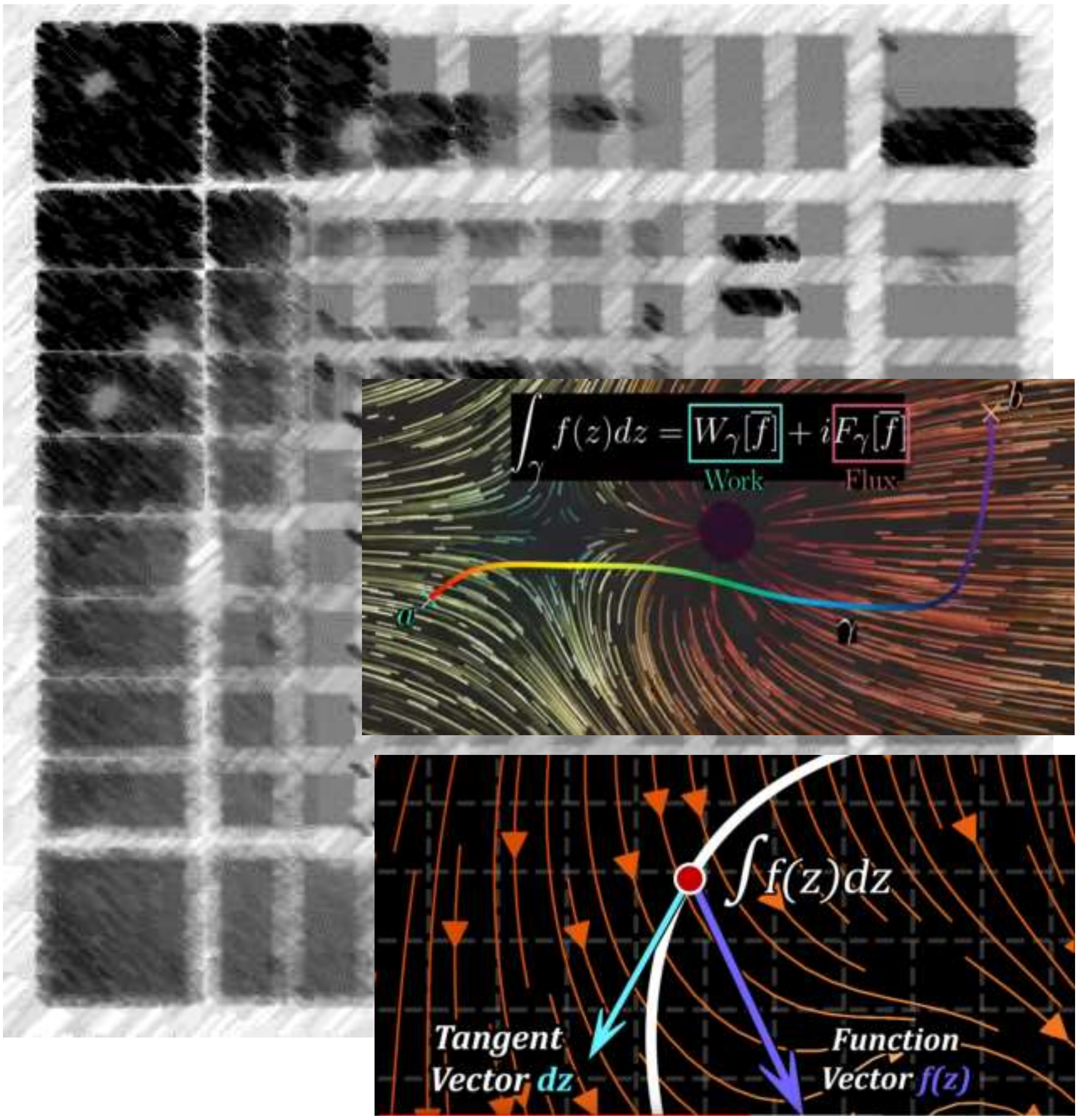
HP-41 Contour ROM

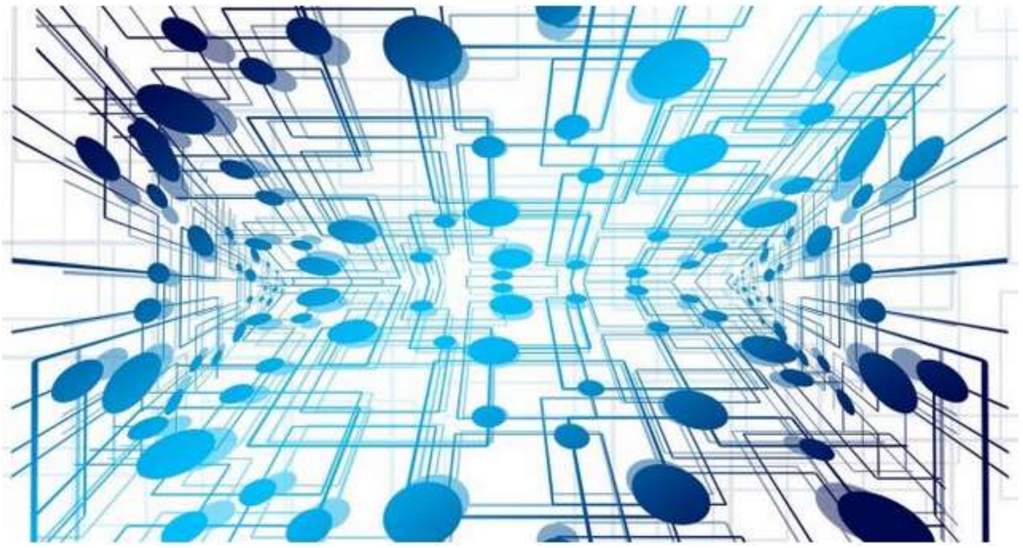
Advanced 41Z / SandMath Apps – Vol. 3


$\int_\gamma f(z)dz = W_\gamma[\bar{f}] + iF_\gamma[\bar{f}]$
Work
Flux
a
b
γ
∫f(z)dz
Tangent Vector dz
Function Vector f(z)

Ángel M. Martin Cañas. *May 2024*

This compilation revision 1.1.1

Copyright © 2024 Ángel Martin



Published under the GNU software license agreement.

Original authors retain all copyrights and should be mentioned in writing by any part utilizing this material. No commercial usage of any kind is allowed.

Front cover image taken from: *https://www.dreamstime.com/royalty-free-stock-photography-mathematics-background-image20849947*

Screen captures taken from V41, Windows-based emulator developed by Warren Furlow.
See www.hp41.org

"CONTOUR" ROM

Table of Contents

Contour_ROM Manual
HP-41 Module

Introduction and Credits.

Welcome to the Contour ROM, gathering a few advanced math applications showcasing the prowess of the SandMath and 41Z modules. You'll find HP-41 versions of classic HP-15 advanced application examples, such as the Contour Integration (which gives this ROM its very name) and the Complex Potential, as well as several other state-of-the-art examples of the usability and effectiveness of the calculator platform that may still surprise you after all these years – such as Valentín Albillo's seminal contribution on the Mandelbrot Set area estimation.

Other programs include additional applications of the SandMath and 41Z in root-finding and differential geometry areas - see the Curve length, Surface of revolution, and areas under generic surfaces.

And make sure you don't miss the Fourier Transform sections, an elusive field for RPN calculators successfully conquered by your trusty HP-41 companion. See the driver program for the 41Z MCODE functions and the seminal version by Narmwon Kim, here enhanced with X-Mem file support.

Overlap with other ROMs

Several applications in this module have been taken from the Advantage Math ROM, some of them for completion sake and other to round the selection in a more logical manner. Consequently, and with some exceptions, they have been removed from the Advantage Math to avoid repetition.

Note that the use of the SandMath for **FINTG** and **FROOT** has been favored over the leaner "Solve & Integrate" ROM. Reasons for that are several, mainly because other SandMath functions (such as DERV) not available in the S&I ROM are also featured in the programs. That's why the section dealing with the Recursive use of FINTEG and FROOT has been included in this module again, no need to plug the S&I ROM for those.

A few other utility functions are sprinkled throughout the module as well, be that on the FFT section or in the others. Finally several number-theory applications are also included for completion sake.

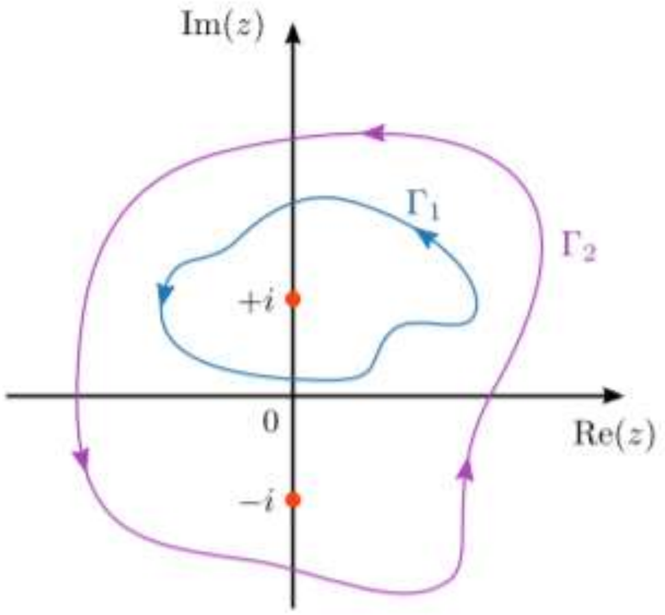
Dependencies.

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. Numerous programs rely on functions from the SandMath and the 41Z modules, thus make sure they're also installed – as well as the Library#4, required by these two.

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

XROM#	Function	Description	Author
16.00	**-Z-CONTOUR**	*Section header*	
16.01	**"ZLITG"**	Complex Line Integral	*HP Co. – Á. Martin*
16.02	**"ITG"**	Integrand function	*HP Co. – Á. Martin*
16.03	**" ZCNTR"**	Contour Integral	*HP Co. – Á. Martin*
16.04	**"ITC"**	Integrand function	*HP Co. – Á. Martin*
16.05	**"ARC"**	R-Circle Contour and Derivative (F0 set)	*Á. Martin*
16.06	**"LIN"**	Line Contour and Derivative (F0 Set)	*Á. Martin*
16.07	**"FZ"**	Complex function - f(z) = exp(iz)/(z+1/z)	*Á. Martin*
16.08	**"ZCD"**	exp(-iz) / (1+z^2) - Cauchy Distribution	*Á. Martin*
16.09	**"ZL2"**	Ln z /(1+z^2)}^2	*Á. Martin*
10.10	**"Z13"**	(z+1)/(z-1).(z-3)	*Á. Martin*
16.11	**"ZFLOW"**	Complex Flow Study	*HP Co. – Á. Martin*
16.12	**"ZFL"**	*f(z) = P(z)-Yo - Function to Solve*	*Á. Martin*
16.13	**"PZ"**	Complex Potential - P(z) = z + 1/z	*Á. Martin*
16.14	**"MBA"**	Mandelbrot Set Area	*Valentín Albillo*
16.15	**"D-Y"**	Delta-Wye Transform	*Á. Martin*
16.16	**"Y-D"**	Wye-Delta Transform	*Á. Martin*
16.17	**"PPL"**	Print Pythagorean Triplets	*Thomas Klemm*
16.18	**-ZFOURIER**	Section header	
16.19	**E3/E+**	Pointer builder	*Á. Martin*
16.20	**EINS _**	Einstein Functions 1-2-3	*Á. Martin*
16.21	**SIGMD**	Sigmoid function	*Á. Martin*
16.22	**"JNYX"**	Bessel J and Y vía Continued Fractions	*Baillard-Martin*
16.23	**"=:"**	Subroutine for JYNX	*Baillard-Martin*
16.24	**"ZDFT+"**	Complex Discrete Fourier Transform	*Á. Martin*
16.25	**"ZIDFT+"**	Complex Inverse Discrete Fourier Transform	*A. Martin*
16.26	**"DFTZ"**	Direct Fourier Transform	*A. Martin*
16.27	**"IFTZ"**	Inverse Fourier Transform	*A. Martin*
16.28	**"FFT"**	Fast Fourier Transform	*Narmwon Kim*
16.29	**"IFF"**	Inverse Fast Fourier Transform	*Narmwon Kim*
16.30	**-2D-ITG/SLV**	Section header	
16.31	**ASWAP**	ALPHA swap around comma	*Á. Martin*
16.32	**CLAC**	Clear ALPHA from Comma	*W&W GmbH*
16.33	**"FITG2"**	Recursive Double Integration	*Á. Martin*
16.34	**"*2D"**	Auxiliary for FITG2	*Á. Martin*
16.35	**"F1XY"**	Examle f1(x,y)	*Á. Martin*
16.36	**"F2XY"**	Example f2(x,y)	*Á. Martin*
16.37	**"FRT2"**	Recursive Root Finder f(x.y)	*Á. Martin*
16.38	**"*FG"**	Auiliary for FRT2	*Á. Martin*
16.39	**"FG1"**	Example f1 and g1	*Á. Martin*
16.40	**"FG2"**	Example f2 abd g2	*Á. Martin*
16.41	**FNRM**	Finite Nested Radicals	*Martin-Baillard*
16.42	**INRM**	Infinite Nested Radicals	*Martin-Baillard*
16.43	**-SANDMATH+**	Section header	
16.44	**"CLEN"**	Curve Length	*Á. Martin*
16.45	***CL"**	Auxiliary for CLEN	*Á. Martin*
16.46	**"LNG"**	Arc Length of a Curve	*JM Baillard*
16.47	**"SRV"**	Area of Surface of Revolution	*JM Baillard*
16.48	**"SKS"**	Area of Surface	*JM Baillard*

16.49	**"*RM"**	Romberg Routine	*JM Baillard*
16.50	**"XHALL"**	Halley's Method for real roots	*Á. Martin*
16.51	**"XNWT"**	Newton Method for real roots	*Á. Martin*
16.52	**"ZNWT"**	Complex-Step-Differentiation for real roots	*Á. Martin*
16.53	**"ZROOT"**	Complex Root finder	*Albillo-Martin*
16.54	**BELL**	Bell Numbers	*Á. Martin*
16.55	**BN2**	Bernouilly Numbers	*Á. Martin*
16.56	**BINETN**	Binet formula – Integer order	*Á. Martin*
16.57	**BINETX**	Binet formula – Real order	*Á. Martin*
16.58	**FIB**	Fibonacci Numbers	*Á. Martin*
16.59	**IFIB**	Inverse Fibonacci numbers	*Á. Martin*
16.60	**MLN**	Mutinomial Coefficients	*Martin-Baillard*
16.61	**ULAM**	Ulam's Conjecture	*Á. Martin*
16.62	**ΣFIB**	Sum of Fibonacci numbers	*Á. Martin*
16.63	**ΣIFIB**	Sum of Inverse Fibonacci numbers	*Á. Martin*



$$\oint_{\Gamma_1} dz\, f(z) = 2\pi i\, \mathrm{Res}\left[\, f(z)\,\right]_{z=i}$$

$$\oint_{\Gamma_2} dz\, f(z) = 2\pi i \cdot \left[\frac{1}{2i} - \frac{1}{2i}\right] = 0.$$

Contour Integration on the HP-41.

What follows is just a quick adaptation of the parameterized complex integral examples from the HP-15C Advanced Functions manual, see pages 73 and following.

Perhaps a little brute-force-ish, nevertheless a good example of a combined application of the 41Z functions and the SandMatrix for the numerical integration task. Surely it is restricted to easy contours like the *straight line segments* used in the example below, so the general-purpose case (Residues theorem, analytical functions, etc.) remains a challenge to be cracked.

You can use FINTG to evaluate the contour integral $\int_C f(z)dz$ here C is a curve in the complex plane. First parameterize the curve C by:

$$z(t) = x(t) + i\, y(t)\ ;\ \text{for } t1 \le t \le t2.$$

Let $G(t)=f(z(t)).z'(t)$. Then $\int_C f(z)dz = \int_{t_1}^{t_2} G(t)dt$

$$= \int_{t_1}^{t_2} \mathrm{Re}(G(t))dt + i\int_{t_1}^{t_2} \mathrm{Im}(G(t))dt$$

These integrals are precisely the type that **FINTG** evaluates. Since G(t) is a complex function of a real variable t, **FINTG** will sample G(t) on the interval t1 ≤ t ≤ t2 and integrate Re(G(t))—the value that your function returns to the real X-register. For the imaginary part, integrate a function that evaluates *G*(*t*) and uses RE<>IM to place Im(*G*(*t*)) into the real X-register.

Program #1. Integral along line segment [a, b]

$$I = \int_a^b f(z)dz$$

The generalized program listed below evaluates the complex integral **along the straight line from *a* to *b*,** where *a* and *b* are complex numbers such that Im(a)#Im(b). Hence, the parameterized values z(t) use z = a + t.(b-a), with t1=0, t2=1.The program assumes that your complex function subroutine has a global label and evaluates the complex function f(z), and that the limits ***a*** and ***b*** are in the complex W- and Z-registers, respectively. The complex components of the integral $\cdot I$ and the uncertainty ΔI are returned in the X- and Y-registers respectively.

The parameterization is for this case quite simple:

z(t) = a + t.(b-a), with t1=0, t2=1
z'(t) = (b-a)

This has the additional benefit that there's no need to write a global label subroutines for either the contour or the derivative curves.

Note that since the derivative of the contour is not dependent on t it could therefore be taken out of the integral - however the requirement of using the imaginary part of the integrand advises to leave the derivative inside.

```
LBL "ZLITG+"     Data entry .
"F(Z)? "         line segment
PMTA
ASTO 00          FName in R00
"Z1=?"
PROMPT           saves a in ZR01
ZSTO 01          Re - R02, Im - R01
"Z2=?"
PROMPT           b
ZRC- 01          (b-a)
GTO 00
LBL "ZLITG"      Cpx. Line Intg
ASTO 00          FName in R00
Z<>W             a
ZSTO 01          saves a in ZR01
Z-               (b-a)
LBL 00
ZSTO 02          saves (b-a) in ZR02
"ITG"            integrands
0                t1 limit
ENTER^
1                t2 limit
SF 01            Imaginary parts
FINTG            calculates Im (I, Δ)
STO 07           saves Im(I) in R07
RDN
STO 09           saves Im(ΔI) in R03
RDN              same limits
CF 01            flag real parts
FINTG            calculates Re(I, Δ)
STO 06           saves Re(I) in R06
RDN
STO 08           saves Re(ΔI) in R08
RCL 09           presents ΔI in W
X<>Y
ZENTER^          saves ΔI in W
RCL 07           presents I in Z
RCL 06
ZAVIEW           shows result
TONE 2
RTN              done.
LBL "ITG"        Integrals
0                no Imaginary
X<>Y             current t
ZRC* 02          (b-a).t
ZRC+ 01          a +(b-a).t
XEQ IND 00       f(a + (b-a).t)
ZRC* 02          f(z).z'(t)
FS? 01           Imaginary?
X<>Y             yes, use it
END              done.
```

To use **ZLITG** you need to write a subroutine to calculate the complex function f(z), place its global label in ALPHA and the two complex integration limits that define the ends of the straight line that your function will be integrated along in the complex stack levels W and Z.

The driver program **ZLITG**+ offers prompts to input the data sequentially, so it's more convenient for the casual user. Note that f(z) still must be written prior to executing the program.

Note that in this case $z(t) = a + t.(b-a)$,

hence $z'(t) = (b-a)$,

and thus, not depending on the real variable t, it can be taken out of the integral instead of being part of the subroutine programming f(z). This facilitates the calculations and speeds up the execution.

Example 1. Approximate the integrals: $I_1 = \int_1^{\infty} \frac{\cos x}{x+1/x}dx$ and $I_2 = \int_1^{\infty} \frac{\sin x}{x+1/x}dx$.

These integrals decay very slowly as x approaches infinity and therefore require a long interval of integration and a long execution time. You can expedite this calculation by deforming the path of integration from the real axis into the complex plane. According to complex variable theory, these integrals can be combined as

$$I_1 + iI_2 = \int_1^{1+i\infty} \frac{e^{iz}}{z+1/z}dz$$ with: $$f(z) = \frac{e^{iz}}{z+1/z}.$$

This complex integral, evaluated along the line x=1 and $y \geq 0$, decays rapidly as y increases — like exp($-y$). To use the previous program to calculate both integrals at the same time, we write a subroutine to evaluate f(z). This result I is calculated much more quickly than if $I1$ and $I2$ were calculated directly along the real axis. .

```
01    LBL "FZ"
02    ZENTER^
03    ZINV        1/z
04    LASTZ       z
05    Z+          z+1/z
06    ZINV        1/(z+1/z)
07    Z<>W        z
08    Z*I         can be replaced with {X<>Y, CHS}
09    ZEXP        exp(iz)
10    Z*          f(z)
11    END
```

Approximate the complex integral by integrating the function from a = 1 + 0i to b = 1 + 6i using a FIX 3 display format to obtain three significant digits. (The integral beyond 1 + 6i doesn't affect the first three digits.)

0, ENTER^, 1, ZENTER^, 6, ENTER, 1, puts the lower limit in W and the upper one in Z

```
ALPHA, "FZ", ALPHA, XEQ "ZLITG"    =>  -0.324+J0.382
Z<>W                               =>   0.001(1+J)
```

This result **I** is calculated much more quickly than if I1 and I2 were calculated directly along the real axis.

Using FIX 6 instead returns after a substantially longer time: -0.324350+J0.382053

And here the upper limit does have an impact, for instance moving it up to b=1+7i:

I= -0.324267+J0.382280

Program #2.- Extension to a more general contour.

The next step is an extension of this method to more general contours, beyond the straight-line (vertical or not) segment used before. For this we'll need to program the different contour sections as parameterized formulas of the real variable **t**, i.e. z(t) in the contour, with t going from an initial (lower) value of the parameter t1, to a final (upper) value t2.

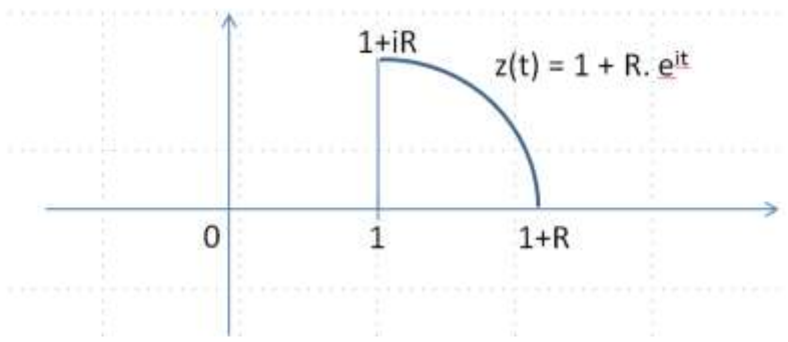
Besides that parameterized curve we'll also need its derivative as another component of the complex integral equivalent once the change of variable is applied: G(t) = f(z(t)).z'(t).

Therefore we see that in principle *three global labels* are going to be required – although the parameterized equations are likely to be rather simple ones given the nature of the usual contours used for these integrals – typically line segments and circle arcs.

Let's see as example the integral of the previous function f(z) but this time using as contour the arc of circumference with radius R and centered at z=1, taken in a direct (counter-clockwise) direction from z1 = 1+ R to z2= 1+ Ri

z(t) = 1 + R.exp(it) with t in the interval [0, π/2]

z'(t) = i.R exp(it)



$$CI_R = \oint_{\gamma_R^2} \frac{e^{iz}}{z+1/z}\,dz,$$
$$\gamma_R^2(t) = Re^{it} + 1, \quad 0 \le t \le \frac{\pi}{2},$$

The program #2 in next page is a straightforward extension of the previous one, with the obvious difference this time that within the integrand routine we call the parameterized z'(t) and multiply its value by the value of the function f(z(t)) as required by the definition G(t) formula.

The program has a data input section where the names of the three global labels are saved in data registers R00, R01, and R02 using the OS/X utility function **PMTA**. Also the value of the radius R and the parameterized integration limits are required at this stage.

Then the runtime main body starts at LBL C – which assumes all input values have been already entered. The arrangement will be convenient to do repeated calculations with different values of the radious R, as the point we're really after is ***checking whether the integral values decrease with R***, hinting at a final zero result when R goes to an infinite limit.

The function f(z) was already taken into account by the "FZ" routine in the first example, so it won't be repeated in the listings below – refer to the previous example if needed.

```
1   LBL "ZCNTR"     main driver program        29  X<>Y
2   "FZ? "          global LBL name            30  STO 07       saves □(Im(I))
3    PMTA           for f(z) routine           31  CF 01        flags Real parts
4   ASTO 02         saved in R02               32  RCL 04       lower linit t1
5   "Z(T)? “        global label name          33  RCL 05       upper limit t2
6   PMTA            for z(t) routine           34  FINTG        does the integration
7   ASTO 00         saved in R00               35  STO 08       saves Re(I)
8   "Z'(T)?  “      global LBL name            36  X<>Y
9   PMTA            for z'(t) routine          37  STO 09       saves □(Re(I))
10  ASTO 01         saved in R01               38  RCL 07       □(Im(I))
11  “Z0=?”          anchor point               39  X<>Y         □(Re(I))
12  PROMPT                                     40  ZENTER^      pushes □ in level W
13  ZSTO 05         saved in ZR05              41  RCL 06       Im(I)
14  "R=?"           value of radius            42  RCL 08       Re(I)
15  PROMPT          ignore if not needed       43  ZAVIEW       shows result
16  STO 03          saved in R03               44  TONE 2
17  "T1^T2=?"       integration limits         45  RTN          done.
18  PROMPT          for parameter t            46  LBL "ITC”    Integrand routine
19  STO 05          t2 saved in R05            47  STO 08       saves t in R08
20  X<>Y                                       48  XEQ IND 00   calculates z(t)
21  STO 04          t1 saved in R04            49  XEQ IND 02   f(z(t))
22  LBL C           for repeat use             50  ZENTER^      saves f(z) in W
23  RCL 04          lower limit t1             51  RCL 08       recalls t
24  RCL 05          upper limit t2             52  XEQ IND 01   calculates z'(t)
25  "ITC"           integrand routine          53  Z*           z’(t).f(z(t))
26  SF 01           flags Imaginary parts      54  FS? 01       Imaginary?
27  FINTG           does integration           55  X<>Y         yes, take Im part
28  STO 06          Im(I) in R06               56  END          done.
```

And finally the parameterized curves are programmed as follows:

```
01  LBL "ZP"      derivative z’(t)           08  RTN
02   0            pure imaginary (0+it)      09  LBL "ZT"      parameterized  z(t)
03  ZEXP          exp(it)                    10  XEQ “ZP”      opportunistic
04  RCL 03        R                          11  Z/I           undoes Z*I
05  ST* Z                                    12  ZRCL 05       adds anchor
06  *             R.exp(it)                  13  Z+            a + R.exp(i.t)
07  Z*I           i.R.exp(i.t)               14  END           done.
```

Example 2.- Obtain the integral results for the different values of R=1, R=10, R=100, and R=1000 and see if they show a decreasing trend as R increases.

Radius	Re(Intg)	Im(Intg)	Magnitude
1	-3.1417 E-01	-2.3007 E-01	|Z|=0.3902
10	8.887 E-02	-7.403 E-03	|Z|=0.0892
100	-4.387 E-03	8.873 E-03	|Z|=0.0099
1000	-9.191 E-04	-3.906 E-04	|Z|=0.0010

Finally let's close the circle (pun intended) using the general-purpose program #2 to re-calculate the first example, where the contour in this case is the straight segment: z(t) = a + (b-a).t with 0 ≤ t ≤ 1

```
01   LBL "LP" derivative
02   6              (b-a)
03   ENTER^
04   0
05   RTN            0 + 6i
06   LBL "LT" contour
07   XEQ "LP"       0 + 6.i
08   RCL 08         t
09   ST* Z          t.(b-a)
10   *              0 + t.(b-a)
11   1              a
12   +
13   RTN            a + t.(b-a)
```

XEQ "ZCNTR" with FZ, LT, and LP as global labels, plus t1=0 and t2=1 (R can be ignored)

Using FIX 4 it gives the same result as before,

Result: ZIT = -0.324+J0.382

Combining Curve and Derivative

In order to reduce the number of global labels in the ROM (where FAT space is always at a premium), the programs in the module has been modified to use flag F00 to determine whether to calculate the contour (F0 clear) or its derivative (F0 set). The main program will manage the status of F00 appropriately, setting and clearing F00 appropriately before calling the (now combined) parameter curve routines. Besides that, the prompt for the derivative subroutine "Z'(T)" has been eliminated – freeing register R01 for other purposes.

This changes the previous routines listing into the following version:

```
01   LBL "LT/LP"    single entry
02   FC? 00
03   GTO 00
04   LBL 01         derivative
05   6
06   ENTER^
07   0              (b-a)
08   RTN
09   LBL 00         contour
10   XEQ 01         (b-a)
11   RCL 08         t
12   ST* Z
13   *              t.(b-a)
14   ZRCL 05 a
15   Z+             a+t.(b-a)
16   END.
```

And likewise for the arc of circumference contour:

```
01  LBL "ZT/ZP"      single FAT entry)
02  FC? 00           derivative?
03  GTO 00           no, branch off
04  LBL 01           derivative
05  0                pure imaginary (0+it)
06  ZEXP             exp(it)
07  RCL 03           R
08  ST* Z
09  *           R.exp(it)
10  Z*I              i.R.exp(i.t)
11  RTN
12  LBL 00           parameterized  z(t)
13  XEQ 01           opportunistic
14  Z/I              undoes Z*I
15  1           adds anchor
16  +           1+R.exp(i.t)
17  END              done.
```

Program #3.- Final consolidated version

Rewriting the data entry section and using **PMTK** in the OS/X Module we can combine both cases in a single program, as listed below. This has the advantage of using the same integrand routine for both cases (ITG and ITC), and thus saves one more FAT entry in the module. Note that iy uses PMTK in the OS/X module to select the case, either ARC ("Ä") or LINE ("L") – and even a custom contour denoted by "X", which would need its custom routine to compute the curve and its derivative.

```
01    LBL "ZCNTR+"
02    "FZ? "
03    PMTA           f(z) Name
04    ASTO 00
05    "TYPE ALX"
06    PMTK           either 1 0r 2
07    GTO IND X      dispatch choice
08    LBL 03
09    "Z(T)? "
10    PMTA           contour name
11    ASTO 01
12    GTO 00                   merge
13    LBL 02         ine segment
14    "Z1=?"
15    PROMPT
16    ZSTO 01
17    "Z2=?"
18    PROMPT
19    ZRC-  01
20    ZSTO 02
21    0              initial param
22    ENTER^
23    1              final param
24    "LIN"          contour name
25    GTO 00         merge
26    LBL 01         Circular ARC
27    "A=?"
28    PROMPT
29    STO 02         Aux. Param.
30    "R=?"
31    PROMPT
32    STO 03         Circle radius
33    "T1^T2=?"
34    PROMPT
35    STO 05         final angle
36    X<>Y
37    STO 04         initial angle
38    X<>Y           final angle
39    "ARC"          contour name
40    LBL 00         merged code
41    ASTO 01
42     "ITC"         integrand
43    LBL C
44    SF 01          imaginary
45    FINTG
46    STO 07         Im(ITG)
47    RDN
48    STO 09         ∆(Im)
49    RDN            same limits!
50    CF 01          real part
51    FINTG
52    STO 06         Re(ITG)
53    X<>Y
54    STO 08         ∆(Re)
55    RCL 09         ∆(Im)
56    X<>Y
57    ZENTER^
58    RCL 07         Im(ITG)
59    RCL 06         Re(ITG(
60    ZAVIEW         show result
```

```
61    TONE 0
62    RTN
63    LBL "ITC"     integrand
64    STO 08        current t
65    CF 00         contour
66    XEQ IND 01    contour z(t)
67    XEQ IND 00    complex f(z)
68    ZSTO 05       save for later
69    RCL 08        current t
70    SF 00         derivative
71    XEQ IND 01    z'(t)
72    ZRC* 05       f(z).z'(t)
73    FS? 01        imaginary?
74    X<>Y          yes, oblige
75    END           done

01    LBL "ARC"     circular
02    0
03    ZEXP          exp(i.t)
04    RCL 03        R
05    ST* Z
06    *             R.exp(i.t)
07    X<>Y          do Z*I
08    CHS           i.R.exp(i.t)
09    FS?C 00       derivative?
10    RTN           yes, return
11    CHS           no, undo Z*I
12    X<>Y          R.exp(i.t)
13    RCL 02        anchor point
14    +             a+R.exp(i.t)
15    RTN

16    LBL "LIN"     line segment
17    ZRCL 02       (b-a)
18    FS?C 00`      derivative?
19    RTN           yes, return
20    RCL 08        t
21    ST* Z
22    *             t.(b-a)
23    ZRC+ 01       a+t.(b-a)
24    END
```

Where the last two routines are the combined contour & derivative calculation for the cases of a circular arc and a straight line segment,

See the registers used in the table below:

Register #	ZLITG	ZCNTR - LINE	ZCNTR - ARC
R00	f(z) - function Name		
R01	*unused*	Z(t) - Contour Name	
R02	Re(z1)		Anchor point A
R03	Im(z1)		Radius R
R04	Re(z2-z1)		t1
R05	Im(z2-z1)		t2
R06	Re(Intg)		
R07	Im(Intg)		
R08	Re(Delta)		
R09	Im(Delta)		
R10	*unused*	Re(z'(t))	
R11	*unused*	Im(z'(t))	

Thanks to this common register mapping across the three programs we'll be able to use subroutines valid for all applicable cases, therefore saving further space in the ROM.

$= \pi.$

Examples from Wikipedia: https://en.wikipedia.org/wiki/Contour_integration

Example 1 – Unit circle

A fundamental result in complex analysis is that the contour integral of f(z)=1/z is 2πi, where the path of the contour is taken to be the unit circle traversed counterclockwise (or any positively oriented Jordan curve about 0). In the case of the unit circle |z|=1 there is a direct method to evaluate the integral

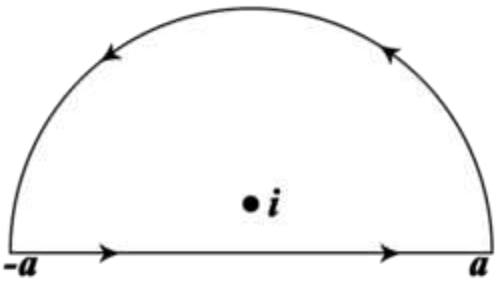
$$\oint_C \frac{1}{z}\,dz = \int_0^{2\pi} \frac{1}{e^{it}} ie^{it}\,dt = i\int_0^{2\pi} 1\,dt = i\,t\Big|_0^{2\pi} = (2\pi - 0)\,i = 2\pi i$$

Example 2 – Cauchy distribution.

The integral $\displaystyle\int_{-\infty}^{\infty} \frac{e^{itx}}{x^2+1}\,dx$

which arises
 in probability theory as a scalar multiple of the characteristic function of the Cauchy distribution) resists the techniques of elementary calculus. We will evaluate it by expressing it as a limit of contour integrals along the contour *C* that goes along the real line from −*a* to *a* and then counterclockwise along a semicircle centered at 0 from *a* to −*a*. Take *a* to be greater than 1, so that the imaginary unit *i* is enclosed within the curve. The contour integral is

$$\int_C \frac{e^{itz}}{z^2+1}\,dz.$$



Since e^{itz} is an entire function (having no singularities at any point in the complex plane), this function has singularities only where the denominator z2 + 1 is zero. Since z2 + 1 = (z + i)(z − i), that happens only where z = i or z = −i. Only one of those points is in the region bounded by this contour. The residue of f(z) at z = i is:

$$\lim_{z\to i}(z-i)f(z) = \lim_{z\to i}(z-i)\frac{e^{itz}}{z^2+1} = \lim_{z\to i}(z-i)\frac{e^{itz}}{(z-i)(z+i)} = \lim_{z\to i}\frac{e^{itz}}{z+i} = \frac{e^{-t}}{2i}.$$

According to the residue theorem, then, we have

$$\int_C f(z)\,dz = 2\pi i \operatorname{Res}_{z=i} f(z) = 2\pi i \frac{e^{-t}}{2i} = \pi e^{-t}.$$

According to Jordan's lemma, if t > 0 then the integral along the arc of circumference tends to zero as R tends to infinite. Therefore, if t > 0 then

$$\int_{-\infty}^{\infty} \frac{e^{itx}}{x^2+1}\,dx = \pi e^{-t}.$$

A similar analysis can be made for values t<0, leading to the final consolidated result shown below:

$$\int_{-\infty}^{\infty} \frac{e^{itx}}{x^2+1}\, dx = \pi e^{-|t|}.$$

If $t = 0$ then the integral yields immediately to real-valued calculus methods and its value is π

Example 3 – Squared Logarithm

This example treats a type of integral of which

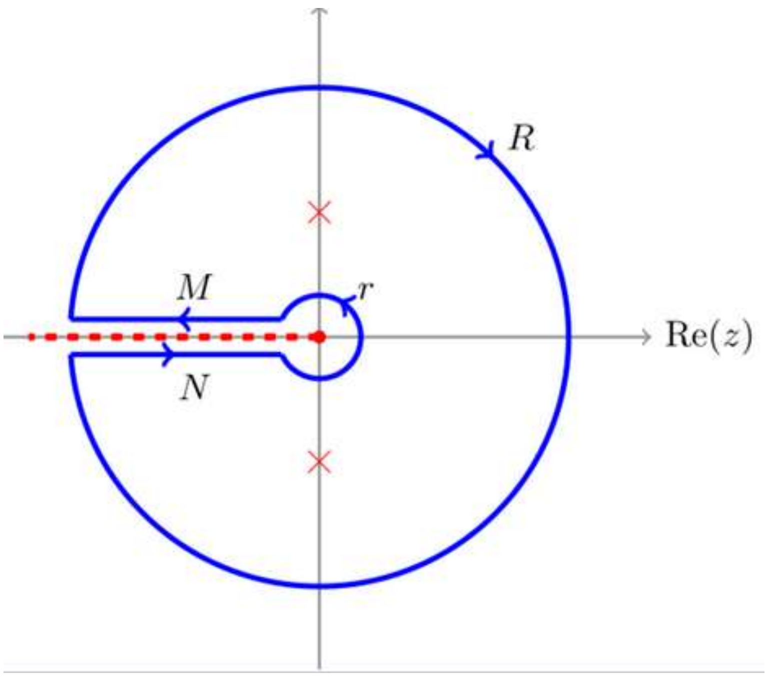
$$\int_0^{\infty} \frac{\log x}{(1+x^2)^2}\, dx$$

To calculate this integral, one uses the function

$$f(z) = \left(\frac{\log z}{1+z^2}\right)^2$$

And to avoid singularities in the integration path we use the branch of the logarithm corresponding to $-\pi < \arg z \le \pi$.

We will calculate the integral of f(z) along the **keyhole contour** shown at right. As it turns out this integral is a multiple of the initial integral that we wish to calculate and by the Cauchy residue theorem (there are two poles at z=i and z=-i) we have



2π.i.(ΣRes{I, -i}) = -π^2

Let R be the radius of the large circle, and r the radius of the small one. We will denote the upper line by M, and the lower line by N. As before we take the limit when R → ∞ and r → 0. *The contributions from the two circles vanish.*

In order to compute the contributions of M and N we set z = −x + iε on M and z = −x − iε on N, with 0 < x < ∞: Replacing z by hose values and performing some simplification we obtain:

$$\left(\int_M + \int_N\right) f(z)\, dz \quad = 4\pi i \int_0^{\infty} \frac{\log x}{(1+x^2)^2}\, dx$$

and after isolating the sought for integral it gives

$$\int_0^\infty \frac{\log x}{(1+x^2)^2}\,dx = -\frac{\pi}{4}.$$

we'll see that the chosen branch of the logarithm is a rather relevant point for a proper usage of the ZLN function. That, together with the winding direction being clock-wise, must be taken into account to obtain he correct results!

Application with the module functions.

The module includes global labels for straight segments - defined by the two complex points at its end - and circumferences centered in 0 and with a radius R. Choosing initial and final angles is the way to provide a general-purpose arc, such as the one used in the previous example with initial angle zero and final angle $\pi/2$.

The module also includes the complex functions routines for the examples described above:

- LBL "1/Z" for the first example
- LBL "ZCD" for the Cauchy distributions
- LBL "ZLN2" for the keyhole contour example

Combining these resources is therefore a simple task.

Example 1 - Keystrokes,

A straightforward case for which we have both the complex function and the contour:

XEQ "ZCNTR"	FZ?
"1/Z", R/S	Z(T)?
"RC", R/S	A=?
0, R/S	R=?
1, R/S	T1^T2=?
0, PI, 2, *, R/S	-2,435E-11+J6,283
X<>y, FIX 9	6,283185308

Note that the „A" parameter is irrelevant for this example, so zero is as good as any other value

Example 2 - Keystrokes

Here the routines can be used to verify that the contribution of the semi-circumference tends to zero as its radius increases, i.e. similar to the analysis made at the beginning of this section.

Running if for the cases R=10, R-100, R-1,000 and R=10,0000 we compile the following table:

Radius	Result	Magnitude
10	0,100+J0,100	|Z|=0,141
100	0,010+J0,010	|Z|=0,014
1.000	0,001+J0,001	|Z|=0,001
10.000	E-4+J 1,000E-4	|Z|= 1,415E-4

XEQ "ZCNTR"	FZ?	
"ZCD", R/S	Z(T)?	
"RC", R/S	A=?	
0, R/S	R=?	will repeat for different radius
1, R/S	T1/T2=?	
0, PI, 2, /, R/S	0,100+J0,100	

Repeating for R=100, 1,000 and 10,000 is as simple as modifying the radius value in data register R03 and executing the program from the local label "C", as shown below:

100, STO 03, XEQ C
1000, STO 03, XEQ C
10000, STO 03, XEQ C
etc…

Example 3 - keystrokes

Here too the contribution of the module is limited to the verification of the diminishing results in the outer and inner circles of the keyhole contour. It is left to the reader for practice…

This contour and the choice of the branch of the logarithm introduce two points for consideration.

- The first one is that the contour winds in clockwise direction, therefore the sign changes when compared to the "natural" convention.
- The second is the function **ZLN** in the 41Z module uses the principal branch of the logarithm, which removes the negative semi-axis and therefore it is the appropriate one for this example. This is mentioned just to alert you that it may not always be the case, depending on the case.

Program Listings

01	**LBL "1/Z"**	
02	**ZINV**	
03	RTN	
04	**LBL "ZLN2"**	{Ln z /(1+z^2)}^2
05	**ZLN**	
06	**LASTZ**	
07	**Z^2**	
08	1	
09	+	
10	**Z/**	
11	**Z^2**	
12	RTN	
13	**LBL "ZCD"**	Cauchy Distr
14	**ZENTER^**	exp(-i.t.z) / (1+z^2)
15	X<>Y	
16	CHS	i.z
17	RCL 02	argument " t"
18	ST* Z	
19	*	i.t.z
20	**ZEXP**	exp(i.t.z)
21	**Z<>W**	
22	**Z^2**	z^ 2
23	1	
24	+	1+Z^ 2
25	**Z/**	exp(-i.t.z) / (1+z^2)
26	RTN	
27	**LBL "RC"**	R-Circle
28	0	0+i.t
29	**ZEXP**	exp(it)
30	RCL 03	get radius
31	ST* Z	
32	*	R.exp(i.t)
33	FC?C 00	derivative?
34	RTN	no, return
35	X<>Y	yes, multiply by i
36	CHS	i.R.exp(i.t)
37	END	done

Complex Potentials.

What follows is just a quick adaptation of the complex potentials examples from the HP-15C Advanced Functions manual, see pages 76 and following.

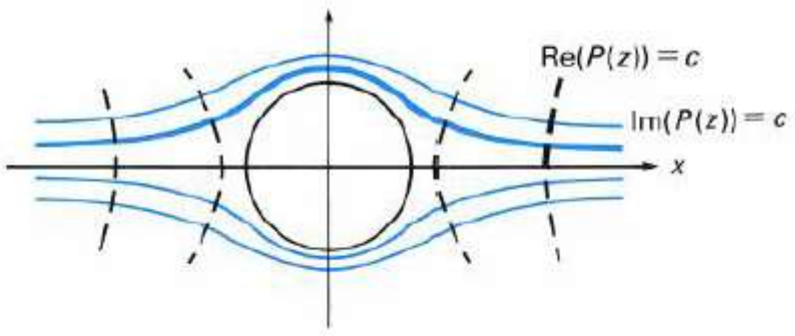
Conformal mapping is useful in applications associated with a complex potential function. The discussion that follows deals with the problem of fluid flow, although problems in electrostatics and heat flow are analogous.

Consider the potential function P(z). The equation Im(P(z)) = c defines a family of curves that are called **streamlines of the flow**. That is, for any value of c, all values of z that satisfy the equation lie on a streamline corresponding to that value of c. To calculate some points zk on the streamline, specify some values for xk and then use FROOT to find the corresponding values of yk using the equation

$$\mathrm{Im}(P(x_k + iy_k)) = c.$$

If the xk values are not too far apart, you can use yk-1 as an initial estimate for yk. In this way, you can work along the streamline and calculate the complex points zk = xk + iyk. Using a similar pocedure, you can define the **equipotential lines**, which are given by

$$\mathrm{Re}(P(z)) = c.$$



The program listed below is set up to compute the values of yk from evenly spaced values of xk. You must provide a subroutine labeled with a global label in memory that places Im(P(z)) in the real X-register. The program uses inputs that specify the step size h, the number of points n along the real axis, and z0 = x0 + iy0, the initial point on the streamline. You must enter n, h, and z0 into the Z-, Y-, and X-registers before running the program.

The program computes the values of zk and stores them in data file in X-Mem in the form ak1 = xk-1 and ak2 = yk-1 for k = 1, 2, ... , n. Data entry includes prompting for the flow conditionas and allows for either streamlines or equipotentials to be computed. In addition to the 41Z module to define the complex flow, and the Solve & Integrate module (for FROOT, hence the ancilliary subroutine "ZFL" for the equation to solve), the AMC_OS/X module is required for PMTA and PMTK. The listing below also includes the potential flow example "PZ" given by: $P(z) = z + 1/z$

One special feature of this program is that if an xk value lies beyond the domain of the streamline (so that there is no root for _ to find), then the step size is decreased so that xk approaches the boundary where the streamline turns back. This feature is useful for determining the nature of the streamline when yk isn't a single-valued function of xk. If h is small enough, the values of zk will lie on one branch of the streamline and approach the boundary. (The second example below illustrates this feature.)

```
01 LBL "ZFLOW"
02 "FNAME?"      function name
03 PMTA
04 ASTO 08
05 "#N=?"        data points
06 PROMPT
07 STO 09
08 "H=?"         step size
09 PROMPT
10 STO 04
11 "Z0=?"        initial point
12 PROMPT        enter IM in Y, Re in X
13 ZSTO 00       saved in {R00, R01}
14 "TYPE? SV"    stream/velocity
15 PMTK
16 CF 00         default is streamlines
17 2
18 X=Y?          chose "V"?
19 SF 00         flags velocity
20 LBL C         main section
21 RCL 09        number of points
22 ST+ X         double size
23 "ZFL"         data file name
24 SF 25
25 PURFL         purge if there
26 CF 25
27 CRFLD         create it (again)
28 CLX
29 SEEKPTA       sets pointer to top
30 RCL 09        # of points
31  E
32 –             n-1
33 E3/E+         1,00(n-1)
34 STO 07 `      save counter
35 ZRCL 00       initial point
36 STO 02        x0 in R02
37 SAVEX         and in data file
38 X<>Y
39 STO 03        y0 in R03
40 SAVEX         and in data file
41 X<>Y          restore order
42 XEQ IND 08    compute function
43 STO 05        save result
```

```
44 LBL 02
45 RCL 04        h
46 RCL 07        counter
47 INT           index k
48 *             k.h
49 RCL 02        xk
50 +             xk+1 = xk+k.h
51 STO 06
52 RCL 03        yk as guess1
53 ENTER^        and as guess2
54 FROOT
55 GTO 04        root found!
56 4             if not, adjust search
57 ST/ 04        new # of points
58 ST* 07        new step size
59 GTO 02        try again
60 LBL 04        root was found
61 RCL 06
62 VIEW X        show Re(xk+1)
63 SAVEX         and save in file
64 RDN
65 STO 03        make yk = yk+1
66 VIEW X        show Im(xk+1)
67 SAVEX         and save in file
68 ISG 07        increase counter
69 GTO 02        do next if not last
70 CLX
71 SEEKPT        set pointer to top
72 RTN           done.
73 LBL "ZFL"     ancillary routine
74 RCL 06        Yk in Y, xk in Xk
75 XEQ IND 08    compute P(z)
76 RCL 05        xk ot yk
77 –             subtract it
78 RTN           done.
79 LBL "PZ"      example potential
80 ZENTER^
81 ZINV
82 Z+
83 FC? 00        streamline?
84 X<>Y          yes, get Im part
85 END
```

Let's see next a couple of examples to check the program

.

<u>Example-1</u>: Calculate the *streamline* of the potential $P(z) = 1/z + z$ passing through $z = -2 + 0.1i$.
Using nine data points and a step size of 0.1 we obtain the results shown below...

<u>Example-2</u>: For the same potential as the previous example, $P(z) = 1/z + z$, compute the *velocity equipotential line* starting at $z = 2 + i$ and proceeding to the left.
We\ll try with n = 6 and h = −0.5. (Notice that h is negative, which specifies that x_k *will be to the left of* x_0*)*

Example-1 Results

xk	yk
−2.0	0.1000
−1.5	0.1343
−1.0	0.4484
−0.5	0.9161
0.0	1.0382
0.5	0.9161
1.0	0.4484
1.5	0.1343
2.0	0.1000

Example-2 Results

xk	yk
2.0000	1.0000
1.8750	0.2363
1.8672	0.1342
1.8452	0.0941
1.8647	0.0844
1.8646	0.0775

The example-2 results show the nature of the top branch of the curve (the heavier dashed line in the graph for the previous example). Note that the step size h is automatically decreased in order to follow the curve-rather than stop with an error-when no y-value is found for x < 1.86.

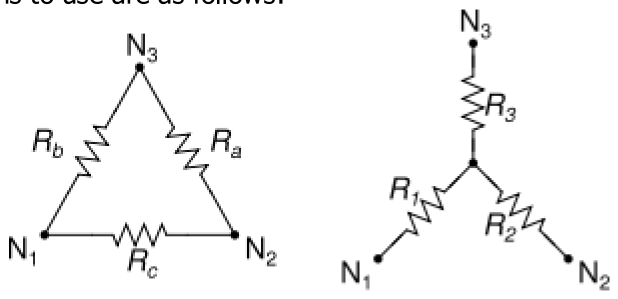
To review the results you could set user flag 21 to halt the displaying while the calculations are being made, or alternatively review the values saved in the data file with the utility listed below:

```
01  LBL "DFED"
02  FLSIZE
03  1
04  –
05  E3
06  /`                 0.00(n-1)
07  LBL 00
08  SEEKPT
09  GETX               get current
10  X<>Y               index to X
11  "D"
12  ARCLI
13  >"="
14  X<>Y               value to X
15  ARCL X
16  CF 22              set data entry flag
17  FC? 08             edit mode?
18  >"?"               yes, add question mark
19  PROMPT
20  FC?C 22            was data entered?
21  GTO 02             no, branch off
22  FS? 08             yes, edit mode?
23  GTO 01             no, skip over
24  X<>Y               yes, prepare stage
25  RDN
26  X<>Y               get pointer
27  SEEKPT             set pointer
28  X<>Y               new value
29  SAVEX              save it in file
30  LBL 02
31  X<>Y
32  LBL 01
33  ISG X              increase counter
34  GTO 00             loop for next
35  "DONE"
36  AVIEW
37  CLA
38  END
```

41Z Application: Delta-Wye Transformation.

Here's a token of appreciation for the EE folks in the audience – using the 41Z to tackle a classic: Delta-Wye impedance transformation for 3-phase systems.

The expressions to use are as follows:



$$R_a = \frac{R_1R_2 + R_2R_3 + R_3R_1}{R_1} \qquad R_1 = \frac{R_bR_c}{R_a + R_b + R_c}$$

$$R_b = \frac{R_1R_2 + R_2R_3 + R_3R_1}{R_2} \qquad R_2 = \frac{R_aR_c}{R_a + R_b + R_c}$$

$$R_c = \frac{R_1R_2 + R_2R_3 + R_3R_1}{R_3} \qquad R_3 = \frac{R_aR_b}{R_a + R_b + R_c}$$

Examples.

Compute the Delta impedances equivalent to the Wye configuration given by:

z1 = 1+2i,   z2=3+4i, and z3=5+6i

We type:

CF 00, GTO "Y-D"

```
2, ENTER^, 1, ZENTER^          1+J2
4, ENTER^, 3, ZENTER^          3+J4
6, ENTER^, 5, XEQ C            7,720+J11,040
ZRDN                           21,400+J21,200
ZRDN                           4,574+J7,311
```

and for the reverse direction we take advantage that the three values are already in the complex stack, thus there's no need to re-enter them.

```
SF 00, XEQ C                   5,000+J6
ZRDN                           3,000+J4
ZRDN                           1+J2
```

The simple program below is all there is to it – behold the power of the 41Z complex stack in action :-)

Delta <-> Wye conversions				
1	LBL "D-Y	01	LBL "DYD"	
2	SF 00	02	ZRCL 00	Za / Zab
3	GTO 00	03	ZRC+ 01	Za+Zb / Zab+Zbc
4	LBL "Y-D	04	FC? 00	
5	CF 00	05	GTO 01	
6	LBL 00	06	ZRC+ 02	Zab+Zbc+Zca
7	"Za"	07	ZINV	1/(Zab+Zbc+Zca)
8	FS? 00	08	ZRPL^	
9	"|-b"	09	ZRCL 00	Zab
10	"|-=?"	10	ZRC* 02	ZabZca
11	PROMPT	11	Z*	Za = ZabZca
12	ZSTO 00	12	Z<>W	1/(Zab+Zbc+Zca)
13	"Zb"	13	ZRCL 01	Zbc
14	FS? 00	14	ZRC* 00	ZabZbc
15	"|-c"	15	Z*	Zb = ZabZbc/(Zab+Zbc+Zca)
16	"|-=?"	16	ZRUP	1/(Zab+Zbc+Zca)
17	PROMPT	17	ZRCL 02	Zca
18	ZSTO 01	18	ZRC* 01	ZbcZca
19	"Zc"	19	Z*	Zc = ZbcZca/(Zab+Zbc+Zca)
20	FS? 00	20	RTN	
21	"|-a"	21	LBL 01	
22	"|-=?"	22	LASTZ	Zb
23	PROMPT	23	ZRC* 00	ZaZb
24	ZSTO 02	24	ZRC/ 02	ZaZb/Zc
25	XEQ "DYD"	25	Z+	Zab = Za+Zb+ZaZb/Zc
26	ZSTO 02	26	ZRCL 01	Zb
27	ZRDN	27	ZRC/ 00	Zb/Za
28	ZSTO 01	28	ZRC* 02	ZbZc/Za
29	ZRDN	29	LASTZ	Zc
30	ZSTO 00	30	Z+	Zc+ZbZc/Za
31	ZRDN	31	ZRC+ 01	Zb+Zc+ZbZc/Za
32	ZRDN	32	ZRCL 00	Za
33	ZVIEW 00	33	ZRC/ 01	Za/Zb
34	ZVIEW 01	34	ZRC* 02	ZaZc/Zb
35	ZVIEW 02	35	LASTZ	Zc
36	RTN	36	Z+	Zc+ZaZc/Zb
		37	ZRC+ 00	Za+Zc+ZaZc/Zb
		38	END	

Note that to reduce the number of FAT entries, the version in this ROM has replaced the global label DYD with the local label C, to be used as a soft key assignment.

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 0.25
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
```

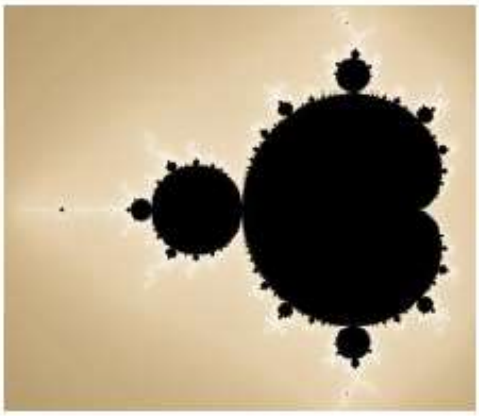
Boldly Going - Mandelbrot Set Area

© 2020 Valentín Albillo

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 ***1.5065918849*** 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:

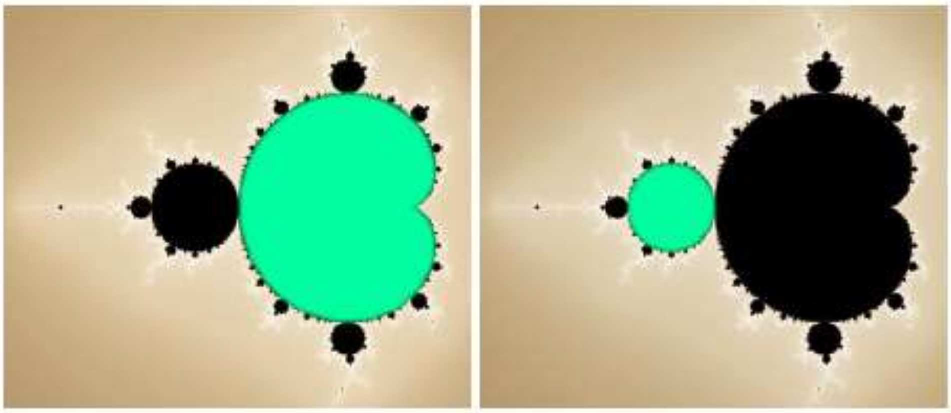
- Each point *(x,y)* will be generated as a random complex number *z* within a rectangular box enclosing **M**. Actually, the leftmost extreme of **M** is at $x = -2$, the righmost extreme is at $x = 0.471185334933396+$, the topmost extreme is at $y = 1.122757063632597+$ and the downmost extreme is at $y = -1.122757063632597+$.

- 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 $\mathbf{M}$ is a very time-consuming iterative process (which will reach the maximum number of iterations if z actually belongs to $\mathbf{M}$) we can try and avoid it altogether for those z which we can easily ascertain in advance as belonging to $\mathbf{M}$ 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 $3\pi/8 = 1.178097+$ (about *78.20%* of the total area), while the main disk has an area of $\pi/16 = 0.196350+$, (another *13.03%)* and their combined total is $7\pi/16 = 1.374447+$, which already accounts for *91.23%* of the total area of $\mathbf{M}$ 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 $\mathbf{M}$ 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 $\mathbf{M}$ 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"**	26	CF 21	51	ABS	76	GTO 00
	2.5		"Working…"		X<Y?		**LBL 03**
	STO 06		AVIEW		GTO 04		RCL 00
	2		CF 00		SIGN		RCL 03
05	STO 07	30	X≠0?	55	RCL+ ST Z	80	MOD
	1.2		SF 00		ABS		X≠0?
	STO 08		**LBL 00**		RCL 09		RTN
	0.25		RCL 05		X>Y?		CLA
	STO 09		STO 01		GTO 04		RCL 04
10	1	35	FS? 00	60	R↑	85	RCL- 00
	SEED		***XEQ 03***		RCL 07		X=0?
	"Points?"		RAN		RCL ST Y		RTN
	PROMPT		RCL× 06		**LBL 01**		AIP
	STO 04		RCL- 07		X↑2		├"→"
15	STO 00	40	RAN	65	RCL+ ST Z	90	RCL 02
	256		RCL× 08		ABS		AIP
	"Iters?"		COMPLEX		X≥Y?		RCL÷ ST Y
	PROMPT		ENTER		GTO 02		6
	STO 05		ENTER		X<> ST L		×
20	CLX	45	SIGN	70	DSE 01	95	├"␊Area="
	STO 02		RCL- 07		GTO 01		ARCL ST X
	"Every?"		RCL× ST L		**LBL 04**		AVIEW
	PROMPT		ABS		ISG 02	98	**END**
	STO 03		RCL× 09		**LBL 02**		
25	RECT	50	X<>Y	75	DSE 00		

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:

$\boxed{\text{XEQ}}$ "AM"	→ *Points?*	*{ asks for the number of points to generate, N }*
N $\boxed{\text{R/S}}$	→ *Iters?*	*{ asks for the max.num. of iterations*[1]*, K. Default=256, just press* $\boxed{\text{R/S}}$ *}*
K $\boxed{\text{R/S}}$	→ *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* $\boxed{\text{R/S}}$ *}*
P $\boxed{\text{R/S}}$	→ *Point* $_P$ → *Count* $_P$	*{ the intermediate tally of points generated and resulting counts }*
	Area ~ Area $_P$	*{ the intermediate estimations of the area }*
	...	
	→ *Point* $_N$ → *Count* $_N$	*{ the final tally of points generated and resulting count }*
	Area ~ Area $_N$	*{ 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 $\boxed{\text{FIX 05}}$ 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.

$\boxed{\text{XEQ}}$	"AM"	→	*Points?*	
10000	$\boxed{\text{R/S}}$	→	*Iters?*	*{ we'll use 256 iters. which is the default so just press* $\boxed{\text{R/S}}$ *}*
	$\boxed{\text{R/S}}$	→	*Every?*	*{ we just want the final result so just press* $\boxed{\text{R/S}}$ *}*
	$\boxed{\text{R/S}}$	→	*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.

$\boxed{\text{XEQ}}$	"AM"	→	*Points?*		
10000	$\boxed{\text{R/S}}$	→	*Iters?*		
512	$\boxed{\text{R/S}}$	→	*Every?*		
2000	$\boxed{\text{R/S}}$	→	*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}:

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

$\boxed{\text{XEQ}}$	"AM"	→	*Points?*		*{ as above, still just 100,000 }*
100000	$\boxed{\text{R/S}}$	→	*Iters?*		*{ now we'll use 256 iterations, so just press* $\boxed{\text{R/S}}$ *}*
	$\boxed{\text{R/S}}$	→	*Every?*		*{ we won't be monitoring progress either }*
	$\boxed{\text{R/S}}$	→	*100000 →25501*	*Area~1.53006*	*{ the value of* $Area_{100000,256}$ *[1'58"] }*
				$\boxed{\text{STO÷ 10}}$	*{* R_{10} *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 ...

$\boxed{\text{XEQ}}$	"AM"	→ *Points?*		*{ we'll use the full 500,000 points }*
500000	$\boxed{\text{R/S}}$	→ *Iters?*		*{ we'll use 256 iterations, so just press* $\boxed{\text{R/S}}$ *}*
	$\boxed{\text{R/S}}$	→ *Every?*		*{ we'll monitor progress every 100,000 points }*
100000	$\boxed{\text{R/S}}$	→*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:

$\boxed{\text{RCLx 10}}$ → <u>*1.50658*</u> *{ more precisely, 1.50658_263 vs. Förstemann's 1.50659_188 }*

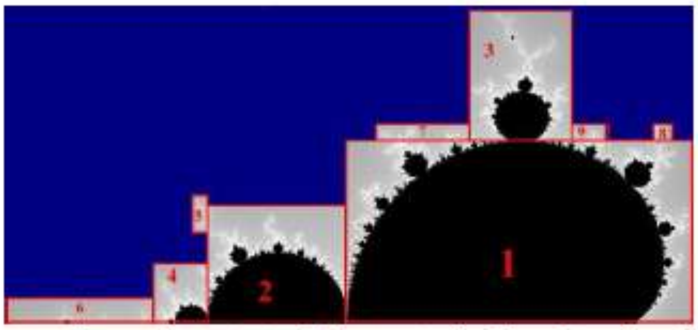
which is my final computed estimation for the area of **M** and it's correct to 6 digits within less than one *ulp* (unit in the last place). It differs from Förstemann's *88-trillion-pixels-calculated-at-8.6-billion-iterations-per-pixel* result by just ~ *0.00000925*, an error of ~ *0.000614%*.

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.

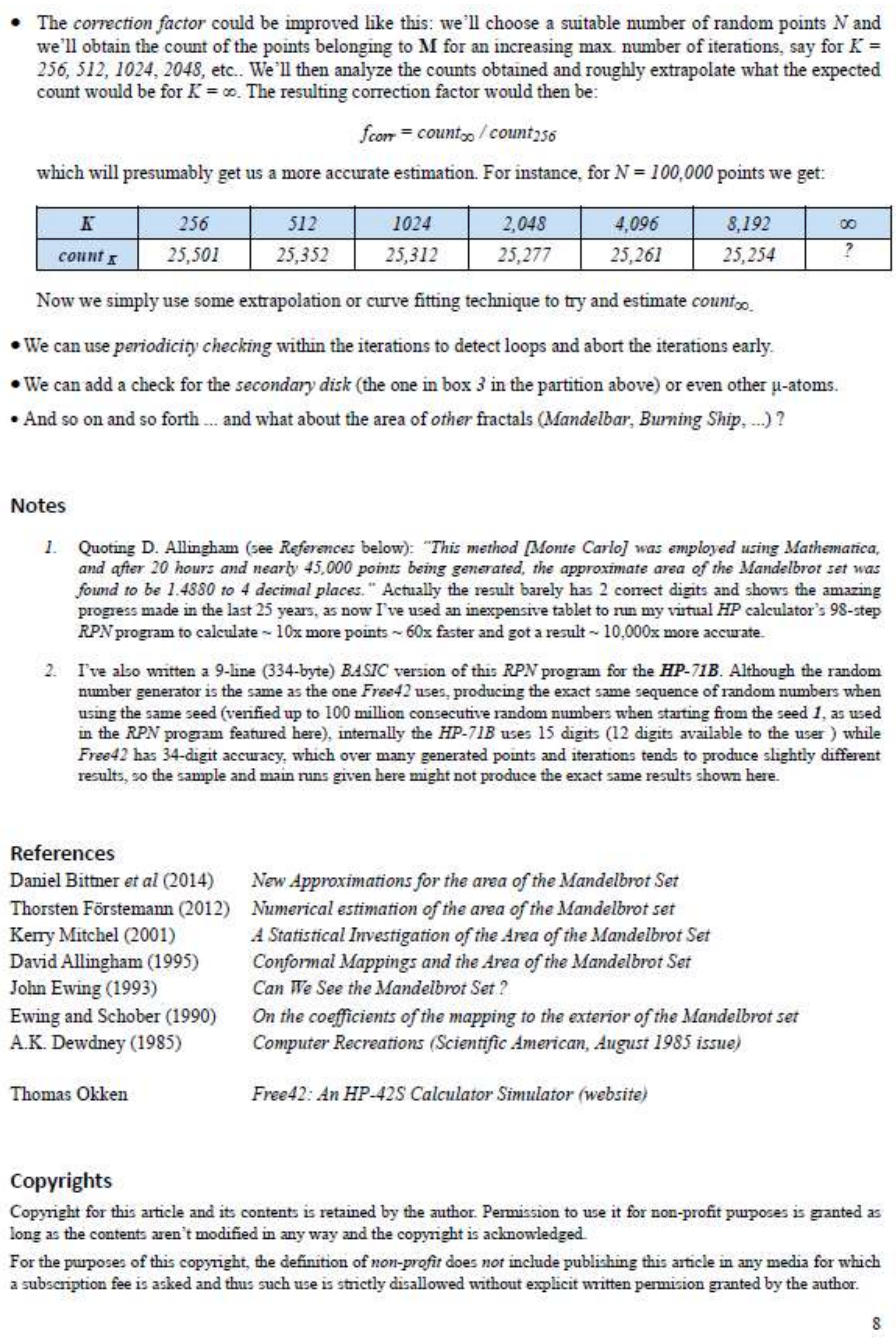
  

  It is important to distribute the total number of points **N** among the boxes proportionally to the area of each box so that the density of points is the same. Otherwise we would be adding areas computed with different precisions and this is wasteful as the resulting sum will be no more accurate than 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

Copyright for this article and its contents is retained by the author. Permission to use it for non-profit purposes is granted as long as the contents aren't modified in any way and the copyright is acknowledged.

For the purposes of this copyright, the definition of *non-profit* does *not* include publishing this article in any media for which a subscription fee is asked and thus such use is strictly disallowed without explicit written permision granted by the author.

Discrete Fourier Transform

This module includes several programs related to the DFT subject.

- The first one is just a driver for the functions **ZDFT** and **ZIDFT** included in the 41Z module, used to input the data points throughout the execution. This driver program was not included in the 41Y due to the lack of enough available space.
- The second one is a FOCAL equivalent to the MCODE implementation in the 41Z , therefore should be equivalent to the first one only considerably slower of course.
- The third one is a Fast Fourier Transform implementation written by Narmwon Kim, and published in the US Users' Library

Program #1.- Driver for **ZDFT** and **ZIDFT** in the 41Z module.

```
01  LBL "ZDFT+"
02  CF 00
03  GTO 00
04  LBL "ZIDFT+"
05  SF 00
06  LBL 00
07  "#PTS=?"
08  PROMPT
09  E3/E+          build pointer
10  STO 00
11  ZINPT          data entry
12  LBL 04
13  RCL 00         control word
14  FC? 00         direct?
15  ZDFT           direct DFT
16  FS? 00         inverse?
17  ZIDFT          inverse DFT
18  RCL M          number of pts.
19  E3/E+          build pointer
20  STO 00         inputl word
21  RDN            results word
22  ZOUTP          data output
23  RTN            done.
24  LBL C          Undo
25  0
26  TF             toggles F0
27  RCL 00         bbb.eee
28  FRC            0.eee
29   E3
30  *              eee
31  ST+ X          2.(eee)
32  ENTER^
33  ENTER^
34   E6
35  /              0.000|2.(eee)
36  +              0.000|2.(eee)
37  2.002
38  +              (e+2),002|(2e)
39  REGMOVE
40  GTO 04
41  END
```

In addition to facilitating the data entry process, this program offers the option to undo the last transformation to verify that the results obtained were correct, by doing the inverse calculation again which should equal the same original data set. If you want to use such option simply press R/S after all points have been outputted, or press XEQ C at any time afterwards. Note that function **TF** in the OS/X module is used here to toggle the status of user flag F00.

Program #2.- A FOCAL counterpart.

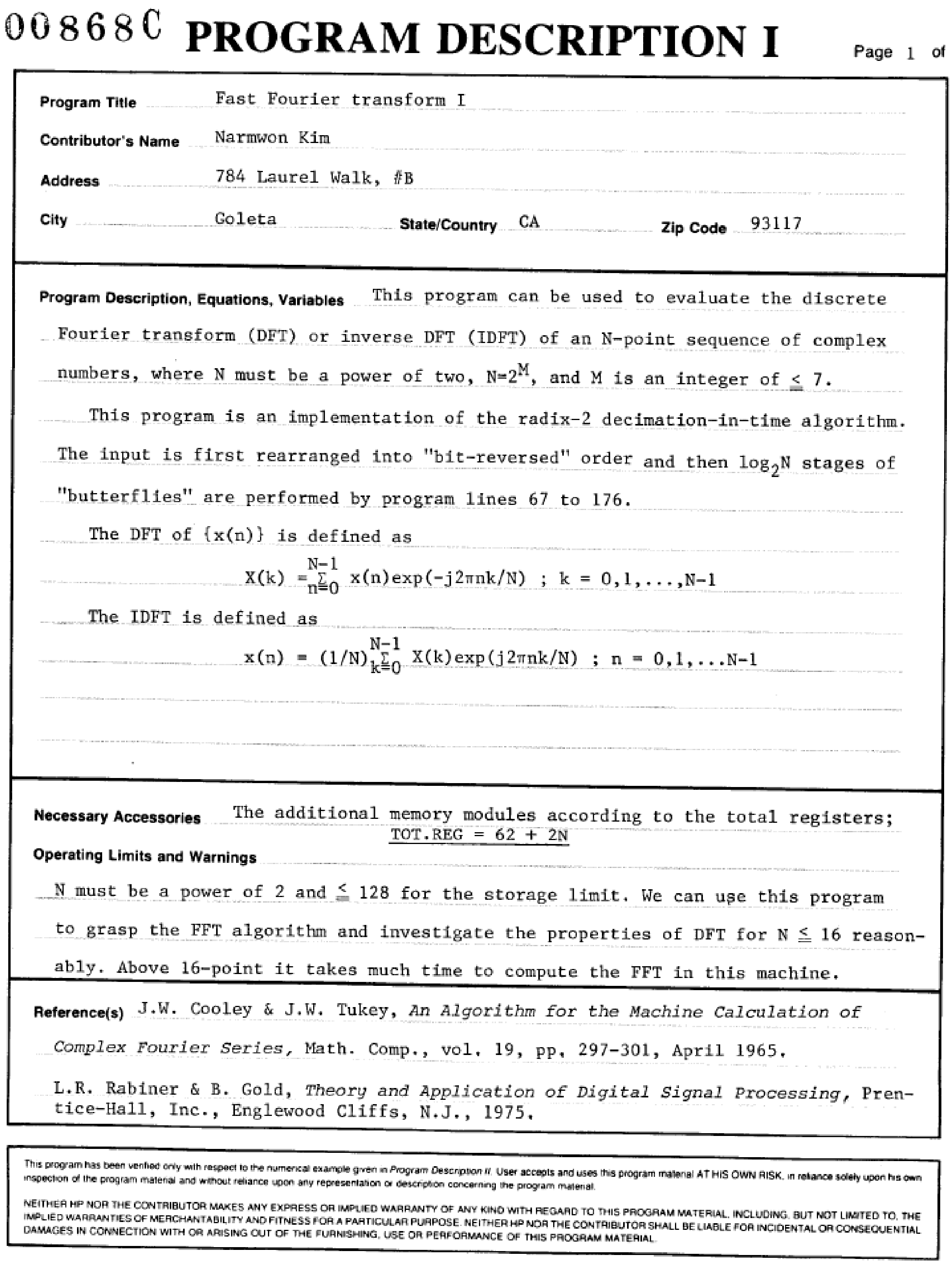
The FOCAL program below is a rough equivalent of the MCODE function. Execution times for this program are about four to five times longer than the MCODE counterpart.

```
01  LBL "ZDFT"
02  CF 01
03  GTO 00
04  LBL "ZIDFT"
05  SF 01
06  *LBL 00
07  STO 00        N
08  E3/E+
09  STO M(5)      j,00N
10  *LBL 01       outer loop
11  VIEW M(5)
12  RCL 00        N
13  STO N(6)
14  E3/3+
15  STO O(7)      k,00N
16  RCL 5(M)      j,00N
17  INT           j
18  ST+  N(6)     dest: ZR(N+j)
19  E
20  -             j-1
21  PI
22  *
23  ST+ X(3)      2p.(j-1)
24  RCL 00        N
25   /            2p.(j-1)/N
26  STO 01
27  CLZ
28  ZSTO IND N(6)     reset destination
29  *LBL 02           inner loop
30  RCL 0(7)          k,00N
31  INT               k
32  E
33  -                 k-1
34  RCL 01            2p.(j-1)/N
35  *                 2p.(j-1)(k-1)/N
36  FC? 01
37  CHS
38  E
39  P-R
40  ZRC* IND O(7)
41  ZST+ IND N(6)
42  ISG O(7)          next k
43  GTO 02            loop back
44  FC? 01
45  GTO 00
46  ZRCL IND 01
47  RCL 00
48  ST/ Z
49   /
50  ZSTO IND 01
51  *LBL 00
52  ISG M(5)          next j
53  GTO 01            loop back
54  END
```

Note that contrary to the functions in the 41Z, this program will not check that enough data registers are available. If not, the "NONEXISTENT" message will be presented; adjust the size and try again. Make sure complex data register ZR00 is not used to store the sample – which must start at ZR01. This is because (real) data registers R00 and R01 are used for scratch calculations by these functions.

Program #3. – Fast and Furious.

Last in this section is an ***enhanced version*** *using extended memory for the data storage* of the contribution to the User's Library by Narmwon Kim (reference 008068C) with a Fast Fourier Transform program, using the well-known Cooley & Tuckey FFT algorithm. Some of the original UPL forms reproduced here, but the program listing is more elaborate for the additional features.



00868C **PROGRAM DESCRIPTION I** Page 1 of

Program Title	Fast Fourier transform I
Contributor's Name	Narmwon Kim
Address	784 Laurel Walk, #B
City	Goleta
State/Country	CA
Zip Code	93117

Program Description, Equations, Variables This program can be used to evaluate the discrete Fourier transform (DFT) or inverse DFT (IDFT) of an N-point sequence of complex numbers, where N must be a power of two, $N=2^M$, and M is an integer of ≤ 7.

This program is an implementation of the radix-2 decimation-in-time algorithm. The input is first rearranged into "bit-reversed" order and then $\log_2 N$ stages of "butterflies" are performed by program lines 67 to 176.

The DFT of {x(n)} is defined as

$$X(k) = \sum_{n=0}^{N-1} x(n)\exp(-j2\pi nk/N) \ ; \ k = 0,1,\ldots,N-1$$

The IDFT is defined as

$$x(n) = (1/N)\sum_{k=0}^{N-1} X(k)\exp(j2\pi nk/N) \ ; \ n = 0,1,\ldots N-1$$

Necessary Accessories The additional memory modules according to the total registers; <u>TOT.REG = 62 + 2N</u>

Operating Limits and Warnings

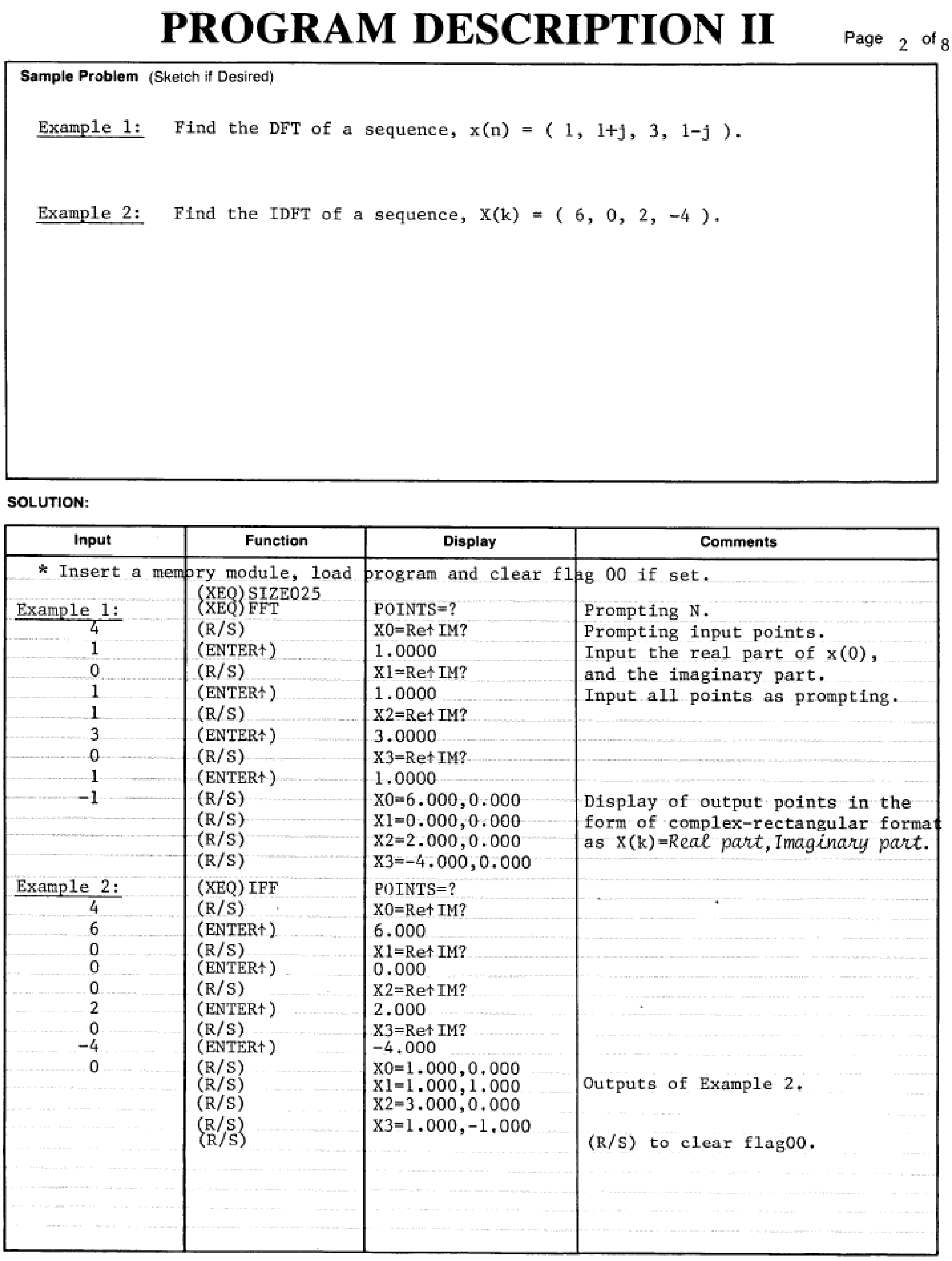
N must be a power of 2 and $\leq$ 128 for the storage limit. We can use this program to grasp the FFT algorithm and investigate the properties of DFT for N $\leq$ 16 reasonably. Above 16-point it takes much time to compute the FFT in this machine.

Reference(s) J.W. Cooley & J.W. Tukey, *An Algorithm for the Machine Calculation of Complex Fourier Series*, Math. Comp., vol. 19, pp. 297-301, April 1965.

L.R. Rabiner & B. Gold, *Theory and Application of Digital Signal Processing*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1975.

This program has been verified only with respect to the numerical example given in *Program Description II*. User accepts and uses this program material AT HIS OWN RISK, in reliance solely upon his own inspection of the program material and without reliance upon any representation or description concerning the program material.

NEITHER HP NOR THE CONTRIBUTOR MAKES ANY EXPRESS OR IMPLIED WARRANTY OF ANY KIND WITH REGARD TO THIS PROGRAM MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. NEITHER HP NOR THE CONTRIBUTOR SHALL BE LIABLE FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH OR ARISING OUT OF THE FURNISHING, USE OR PERFORMANCE OF THIS PROGRAM MATERIAL.



PROGRAM DESCRIPTION II

Page 2 of 8

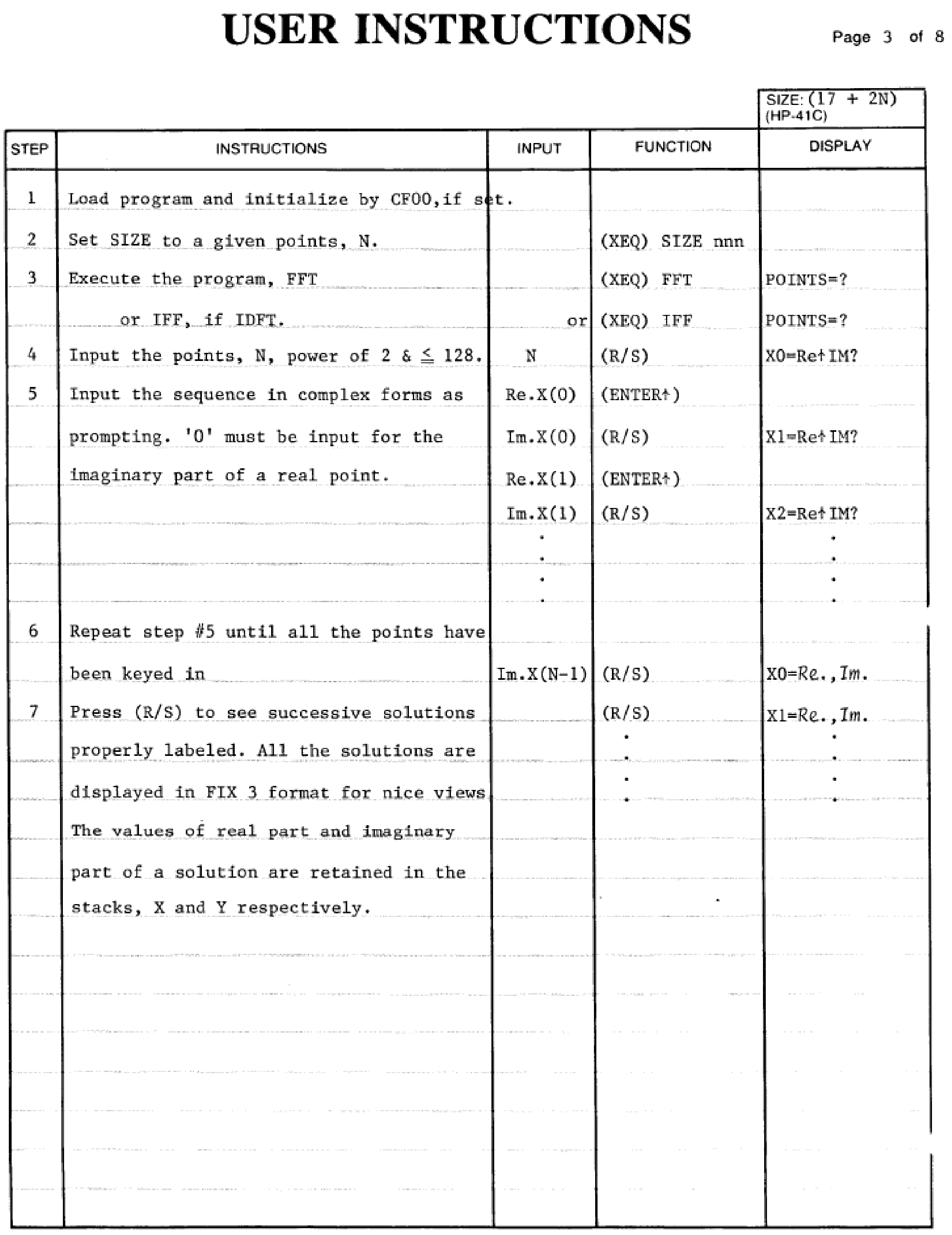
Sample Problem (Sketch if Desired)

Example 1: Find the DFT of a sequence, x(n) = (1, 1+j, 3, 1-j).

Example 2: Find the IDFT of a sequence, X(k) = (6, 0, 2, -4).

SOLUTION:

Input	Function	Display	Comments
* Insert a memory module, load program and clear flag 00 if set.			
	(XEQ)SIZE025		
Example 1:	(XEQ)FFT	POINTS=?	Prompting N.
4	(R/S)	X0=Re↑IM?	Prompting input points.
1	(ENTER↑)	1.0000	Input the real part of x(0),
0	(R/S)	X1=Re↑IM?	and the imaginary part.
1	(ENTER↑)	1.0000	Input all points as prompting.
1	(R/S)	X2=Re↑IM?	
3	(ENTER↑)	3.0000	
0	(R/S)	X3=Re↑IM?	
1	(ENTER↑)	1.0000	
-1	(R/S)	X0=6.000,0.000	Display of output points in the
	(R/S)	X1=0.000,0.000	form of complex-rectangular format
	(R/S)	X2=2.000,0.000	as X(k)=*Real part,Imaginary part*.
	(R/S)	X3=-4.000,0.000	
Example 2:	(XEQ)IFF	POINTS=?	
4	(R/S)	X0=Re↑IM?	
6	(ENTER↑)	6.000	
0	(R/S)	X1=Re↑IM?	
0	(ENTER↑)	0.000	
0	(R/S)	X2=Re↑IM?	
2	(ENTER↑)	2.000	
0	(R/S)	X3=Re↑IM?	
-4	(ENTER↑)	-4.000	
0	(R/S)	X0=1.000,0.000	
	(R/S)	X1=1.000,1.000	Outputs of Example 2.
	(R/S)	X2=3.000,0.000	
	(R/S)	X3=1.000,-1.000	
	(R/S)		(R/S) to clear flag00.
	(R/S)		

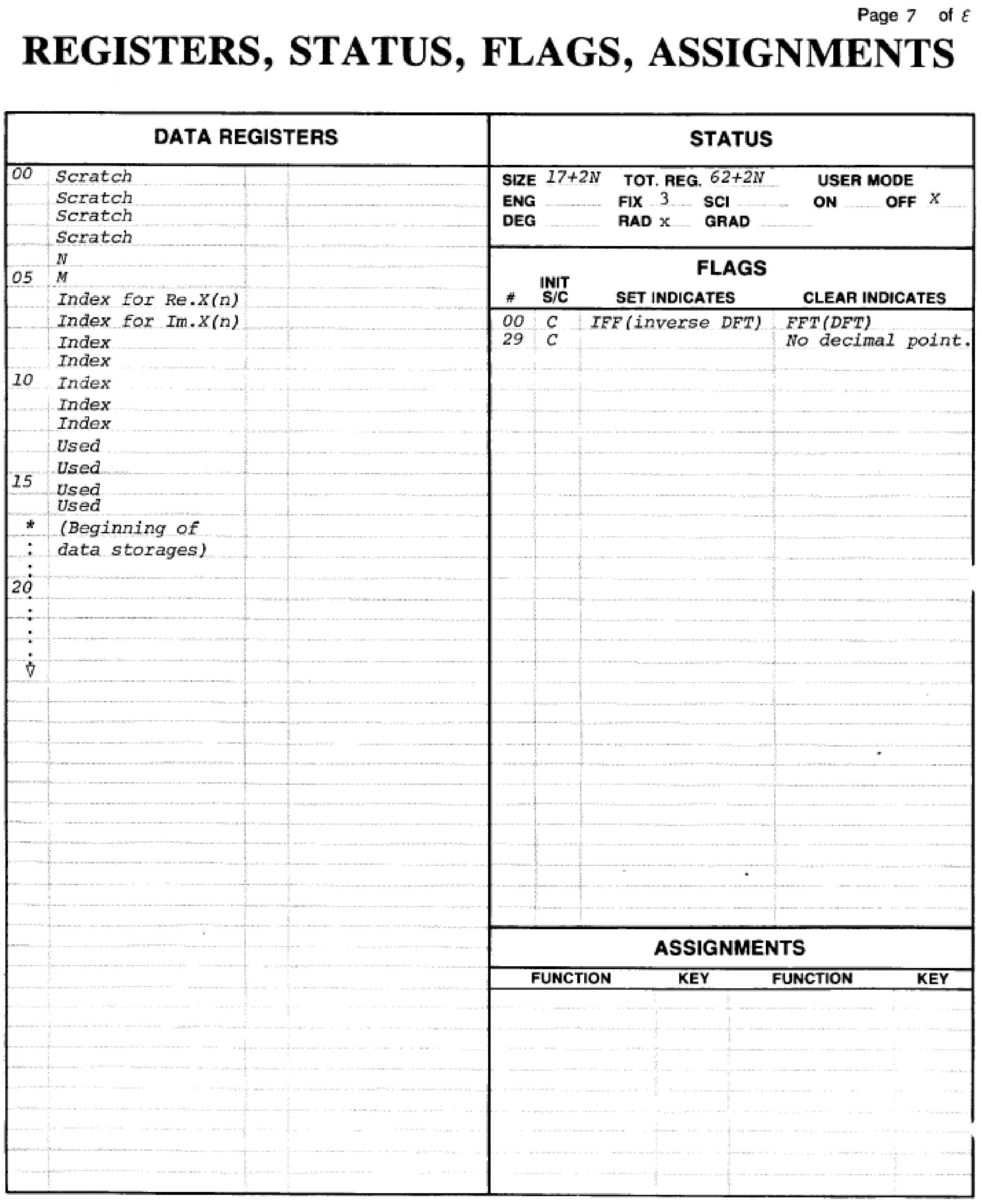


USER INTRUCTIONS

Page 3 of 8

SIZE: (17 + 2N)
(HP-41C)

STEP	INSTRUCTIONS	INPUT	FUNCTION	DISPLAY
1	Load program and initialize by CF00,if set.			
2	Set SIZE to a given points, N.		(XEQ) SIZE nnn	
3	Execute the program, FFT		(XEQ) FFT	POINTS=?
	or IFF, if IDFT.	or	(XEQ) IFF	POINTS=?
4	Input the points, N, power of 2 & ≤ 128.	N	(R/S)	X0=Re↑IM?
5	Input the sequence in complex forms as	Re.X(0)	(ENTER↑)	
	prompting. '0' must be input for the	Im.X(0)	(R/S)	X1=Re↑IM?
	imaginary part of a real point.	Re.X(1)	(ENTER↑)	
		Im.X(1)	(R/S)	X2=Re↑IM?
		⋮		⋮
6	Repeat step #5 until all the points have			
	been keyed in	Im.X(N-1)	(R/S)	X0=Re.,Im.
7	Press (R/S) to see successive solutions		(R/S)	X1=Re.,Im.
	properly labeled. All the solutions are		⋮	⋮
	displayed in FIX 3 format for nice views			
	The values of real part and imaginary			
	part of a solution are retained in the			
	stacks, X and Y respectively.			



Page 7 of 8

REGISTERS, STATUS, FLAGS, ASSIGNMENTS

DATA REGISTERS	
00	Scratch
	Scratch
	Scratch
	Scratch
	N
05	M
	Index for Re.X(n)
	Index for Im.X(n)
	Index
	Index
10	Index
	Index
	Index
	Used
	Used
15	Used
	Used
*	(Beginning of
:	data storages)
20	
↓	

STATUS

SIZE 17+2N	TOT. REG. 62+2N	USER MODE
ENG	FIX 3 SCI	ON OFF X
DEG	RAD x GRAD	

FLAGS

#	INIT S/C	SET INDICATES	CLEAR INDICATES
00	C	IFF(inverse DFT)	FFT(DFT)
29	C		No decimal point.

ASSIGNMENTS

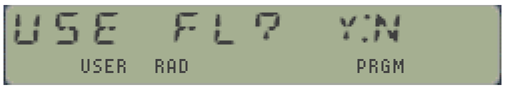
FUNCTION	KEY	FUNCTION	KEY

Using Extended functions makes this program easier to use and provides a more stable repository for the different data sets, which are saved as data files in X-Mem.

Example. Do the transformation for the following data set:

{ 1, 1+j, 3, 3+j }

The first time we'll not be using an existing data file for the data set, even if it already exists - therefore we'll choose "N" to the pertinent question if it appears:



```
USE FL? Y:N
  USER  RAD          PRGM
```

XEQ "FFT	USE FL? Y:N
"N"	#POINTS=?
4, R/S `	Z0=? RE?IM
1, ENTER, 0, RS	Z1=? RE?IM
1, ENTER^, 1, R/S	Z2=? RE?IM
3, ENTER^, 0, R/S	Z3=? RE?IM
3, ENTER^1, R/S	Z0=8+J2
R/S	Z1=-2(1-J)
R/S	Z2=0-J2
R/S	Z3=-2(1+J)
R/S	0,000

At this point the data file FFT contains the four results for a more permanent repository, one that can even be used to obtain the inverse and check the accuracy of the programs:

XEQ "IFF"	USE FL? Y:N
"Y"	Z0= 1+J0
R/S	Z1= 1(1+J)
R/S	Z2= 3+J0
r/s	Z3= 3+J1
R/S	0,000

Note: When the use of the data file is selected the program expects the pointer to be set at the first element. This is normally the case, as both **FFT** and **IFF** will leave it in that setting – but if you manually alter the pointer then an END OF FL error message will probably come up. You can make sure with the sequence 'FFT" . 0, SEEKPTA before running the programs.

```
01   *LBL "IFF"
02   SF 00
03   GTO 03
04   *LBL "FFT"
05   CF 00
06   *LBL 03
07   RAD
08   CF 29
09   SIZE?
10   "FFT"
11   SF 25
12   FLSIZE
13   FC?C 25
14   GTO 03
15   CF 01
16   "USE FL? YN"
17   PMTK
18    E
19   -
20   X#0?
21   GTO 03
22   SF 01
23   RDN
24   ENTER^
25   ENTER^
26   2
27   /
28   STO 04
29   RDN
30   GTO C
31   *LBL 03
32   "#POINTS=?"
33   PROMPT
34   STO 04
35   ST+ X
36   "FFT"
37   SF 25
38   PURFL
39   CF 25
40   CRFLD
41   17
42   +
43   X>Y?
44   PSIZE
45   *LBL C
46   CLX
47   SEEKPTA
48   RCL 04
49   STO 08
50   LOG
51   2
52   LOG
53   /
54   FIX 8
55   RND
56   STO 05
57   FRC
58   STO 00
59   FACT
60   *LBL 01
61   "Z"
62   RCL 00
63   ARCLI
64   "`=? Re^IM"
65   RCL 05
66   STO 01
67   RCL 00
68   STO 03
69   CLX
70   STO 02
71   *LBL 05
72   RCL 03
73   ENTER^
74   ENTER^
75   2
76   /
77   INT
78   STO 03
79   ST+ X
80   -
81   RCL 02
82   ST+ X
83   +
84   STO 02
85   DSE 01
86   GTO 05
87   17
88   RCL 02

89   ST+ X
90   +
91   STO 06
92    E
93   +
94   STO 07
95   FC? 01
96   PROMPT
97   FC? 01
98   GTO 00
99   GETX
100  GETX
101  *LBL 00
102  STO IND 07
103  X<>Y
104  STO IND 06
105  1
106  ST+ 00
107  DSE 08
108  GTO 01
109  1
110  STO 03
111  STO 02
112  *LBL 15
113  2
114  RCL 03
115  Y^X
116  STO 01
117  RCL 02
118  STO 08
119  CLX
120  STO 09
121  PI
122  ST+ X
123  FC? 00
124  CHS
125  RCL 01
126  /
127  RCL 02
128  P-R
129  STO 10
130  X<>Y
131  STO 11
132  RCL 02

133  STO 12
134  *LBL 16
135  RCL 12
136  STO 00
137  *LBL 02
138  RCL 02
139  15
140  RCL 00
141  ST+ X
142  +
143  STO 13
```

```
144   +
145   STO 14
146   LASTX
147   RCL 01
148   +
149   STO 15
150   RCL 02
151   +
152   STO 16
153   RCL IND 15
154   RCL 08
155   *
156   RCL IND 16
157   RCL 09
158   *
159   -
160   STO 06
161   RCL IND 15
162   RCL 09
163   *
164   RCL IND 16
165   RCL 08
166   *
167   +
168   STO 07
169   RCL IND 14
170   RCL 07
171   -
172   STO IND 16
173   RCL IND 13
174   RCL 06
175   -
176   STO IND 15
177   RCL IND 13
178   RCL 06
179   +
180   STO IND 13
181   RCL IND 14
182   RCL 07
183   +
184   STO IND 14
185   RCL 01
186   ST+ 00
187   RCL 04
188   RCL 00
189   X<=Y?
190   GTO 02
191   RCL 10
192   RCL 08
193   *
194   RCL 11
195   RCL 09
196   *
197   -
198   RCL 08
199   RCL 11
200   *
201   RCL 09
202   RCL 10
203   *
204   +
205   STO 09
206   X<>Y
207   STO 08
208   RCL 02
209   ST+ 12
210   RCL 01
211   2
212   /
213   RCL 12
214   X<=Y?
215   GTO 16
216   RCL 02
217   ST+ 03
218   RCL 05
219   RCL 03
220   X<=Y?
221   GTO 15
222   BEEP
223   CLX
224   STO 00
225   17
226   STO 08
227   18
228   STO 09
229   RCL 04
230   STO 01
231   0
232   SEEKPT
233   *LBL 11
234   RCL IND 09
235   RCL IND 08
236   FC? 00
237   GTO 06
238   RCL 04
239   /
240   X<>Y
241   LASTX
242   /
243   X<>Y
244   *LBL 06
245   FIX 8
246   ZRND
247   SAVEX
248   X<>Y
249   SAVEX
250   X<>Y
251   FIX 3
252   ZAVIEW
253   "`Z"
254   RCL 00
255   ARCLI
256   "`="
257   -3
258   AROT
259   PROMPT
260   RCL 02
261   ST+ 00
262   2
263   ST+ 08
264   ST+ 09
265   DSE 01
266   GTO 11
267   CLX
268   SEEKPT
269   END
```

Direct Bessel fns. via Continued Fractions

The SandMath contains a very competent set of Bessel functions, both for the direct (J, Y) and the modified kinds (I, K). The implementation is a hybrid of MCODE and Focal routines, really optimized for the applicable valid range of the functions.

And therein lays the only caveat: that implementation does a direct sum of the alternating terms of the series, which isn't valid for asymptotic cases, where either the order or the argument (or their sum!) are very large. To palliate this, the SandMath also includes an iterative approach for JNX, using recurrence formulas – but alas, the execution time can be really long.

Is there another way to skin this cat? Well as it turns out yes, at least for the non-modified cases there's a very intriguing approach based on continued fractions, which after all are another way to iterate for the solution – only that we can take advantage of the MCODE implementation in both the SandMath and the 41Z Modules, because there are two different continued fractions involved, one of them in the complex variable – even for the real Bessel J case!

Here too the routine is a direct modification of Jean-Marc Baillard's FOCAL program available on his web site (cf #5 in http://hp41programs.yolasite.com/bessel.php), adapted to use the MCODE functions **CF2V** and **ZCF2V** instead of the FOCAL subroutines – faster and shorter code. A real beauty to see the SandMath and 41Z joining forces to crack this one!

The formulas used are as follows:

With $p + i.q = -1/(2x) + i + (i/x) [(0.5^2 - n^2)/(2x + 2i + (1.5^2 - n^2)/(2x + 4i + ...))]$
and $g_n = -1/(((2n + 2)/x) - 1/(((2n + 4)/x) - .....))$

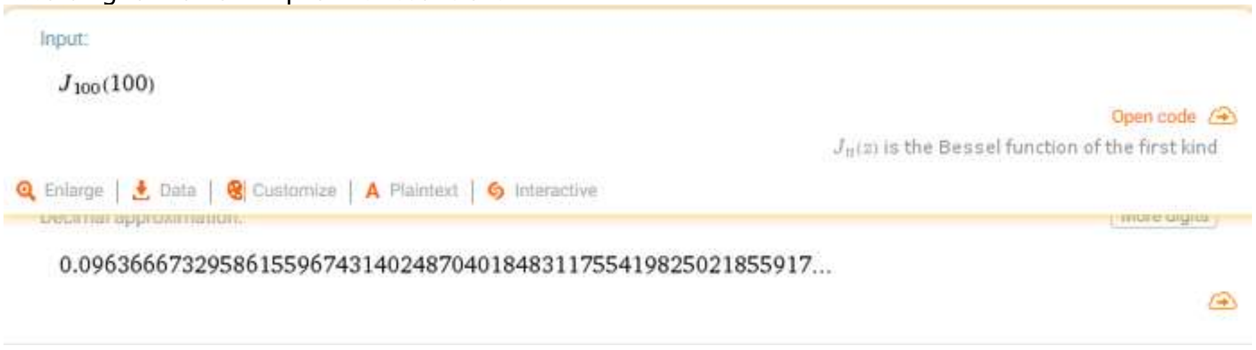
Then, calling D = the denominator of the second continued fraction:

$$J_n(x) = \text{sign}(D) [(2q/(x.Pi)) / (q^2 + (p - g_n - n/x)^2)]^{1/2}$$
$$Y_n(x) = [(p - g_n - n/x)/q] J_n(x)$$

One must pay careful attention to the data registers requirements by these functions for the successions used to define the continued fractions, which are programmed under the global labels "#" for the real one and '=" for the complex one.

<u>Example</u>: Calculate the Bessel J and Y of order 100 for the argument x=100

According to Wolfram Alpha the results are:


Input:

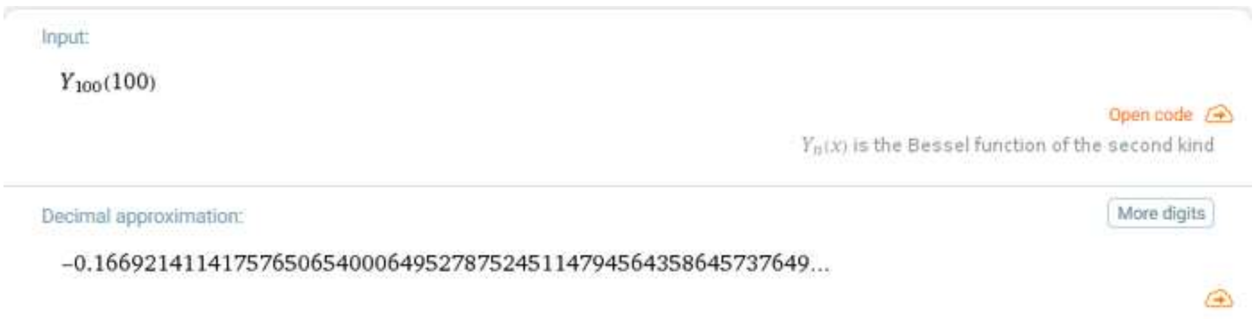
$J_{100}(100)$

$J_n(z)$ is the Bessel function of the first kind

Decimal approximation:

0.096366673295861559674314024870401848311755419825021855917...

and:


Input:

$Y_{100}(100)$

$Y_n(x)$ is the Bessel function of the second kind

Decimal approximation:

−0.16692141141757650654000649527875245114794564358645737649...

Which sure enough is what we obtain (with ten digit precision) using our routine:

100, ENTER^, XEQ "JYNX" => 0.0963666738

X<>Y -0.166921 4116

Program Listing:

```
01  LBL "JYNX"      32  RCL 09      63  GTO 00
02  STO 01          33  +           64  RCL 12
03  X<>Y            34  RCL 13      65  ST+ X
04  STO 13          35  RCL 01      66  RCL 02
05  "="             36  /           67  ST+ X
06  CLST            37  -           68  ZENTER^
07  ZENTER^         38  STO 11      69  RCL 12
08  .               39  RCL 10      70  0.5
09  RCL 01          40  R-P         71  -
10  SF 02           41  LASTX       72  X^2
11  ZCF2V           42  ST+ X       73  RCL 13
12  RCL 02          43  PI          74  X^2
13  STO 01          44  RCL 01      75  -
14  ST/ Z           45  *           76  0
15  /               46  /           77  X<>Y
16  E               47  SQRT        78  RTN
17  +               48  X<>Y        79  LBL 00
18  STO 10          49  /           80  X<>Y
19  X<>Y            50  RCL 05      81  STO 05
20  CHS             51  SIGN        82  X<>Y
21  RCL 01          52  *           83  RCL 02
22  ST+ X           53  STO 12      84  RCL 13
23  1/X             54  RCL 11      85  +
24  -               55  *           86  ST+ X
25  STO 09          56  RCL 10      87  RCL 01
26  "="             57  /           88  /
27  0               58  RCL 12      89  -1
28  RCL 01          59  CLD         90  END
29  CF 02           60  RTN
30  CF2V            61  LBL "="
31  CHS             62  FC? 02
```

Note: ensure that the module is plugged in <u>a page before the SandMath</u>. This is required because there is another global label "=" in the SandMath and we don't want the routine to use the incorrect one for the calculation! (besides, this would result in NONEXISTENT, so you'll know right away).

Nested Radicals of m-th order.

FNRM and **INRM** are MCODE functions to calculate finite and infinite Nested Radicals or root-order m. The definition of the radical is given in a user-provided function under a global label, to generate the n terms that contribute to the radical R(n).

- For the finite case the calculation ends when all the terms are provided and used in the radical.
- For the infinite case, a series of finite radicals of increasing sizes are computed until two of them are equal. This means R(n) = R(n+1), for a given n large enough.

An **initial size n0** needs to be provided by the user, which ideally is a balance between the radical size and the number of subsequent radicals to calculate: the larger the radical the longer calculation time, but the less number of radicals likely to calculate.

STACK	INPUTS	OUTPUTS
Y	k	
X	no	NR
ALPHA	F.NAME	/

FNRM and **INRM** use data registers {R00 – R05} as well as user flags UF 001 and UF 01. Refrain from using these resources in the definition of your radicand functions. Note that both the root order m and the term n are available for your user function to use – even if normally only n is used. This allows for more elaborate expressions in the definitions.

For example, let's calculate the value of an infinite nested radical with f(n) = n, as per the expression below:

$$\sqrt{n+\sqrt{n+\sqrt{n+\sqrt{n+\cdots}}}} = \frac{1}{2}\left(1+\sqrt{1+4n}\right)$$

For the case n=1 this happens to be the golden ratio Φ = ½ (1+sqr(5)

A trivial user program like this: {LBL "PH", 1, RTN}, say we set FIX 9 and then we type:

```
2, ENTER^ 4, XEQ "INRM"_ PH" =>   1.618033989
```

Using cubic roots instead we'll obtain the "Plastic" Constant:

```
3, ENTER^, 4, XEQ "INRM"_"PH =>   1.324717957
```

<u>Example2</u>.Calculate the cubic and quartic root nested radicals for the function F(n) = n^4

Using n0=4 and the trivial user function {LBL "NR4", X^2, X^2, END} we get:

```
4, ENTER^, 4, XEQ "INRM"_"NR4"       =>1.325706774   quartic case
3, ENTER^, 4, XEQ "INRM"_"NR4"       =>1.551416993   cubic case
```

Example 3. Calculate the square nested radical for the function F(n) = n {LBL "NR1", RTN"}

2, ENTER^, 4, XEQ "INRM"_"NR1" =>1.757932757

Programmer's notes.

These functions use a special technique to call user-programs within the MCODE. This technique was developed by Greg McClure for the Derivatives and Continued fractions (DERV and CF2V) applications available in both the SandMath and the 41Z, and has been ported here as well. The method requires ancillary housekeeping functions to manage the transitions between User- and M-Code. These auxiliary functions are stealth under the FAT section headers, as they don't require any user interaction or utilization beyond its automated purpose.

Execution flow:

1. Search for User function using [ASRCH]
2. Save its RAM address (in data register)
3. Prepare variables and check data regs available
4. transfer to FOCAL stub code (call to [XMR20]
    a. add address to FOCAL RTN stack with [SAVRTN]
    b. execute user function via [XGI07] (but **can't use** XEQ IND nn !!)
    c. return to MCODE, popping the FOCAL RTN with [XRTN]
5. Loop back to task #3 as needed

Where tasks 4.a, 4.b and 4.v are performed by XQRTN, a dedicated (stealth) function used in the FOCAL stub. It is called twice, controlled by UF 00 to determine which one of the tasks to perform:

```
-FOCAL stub code -
      01  SF 00
      02  XQRTN       - first time does 4.a and 4.b
      03  CF 00
      04  XQRTN       - second time does 4.c
```

Newton and Halley Methods Revisited

The idea of using the MCODE functions in the SandMath and the 41Z is also at the heart of these final applications. This time we'll use the first & second derivatives function **DERV** as an auxiliary tool to calculate the derivatives of the function whose roots we're trying to obtain, directly and without any additional conditioning regardless of the function in case.

The formulas involved are well known:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad ; \quad x_{n+1} = x_n - \frac{2f(x_n)f'(x_n)}{2[f'(x_n)]^2 - f(x_n)f''(x_n)}$$

As usual, you need to provide the boundaries [a, b] in the Y,X registers and the function name in ALPHA. The user is required to program the function in a FOCAL routine under a global label, which cannot use data registers R00 to R08 as explained below.

Remember that **DERV** uses R00 to R04 (see the documentation in the SandMath manual for details), and in addition to these the routines use R05 for the function global label name, and R06 – R08 to save the initial guesses and as scratch. As it's already customary, the successive approximations to the root will be displayed if user flag 10 is set.

```
 1  *LBL "XNWT"
 2  CF 01
 3  GTO 01
 4  *LBL "XHALL"
 5  SF 01
 6  *LBL 01
 7  ASTO 05
 8  X<>Y
 9  STO 08
10  X<>Y
11  *LBL 00
12  FS? 10
13  VIEW X
14  STO 06
15  XEQ IND 05
16  STO 07
17  RCL 08
18  RCL 06
19  DERV
20  FC? 01
21  ST/ 07
22  FS? 01
23  XEQ 02
24  RCL 06
25  RCL 06
26  RCL 07
27  -
28  X#Y?
29  GTO 00
30  CLD
31  RTN
32  *LBL 02
33  X^2
34  ST+ X
35  RCL 07
36  RCL 01
37  *
38  -
39  1/X
40  RCL 07
41  *
42  RCL 00
43  *
44  ST+ X
45  STO 07
46  END
```

This really can't get any shorter; my kinda routine that clearly showcases that with a powerful engine behind doing the heavy lifting (**DERV** in this case) the rest is a downhill trip.

Example: obtain a root for the equation below, which we program easily as shown. Then we use some obviously non-optimal guesses to stress the algorithm:

$$y = \sqrt[3]{x}(x+4)$$

{ LBL "X1", CBRT, LASTX, 4, +, *, END }, and then

```
      ALPHA,"X1", ALPHA, 1, 2, XEQ "XNWT"  =>  -4.00000000
Or:`  1, 2, XEQ "XHALL"                    =>  -4.00000000
```

Newton's Method with Complex Step Differentiation.

And the proverbial last but not least is reserved for the "complex step derivative" method to calculate real function derivatives, just as a quasi-magical application of complex variables. Complex step differentiation is a technique that employs complex arithmetic to obtain the numerical value of the first derivative of a real valued analytic function of a real variable, avoiding the loss of precision inherent in traditional finite differences. This is then used n Newton's method in the usual way.

We're concerned with an *analytic* function. Mathematically, that means the function is infinitely differentiable and can be smoothly extended into the complex plane. Computationally, it probably means that it is defined by a single "one line" formula, not a more extensive piece of code with if statements and for loops.

Let $F(z)$ be such a function, let x_0 be a point on the real axis, and let h be a real parameter. Expand $F(z)$ in a Taylor series off the real axis.

$$F(x_0+ih) = F(x_0) + i.hF'(x_0) - h^2F''(x_0)/2! - ih^3F^{(3)}/3! + \ldots$$

Take the imaginary part of both sides and divide by h

$$F'(x_0) = Im(F(x_0+ih))/h + O(h^2)$$

Armed with the 41Z arsenal of functions it's very likely that your real function can be programmed as an equation in the complex variable too. Then all it takes is to calculate the value of said complex function in a complex point close to the real argument x_0, offset by a very small amount in the imaginary axis ih. The program expects the function name in ALPHA and the values of h and x0 in the Y,X stack registers, and it returns the real derivative value in X. It uses data registers R00 to R02.

```
LBL "ZNWT"
 ASTO 02
 ZSTO 00
 LBL 00
FS? 10
VIEW 00
ZRCL 00
XEQ IND 02
X<>Y
/
RCL 01
*
ST- 00
RND
X#0?
GTO 00
RCL 00
END
```

What's remarkable is that with just one execution of the complex function we calculate both the real function's value (the real part) and its derivative (the imaginary part with correction) at the same time. Note also the clever use of complex data register C00 to store z0 = x0 +ih, and then how it keeps calculating the complex function value until two successive iterations are equal for the current FIX selected in the calculator.

You can tell something's remarkable when the root-finding routine is almost shorter than the equation used to program the function!

Time now for some examples. The first one just a simple polynomial to try our hand with the new method, taken from the MoHPC forum: https://www.hpmuseum.org/forum/thread-6667.html

Calculate the three roots of the third degree polynomial: $x^3–x^2–x+0,5 = 0$

We program the equation as shown below:

```
01  LBL "Z3"          06  Z-
02  Z^3               07  .5
03  LASTZ             08  +
04  Z^2               09  END
05  Z+
```

And type:

```
ALPHA, "Z1", ALPHA
,01, ENTER^, 0, XEQ "ZNWT"        =>   0.40301587
.01, ENTER^, 2, XEQ "ZNWT"        =>   1.45174468
.01, ENTER^, -2, XEQ "ZNWT"       =>  -0.85476055
```

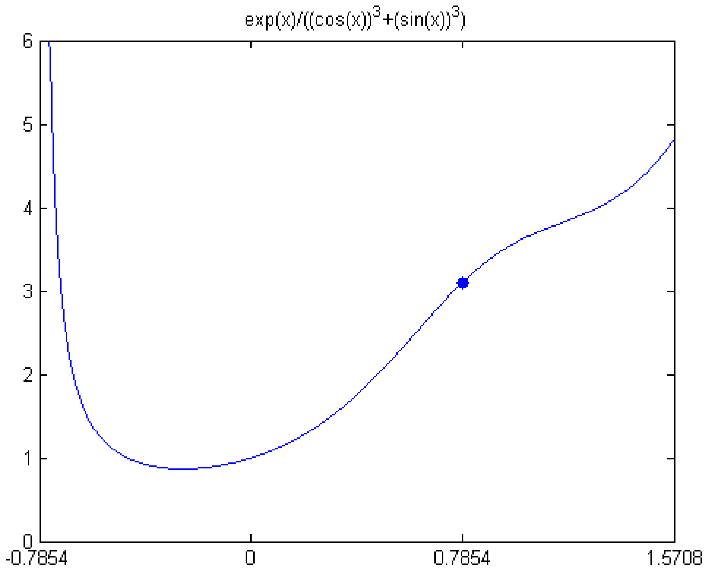
And then a more elaborate example adapted from the seminal reference: https://blogs.mathworks.com/cleve/2013/10/14/complex-step-differentiation/

The blog uses the function F(x) given below, which does not have any real roots:

$$F(x) = \frac{e^x}{(\cos x)^3 + (\sin x)^3}$$

For our purposes let's calculate the roots of, say g(x) = F(x) – π

```
01  LBL "Z2"
02  ZEXP
03  LASTZ
04  ZSIN
05  LASTZ
06  ZCOS
07  3
08  Z^X
09  Z<>W
10  3
11  Z^X
12  Z+
13  Z/
14  PI
15  -
16  END
```



And type:

```
ALPHA, "Z2", ALPHA
,01, ENTER^, 1, XEQ "ZNWT"        =>   0.79830245
```

Halley's Method for Complex Functions

To complement the choices already available in the 41Z (programs **ZSOLVE** and **ZHALL**), a third program is included in the Contour module as well.

This program is based on Valentín Albillo's article "*Going back to the roots*", where he presented an HP-35S solution to the problem. The final version shown here was aided by a first port to the HP-41 platform by Vincent Weber, contributed to the MoHP forum

see: *https://www.hpmuseum.org/forum/thread-21615.html*) and

https://albillo.hpcalc.org/articles/HP%20Article%20VA031%20-%20Boldly%20Going%20-%20Going%20Back%20to%20the%20Roots.pdf)

User instructions:

Just type in ALPHA the name of the global label the function has been programmed under, and the guess value in stack registers Y,X (i.e. complex stack level Z), then call the program. After a while the root found is presented in the displat. Execution time depends on the initial guess value and the number of decimal places used for the precision setting.

Example: obtain one root for the expression $f(z) = z^\wedge z - \pi$

We program the function under LBL "ZZ" as follows:

```
01  LBL "ZZ"
02  ZENTER^
03  W^Z
04  PI
05  –
06  END
```

Next, we enter a guess value (imaginary part in Y, real part in X), and call **ZROOT**.
After a while the result is shown in the display

```
ALPHA, "ZZ", ALPHA
0,  ENTER^, 1, XEQ "ZROOT"        →      1,854+J0

verification:   XEQ "ZZ"          →      0+J0
```

The program is listed below. Being a port from another machine I decided to leave parts unchanged, not using 41Z functions in them to maintain the original ideas. Nevertheless the MCODE 41Z functions are profusely used all throughout the code, contributing to a faster execution and more accurate results.

Program listing.

```
01    *LBL "ZROOT"
02    ZSTO 00
03    ASTO 02
04    E-4
05    STO 10
06    X^2
07    STO 11
08    ,5
09    STO 03
10    *LBL 02
11    ZRCL 00
12    XEQ IND 02
13    RCL 03
14    ST/ Z
15    /
16    ZSTO 02
18    RCL 10
19    ST+ 00
20    ZRCL 00
21    XEQ IND 02
22    ZSTO  03
24    RCL 10
25    ST- 00
26    ST- 00
27    ZRCL 00
28    XEQ IND 02
29    ZSTO  04
31    ZRC+ 03
33    ZRC- 02
35    RCL 11
36    ST/ Z
37    /
38    ZRCL 03
40    ZRC- 04
42    RCL 10
43    ST+ 00
44    ST/ Z
45    /
46    RCL 03
47    ST* Z
48    *
49    ZSTO 04
51    Z/
52    ZSTO 03
54    ZRC* 02
56    ZRC/  04
58    1
59    -
60    CHS
61    X<>Y
62    CHS
63    X<>Y
64    RCL 03
65    Z^X
66    1
67    -
68    ZRC/ 03
70    ZST+ 00
71    ZRC/ 00
72    R-P
73    RCL 11
74    X<Y?
75    GTO 02
76    ZRCL  00
77    R-P
78    RDN
79    STO 08
80    SIN
81    ABS
82    RCL 11
83    X<=Y?
84    GTO 03
85    RCL 08
86    COS
87    ENTER^
88    ABS
89    X#0?
90    /
91    RCL 01
92    RCL 00
93    R-P
94    X<>Y
95    RDN
96    *
97    STO 00
98    *LBL 03
99    ZRCL 00
100   ZAVIEW
101   END
```

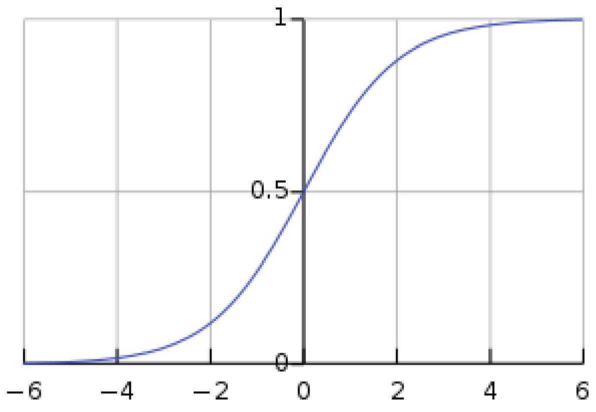
Note that the line numbers reflect the non-merged character of some 41Z functions, taking two standard lines (that have been merged in the listing).

Sigmoid and Einstein functions,

SIGMD calculates the Sigmoid of the argument in x. This function is relevant in machine learning and data mining fields. It is defined as:

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1} = 1 - S(-x).$$

The result is placed in X and the original argument is saved in LastX. Y,Z,T are untouched and no data registers are used either.



Examples:

1, XEQ “SIGMD => 0.731058579
2, XEQ “SIGMD => 0.880797078

The Sigmoid function is also known as the *Standard Logistics function*, which will appear linked to the Logistics Map in the discrete domain – refer to the CHAOS Module for additional applications.

Derivative and Integral of the Sigmoid function.

The derivative is known as the density of the logistic distribution:

$$\frac{\mathrm{d}}{\mathrm{d}x} f(x) = \frac{e^x \cdot (1+e^x) - e^x \cdot e^x}{(1+e^x)^2} = \frac{e^x}{(1+e^x)^2} = f(x)\big(1 - f(x)\big)$$

Conversely, its antiderivative can be computed by the substitution u = 1+e^x , since f(x) = u'/ u, so (dropping the constant of integration)

$$\int \frac{e^x}{1+e^x}\, dx = \int \frac{1}{u}\, du = \ln u = \ln(1+e^x).$$

In artificial neural networks, this is known as the *softplus* function and (with scaling) is a smooth approximation of the ramp function, just as the logistic function (with scaling) is a smooth approximation of the Heaviside step function.

Finally, SIGMD is a rather simple function. The MCODE listing is shown below.

Header	AEAC	084	"D"	
Header	AEAD	00D	"M"	Sigmoid Function
Header	AEAE	007	"G"	sig = 1/(1 + e^-x)
Header	AEAF	009	"I"	
Header	AEB0	013	"S"	Ángel Martin
SIGMD	AEB1	0F8	READ 3(X)	
	AEB2	361	?NC XQ	(includes SETDEC)
	AEB3	050	->14D8	[CHK_NO_S]
	AEB4	2BE	C=-C-1 MS	Sign change
	AEB5	044	CLRF 4	standard version (w/out "-1")
	AEB6	029	?NC XQ	e^-x
	AEB7	068	->1A0A	[EXP10]
	AEB8	001	?NC XQ	1+e^-x
	AEB9	060	->1800	[ADDONE]
	AEBA	239	?NC XQ	1/(1+e^-x)
	AEBB	060	->188E	[ON/X13]
	AEBC	331	?NC GO	Overflow, DropST, FillXL & Exit
	AEBD	002	->00CC	[NFRX]

Here's a minimalistic FOCAL routine for the derivative and the antiderivative:

```
01 LBL "SGD"
02 SIGMD
03 ENTER^
04 CHS
05 E
06 +
07 *
08 RTN
09 LBL "SGI"
10 SIGMD
11 LN1+X
12 END
```

Einstein functions.

Typically four functions are considered under this classification, as follows:

$$E_1(x) = \frac{x^2 e^x}{(e^x - 1)^2}$$

$$E_2(x) = \frac{x}{e^x - 1}$$

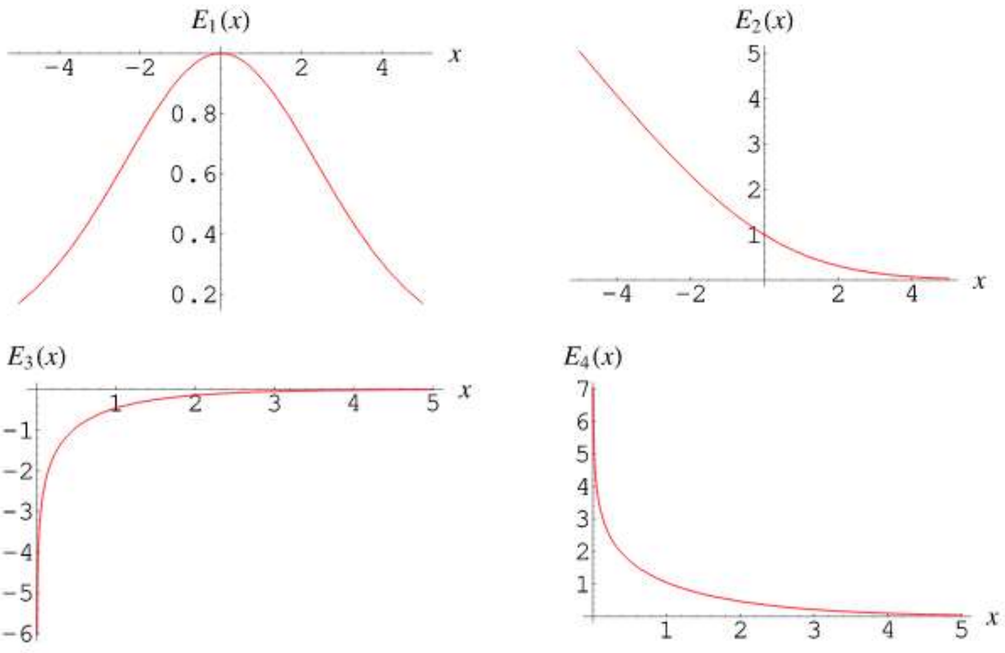
$$E_3(x) = \ln(1 - e^{-x})$$

$$E_4(x) = \frac{x}{e^x - 1} - \ln(1 - e^{-x}).$$

Clearly E4(x)= E2(x) – E3(x), thus no dedicate function for it exists in the module.

The module uses a prompting field for a parameter value from 1 to 3 to select the specific function to calculate. Any input larger than 3 will calculate E3(x), whereas entering zero returns a DATA ERROR message. Besides that, in program mode you need to add the parameter as a second program line after EINS

See below the graphics for these in the range x around the origin



E1(x) has an inflection point at:

$$E_1''(x) = \frac{1}{8}\operatorname{csch}^4\left(\frac{1}{2}x\right)\left[\left(x^2+2\right)\cosh x + 2\left(x^2 - 2x\sinh x - 1\right)\right] = 0,$$

which can be solved numerically to give x=+/-2.34694130...

Example: Calculate E1, E2, and E3 for x = 1

```
1, XEQ "EIN" ,1          => 0.581976707
1, XEQ "EIN" ,2          => 0.920673594
1, XEQ "EIN" ,3          => -0.458675145
```

Example: Calculate E1(E2(E3(x))), and E3(E2(E1(x))) for x = 1

```
1, EINS-1, EINS-2, EINS-3        => -0.475188625

1, EINS-3, EINS-2, EINS-1        => 0.587875507
```

Arc Length of a Curve defined by y = f(x)

-The arc length of the curve y = f(x) (a < x < b) is given by

$$s = \int_a^b \sqrt{1 + \left(\frac{dy}{dx}\right)^2}\, dx.$$

The module includes two programs to calculate the arc length. The first one "**CLEN**" is a direct (i.e. brute-force) application of this formula using **FINTG** and **DERV** in the SandMath. It clearly is simpler to program but foreseeably with longer execution time than a dedicated approach. It also requires a second FAT entry for the auxiliary program that defines the integrand, as you can see in the program listing below.

To use this program, just type the function's program label name in ALPHA, and enter the integration limits a in Y, b in X.

```
01  LBL "CLEN"
02  ASTO 05
03  "*CL"
04  FINTG
05  RTN
06  LBL "*CL"
07  CLA
08  ARCL 05
09  0.1
10  X<>Y
11  DERV
12  X^2
13   E
14  +
15  SQRT
16  END
```

As always, **FINTG** determines the precision of the result by the number of decimal places set in the calculator. Using FIX 9 yields the maximum accuracy but takes the longest time to compute it.

The second one **"LNG"** doesn't use this formula and so it avoids the calculation of dy/dx . It simply applies Pythagoras' theorem. "LNG"was written by Jean-Marc Baillard, and it is included in his DERIVE+ module, see: http://www.hp41.org/LibView.cfm?Command=View&ItemID=1315

Data Registers: • R00 = Function name
(Register R00 is to be initialized before executing "LNG")

R01 = a   R04 to R07: temp "
R02 = b   R20, R21, ... are used by "ROM
R03 = L

Flag: F02 is cleared

Subroutines: "ROM" , plus a program that takes x in X-register and returns f(x) in X-register

STACK	INPUT	OUTPUT
Y	a	/
X	b	L(a,b)

Example: Calculate the arc length of the curve y = ln x   1 < x < 3

```
01  LBL "T"
02  LN
03  RTN
```

Manual data entry:

ALPHA , "T", ASTO 00, ALPHA
FIX 4, 1, ENTER^, 3,

Using the direct approach:

XEQ "CLEN" >>>> 2,30 1987548
---Execution time = 1m 35s---

Using the iterative approach in manual way, skipping the data entry prompts:

GTO "LNG", XEQ C >>>> 2.30 1987533 ---
Execution time = 75s---

Notes:

The HP41 displays the successive approximations
The precision depends on the display format: for instance, FIX 6 would be faster but less accurate.

The exact result is L = 2.301987535 (rounded to 9 decimals)

The program listing below includes the Arc Length and the Surgace of Revolution described in next section – both can be combined into a single application with considerable byte savings.

The program starts with a data entry section, prompting for the required information on the function and integration limits. You can skip these steps if you prefer a manual data entry using the soft-Label "C"

Note the use of function **PMTA** in the OS.X module to enter the function's global label name. It can be replaced by { AON, PROMPT, AOFF} as well.

```
01   *LBL "LNG"
02   CF 02
03   GTO 00
04   *LBL "SRV"
05   SF 02
06   *LBL 00
07   "FNAME? "
08   PMTA
09   ASTO 00
10   "a^b=?"
11   PROMPT
12   *LBL C
13   STO 02
14   X<>Y
15   STO 01
16   1
17   STO 20
18   *LBL 11
19   CLX
20   STO 04
21   RCL 02
22   RCL 01
23   STO 05
24   -
25   RCL 20
26   STO 06
27   /
28   STO 03
29   ST+ 05
30   RCL 01
31   XEQ IND 00
32   STO 07
33   *LBL 12
34   RCL 05
35   XEQ IND 00
36   ENTER^
37   ENTER^
38   X<> 07
39   ST+ Z
40   -
41   X^2
42   RCL 03
43   ST+ 05
44   X^2
45   +
46   SQRT
47   FS? 02
48   *
```

```
49    ST+ 04
50    DSE 06
51    GTO 12
52    PI
53    FS? 02
54    ST* 04
55    RCL 04
56    XROM "*RM"
57    X#0?
58    GTO 11
59    RDN
60    STO 03
61    RTN
62    *LBL "*RM"
63    RCL 20
64    X<>Y
65    SIGN
66    ST* X
67    X#Y?
68    GTO 01
69    STO 23
70    LASTX
71    STO 25
72    GTO 03
73    *LBL 01
74    4
75    STO 21
76    25
77    STO 22
78    RCL 23
79    STO 24
80    LASTX
81    ISG 23
82    *LBL 02
83    ENTER^
84    ENTER^
85    X<> IND 22
86    -
87    RCL 21
88    4
89    ST* 21
90    SIGN
91    ST+ 22
92    -
93    /
94    +
95    DSE 24
96    GTO 02
97    STO IND 22
98    VIEW X
99    RND
100   X<>Y
101   RND
102   X#Y?
103   GTO 03
104   RCL IND 22
105   0
106   RTN
107   *LBL 03
108   RCL 20
109   ST+ 20
110   END
```

The “*RM” routine could be replaced by FINTG as well…

Romberg Method

Suppose that a sequence {Ln} tends to L as n tends to infinity and that the "errors" L -Ln are nearly proportional to 1/n^2

If we want to use Romberg method to estimate the limit L "RM" must be called by a program with the following specifications:

L must be stored in R20 at the beginning
Then, a loop - say LBL 01 - calculates the value of Ln in X-register corresponding to n in R20
The last instructions must be XEQ "ROM" X#0? GTO 01 RDN END

See the paragraphs above for several examples ("CRVL" "CRVLN" "LNG" "SRV" "SKS")
You can also use it for your own programs, provided that registers R20 R21 ..... are not disturbed.

Area of a Surface of Revolution

The rotation of the curve y = f(x) (a < x < b) around x-axis generates a surface of revolution given by

$$A_x = 2\pi \int_a^b y\sqrt{1+\left(\frac{dy}{dx}\right)^2}\,dx = 2\pi \int_a^b f(x)\sqrt{1+\left(f'(x)\right)^2}\,dx$$

The program included in the module "SRV"was written by Jeam-Marc Baillard. "SRV" avoids the calculation of dy/dx : the area of a truncated cone is used with Romberg method.

Data Registers: • R00 = Function name
(Register R00 is to be initialized before executing "SRV")

R01 = a	R04 to R07: temp
R02 = b	R20, R21, ... are used by "ROM"
R03 = A	

Flag: F02 is set

Subroutines: "ROM" & 1 program that takes x in X-register and returns f(x) in X-register

STACK	INPUT	OUTPUT
Y	a	/
X	b	A(a,b)

Example: The sin of revolution.

Evaluate the area of the surface of revolution generated by the rotation of the curve
y = sin x (0 < x < pi) around the x-axis.

```
01  LBL "T"
02  SIN
03  RTN
```

Using a manual approach that skips the data entry prompts:

ALPHA, "T" , ASTO 00
FIX 9, 0, ENTER^, PI,

GTO "SRV", XEQ C >>>> 14,42359950 ---
Execution time = 168s---

Notes:

The HP41 displays the successive approximations. The precision depends on the display format: for instance, FIX 6 would be faster but less accurate

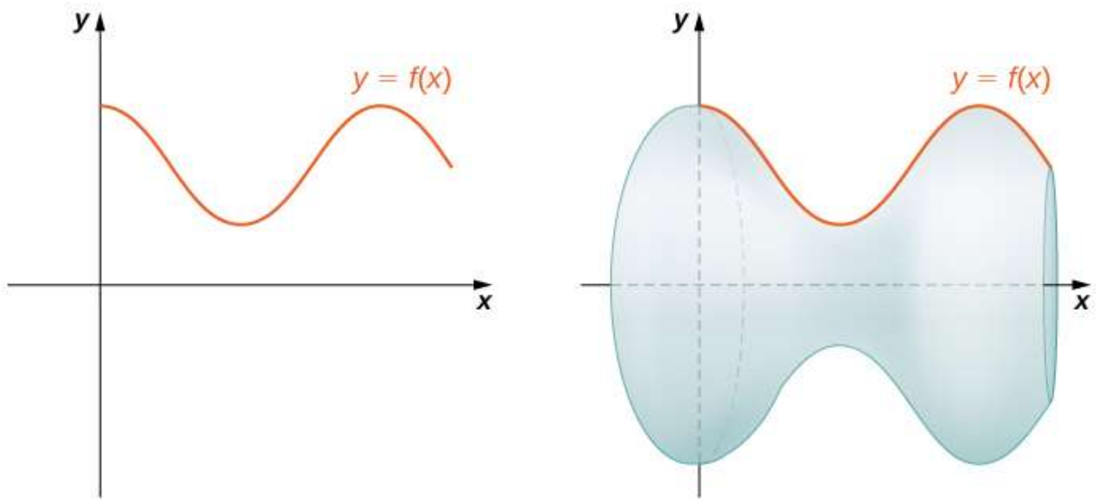
-The exact result is A = 14.42359945 (rounded to 8 decimals).

Note that in this case the module doesn't include the direct approach based on FINTG and DERV. If you're interested it'd be very simple to modify CLEN to do it, as follows:

```
01  LBL "SREV"
02  ASTO 05        function LBL
03  "*SR"          integrand LBL
04  FINTG
05  PI
06  ST+  X         2.π
07  *
08  RTN
09  LBL "*SR"      integrand
10  STO 06         saves x in R06
11  CLA
12  ARCL 05
13  0.1            step size
14  X<>Y
15  DERV
16  X^2
17   E
18  +
19  SQRT           partial value
20  X<>  06        x
21  XEQ IND 05     f(x)
22  RCL 06         previous value
23  *              integrand
24  END
```

Using this ad-hoc program the results for example 1 are EXACTLY as follows:

ALPHA, "T", ALPHA, 0, PI, XEQ "SREV" >>> 14,42359945



Reference: this web site is an excellent reference on this subject, also providing some examples to check the programs described before.

https://math.libretexts.org/Courses/University_of_California_Davis/UCD_Mat_21B%3A_Integral_Calculus/6%3A_Applications_of_Definite_Integrals/6.4%3A_Areas_of_Surfaces_of_Revolution

Area of a Surface defined by z = f(x,y)

"SKS" computes the area of a surface defined by: $z = f(x,y)$ $a < x < b$, $c < y < d$

The result could be obtained by the double integral

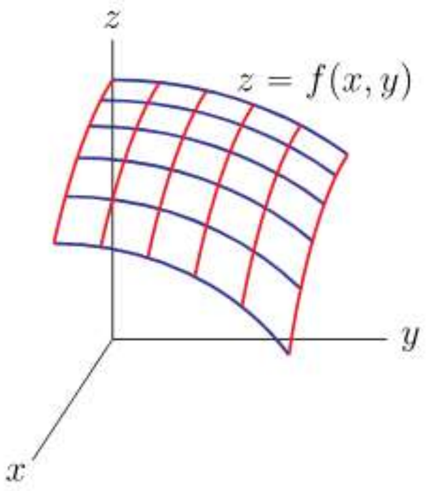
$$= \iint_T \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2 + 1}\, dx\, dy$$

where fx = df/dx and fy = df/dy are the partial derivatives with respect to x and y respectively.

But "SKS" avoids the calculation of the partial derivatives:

The intervals [a,b] and [c,d] are divided into n parts, and the approximate area is the sum of the areas of triangles.

"*RM" uses Romberg method to obtain more and more accurate approximations.



Data Registers:

R00 = Function name

R01 = a	R04 = d	R06 to R16: temp
R02= b	R05 = A	R20, R21, ... are used by "ROM"
R03 = c		

Subroutines: "*RM" plus a program that takes x in X-register & y in Y-register and returns f(x,y) in X-register

STACK	INPUTS	OUTPUTS
T	a	/
Z	b	/
Y	c	/
X	d	A

Example: Evaluate the area of the surface defined by
z = (25 - x2 - y2)1/2 , 0 < x < 2 , 0 < y < 3

To get faster result, store 25 in an unused register, for instance R17, 25 STO 17

```
01  LBL "T"
02  X^2
03  X<>Y
04  X^2
05  +
06  RCL 17
07  X<>Y
08  -
09  SQRT
10  RTN
```

And using manual data entry:

```
ALPHA, "T",  ASTO 00,  ALPHA
FIX 6,  0,  ENTER^, 2,   ENTER^, 0,  ENTER^, 3
GTO "SKS",  XEQ C                      >>>>   6.654397
 ---Execution time = 5m06s---
```

Notes:

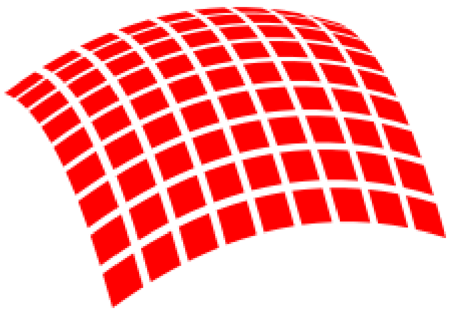
The HP41 displays the successive approximations

The precision depends on the display format: for instance, FIX 9 would give more accurate results but with a much longer execution time as the price to pay for it.

With V41 & FIX 9 we get: 6.654396106

The exact result is A = 6.654396117

As usual with Romberg method, n is multiplied by 2 at each iteration, but here execution time is multiplied by 4 because we are approximating a double integral.



$$= \iint_T \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2 + 1}\, \mathrm{d}x\, \mathrm{d}y$$

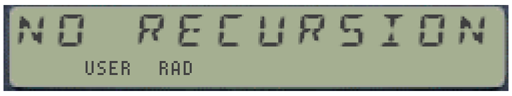
Recursive Utilization of FINTG and FROOT.

Like the original SOLVE and INTEG did, both **FROOT** & **FINTG** in the SandMath support "crossed" nested calls from one another, i.e. you can call FROOT from an integrand function being used by FINTG, and you can call FINTG in the root-finding function definition for FROOT. However, it is not possible to recursively call either one of these functions sequentially from within a FOCAL routine. Any attempt to do so triggers the "*RECURSION*" error message and the execution aborts.

This ROM provides a set of MCODE functions and two FOCAL routines to overcome this limitation. Each time FROOT/FINTG is executed it creates a dedicated memory buffer to store the application data and to perform all the math. The basis of the recursive operation is the use of a secondary memory area for the nested call of the function, not conflicting with the initial memory buffer created in the first call. The main loop uses the initial buffer #14, and the operand function in turn creates a secondary buffer #14 to use for the nested loop – deleting it after it's complete.

In order to reuse the existing code, we'll trick the OS changing the id# of the initial buffer #14 right before the second call – *not deleting it but cloaking it in the I/O Memory area of the calculator*. The operand function re-labels the buffer with id#13 (using function **CLOAK**), then the nested call to FROOT/INTEG creates and uses a new buffer #14 to perform its task and deletes it upon completion – returning the execution to the "operand" function FOCAL routine. Before the execution is returned to the driver program, the cloaked buffer is re-issued as id#14 (using function **EXPOSE**) so things can be picked up exactly where there were left off before calling the nested subroutine.

If you must know, all **CLOAK** and **EXPOSE** do is changing the buffer id#' of the initial buffer created in the first call to FROOT/INTEG - first from 14 to 13, and then back to 14. Prior to all this a third function (**RESET**) is used to check for pre-existing buffers with id#13 – deleting it if found.


NO RECURSION
USER RAD

2D Driver Routines and Rules of Engagement.

The main programs for double integrals and system of 2 equations are **FITG2** and **FRT2**. Each one has an auxiliary routine associated with it, which acts as the first level operand function and issues a second nested call for the integrand or the second equation appropriately, as follows:

For **FITG2**, the function name f(x,y) is expected in ALPHA, and the four integral limits in the stack in the pattern "y1, y2, x1, x2" – (y1,y2) for the outer integral, and (x1,x2) for the inner one.

- *The integrand function is to be programmed assuming x is in R01, and y <u>in the stack</u>.*

For **FRT2**, both function names are expected to be in Alpha separated by comma (like "F1,F2"), and the guesses entered in the stack, with the pattern "x1, x2, y1, y2" - with (x1, x2) for f1(x,y) and (y1, y2) for f(2(x,y).

- *The second operand function f2(x,y) is executed first. It assumes x in R01 and y in the stack.*
- *The first operand function f1(x,y) assumes x in R01 and y in R02.*
- *You decide which one is F1 and F2 by their order in the ALPHA string*

All buffer management is made automatically by the auxiliary routines ***2D** and ***FG**.

Routine Listings.

Here are the routine listings for your perusal. Notably **FRT2** introduces more complexity to process the function names – entered as comma-separated strings in ALPHA – and due to the indirect call to f1(x,y) at the end of the auxiliary routine ***FG** - which is not required by ***2D** in the double integration case, as it's just one function involved. **CLAC** and **ASWAP** are borrowed from the ALPHA ROM – and need the Library#4 present in the calculator. They're only used for **FRT2**.

```
01   LBL "FRT2"
02   CLKEYS        no keys assigned
03   ASTO 00       save string
04   ASWAP         swap around ","
05   CLAC          remove second
06   ASTO 05       save in R05
07   CLA
08   ARCL 00       recall string
09   CLAC          remove second
10   ASTO 00       save in R00
11   STO 04        upper guess2
12   RDN
13   STO 03        lower guess2
14   RDN
15   RESET         reset buffers
16   "*FG"         first level operand
17   FROOT         call first round
18   GTO 00
19   *LBL 01       Not found
20   RESET
21   "NO ROOT"
22   AVIEW
23   *LBL 00       Found
24   RCL 02        y solution
25   X<>Y          arrange in stack
26   CLA           appends
27   ARCL 00       f1(x,y) name
28   "|-,"
29   ARCL 05
30   RTN           done(!)
31   *LBL "*FG"
32   STO 01        save x for later
33   CLOAK         mask buffer id#
34   RCL 03        lower guess 2
35   RCL 04        upper guess 2
35   CLA
36   ARCL 05       f2(x,y)
37   FROOT         nested call
38   GTO 00        Foundyo, skip
39   GTO 01        Not found!
40   *LBL 00
41   EXPOSE        re-issue buf id#
42   STO 02        Save yo result
```

```
01   *LBL " FITG2"
02   CLKEYS        no keys assigned
03   ASTO 00       save in R00
04   STO 03        upper limit2
05   RDN
06   STO 02        lower limit2
07   RDN
08   RESET         reset buffers
09   "2D"          first level operand
10   FINTG         call first round
11   RTN           done
12   "NO SOL"
13   AVIEW
14   RESET
15   RTN           done.
16   *LBL "*2D"
17   STO 01        Save x for later
18   CLOAK         mask buffer id#
19   RCL 02        lower limit2
20   RCL 03        upper limit2
21   CLA
22   ARCL 00       f(x,y)
23   FINTG         nested call
24   EXPOSE        re-issue buf id#
25   END           ready
```

```
43    XEQ IND 00    calculates f1(x,Yo)
44    END
```

FITG2 uses registers {R00-R03} and leaves the results in X and R01. The function name is left in ALPHA (6-chars max).

FRT2 uses registers {R00-R05} and leaves the results in the stack registers {X, Y} and {R01, R02} for the 2-equation roots. The comma-separated function names string is left in ALPHA (6-chars max for each name).

Comments.

The new functions to support the nested configuration are simplified versions of some general-purpose buffer utilities, available in other extension modules as follows:

- **RESET** is equivalent to the sequence { 13, **B?**, **CLB,** RDN }
- **CLOAK** is equivalent to the sequence { 14.013 , **REIDBF**, RDN}
- **EXPOSE** is equivalent to the sequence: { 13.014 , **REIDBF ,** RDN }

B? and **CLB** are available in the OS/X ROM, and **REIDBF** in the RAMPAGE ROM.

Using the simplified versions is more intuitive for math-oriented users, and besides it freed up some space for additional examples in the SIROM.

While you can use **RESET** at any time (which will delete buff #13 if present, or do nothing if not present), using **CLOAK** and **EXPOSE** will generally result in the error message "BUF ERR". They're meant to be used only while buffer #14 exists, which is tightly controlled by the code in FINTG and FROOT – and furthermore, the SIROM uses the I/O_PAUSE interrupt as a "search & destroy" for buffer#14 at all times. Refer to the corresponding section in the **SandMath** manual to read more on this subject.

Caveat emptor:

- There's a price to pay for this buffer trickery, and that's the loss of the USER key assignments. As you can see in the listings above, the main routines call **CLKEYS** to make the operation more reliable (this avoids spurious buffer errors due to memory overwrites). You can save them in an X-Mem file using **SAVEKA** and then recover them with **GETKA** after the fact (both functions are also available in the AMC_OS/X ROM).

- These routines are not fast, their interest is in the methodology - not optimized for speed to say the least. If you need faster responses, then the SandMath provides dedicated MCODE functions for many of these and yet some more.

- Bear in mind that the INTEG-based method to define special functions is not an efficient one from the mathematical standpoint, but it is a godsend for engineering problems. Also FROOT is not perfect or fool-proof either, so choosing a good initial guess is of high importance. If FRT2 fails to find a root (in either variable), it'll present the error message "NO ROOT" – Change the limits and try again.

The following examples should provide a good overview into the details of the programming.

Example 1. Calculate the integral of the Bessel Jn function, ITJ(1,3) = **INT** (0,3) { J(1,t).dt} using the integral definition as reference:

$$J_n(x) = \frac{1}{\pi}\int_0^{\pi} \cos(n\tau - x\sin\tau)\,d\tau.$$

Program Code is below. Note that you don't need to worry about the buffer management, that's done automatically by the driver routines all transparently to the user.

01	**LBL "ITJB"**		13	**LBL " *JN"**	*inner variable t in stack*
02	**X<>Y**	*order n to X*	14	**RAD**	*angular mode*
03	**STO 04**	*order saved in R04*	15	**RCL 04**	*get order*
04	**CLX**	*lower outer limit*	16	*****	*n.t*
05	**X<>Y**	*upper outer limit*	17	**X<>Y**	*inner variable t*
06	**0**	*lower inner limit*	18	**SIN**	*sin t*
07	**PI**	*upper inner limit*	19	**RCL 01**	*outer variable*
08	*"*JN"*	*function name*	20	*****	*x.sin t*
09	**XROM " ITG2"**	*double integration*	21	**-**	*n.t - x.sin t*
10	**PI**	*adjust factor*	22	**COS**	*cos (n.t - x.sin t)*
11	**/**	*final result*	23	**END**	*integrand complete.*
12	**RTN**	done.			

As mentioned before, *speed is not this method's forte*. Even on V41 in turbo mode it'll take a good 75 seconds to return 1.260052 (in FIX 6). This was not the goal of the example, but to clarify the general guidelines and showcase the conceptual approach. If you want a fast result you're encouraged to use **JBS** in the SandMath, or even better the **ITJ** (sub)function also in the SandMath, which uses the Generalized, Regularized Hypergeometric function for the calculation – a world of differences…

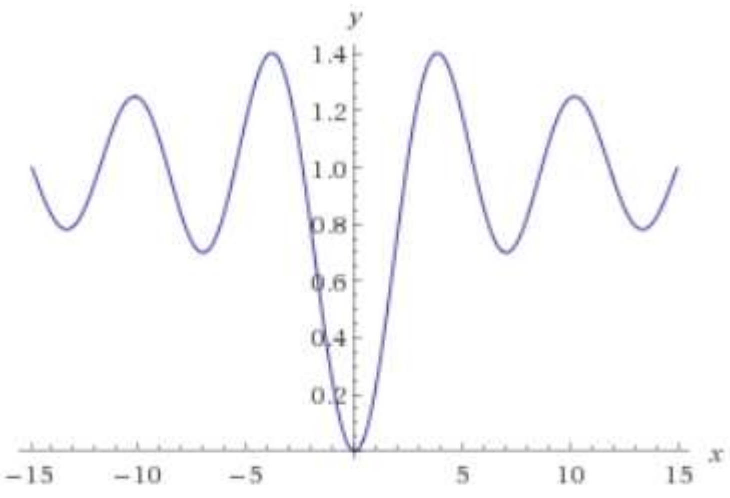
Comment. This particular example is of course much better dealt with using the well-known expression between the Bessel function J1 and J0 shown below (proving once again that it's always good to check your math before embarking in long and winding paths):

$$\int_0^x J_1(t)\,dt = 1 - J_0(x)$$

thus:

$$\int_0^3 J_1(t)\,dt = 1 - J_0(3) \approx 1.26005$$

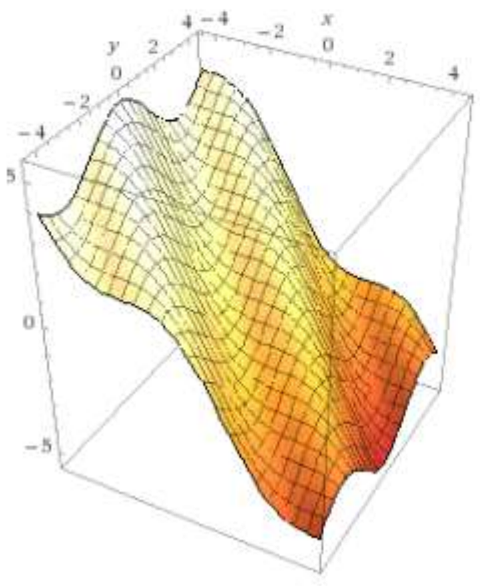
Here's an interesting plot showing the integral function of J1(x) between]-15 . 15[

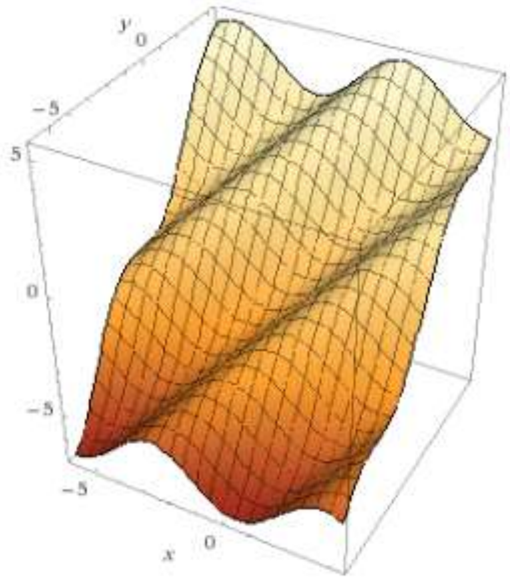


Example 2. Calculate the solution for the system of non-linear equations below:

$$\begin{cases} f1(x,y) = x - \sin(x + y) \\ f2(x,y) = y - \cos(x - y) \end{cases}$$

Solution: x = 0,935082064
y = 0,998020058





The equations are programmed as shown below. Note how the convention is observed, with the y value assumed in the stack for the second function and in R02 for the first one; whilst x is always assumed in R01 for both functions. The solutions are obtained in about 3 seconds (FIX 9) using V41 in Turbo mode.

ALPHA, "FG1,FG2" , ALPHA, ENTER^, 2, ENTER^, 1, ENTER^, 2, **CF 01,** XEQ "FRT2"

01	**LBL "FG1"**	*2 sets combined*
02	RCL 01	x
03	FS? 01	
04	GTO 01	
05	RAD	*example #3*
06	RCL 02	y
07	+	x+y
08	SIN	sin(x+y)
09	RCL 01	x
10	-	-x+sin(x+y)
11	RTN	
12	LBL 01	*example #2*
13	X^2	x^2
14	RCL 02	y
15	X^2	y^2
16	+	x^2+y^2
17	5	
18	–	x^2+y^2-5
19	RTN	

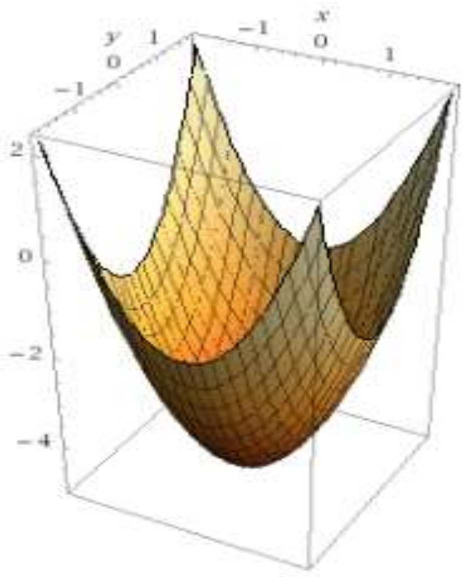
20	**LBL "FG2"**	*2 sets combined*
21	FS? 01	
22	GTO 01	
23	RAD	*example #3*
24	CHS	-y
25	RCL 01	x
26	+	
27	COS	cos(x-y)
28	X<>Y	y
29	–	-y+cos(x-y)
30	RTN	
31	LBL 01	*example #2*
32	X^2	y^2
33	CHS	-y^2
34	RCL 01	x
35	X^2	
36	+	x^2 – y^2
37	3	
38	–	
39	END	

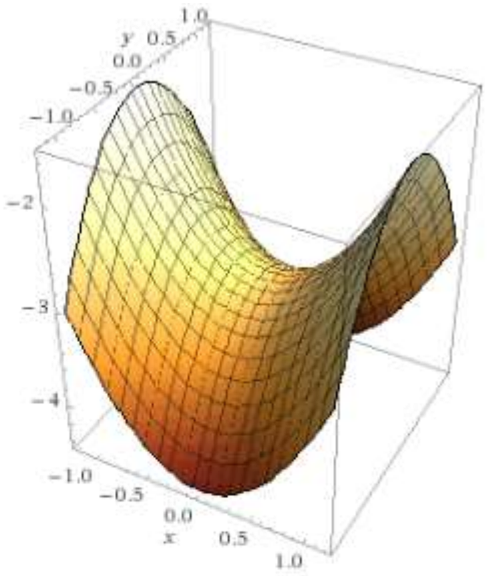
<u>Note:</u> *to save main FAT entries the example #2 and #3 function sets included in the ROMS have been combined to use a single global label for both sets, aided by the user flag F1 to determine which set would be computed. CF 01 – uses F1(x,y) and F2(x,y) ; SF 01 – uses G1(x,y) and G2(x,y)*

Obviously this approach won't be needed with your own examples, which will likely have one global label per set of two functions – i.e. **not combined** with more sets.

Example 3. Obtain the roots for the system of two equations below (available as "FG1" and "FG2" with F1 clear)

$$\begin{cases} g1(x,y) = x^2 + y^2 - 5 \\ g2(x,y) = x^2 - y^2 - 3 \end{cases} \qquad \text{Solution:} \qquad \begin{matrix} x = 2 \\ y = 1 \end{matrix}$$





This is an interesting case because FRT2 not only is much slower (as we knew it was going to be), but also fails to find a root using initial guesses equal to the solutions, i.e. x0 = 2, y0=1.

<u>Other Examples.</u>

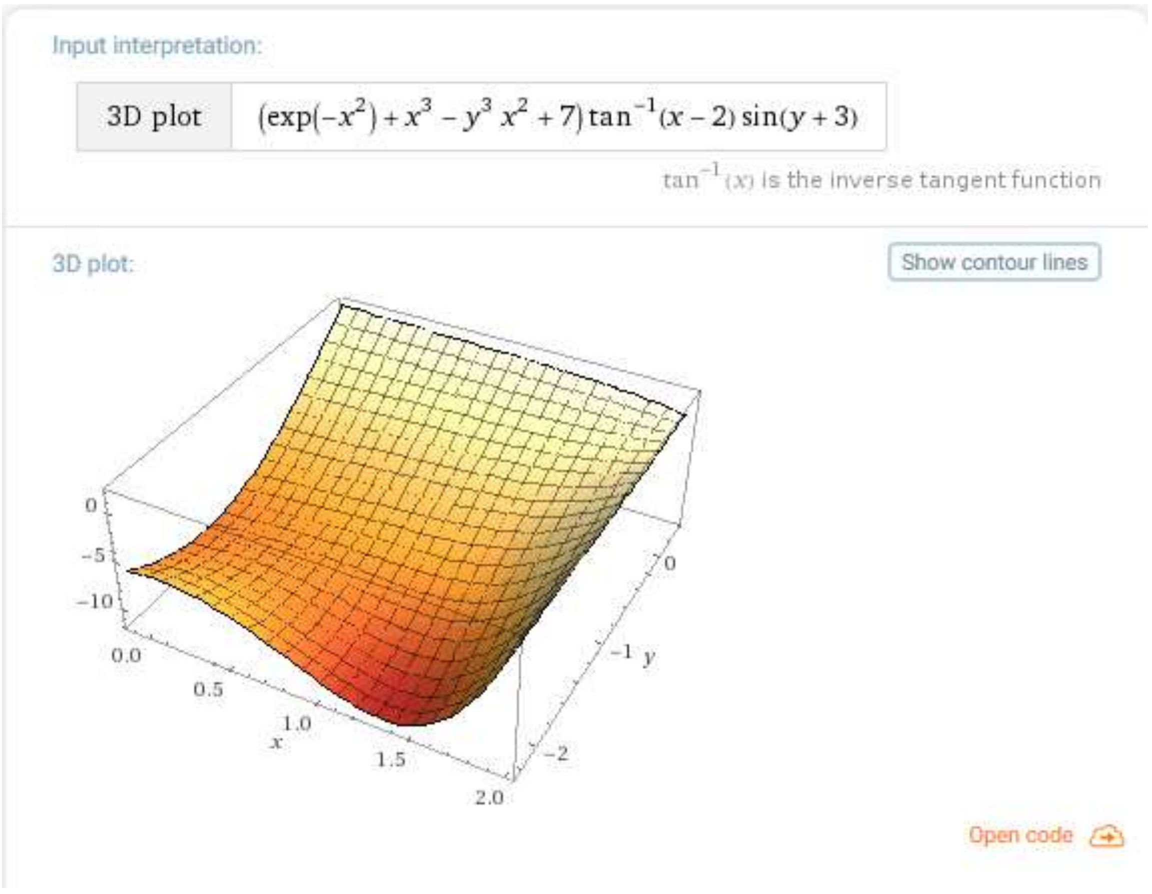
Let's use Valentín Albillo's neat examples from DataFile for Double Integrals - as follows:

$$I = \int_0^1 \int_1^2 (x^2 + y^2)\ .dy.dx \quad ; \quad I = \int_3^4 \int_1^2 1/(x + y)^2\ .dy.dx$$

$$I=\int_{-2.3}^{1.6} \int_{3.9}^{6.1} (e^{-x*x} + x^3 - y^3 * x^2 + 7) * \tan^{-1}(x-2) * \sin(y+3)\ .dy.dx$$

See the original article for details, available at: http://web.archive.org/web/20110906135412/http://membres.multimania.fr/albillo/calc/pdf/DatafileVA024.pdf

The results are: I1 = 8/3 = 2.6666666
I2 = Ln(25/24) = 0.040821
I3 = 1,321.275779

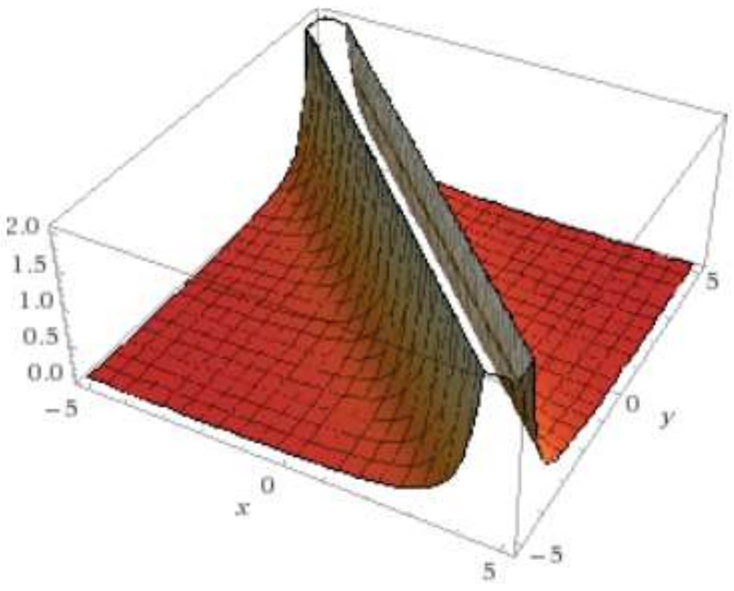

Input interpretation:

3D plot	$(\exp(-x^2)+x^3-y^3x^2+7)\tan^{-1}(x-2)\sin(y+3)$

$\tan^{-1}(x)$ is the inverse tangent function

3D plot:

3D plot	$\dfrac{1}{(x+y)^2}$



Appendix: MCODE listing for dedicated functions

BUFERR	AEC0	20D	?NC XQ	Build Msg - UF25 Clear
	AEC1	0FC	->3F83	[APERMSG]
	AEC2	002	"B"	
	AEC3	015	"U"	
	AEC4	006	"F"	"NO BUF" or
	AEC5	020	" "	"DUP BUF"
	AEC6	005	"E"	
	AEC7	012	"R"	
	AEC8	212	"R"	
	AEC9	1F1	?NC GO	LeftJ, Show and Halt
	AECA	0FE	->3F7C	[APEREX]
Header	AECB	08B	"K"	
Header	AECC	001	"A"	
Header	AECD	00F	"O"	
Header	AECE	00C	"L"	
Header	AECF	003	"C"	Ángel Martin
CLOAK	AED0	388	SETF 0	
	AED1	130	LDI S&X	
	AED2	00E	CON: 14	buffer id# = "E"
	AED3	053	JNC +10d	[MERGE]
Header	AED4	085	"E"	
Header	AED5	013	"S"	
Header	AED6	00F	"O"	
Header	AED7	010	"P"	
Header	AED8	018	"X"	
Header	AED9	005	"E"	Ángel Martin
EXPOSE	AEDA	384	CLRF 0	
	AEDB	130	LDI S&X	
	AEDC	00D	CON: 13	buffer id# = "D"
MERGE	AEDD	106	A=C S&X	
	AEDE	000	NOP	
	AEDF	2A5	?NC XQ	
	AEE0	10C	->43A9	[CHKBFA]
NOBUF2	AEE1	2FB	JNC -33d	[NOBUF]
BFOUND2	AEE2	2DC	PT= 13	
	AEE3	38C	?FSET 0	cloaking?
	AEE4	027	JC + 04	yes, skip
MAKE14	AEE5	390	LD@PT- E	change id# to "EE"
	AEE6	390	LD@PT- E	
	AEE7	01B	JNC +03	
MAKE13	AEE8	350	LD@PT- D	change id# to "DD"
	AEE9	350	LD@PT- D	
	AEEA	2F0	WRTDATA	
	AEEB	3C1	?NC GO	Normal Function Return
	AEEC	002	->00F0	[NFRPU]

Header	AEAE	094	"T"	
Header	AEAF	005	"E"	Deletes Buffer 13
Header	AEB0	013	"S"	
Header	AEB1	005	"E"	
Header	AEB2	012	"R"	Ángel Martin
RESET	AEB3	130	LDI S&X	
	AEB4	00D	CON: 13	buffer id# = "D"
	AEB5	106	A=C S&X	
	AEB6	2A5	?NC XQ	
	AEB7	10C	->43A9	[CHKBFA]
NOTFUND	AEB8	3E0	RTN	
BFOUND	AEB9	05E	C=0 MS	delete only the first nybble
	AEBA	2F0	WRTDATA	so the OS will do the rest!
	AEBB	04E	C=0 ALL	
	AEBC	270	RAMSLCT	
	AEBD	051	?NC GO	Pack IO/KA area
	AEBE	086	->2114	[PKIOAS]

CHKBFA	43A9	0A6	A<>C S&X	recall id# to C(0)
CHKBF4	43AA	23C	RCR 2	id# to C(12)
	43AB	35C	PT= 12	
	43AC	130	LDI S&X	
	43AD	0BF	CON: 191	Fisrt possible reg -1
	43AE	10E	A=C ALL	store id# & addr in A
CB10	43AF	166	A=A+1 S&X	Increase reg# address
CB20	43B0	046	C=0 S&X	
	43B1	270	RAMSLCT	Select Chip 0
	43B2	378	READ 13(c)	.END.
	43B3	306	?A<C S&X	did we reach the .END. Chainhead?
	43B4	3A0	?NC RTN	yes -> Not Found
	43B5	0A6	A<>C S&X	addr to C[S&X]
	43B6	270	RAMSLCT	Candidate address for header
	43B7	0A6	A<>C S&X	id# to A(12) & addr to A[S&X]
	43B8	038	READATA	Candidate Value for header
	43B9	2EE	?C#0 ALL	Carry if not empty register
	43BA	3A0	?NC RTN	**empty reg -> Not Found**
	43BB	23E	C=C+1 MS	Carry if id#="F" (KAR)
	43BC	39F	JC -13d	Key Assignment Register
	43BD	362	?A#C @PT	is this IO Buffer?
	43BE	037	JC +06	NO , keep searching
	43BF	1B0	POPADR	YES !
	43C0	23A	C=C+1 M	Return to (P+2)
	43C1	170	PUSHADR	
	43C2	038	READATA	Return with Header in C
	43C3	3E0	RTN	**and BuffAdr in A - rg# selected**
CB30	43C4	0FC	RCR 10	Skip Buffer
	43C5	056	C=0 XS	
	43C6	146	A=A+C S&X	add buffer size
	43C7	34B	JNC -23d	[CB20]

Binet Formulas

Function	Description	Input	Output
BINETN	Binet formula for integers	n in X	f(n)
BINETX	Binet formula for real values	x in X	f(x)
MLN	Multinomial Coefficient	n in Y, k in X	C(n,k)

- **BINETN** implements the well-known Binet formula for integer input values. The result is the n-th Fibonacci number obtained directly without any iterations.

  $$F_n = \frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}}$$

  Example: Calculate f(9)

  9, XEQ "BINETN" => 34.00000000

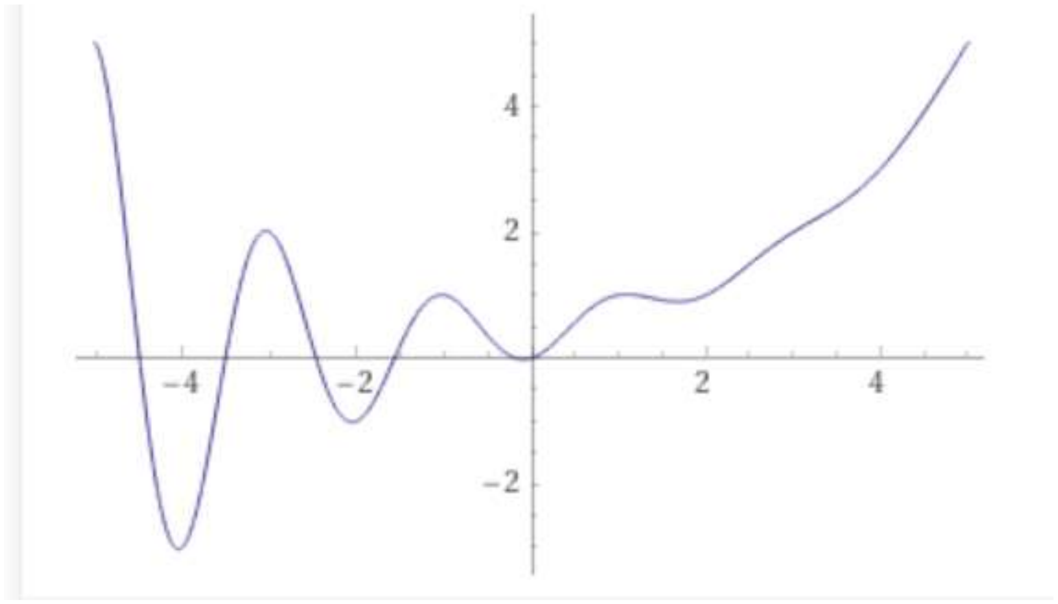
- **BINETX** implements an extension for non-integer real input values to calculate the interpolated Fibonacci numbers. This provides an easy expression for the determination that guarantees real values also for the interpolated Fibonacci numbers:

  $$f_x^* = \frac{\varphi^x - \cos(\pi x)\varphi^{-x}}{\sqrt{5}}$$

  Example: Calculate f(π)

  PI, XEQ "BINETX" => 0.043896342

  See below the graphical representation of Binet(x) for arguments between [-5 . 5]



Obviously, the values for integer arguments coincide with the natural Fibonacci number, since the term cos(πn) is equal to +/- one.

In fact, this modified formula produces the real parts of the complex results obtained applying Binet's formula directly with complex arguments – where the term -φ^-n clearly yields a result in the complex domain: (-φ)^(-n) = exp(-n . ln (-φ))

Note: You can refer to the 41Z Module manual for the complex case, implemented in that module with the function **ZFIB**.

Multinomial Coefficients. { **MLN** } (See JM Baillard's *reference page*.)

Multinomial coefficients are an extension of the Binomial coefficient, using multiple indexes instead of two. For example, if "k" is the number of variables we have:

P = (n1 , n2 , ... nk) ! = n ! / (n1! n2! .... nk!) ; where n = n1 + n2 + ... + nk

$$\binom{n}{k_1, k_2, \ldots, k_{r-1}} = \frac{n!}{k_1!k_2!\ldots k_{r-1}!k_r!}$$

The function **MLN** expects the input values stored in data registers starting in R01, The number of variables "k" is entered in the stack' X-register.

Example: Calculate (76 , 107 , 112 , 184) !

```
16  STO 01    24  STO 02    41  STO 03    48  STO 04
4   XEQ "MLN"            =>        P = 9.227558919  E69
```

Bell and Bernoulli Numbers

Function	Description	Input	Output
BELL	Bell Numbers	Index n in X	n-th. Bell number
BN2	Bernoulli Numbers	Index n in X	n-th. Bernoulli number

Bell Numbers. {**BELL**} (See JM Baillard's *reference page*)

In combinatorial mathematics, the Bell numbers count the possible partitions of a set, i.e. the Bell number Bn counts the number of different ways to partition a set that has exactly n elements.

Bell numbers are defined by the iterative sequence below:

B(0) = 1 and
B(n+1) = Σ{k=0..n} Cn,k B(k) if n > 1

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k.$$

where Cnk = n!/ [k!(n-k)!] are the binomial coefficients.

Examples:

10, XEQ "BELL" => 115,975.0000
89, XEQ "BELL" =>5.225728472 E99

Bernoulli Numbers{ **BN2** } (see JM Baillard *reference page*)

The Bernoulli numbers could be computed by the relations:

B(0) = 1 ;
B(0) + Cn+1,1 B(1) + Cn+1,2 B(2) + ... + Cn+1,n B(n) = 0

where Cnk = n!/ [k!(n-k)!] are the binomial coefficients

If the convention B1=−1/2 is used, this sequence is also known as the first Bernoulli numbers; with the convention B1=+1/2 is known as the second Bernoulli numbers. Except for this one difference, the first and second Bernoulli numbers agree. Since Bn=0 for all odd n>1, and many formulas only involveeven-index Bernoulli numbers, some authors write Bn instead of B2n.

Example:

10, XEQ "BN2" =>B(10) = -0.075757576

Note however that this recurrence relation is unstable, and the results are quite inaccurate for large n. The generating function below is often used to avoid that:

$$\frac{t}{e^t - 1} = \frac{t}{2}\left(\coth\frac{t}{2} - 1\right) = \sum_{m=0}^{\infty} \frac{B_m^- t^m}{m!}$$

Fibonacci Numbers

Function	Description	Input	Output
FIB	Fibonacci Numbers	Index n in X	n-th. Fibonacci number
FIBI	Inverse Fibonacci	Index n in X	n-th/ inverse Fibonacci
ΣFIB	Sum of Fibonacci	Range n in X	Sum[fib(n)]
ΣIFIB	Sum of Inverse Fibonacci	Range n in X	Sum[1/fib(n)]

Fibonacci Numbers { **FIB** , **FIBI** }

These functions calculate the Fibonacci andthe Fibonacci Inversenumbers using the well-known recurrent relationship:

f(0) = 0 ,
f(1) = 1;
f(n) = f(n-2) +f(n-1)

And the "Fibonacci Inverse" defined as

f'(0) = 0
f'(1) = 1
f'(n) = 1/f('n-2) + 1/f'(n-1).

Note that this is **not* the same as the inverse of Fibonacci*, which would simply be 1/F(n)

Examples:

10, XEQ "FIB" =>55.00000000 ; LASTX, XEQ : FIBI" =>0.683299104
25, XEQ "FIB" => 75,025.00000 ; LASTX, XEQ "FIBI" =>0.707165965

Sum of Fibonacci numbers { **ΣFIB**, **ΣFIBI**}

Here we're calculating the sum of the first n Fibonacci numbers, starting at f(0)=0 until f(n).

An interesting fact is the sum of the first Fibonacci numbers with odd index up to f(2n–1) is the 2n-th. Fibonacci number, and the sum of the first Fibonacci numbers with even index up to f(2n) is the (2n+1)-th. Fibonacci number minus 1:

Moreover, the general expression below relates the sum to the sequence value:

Σ{0..n)F(n) = f(n+2)-1

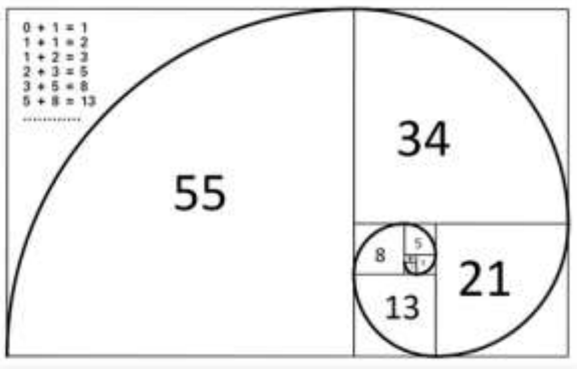
Example:

15, XEQ "ΣFIB" => 1,596.000000

Verifying the formula above:

17, XEQ "FIB" => 1,597.000000

Example:
15, XEQ "ΣFIBI" =>3.357233149



Collatz conjecture. { $\boxed{\textbf{ULAM}}$ }

(see: https://en.wikipedia.org/wiki/Collatz_conjecture)

ULAM shows the successive values in the Collatz conjecture, starting with the argument in X.
It is completely off-topic subject but it sorts of happened while preparing this manual – what an excuse, uh?

The **ULAM** function does a complete path starting with the value in X, all the way until the end when "1" is reached using the well-known Ulam's (or Collatz's) algorithm:

- If odd, multiply by three and add one
- If even, divide by two

$$f(n) = \begin{cases} \frac{n}{2} & \text{if } n \equiv 0 \pmod 2 \\ 3n+1 & \text{if } n \equiv 1 \pmod 2. \end{cases}$$

The function will take the integer part of the absolute value of the number in X. Then all intermediate values are briefly shown, and the total number of "nodes" is left in X upon completion. The starting number is left in X.

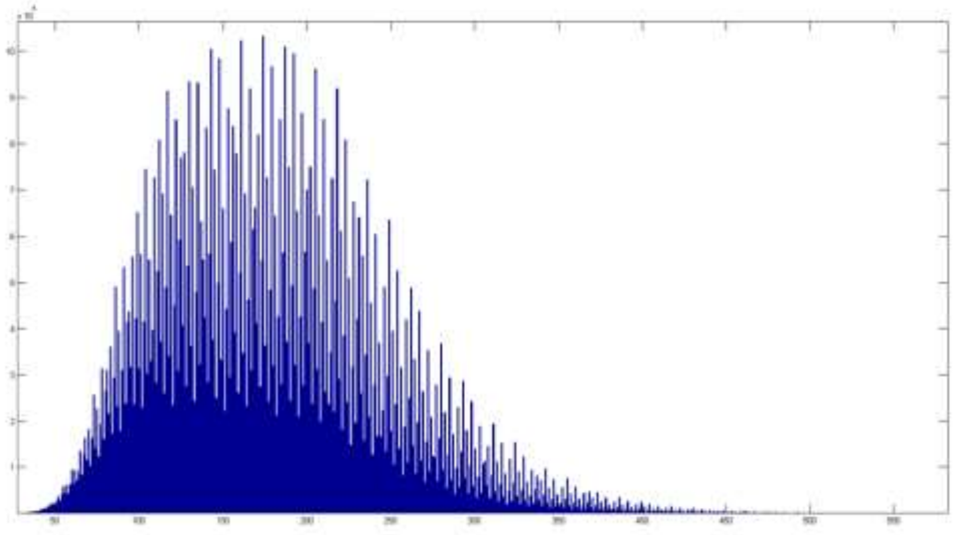
Examples:

41, XEQ "ULAM" -> generates a sequence of 109 numbers

22, ULAM -> generates a sequence of 15 numbers

The sequence for n = 27, listed below, takes 111 steps (41 steps through odd numbers), climbing as high as 9232 before descending to 1.

27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1 (sequence A008884 in the OEIS)



Histogram of total stopping times for the numbers 1 to 10^8. Total stopping time is on the x axis, frequency on the y axis.

MCODE listing

Header	AEBF	*08D*	***"M"***	
Header	AEC0	*001*	***"A"***	***Collatz Conjecture***
Header	AEC1	*00C*	***"L"***	
Header	AEC2	*015*	***"U"***	*Ángel Martin*
ULAM	**AEC3**	**0F8**	**READ 3(X)**	
	AEC4	128	WRIT 4(L)	
	AEC5	*149*	*?NC XQ*	*Integer & Positive*
	AEC6	*134*	*->4D52*	*[CHKZI]*
	AEC7	268	WRIT 9(Q)	
	AEC8	04E	C=0 ALL	
	AEC9	0E8	WRIT 3(X)	*reset the counter*
LOOP1	AECA	00E	A=0 ALL	
	AECB	35C	PT= 12	*Builds "1" in A*
	AECC	162	A=A+1 @PT	
	AECD	278	READ 9(Q)	
	AECE	36E	?A#C ALL	*end of the path?*
	AECF	**3A0**	**?NC RTN**	*yes, end here.*
	AED0	0F8	READ 3(X)	
	AED1	2A0	SETDEC	
	AED2	*01D*	*?NC XQ*	*increase counter*
	AED3	*060*	*->1807*	*[AD2_10]*
	AED4	0E8	WRIT 3(X)	*update value*
	AED5	278	READ 9(Q)	*get current n*
	AED6	*3CD*	*?NC XQ*	*C= MOD[int(C),2]*
	AED7	*100*	*->40F3*	*[MOD2]*
	AED8	2EE	?C#0 ALL	*it is odd?*
	AED9	***02F***	***JC +05***	*yes, skip*
	AEDA	278	READ 9(Q)	
EVEN	AEDB	*3CD*	*?NC XQ*	*{A,B} = {C} /2*
	AEDC	*13C*	*->4FF3*	*[DIVTWO]*
	AEDD	***053***	***JNC +10d***	*show result*
ODD	AEDE	04E	C=0 ALL	
	AEDF	35C	PT= 12	
	AEE0	0D0	LD@PT- 3	
	AEE1	10E	A=C ALL	
	AEE2	278	READ 9(Q)	
	AEE3	*135*	*?NC XQ*	*3*n*
	AEE4	*060*	*->184D*	*[MP2_10]*
	AEE5	*001*	*?NC XQ*	*3*n+1*
	AEE6	*060*	*->1800*	*[ADDONE]*
MERGE	AEE7	268	WRIT 9(Q)	
	AEE8	*099*	*?NC XQ*	*Sends C to display - sets HEX*
	AEE9	*02C*	*->0B26*	*[DSPCRG]*
	AEEA	*1FD*	*?NC XQ*	*wait a little - CL compatible*
	AEEB	*12C*	*->4B7F*	*[WAIT4L] - Enables RAM*
	AEEC	*1FD*	*?NC XQ*	*wait a little - CL compatible*
	AEED	*12C*	*->4B7F*	*[WAIT4L] - Enables RAM*
	AEEE	*2E3*	*JNC -36d*	*[LOOP1]*

The calls to [WAIT4L] ensure compatibility with the SY-41CL – slowing down the output for the user to catch a glimpse of the enumerated values.