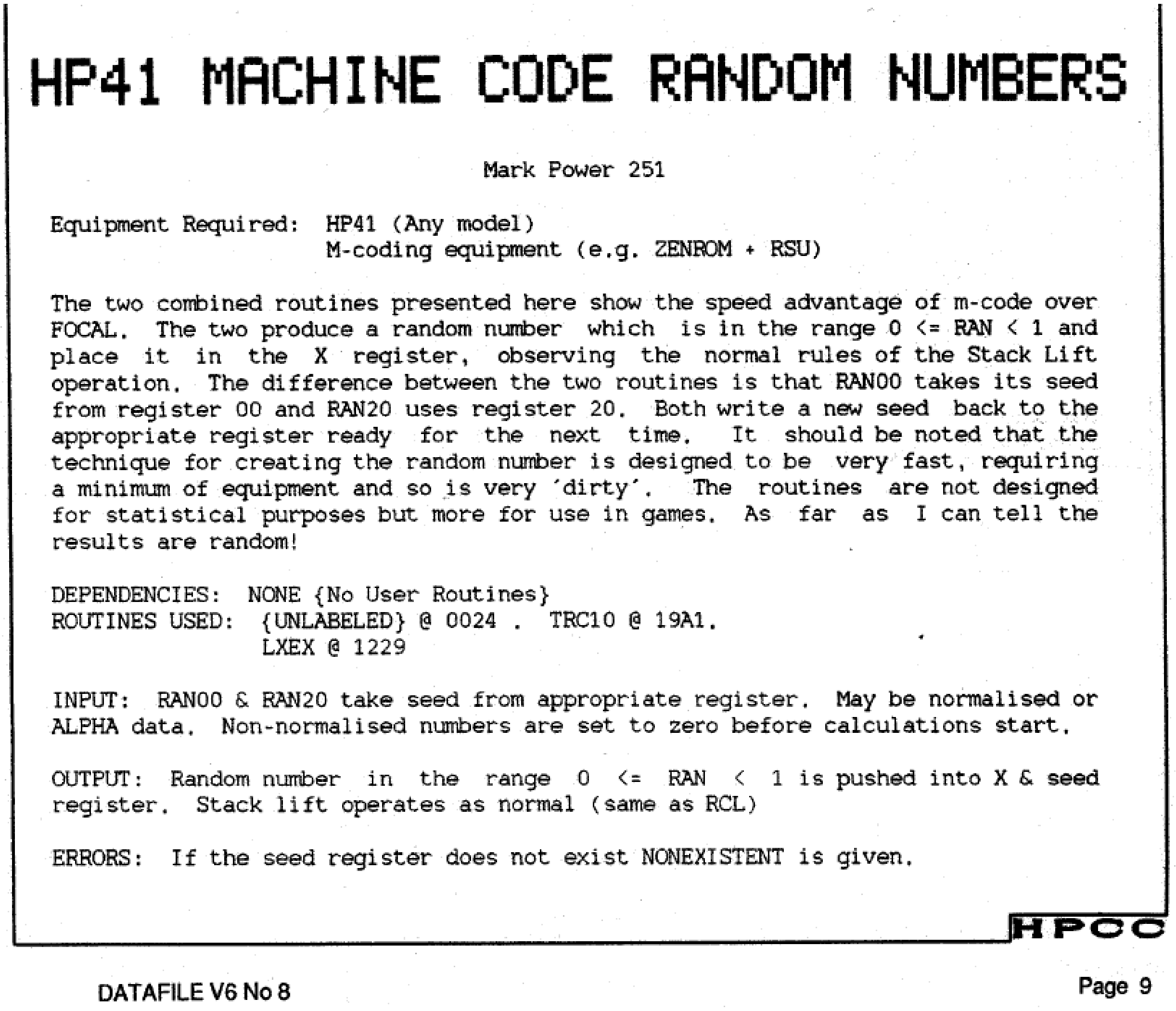
```
44  FIX 4
45  RCL 00
46  SQRT
47  STO 00          Standard dev. of mean of total sample is shown
48  CF 01
49  RTN
50  LBL 04          Manual Input of weights and std. dev. in strata
51  10
52  STO 28
53  16
54  STO 29
55  FIX 0
56  LBL 06
57  1
58  ST+ 28
59  ST+ 29
60  “W”
61  ARCL 02
62  “|-?”
63  PROMPT
64  STO IND 28
65  “S”
66  ARCL 02
67  “|-?
68  PROMPT
69  STO IND 29
70  *
71  ST+ 27
72  ISG 02
73  GTO 06
74  XEQ 09
75  RTN
76  LBL 05          Computation of weights and std. dev. in strata with given function
77  AON
78  FIX 0
79  “NM DF?”        Asks for name of global program label with density function
80  PROMPT
81  ASTO 01
82  AOFF
83  4
84  STO 28
85  “LL S1?”        Asks for lower limit of stratum 1
86  PROMPT
87  STO 04
88  STO 23
89  LBL 07
90  1
```

```
91    ST+ 28
92    "UL S"
93    ARCL 02
94    "|-?"
95    PROMPT          Asks for upper limit of stratum l
96    STO IND 28
97    ISG 02
98    GTO 07
99    FIX 5          Format defines accuracy of integration
100   4
101   STO 28
102   10
103   STO 29
104   XEQ 09
105   LBL 10          Computation of parameters of strata
106   1
107   ST+ 28
108   ST+ 29
109   CLA
110   CF 01
111   ARCL 01
112   RCL 23
113   RCL IND 28
114   STO 24
115   INTEG           Computation of weight of stratum l = integral(f(x))
116   STO IND 29
117   STO 25
118   "STRAX"
119   RCL 23
120   RCL 24
121   INTEG           Computation of integral(x*f(x))
122   RCL 25
123   /
124   STO 26          Mean in stratum
125   SF 01
126   RCL 23
127   RCL 24
128   STO 23
129   INTEG           Computation of integral(x²*f(x))
130   RCL 25
131   /
132   RCL 26
133   X^2
134   -
135   SQRT           std. dev. in stratum
136   RCL 29
137   6
```

```
138   +
139   x<>y
140   STO IND Y
141   RCL 25
142   *               weight in stratum * std. dev. in stratum
143   ST+ 27
144   ISG 02
145   GTO 10
146   XEQ 09
147   RTN
148   LBL 09           Refreshing loop register 02
149   RCL 02
150   FRC
151   1
152   +
153   STO 02
154   RTN
155   LBL 00           Manual input of sample size in stratum
156   "N"
157   ARCL 02
158   "|-?"
159   PROMPT
160   RTN
161   LBL 01          Computation of proportional sample size in stratum
162   X<>Y                     (share of stratum in sample and in universe are equal)
163   RCL 03
164   *
165   GTO 03
166   LBL 02        Computation of optimum sample size in stratum
167   *             (proportional to weight*std. dev in stratum)
168   RCL 27
169   /
170   RCL 03
171   *
172   LBL 03        Output of sample size in stratum
173   AON
174   "N"
175   ARCL 02
176   "|-="
177   ARCL X
178   STOP
179   AOFF
180    END
```

```
01  LBL "STRAX"        Subroutine for computation of integral x*f(x) and x²* f(x)
02  STO 30        depending of status of flag 01
03  XEQ IND 01
04  RCL 30
05  FS? 01
06  x^2
07  *
08  END
```

Appendix – Original DataFile Article by Mark Power



HP41 MACHINE CODE RANDOM NUMBERS

Mark Power 251

Equipment Required: HP41 (Any model)
M-coding equipment (e.g. ZENROM + RSU)

The two combined routines presented here show the speed advantage of m-code over FOCAL. The two produce a random number which is in the range 0 <= RAN < 1 and place it in the X register, observing the normal rules of the Stack Lift operation. The difference between the two routines is that RAN00 takes its seed from register 00 and RAN20 uses register 20. Both write a new seed back to the appropriate register ready for the next time. It should be noted that the technique for creating the random number is designed to be very fast, requiring a minimum of equipment and so is very 'dirty'. The routines are not designed for statistical purposes but more for use in games. As far as I can tell the results are random!

DEPENDENCIES: NONE {No User Routines}
ROUTINES USED: {UNLABELED} @ 0024 . TRC10 @ 19A1.
LXEX @ 1229

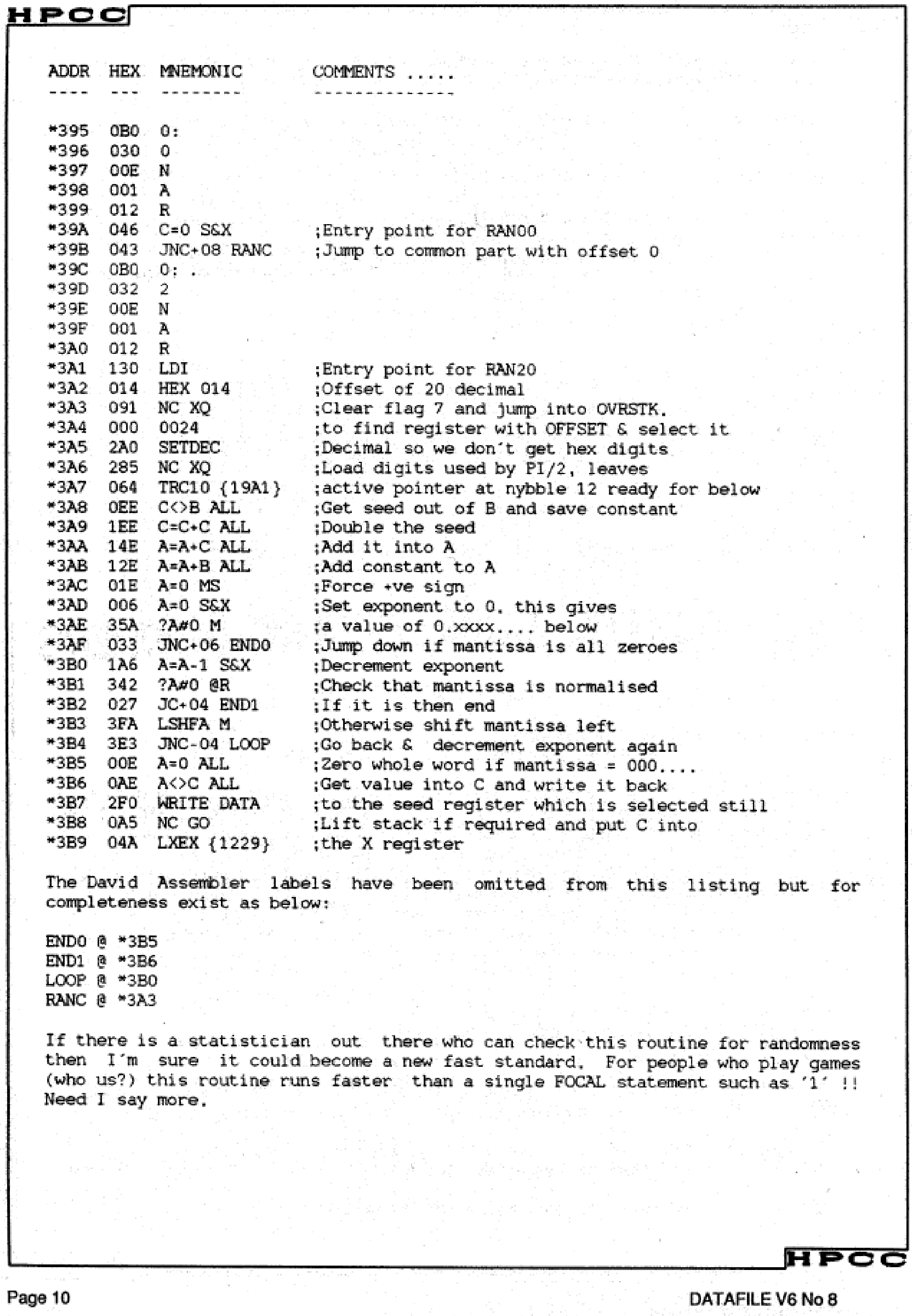
INPUT: RAN00 & RAN20 take seed from appropriate register. May be normalised or ALPHA data. Non-normalised numbers are set to zero before calculations start.

OUTPUT: Random number in the range 0 <= RAN < 1 is pushed into X & seed register. Stack lift operates as normal (same as RCL)

ERRORS: If the seed register does not exist NONEXISTENT is given.

HPCC

DATAFILE V6 No 8 Page 9



```
ADDR  HEX  MNEMONIC        COMMENTS .....
----  ---  --------        --------------

*395  0B0  0:
*396  030  0
*397  00E  N
*398  001  A
*399  012  R
*39A  046  C=0 S&X         ;Entry point for RAN00
*39B  043  JNC+08 RANC     ;Jump to common part with offset 0
*39C  0B0  0: .
*39D  032  2
*39E  00E  N
*39F  001  A
*3A0  012  R
*3A1  130  LDI             ;Entry point for RAN20
*3A2  014  HEX 014         ;Offset of 20 decimal
*3A3  091  NC XQ           ;Clear flag 7 and jump into OVRSTK.
*3A4  000  0024            ;to find register with OFFSET & select it
*3A5  2A0  SETDEC          ;Decimal so we don't get hex digits
*3A6  285  NC XQ           ;Load digits used by PI/2, leaves
*3A7  064  TRC10 {19A1}    ;active pointer at nybble 12 ready for below
*3A8  0EE  C<>B ALL        ;Get seed out of B and save constant
*3A9  1EE  C=C+C ALL       ;Double the seed
*3AA  14E  A=A+C ALL       ;Add it into A
*3AB  12E  A=A+B ALL       ;Add constant to A
*3AC  01E  A=0 MS          ;Force +ve sign
*3AD  006  A=0 S&X         ;Set exponent to 0, this gives
*3AE  35A  ?A#0 M          ;a value of 0.xxxx.... below
*3AF  033  JNC+06 END0     ;Jump down if mantissa is all zeroes
*3B0  1A6  A=A-1 S&X       ;Decrement exponent
*3B1  342  ?A#0 @R         ;Check that mantissa is normalised
*3B2  027  JC+04 END1      ;If it is then end
*3B3  3FA  LSHFA M         ;Otherwise shift mantissa left
*3B4  3E3  JNC-04 LOOP     ;Go back &  decrement exponent again
*3B5  00E  A=0 ALL         ;Zero whole word if mantissa = 000....
*3B6  0AE  A<>C ALL        ;Get value into C and write it back
*3B7  2F0  WRITE DATA      ;to the seed register which is selected still
*3B8  0A5  NC GO           ;Lift stack if required and put C into
*3B9  04A  LXEX {1229}     ;the X register
```

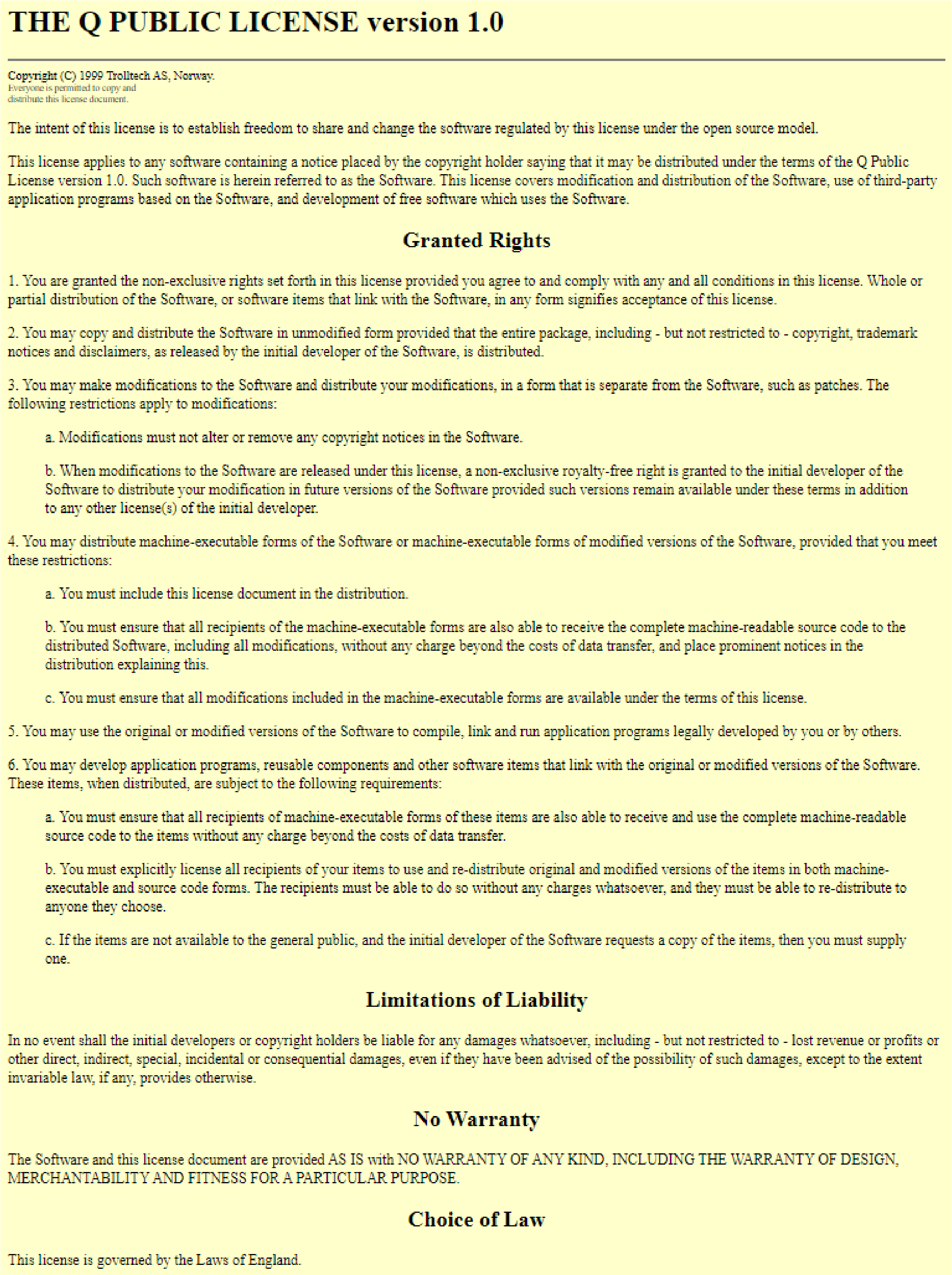
The David Assembler labels have been omitted from this listing but for completeness exist as below:

```
END0 @ *3B5
END1 @ *3B6
LOOP @ *3B0
RANC @ *3A3
```

If there is a statistician out there who can check this routine for randomness then I'm sure it could become a new fast standard. For people who play games (who us?) this routine runs faster than a single FOCAL statement such as '1' !! Need I say more.

Page 10 DATAFILE V6 No 8

Appendix.- Q Public Licence.



THE Q PUBLIC LICENSE version 1.0

The intent of this license is to establish freedom to share and change the software regulated by this license under the open source model.

This license applies to any software containing a notice placed by the copyright holder saying that it may be distributed under the terms of the Q Public License version 1.0. Such software is herein referred to as the Software. This license covers modification and distribution of the Software, use of third-party application programs based on the Software, and development of free software which uses the Software.

Granted Rights

1. You are granted the non-exclusive rights set forth in this license provided you agree to and comply with any and all conditions in this license. Whole or partial distribution of the Software, or software items that link with the Software, in any form signifies acceptance of this license.

2. You may copy and distribute the Software in unmodified form provided that the entire package, including - but not restricted to - copyright, trademark notices and disclaimers, as released by the initial developer of the Software, is distributed.

3. You may make modifications to the Software and distribute your modifications, in a form that is separate from the Software, such as patches. The following restrictions apply to modifications:

> a. Modifications must not alter or remove any copyright notices in the Software.
>
> b. When modifications to the Software are released under this license, a non-exclusive royalty-free right is granted to the initial developer of the Software to distribute your modification in future versions of the Software provided such versions remain available under these terms in addition to any other license(s) of the initial developer.

4. You may distribute machine-executable forms of the Software or machine-executable forms of modified versions of the Software, provided that you meet these restrictions:

> a. You must include this license document in the distribution.
>
> b. You must ensure that all recipients of the machine-executable forms are also able to receive the complete machine-readable source code to the distributed Software, including all modifications, without any charge beyond the costs of data transfer, and place prominent notices in the distribution explaining this.
>
> c. You must ensure that all modifications included in the machine-executable forms are available under the terms of this license.

5. You may use the original or modified versions of the Software to compile, link and run application programs legally developed by you or by others.

6. You may develop application programs, reusable components and other software items that link with the original or modified versions of the Software. These items, when distributed, are subject to the following requirements:

> a. You must ensure that all recipients of machine-executable forms of these items are also able to receive and use the complete machine-readable source code to the items without any charge beyond the costs of data transfer.
>
> b. You must explicitly license all recipients of your items to use and re-distribute original and modified versions of the items in both machine-executable and source code forms. The recipients must be able to do so without any charges whatsoever, and they must be able to re-distribute to anyone they choose.
>
> c. If the items are not available to the general public, and the initial developer of the Software requests a copy of the items, then you must supply one.

Limitations of Liability

In no event shall the initial developers or copyright holders be liable for any damages whatsoever, including - but not restricted to - lost revenue or profits or other direct, indirect, special, incidental or consequential damages, even if they have been advised of the possibility of such damages, except to the extent invariable law, if any, provides otherwise.

No Warranty

The Software and this license document are provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Choice of Law

This license is governed by the Laws of England.