#	Name	Description	Instructions	Author	Source
01	**-SOLINTG 2A**	*Section Header*	*n/a*	*n/a*	*n/a*
02	**FINTG**	HP's INTEG	Fnc in Alpha, a^b in Stack	*HP Co.*	Advantage Pac
03	**FROOT**	HP's SOLVE	Fnc in Alpha, a^b in Stack	*HP Co.*	Advantage Pac
04	***"IG"***	PPC's IG	FNC in R10, a^b in stack	*R. Predmore & J. Kennedy*	PC ROM
05	***"ITG"***	Alternate Integrate	FNC in R10, a^b in stack	*Unknown?*	PPCCJ ??
06	***"ITGD"***	Alternate Double Integral	Prompts for data	*Ernest Gibbs*	PPCCJ V8N4p31
07	**SILOOP**	Solve/Integ Loop	n/a	*HP Co.*	Advantage Pac
08	***"SIRTN"***	Solve/Integ RTN	n/a	*HP Co.*	Advantage Pac
09	***"SV"***	PPC's Solve	FNC in R06, a^b in stack	*J. Kennedy & G. Dennes*	PPC ROM
10	***"SYS2"***	Non-linear 2-eq system	Prompts for data	*F-J Pamies Dura*	UPLE #35006
11	***"2DITG"***	2D Integration	Step in R10	*Valentin Albillo*	DataFile
12	**-APPLIED**	*Section Header*	*n/a*	*n/a*	*n/a*
13	***"CI"***	Cosine Integral	argument in X	*Angel Martin*	This project
14	***"ERF"***	Error Function	argument in X	*Angel Martin*	This project
15	***"JYX"***	Bessel J (integer order)	n^x in stack, F00 clear(!)	*Angel Martin*	This project
16	***"SI "***	Sine Integral	argument in X	*Angel Martin*	This project
17	***"ZJYX"***	Zeros of Bessel J	Prompts for order	*Angel Martin*	This project
18	***"F1XY"***	Example 1 for 2DITG	n/a	*Valentin Albillo*	DataFile
19	***"F2XY"***	Example 2 for 2DITG	n/a	*Valentin Albillo*	DataFile
20	***"F3XY"***	Example 2 for 2DITG	n/a	*Valentin Albillo*	DataFile
21	***"*C"***	Integrand Fnc.	n/a	*Angel Martin*	This project
22	***"*E"***	Integrand Fnc.	n/a	*Angel Martin*	This project
23	***"*JN"***	Integrand Fnc.	n/a	*Angel Martin*	This project
24	***"*S"***	Integrand Fnc.	n/a	*Angel Martin*	This project
25	**?FR**	Free registers finder	returns # available regs	*Ken Emery*	MCODE for beginners

Comments:

1. Display settings are used to set accuracy and largely impact execution times. See original docs.
2. All stack is used. Result uncertainty is returned in Y for FINTG. Previous estimation for FROOT.
3. Integrand/solved function must be programmed using global a global label, even if it's just a native function
4. SILOOP and SIRTN are auxiliary functions needed for the design to work. No user intervention.
5. Clear Flag 00 if you use JYX independently from ZJYX. Use with integer orders only.
6. ITGD will print two integral symbols as special chrs on thermal printers (and emulations)
7. Nested FINTG/FROOT require two functions & global labels - don't use flags to condition execution.
8. CX OS is required only for error messages ("NO ROOM" , and "RECURSION") and for "?FR"
9. FROOT requires 13 free registers, FINTG requires 32 free regs. (not DATA registers but in I/O area)
10. Use "**?FR**" to check the available free registers in the calculator (requires CX OS)
11. Buffer #14 is used - dynamically allocated at the beginning and removed upon the end of execution.
12. Perfectly compatible with Key Assignments, Alarms, and any other buffers present
13. This QRG is no substitute for the original documentation - which is both thorough and a pleasure to read.

Example 1: error function

erf(x) = integ[exp(-t^2) dt]{0,x)

1	LBL "ERF"
2	0
3	X<>Y
4	"*E"
5	FINTG
6	ST+ X
7	PI
8	SQRT
9	/
10	RTN
11	LBL "*E"
12	X^2
13	CHS
14	E^X
15	END

Example 4: solve for x:

intg [sin^2 t dt]{0,x} = 1/2

01	LBL "FX"
02	RAD
03	0
04	X<>Y
05	"*F"
06	FINTG
07	,5
08	-
09	RTN
10	LBL "*F"
11	SIN
12	X^2
13	END

Solution: 1, ENTER^, 2
FROOT
x= 1,277097976

Example 3: sine integral

Si(x) = integ[sint/t dt]{0,x}

1	LBL "SI"
2	0
3	X<>Y
4	"*S"
5	FINTG
6	RTN
7	LBL "*S"
8	SIN
9	LASTX
10	X#0?
11	/
12	END

Example 2: cosine integral

Ci(x) = g+Ln(x)+integ[[cos(t)-1]/t dt]{0,x}

1	LBL "CI"
2	STO 00
3	0
4	X<>Y
5	"*C"
6	FINTG
7	RCL 00
8	LN
9	+
10	GEU
11	+
12	END
13	LBL "*C"
14	RAD
15	COS
16	E
17	-
18	X<>Y
19	X#0?
20	/
21	RTN

Example 5: Bessel Integral

J(n,x) = integ[J(n,t) dt] {0,x}

1	LBL "ITJB"
2	X<>Y
3	STO 00
4	X<>Y
5	0
6	X<>Y
7	"*J"
8	FINTG
9	RTN
10	LBL "ZJB"
11	"ORDER=?
12	PROMPT
13	STO 00
14	0
15	LBL 00
16	E
17	+
18	ENTER^
19	ENTER^
20	3
21	+
22	"*J"
23	FROOT
24	STOP
25	GTO 00
26	LBL "*J"
27	RCL 00
28	X<>Y
29	JBS (*)
30	END

(*) Requires SandMath

FINTG cannot be used within FITNG

Example 6: Bessel Zeros

1	LBL "ZJYX"	
2	"N=?"	
3	PROMPT	
4	STO 00	
4	0	
4	LBL 00	
4	E	
8	+	
9	ENTER^	
10	ENTER^	
11	3	
12	+	
13	"JYX"	solved
14	FROOT	
15	STOP	
16	GTO 00	
17	RTN	
18	LBL "JYX"	
19	STO 01	
20	"*JN"	Integrand
21	0	
22	PI	
23	FINTG	
24	PI	
25	/	
25	RTN	
25	LBL "*JN"	
25	RAD	
29	RCL 00	
30	*	
31	X<>y	
32	SIN	
33	RCL 01	
34	*	
35	-	
36	COS	
37	END	

Example 7: Fourier series Nth. Coeficient

For an explicit Funtion "F(x)"

1	LBL "FOURN"	
2	"F. NAME? "	Function Name
3	AON	
4	PROMPT	
5	AOFF	
6	ASTO 01	
7	"PERIOD=?"	T
8	PROMPT	
9	STO 00	
10	FIX 4	
11	LBL E	
12	"INDEX=?"	n
13	PROMPT	
14	STO 02	
15	CF 00	
16	XEQ 00	Calculates an
17	SF 00	
18	LBL 00	Calculates bn
19	"*FN"	Integrand function
20	0	Lower limit
21	RCL 00	upper limit
22	FINTG	
23	RCL 00	period
24	/	divided by
25	ST+ X	twice
26	"a("	
27	FS? 00	
28	"b("	
29	RCL 02	
30	AINT	coef. To Alpha
31	"|-)="	
32	ARCL Y	
33	PROMPT	show result
34	FC? 00	
35	RTN	
36	GTO E	other coeff.?
36	LBL "*FN"	
36	RAD	
36	STO 03	x
40	XEQ IND 01	f(x)
41	RCL 02	T
42	RCL 03	n
43	*	nx
44	RCL 00	T
45	/	nx/T
46	ST+ X	2nx/T
47	PI	π
48	*	2 π nx/T
49	FC? 00	
50	COS	cos[2pnx/T]
51	FS? 00	
52	SIN	sin[2pnx/T]
53	*	
54	END	f(x)*cos / sin

$$\mathrm{Si}(x) = \int_0^x \frac{\sin t}{t}\,dt \qquad \mathrm{Ci}(x) = \gamma + \ln x + \int_0^x \frac{\cos t - 1}{t}\,dt \qquad \mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,dt.$$

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

$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x)\cos(nx)\,dx, \quad n \geq 0 \qquad b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x)\sin(nx)\,dx, \quad n \geq 1$$

The above formulas are valid for periodic functions in (- π , π) interval

The program is valid for any generic period, but for the case around x=0

Matrix Functions (Continued)

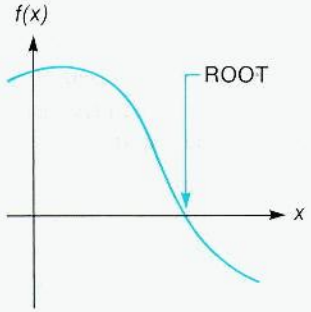
Function Name	Description
SUMAB (p. 49)	Returns sum of absolute values of all elements.
TRNPS (p. 45)	Transposes the matrix in place.
YC+C (p. 49)	Multiplies each element in column *k* by *y*-value and adds product to element in column *l*, replacing the latter.

FINDING THE ROOTS OF AN EQUATION

The SOLVE program finds the roots of an equation of the form

$$f(x) = 0,$$

where x represents a *real root*.*

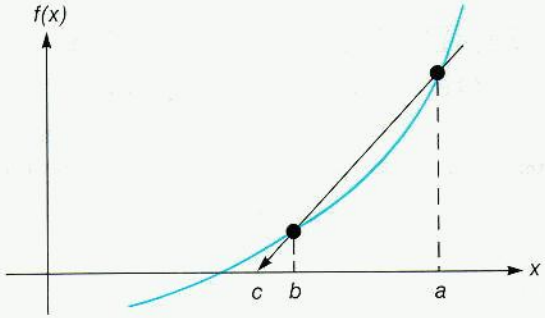


Executing the SOLVE program (SOLVE) employs an advanced numerical technique to find the real roots of a wide range of equations. You supply the equation for the function (in a program) and two initial estimates, and SOLVE does the rest.

Method

SOLVE normally uses the secant method to iteratively find and test x-values as potential roots. It takes the program several seconds to several minutes to do this and produce a result.

* Note that any equation with one variable can be expressed in this form. For example, $f(x) = a$ is equivalent to $f(x) - a = 0$, and $f(x) = g(x)$ is equivalent to $f(x) - g(x) = 0$.



If c isn't a root, but $f(c)$ is closer to zero than $f(b)$, then b is relabeled as a, c is relabeled as b, and the prediction process is repeated. Provided the graph of $f(x)$ is smooth and provided the initial values of a and b are close to a simple root, the secant method rapidly converges to a root.

If the calculated secant is nearly horizontal, then SOLVE modifies the secant method to ensure that $|c - b| \leqslant 100\ |a - b|$. (This is especially important because it also reduces the tendency for the secant method to go astray when rounding error becomes significant near a root.)

If SOLVE has already found values a and b such that $f(a)$ and $f(b)$ have opposite signs, it modifies the secant method to ensure that c always lies within the interval containing the sign change. This guarantees that the search interval decreases with each iteration, eventually finding a root.

If this does not yield a root, SOLVE fits a parabola through the function values at a, b, and c, and finds the value d at the parabola's maximum or minimum. The search continues using the secant method, replacing a with d.

If three successive parabolic fits yield no root or $d = b$, the calculator displays **NO**. In the X- and Z-registers remain b and $f(b)$, respectively, with a or c in the Y-register. At this point you could: resume the search where it left off, direct the search elsewhere, decide that $f(b)$ is negligible so that $x = b$ is a root, transform the equation into another equation easier to solve, or conclude that no root exists.

Instructions

In calculating roots, SOLVE repeatedly calls up and executes a program *that you write* for evaluating $f(x)$. You must also provide SOLVE with two initial estimates for x, providing a *range* for it to begin its search for the root.

Realistic estimates greatly facilitate the speedy and accurate determination of a root. If the variable x has a limited range in which it is meaningful and realistic as a solution, it is reasonable to choose initial estimates within this range. (Negative roots, for instance, are often unrealistic for physical problems.)

- SOLVE requires thirteen unused program registers. If enough spare program registers are not available, SOLVE will not run and the error **NO ROOM** results. Execute [GTO] [·] [·] in Program mode to see how many program registers are available.
- Before running SOLVE you must have a program (stored in program memory or a plug-in module) that evaluates your function $f(x)$ at zero. This program must be named with a *global label*.* SOLVE then iteratively calls your program to calculate successively more accurate estimates of x. Your program can take advantage of the fact that SOLVE fills the stack with its current estimate of x each time it calls your program.
- You then enter two initial estimates for the root, a and b, into the X- and Y-registers.
- Lastly put the name of your program (that evaluates the function) into the Alpha register and then execute [SOLVE].

* This program should *not* include the functions [PASN], [PSIZE], [AK], any card-reader (HP 82104A) functions, or any other function that alters the configuration of the calculator's memory, key assignments, or timer alarms.

When the program stops and the calculator displays a number, the contents of the stack are:

Z = the value of the function at x = root (this value should be zero).*
Y = the previous estimate of the root (should be close to the resulting root).
X = the root (this is what is shown in the display).

If the function that you are analyzing equals zero at more than one value of x, SOLVE stops when it finds any one of these values. To find additional values, key in different initial estimates and execute [SOLVE] again.

Instruction Table for SOLVE

Instructions	Key In:	Display
1. Switch to Program mode and pack memory preparatory to entering a new program. (The display will show you the number of available program registers.)	[PRGM] [GTO] [·][·]	**00 REG** *nnn*
2. Key in a global, Alpha label as program name for the program describing $f(x)$ for $f(x) = 0$.	[LBL] *global label*	**01 LBL**^{T}*label*
3. Key in the lines of the program and end the program with a [RTN] instruction.	⋮ [RTN]	
4. Check that program memory is large enough to run SOLVE (*nnn*≥13).* Then switch out of Program mode.	[GTO] [·][·] [PRGM]	**00 REG** *nnn*
5. Put the name of your program from step 2 into the Alpha register.	[ALPHA] *global label* [ALPHA]	
6. Enter the range for the initial search for x:	*a* [ENTER↑] *b*	*a* *b*

* If the contents of the Z-register are *not* zero, then the X-register does not contain the exact root. Instead, the contents of X and Y are close estimates of the root, bracketing a change in the sign of the function's value.

Instruction Table for SOLVE (Continued)

Instructions	Key In:	Display
7. Execute [SOLVE]. The program runs up to several minutes and then returns the resulting root. If no root is found, the display is **NO**.	[SOLVE] †	*x*
8. To search for another root, repeat steps 6 and 7.		

* If *nnn* is not ≥13, then use [SIZE] to allocate more memory to program registers, or else delete programs. Refer to the HP-41 owner's manual for instructions.
† To execute a program, press [XEQ] [ALPHA] *Alpha name* [ALPHA] or use a User-defined key.

Remarks

Pressing [R/S] aborts the SOLVE program.

Examples

Find the roots of the equation $f(x) = x^2 - 3x - 10 = 0$.

First write a program called TEST to define the function. Then, before executing [SOLVE], put the name of this program into the Alpha register and enter your initial estimates for the root.

Using Horner's method you can rewrite $f(x)$ so that it is more efficiently programmable: $f(x) = (x - 3)x - 10$. (Note that you could also find this root algebraically.) Since the SOLVE program fills the stack with the current estimate of x before calling TEST, TEST can obtain x from the stack when TEST runs.

Keystrokes	Display	
[FIX] 4		Sets the display format used here.
[PRGM] [GTO] [·][·]	**00 REG** *nnn*	Program mode; ready to enter a program to evaluate $(x - 3)x - 10$.
[LBL] [ALPHA] TEST [ALPHA]	**01 LBL**^{T}**TEST**	Global Alpha label "TEST".
3	**02 3**_	

Keystrokes	Display	
[−]	03 −	$(x - 3)$
[×]	04 *	$(x - 3)x$
10	05 10_	
[−]	06 −	$(x - 3)x - 10$
[RTN]	07 RTN	End of program defining $f(x)$.
[GTO] [·] [·]	00 REG *nnn*	Number of available program registers (should be ⩾13).
[PRGM]		Exits Program mode.
[ALPHA] TEST [ALPHA]	TEST_	Puts "TEST" (your program's name) into the Alpha register. This is the necessary first step to running SOLVE.
0 [ENTER↑] 10	10_	Enters initial estimates of zero and ten. Now you're ready to execute [SOLVE]
[XEQ] [SOLVE]	5.0000	Runs the SOLVE program; finds a root of $x = 5.0000$ (in about 12 seconds).

Check that 5.0000 is indeed a root of $f(x) = 0$ by checking the Z-register. Then check for a second root (which is common in quadratic equations) by specifying new initial estimates of 0 and −10 to look for a negative root.

Keystrokes	Display	
[R↓] [R↓]	0.0000	Displays first the Y-register, then the Z-register. Since $f(5) = 0$, 5 is a good root.
0 [ENTER↑] 10 [CHS]	−10_	New initial estimates to look for a second root.
[XEQ] [SOLVE]	−2.0000	Second root.
[R↓] [R↓]	0.0000	This root is also good.

Here is a problem whose root cannot be found algebraically. If champion ridget hurler Chuck Fahr throws a ridget with an upward velocity of 50 meters/second, then how long does it take for it to reach the ground again? Solve for t in the equation

$$h = 5000\,(1 - e^{-t/20}) - 200t$$

Assume h in meters and t in seconds. Naturally we are only interested in a *positive* root, t.

As in the previous example, the program you write to define the function can take advantage of the fact that the stack is filled with the current estimate of x before calling your program.

Keystrokes	Display	
[PRGM] [GTO] [·] [·]	00 REG *nnn*	
[LBL] [ALPHA] HIGH [ALPHA]	01 LBL^T HIGH	Names this program "HIGH" with a global label.
20 [CHS]	02 −20_	
[÷]	03 /	$-t/20$
[e^x]	04 E↑X	
[CHS]	05 CHS	$-e^{-t/20}$
1	06 1_	
[+]	07 +	$1-e^{-t/20}$
5000	08 5000_	
[×]	09 *	$5000\,(1-e^{-t/20})$
[x⇄y]	10 X<>Y	
200	11 200_	
[×]	12 *	$200t$
[−]	13 −	
[RTN]	14 RTN	We now have the full equation so the program is done: $5000(1 - e^{-t/20}) - 200t$
[GTO] [·] [·]	00 REG *nnn*	Is *nnn*⩾13?
[PRGM]		Exits Program mode.
[ALPHA] HIGH [ALPHA]		Puts your program's name into the Alpha register.
5 [ENTER↑] 6	6_	Example of initial estimates for t.
[XEQ] [SOLVE]	9.2843	The root $t = 9.2843$ seconds.
[R↓] [R↓]	0.0000	Shows that $h(9.2843) = 0$.

When No Root Is Found

It is possible that an equation has no real roots. In this case, the calculator displays **NO** instead of a numeric result. This would happen, for example, if you tried to solve the equation

$$|x| = -1,$$

which has no solution since the absolute value function is never negative.

There are three general types of errors that stop SOLVE from running:

- If repeated iterations seeking a root produce a constant nonzero value for the specified function, the calculator displays **NO**.
- If numerous samples indicate that the *magnitude* of the function appears to have a nonzero minimum value in the area being searched, the calculator displays **NO**.
- If an improper argument is used in a mathematical operation as part of your program, the calculator displays **DATA ERROR**.

Programming Information

You can incorporate SOLVE as part of a larger program you create. Be sure that your program provides initial estimates in the X- and Y-registers just before it executes [SOLVE]. Remember also that SOLVE will look in the Alpha register for the name of the program that calculates your function.

If the execution of SOLVE in your program produces a root, then your program will proceed to its next line. If *no* root results, the next program line will be skipped. (This is the "do if true" rule of HP-41 programming.) Knowing this, you can write your program to handle the case of SOLVE not finding a root, such as by choosing new initial estimates or changing a function parameter.

SOLVE uses one of the six pending subroutine returns that the calculator has, leaving five returns for a program that calls SOLVE.

Note that SOLVE cannot be used recursively (calling itself). If it does, the program stops and displays **RECURSION**. You *can* use SOLVE with INTEG, the integration program.

References

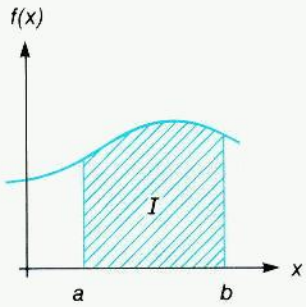
"Using SOLVE Effectively," *HP-15C Advanced Functions Handbook*, Hewlett-Packard Co., 1982.

Kahan, W.M., "Personal Calculator Has Key to Solve Any Equation $f(x)=0$," *Hewlett-Packard Journal*, 30:12, December 1979.

NUMERICAL INTEGRATION

The INTEG program finds the definite integral, I, of a function $f(x)$ within the interval bounded by a and b. This is expressed mathematically and graphically as

$$I = \int_a^b f(x)\, dx.$$



Executing the INTEG program (INTEG) employs an advanced numerical technique to find the definite integral of a function. You supply the equation for the function (in a program) and the interval of integration, and INTEG does the rest.

Method

The algorithm for INTEG uses a Romberg method for accumulating the value of an integral. The algorithm evaluates $f(x)$ at many values of x between the limits of integration. It takes the program from several seconds to several minutes to do this and produce a result.

Several refinements make the algorithm more effective. For instance, instead of using uniformly spaced samples, which can induce a kind of resonance producing misleading results when the integrand is periodic, INTEG uses samples that are spaced nonuniformly. Another refinement is that INTEG uses extended precision (13 significant digits) to accumulate the internal sums. This allows thousands of samples to be accurately accumulated, if necessary.

A calculator using numerical integration can almost never calculate an integral precisely. However, there is a convenient way for you to specify how much error is tolerable. You can set the display format according to how many figures are accurate in the integrand $f(x)$. A setting of [FIX] 2 tells the calculator that decimal digits beyond the second one can't matter, so the calculator need not waste time estimating the integral with unwarranted precision. Refer to the heading, "Accuracy of INTEG."

Instructions

In calculating integrals, INTEG repeatedly executes a program *that you write* for evaluating $f(x)$. You must also provide INTEG with two limits for x, providing an interval of integration.

- INTEG requires 32 unused program registers. If enough spare program registers are not available, INTEG will not run and the error **NO ROOM** results. Execute [GTO] [·][·] in Program mode to see how many program registers are available.
- Before running INTEG you must have a program (stored in program memory or a plug-in module) that evaluates your function $f(x)$. This program must be named with a *global label*.* Your program can take advantage of the fact that INTEG fills the stack with its current estimate of x each time it calls your program.
- You then enter the two limits, a and b, into the X- and Y-registers.
- Lastly put the name of your program (that evaluates the function) into the Alpha register and then execute [INTEG].

When the program stops and the calculator displays the integral, the contents of the stack are:

T = the lower limit of the integration, a.
Z = the upper limit of the integration, b.
Y = the uncertainty of the approximation of the integral.
X = the approximation of the integral (this is what is shown in the display).

* This program should *not* include the functions [PASN], [PSIZE], [AK], any card-reader (HP 82104A) functions, or any other function that alters the configuration of the calculator's memory, key assignments, or timer alarms.

Instruction Table for INTEG

Instructions	Key In:	Display
1. Switch to Program mode and pack memory preparatory to entering a new program.	[PRGM] [GTO] [·][·]	**00 REG** *nnn*
2. Key in a global, Alpha label as program name for the program describing $f(x)$.	[LBL] *global label*	**01 LBL**^{T}*label*
3. Key in the lines of the program and end the program with a [RTN] instruction.	⋮ [RTN]	
4. Check that program memory is large enough to run INTEG ($nnn \geq 32$).* Then switch out of Program mode.	[GTO] [·][·] [PRGM]	**00 REG** *nnn*
5. Put the name of your program from step 2 into the Alpha register.	[ALPHA] *global label* [ALPHA]	
6. Enter the limits for the initial search for the integral:	*a* [ENTER↑] *b*	*a* *b*
7. Set the display format to determine the accuracy of the result.	[FIX] *n* or [SCI] *n* or [ENG] *n*	
8. Execute [INTEG]. The program runs up to several minutes and then returns the resulting integral.	[INTEG] †	*integral*
9. To repeat this calculation using a different level of accuracy, set a new display format, roll down the stack to retrieve the original upper and lower limits, and re-execute [INTEG].	[FIX] *n* or [SCI] *n* or [ENG] *n* [R↓] [R↓] [INTEG]	*b* *integral*

* If *nnn* is not ≥32, then use [SIZE] to allocate more memory to program registers, or else delete programs. Refer to the HP-41 owner's manual for instructions.

† To execute a program, press [XEQ] [ALPHA] *Alpha name* [ALPHA] or use a User-defined key.

Remarks

Pressing [R/S] aborts the INTEG program.

Example 1

The Bessel function of the first kind of order 0 can be expressed as

$$J_0(x) = 1/\pi \int_0^{\pi} \cos(x \sin \theta) d\theta.$$

Find

$$J_0(1) = 1/\pi \int_0^{\pi} \cos(\sin \theta) d\theta.$$

First write a program to define the integrand. Make sure the calculator is set to Radians mode to calculate these trigonometric functions. Then, before executing [INTEG], put the name of your program into the Alpha register and enter the limits of integration. Once you've found the integral, don't forget to multiply it by $1/\pi$.

Keystrokes	Display	
[FIX] 4		Sets the display format used here.
[PRGM] [GTO] [·][·]	**00 REG** *nnn*	Program mode; ready to enter a program to evaluate cos(sin θ).
[LBL] [ALPHA] J01 [ALPHA]	**01 LBL^{T}J01**	Global Alpha label "J01".
[SIN]	**02 SIN**	sin θ.
[COS]	**03 COS**	cos(sin θ).
[RTN]	**04 RTN**	End of program defining $f(x)$.
[GTO] [·][·]	**00 REG** *nnn*	Number of available program registers; is $nnn \geqslant 32$?
[PRGM]		Exits Program mode.
[ALPHA] J01 [ALPHA]	**J01**	Puts "J01" (your program's name) into the Alpha register. This is the necessary first step to running INTEG.
0 [ENTER↑] [π]	**3.1416**	Enters integration limits of zero and π.
[RAD]	**3.1416**	Sets Radians mode. Now you're ready to execute [INTEG].

Keystrokes	Display	
[XEQ] [INTEG]	**2.4040**	Runs INTEG and returns the integral (in about 25 seconds). To complete the equation, don't forget to multiply by the constant outside the integral.
[π]	**3.1416**	
[÷]	**0.7652**	$J_0(1)$.

Accuracy of INTEG

Since the calculator cannot compute the value of an integral exactly, it *approximates* it. The accuracy of this approximation depends on the accuracy of the integrand's function itself as calculated by your program.* This is affected by round-off error in the calculator and the accuracy of empirical constants.

To specify the accuracy of the function, set the display format ([FIX] *n*, [SCI] *n*, or [ENG] *n*) so that *n* is no greater than the number of decimal digits that you consider accurate in the function's values. If you set *n* smaller, the calculator will compute the integral more quickly, but it will also presume that the function is accurate to no more than the number of digits shown in the display format.†

At the same time that the INTEG program returns the resulting integral to the X-register (the display), it returns the *uncertainty* of that approximation to the Y-register.‡ To view this uncertainty value, press [x⇄y].

If the uncertainty of an approximation is greater than what you choose to tolerate, you can decrease it by specifying more digits in the display format and rerunning INTEG.

---

* While integrals of functions with certain characteristics such as spikes or rapid oscillations might be calculated inaccurately, these functions are rare.

† [SCI] and [ENG] determine an uncertainty in the function that is *proportional* to the function's magnitude, while [FIX] determines an uncertainty that is *independent* of the function's magnitude.

‡ No algorithm for numerical integration can compute the exact difference between its approximation and the actual integral. But this algorithm estimates an *upper bound* on this difference, which is returned as the *uncertainty* of the approximation.

To rerun INTEG for the same problem but with a different display format, you do not need to re-enter the limits of integration (*if* you have not made any calculations subsequent to finding the integral). Since they end up in the T- and Z-registers (as shown under "Instructions"), just press [R↓] [R↓] to retrieve them, then execute [INTEG] again.

Example 2

With the display format set to [SCI] 2, calculate the integral in the expression for $J_0(1)$ in example 1. Check the uncertainty of this result. Then calculate a result accurate to four decimal places instead of only two, and check its uncertainty. (Make sure that Radians mode is still set by checking for the **RAD** annunciator, which should be on.) You will have to re-enter the limits of integration for the *first* computation only.

Keystrokes	Display		
[SCI] 2			Sets scientific notation; two decimal places of accuracy.
0 [ENTER↑] [π]	3.14	00	Enters the lower (0) and upper limits (π).
[XEQ] [INTEG]	2.40	00	The integral, accurate to two decimal places.
[x⇄y]	1.57	−03	The uncertainty of the integral.
[SCI] 4	1.5708	−03	Sets four decimal places of accuracy.
[R↓] [R↓]	3.1416	00	Roll down stack until upper limit appears.
[XEQ] [INTEG]	2.4039	00	Integral accurate to four decimal places.
[x⇄y]	1.5708	−05	Uncertainty (much smaller).

Programming Information

You can incorporate INTEG as part of a larger program you create. Be sure that your program provides upper and lower limits in the X- and Y-registers just before it executes [INTEG]. Remember also that INTEG will look in the Alpha register for the name of the program that calculates your function.

INTEG uses one of the six pending subroutine returns that the calculator has, leaving five returns for a program that calls INTEG.

Note that INTEG cannot be used recursively (calling itself). If it is, the program stops and displays **RECURSION**. You *can* use INTEG with SOLVE. A routine that combines INTEG and SOLVE requires 32 available program registers to operate.

References

"Working with [f5]," *HP-15C Advanced Functions Handbook*, Hewlett-Packard Co., 1982.

Kahan, W.M., "Handheld Calculator Evaluates Integrals," *Hewlett-Packard Journal*, 31:8, August 1980.

SV - SOLVE ROUTINE

This routine is a simple root solving program which will approximate a solution to an equation of the form: f(x)=0 using the Secant Method (a simplified form of Newton's Method). SV will find only one root at a time. The program requires an initial guess and an initial step size. The output is an x value which most closely makes f(x)=0. A flag may be set to display the successive approximations as they converge to the final answer. Convergence depends on the initial guess. Accuracy depends on the display setting.

Example 1: Use SV to find the two roots of the quadratic equation $x^2 + 2*x - 15 = 0$.

1. Insure a minimum SIZE 010.
2. Select a display setting of SCI 5. The routine will end when two successive approximations are rounded and found to be equal according to the display setting.
3. Set flag F10 to view the successive approximations.
4. The function on the left side of the equation must be programmed as a subroutine. The input to this subroutine, namely x, is assumed to be in the X-register and can be recalled from R07. The output from this subroutine, namely f(x), is also to be left in the X-register. For this example the following routine may be programmed in RAM program memory.

```
01*LBL "FX1"
02 X↑2
03 LASTX
04 2
05 *
06 +
07 15
08 -
09 RTN
```

5. The name of the global label "FX1" should be stored in R06. Go into alpha mode and key "FX1" ASTO 06.
6. The initial guess (nonzero) is to be entered along with an initial step size which may be zero or may be a small number compared to x. If a 0 step size is entered then the program will calculate the first step as 1% of the intial guess x. The program will also accept a non-zero value as the intial step size. For most applications and for this example use 0 as the initial step size. Choose x=7 as the initial guess for x. Key 0 ENTER↑ 7
7. XEQ "SV". The following consecutive approximations will be displayed.

```
7.00000+00
6.93000+00
3.98682+00
3.30024+00
3.03190+00
3.00115+00
3.00000+00
```

The final solution is returned after about 8 seconds. The true answer is exactly x=3. Since the above quadratic has two roots we will key in another initial guess to search for the other root.
This time we will guess x = -10. Key 0 ENTER↑ 10 CHS and XEQ "SV". The following sequence of numbers will be displayed.

```
-1.00000+01
-9.90000+00
-6.36872+00
-5.47003+00
-5.06539+00
-5.00360+00
-5.00003+00
-5.00000+00
```

The true answer this time is exactly x = -5.

COMPLETE INSTRUCTIONS FOR SV

(Keyboard Operations):

To calculate a root of f(x)=0:

1) Select SIZE. The minimum size required by SV is SIZE 010. The storage requirements for constants and coefficients associated with the function f(x) may dictate a larger size.

2) Select display setting. The display setting will generally determine when SV ends. If an exact solution is found then SV will end on the next iteration, otherwise SV rounds the last two approximations and ends if those rounded values are equal. In general, a display setting of SCI n will produce (optimistically) a solution correctly rounded to n+1 significant figures. A display mode of SCI or ENG is generally preferred to a FIX mode.

3) Specify display option. Flag 10 controls a display option. If F10 is set then the successively calculated approximations will be displayed. In this manner the user may view the progress of the iterations. This is especially recommended in the SV routine since SV may fail to converge if the initial guess is too far away from an actual root. Even when the values stabilize they may oscillate and it is a simple matter for the user to manually stop the program. If F10 is clear only the final x-value is returned.

4) Program the function f(x)=0. The function f(x) represented by one side of the equation must be programmed as a subroutine in program memory which starts with a global label name and ends with a RTN or END instruction. The label name should be of six or less characters and should be stored in R06. The input x and the output f(x) are both assumed to be in the X-register. The input x may also be recalled from R07. Since global label search begins from the bottom of program memory, it is advisable to place f(x) near the bottom of program memory. The f(x) program should not use registers R06-R09 and should not disturb flag F10.

5) Store the global label name from step 4) (six or less characters) in R06. The function subroutine call will be made via an XEQ IND 06 instruction.

6) Specify initial step size and initial guess. SV requires two input values. The first input is the step size which the program uses to determine the approximation for the derivative at the initial guess. The second input is the initial guess and is used as the starting x value by the program. The closer the initial guess is to the true solution the quicker the solution is found. Do not use 0 as an initial guess.

If zero is entered as the initial step size then the program will automatically calculate 1% of x as the acutal step size. A zero step size should prove adequate for the majority of applications. However, the user may enter a non-zero step size which may be finer or coarser than 1% of the initial x.

These two values are keyed in as:

step size ENTER↑ guess

7. XEQ " SV ". If F10 is set the program will display the consecutive approximations. If a printer is plugged in and turned on these approximations will be printed. The final solution will be left in the X-register when the program ends.

MORE EXAMPLES OF SV

Example 2: Solve $f(x) = x^3 - x - 1 = 0$.

1) SIZE 010 minimum
2) Set display mode as SCI 6.
3) Set flag F10 to view the approximations.
4) Key the following routine for f(x) into program memory:

```
LBL*FX2
ENTER↑
 X↑2
  1
  -
  *
  1
  -
 RTN
```

5) Key "FX2" in the alpha register and press ASTO 06.
6) Key in the initial step size as 0 and key in the initial guess as x=4. Key 0 ENTER↑ 4.
7) XEQ " SV ".
The following sequence of approximations will be displayed:

```
4.000000+00
3.960000+00
2.731772+00
2.226515+00
1.780222+00
1.522190+00
1.382556+00
1.333776+00
1.325185+00
1.324722+00
1.324718+00
```

The final solution is returned after about 13 seconds. The solution is correct to the digits displayed.

Example 3: Solve $f(x) = x^3 - 3*x^2 + 4 = 0$.

1) SIZE 010 minimum
2) Set display mode as SCI 4.
3) Set flag F10 to view the approximations.
4) Key the following routine for f(x) into program memory:

```
LBL*FX3
 X↑2
LAST X
  3
  -
  *
  4
  +
 RTN
```

5) Key "FX3" in the alpha register and ASTO 06.
6) Key in an initial step size of 0 and an initial guess of -2. Key 0 ENTER↑ 2 CHS
7) XEQ " SV ".
The following approximations will be displayed:

```
-2.0000+00
-1.9800+00
-1.3283+00
-1.1289+00
-1.0229+00
-1.0018+00
-1.0000+00
```

The final answer is returned after about 8 seconds. The true solution is exactly x = -1.

The following examples contain abbreviated instructions.

Example 4: The following equation is known as Kepler's equation:

$$x - E*SIN(x) - m = 0$$

and plays an important role in astronomy and astrodynamics (space travel). It can be programmed as follows:

```
LBL*FX4
ENTER↑
 SIN
RCL 01        (Note: R01=E)
  *
  -
RCL 02        (Note: R02=m)
  -
 RTN
```

Set RADIANS angle mode. The equation can now be solved for any values of E (R01) and m (R02). When E=0.2 and m=0.8 the function has only one root (9.64334-01) which can be found with any initial guess.

Example 5: SV can be used to find maxima and minima of a function by solving for zeros of its derivative. For example, if $f(x) = \sin(x)/x$ then the derivative $f'(x) = [x*\cos(x) - \sin(x)]/x^2$
Zeros of f' occur where the numerator is 0. Consequently, solutions can be found by applying SV to the following function which represents the numerator $g(x) = x*\cos(x) - \sin(x)$

```
LBL*GX5
ENTER↑
 COS
  *
RCL 07
 SIN
  -
 RTN
```

Assuming SCI 6 display mode and RADIANS angle mode. Store "GX5" in R06. Key in small initial guesses using a step size of 0 and SV will find the first few roots as 0, ±4.49341+00, ±7.72525+00. (Note that instead of taking the derivative algebraically, the ROM routine FD might be used).

Continuting this same example for another root:
Initial step size = 0
Initial guess = -10
See:

```
-1.000000+01
-9.900000+00
-7.959395+00
-7.135530+00
-6.438967+00
-6.086301+00
-5.945260+00
-5.917701+00
-5.915863+00
-5.915842+00
```

Result: -5.915842+00 after about 14 seconds

FURTHER DISCUSSION OF SV

SV is not a sophisticated root solver and is subject to all the difficulties and error traps that confront all other root solvers. Limited space in the PPC ROM did not allow protection schemes to detect or rectify possible trouble areas. The method used, strictly speaking, is the Secant Method, however, it can be considered a form of Newton's Method where a numerical approximation is used for the derivative. A secant line is used to approximate the true tangent line. If SV fails to converge then another initial guess must be tried. SV can be effective as a subroutine in a program provided the user has knowledge of the range of appropriate values for the given function.

The display setting will help control the accuracy of the final result. When in SCI n display mode the final answer will (usually) be accurate to n+1 significant digits. However, sometimes this is not the case and the final answer will not be as accurate as the display setting would indicate. Every floating point operation in a computational process can give rise to rounding error which, once generated, may then be increased in subsequent operations.

For example, let $f(x) = x^2 - 6*x + 9$ and use SV to solve for f(x)=0. In FIX 9 the answer SV returns may be 3.000030072 which is accurate to five digits only. The true solution is a double root at x=3. Thus the display setting has not determined the accuracy in this example. This is basically caused by using only ten digits internally in the calculator, causing each and every calculation to have its solution rounded to ten digits. The root solver itself cannot then be held to ransom when the f(x) routine is affected by rounding errors.

This example highlights the action of SV when the secant line is horizontal, that is, when f(a)=f(b) where a and b are successive approximations. This situation may occur at multiple roots where the first derivative shares a root with the original function. (When the first derivative is zero the tangent line is horizontal). In general, do not select a display n value any larger than necessary. The use of SCI or ENG display modes are generally preferred to the FIX display mode. And do not blindly accept any solution given by this or any other root-solving program. Any potential real solution can be validated by applying the f(x) subroutine to see how close f(x) really is to 0.

Because the HP-34C calculator's SOLVE routine uses a similar method as SV (with many refinements), users are urged to study Reference 5. In that informative article some of the problems of root solving and a description of the mathematics of the secant method are discussed. Reference 1 provides a broad background to the subject. Further information may be found in most university and college libraries.

THE SELECTION OF A METHOD FOR SV

The oldest know method of root-solving is the Method of False Position, or Regula Falsi. Commencing with two estimates, lying on either side of the actual root, inverse linear interpolation is applied to produce a new estimate. Here, a linear function (a straight line) is used to approximate the true function f(x) over the interval of interest. This can be seen to be "reasonable" approximation so long as the interval is "small". As the two estimates must always straddle or bracket the root, convergence is always guaranteed. Of course, after calculating the new estimate, a decision has to be made as to which previous estimate is to be discarded.

The Method of False Position has a unity order of convergence, making the iteration time reasonably long. However, the solution is always obtained.

A method similar to False Position is the Secant Method. Although the mathematics of the two methods are identical, the difference lies in the fact that the Secant Method uses the approximations in strict sequence. Thus, the bracketing of the root is no longer necessary, and a secant is used to approximate the function f(x) over the interval of interest. Then, inverse linear interpolation is applied to produce the next estimate of the root. The Secant Method's order of convergence is approximately 1.62, which is higher than the Method of False Position, but convergence to the root is no longer guaranteed.

Both the Method of False Position and the Secant Method are in the class of two-point iterative methods, which, while the order of convergence is not high, nevertheless have high stability.

Another popular method derived from calculus using a Taylor Series is commonly known as the Newton-Raphson, or Newton's Method. In this method a tangent line to the function is used to determine the direction and amount of displacement to move from the current estimate to the new estimate. Newton's Method belongs to the class of one-point iterative methods. Newton's Method is the official and familiar name of tangent sliding philosophy, and has an order of convergence of two.

The mercurial properties of Newton's Method arise from its use of derivative information gathered at one point. Both the function and its derivative must be evaluated at each iteration. (This also results in a greater programming effort for two functions are really being evaluated). The time for an iteration is longer than the False Position and Secant Methods, which both require only one function evaluation per iteration. However, the convergence rate of Newton's Method is greater than both.

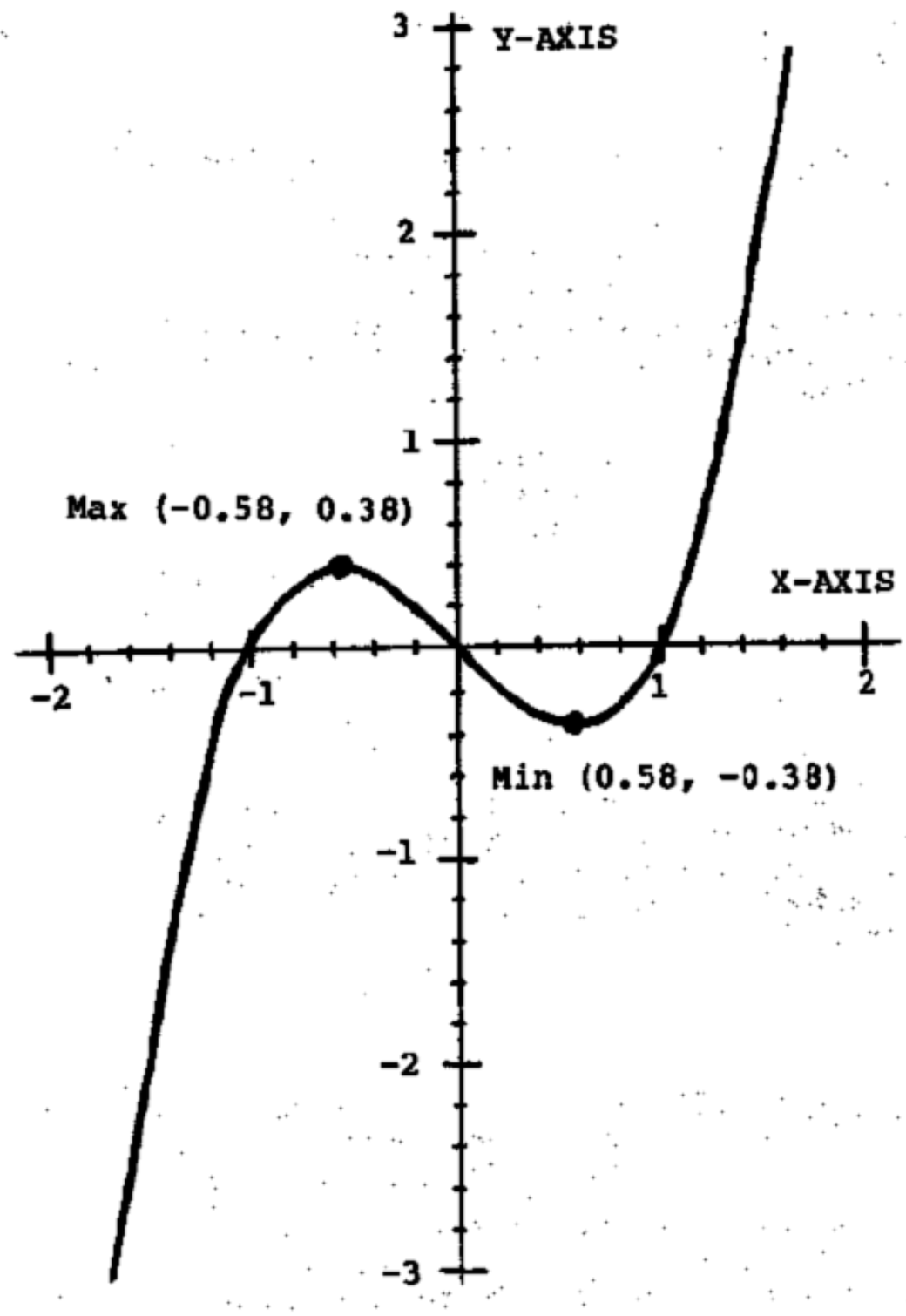
Example 6: Use SV to find all three roots of the cubic equation:

$$x*(x^2-1) = 0$$

The following should be programmed:

```
LBL*FX6
ENTER↑
 X↑2
  1
  -
  *
 RTN
```

A sketch of the graph of this function will prove useful in understanding how the inital guesses determine which root is found.



GRAPH OF $Y = X^3 - X = X(X^2 - 1) = X(X+1)(X-1)$

The initial step size is 0 for each of the guesses suggested below. An initial guess greater than 0.6 for this example will find the root x=1.0, while a guess between -0.49 and +0.49 will find the root at x=0.0. However, guesses too close to the local peaks of the function at $x=\pm 1/\sqrt{3}$, where the slope of the tangent line is zero, lead to oscillations that fail to converge. Try an initial guess of x=0.58 to observe this behavior. This example also illustrates that the root found is not necessarily the one nearest to the initial guess. Try x=0.52 which finds the root x = -1.0.

Example 7: $f(x) = x*LN(x) - 1.2 = 0$

Display mode: SCI 6
Initial step size = 0
Initial guess = 3
See:

3.000000+00
2.970000+00
1.998929+00
1.902018+00
1.888327+00
1.888087+00

Result: 1.888087 after about 8 seconds

Example 8: $f(x) = 3*x - COS(x) - 1 = 0$

Display mode: SCI 6
Use RADIANS angle mode
Initial step size = 0
Initial guess = 2
See:

2.000000+00
1.980000+00
6.159990-01
6.078243-01
6.071024-01
6.071016-01

Result: 6.071016-01 after about 10 seconds

Example 9: $f(x) = x^2 + 4*SIN(x) = 0$

Display mode: SCI 6
Use RADIANS angle mode
Initial step size = 0
Initial guess = -4
See:

-4.000000+00
-3.960000+00
-2.210789+00
-2.037220+00
-1.946227+00
-1.934406+00
-1.933758+00
-1.933754+00

Result: -1.933754 after about 12 seconds

Example 10: $f(x) = x^4 - 26x^2 + 49x - 25 = 0$

Display mode: SCI 6
Initial step size = 0
Initial guess = 5
See:

5.000000+00
4.950000+00
4.310588+00
4.077156+00
3.927333+00
3.883035+00
3.876065+00
3.875777+00
3.875775+00

Result: 3.875775 after about 14 seconds

Comparison of Methods

A. Method of False Position.
   Advantages:
   1. Convergence is guaranteed.
   2. Only one function evaluation is needed at each iteration.
   Disadvantages:
   1. Low (unity) order of convergence.
   2. A decision is needed as to which estimate to discard to insure root-bracketing occurs.
   3. The root-bracketing requirement prevents its use at multiple, even-order roots.

B. The Secant Method.
   Advantages:
   1. Medium (approx. 1.62) order of convergence.
   2. Estimates are used in strict sequence, so no decision is needed on which estimate to discard.
   3. Only one function evaluation is needed at each iteration.
   4. Can be very stable.
   Disadvantages:
   1. Convergence is not always guaranteed.
   2. May have difficulty at multiple even-order roots.

C. Newton's Method.
   Advantages:
   1. High (2.0) order of convergence.
   Disadvantages:
   1. Convergence is not always guaranteed.
   2. Both the function and its derivative must be evaluated at each iteration, which increases the time per iteration.
   3. The derivative must be known explicitly.
   4. Can be very unstable.
   5. Has difficulty at multiple, even-order roots, where the function and its first derivative both have the same root.

Summary

From the above comparison, the Secant Method was considered to be the optimum algorithm, and was selected for SV. It combines a reasonable rate of convergence, a (usually) stable two-point step, uses the calculated approximations in strict sequence, and does not require evaluation of the derivative, dispensing with the need to provide the derivative explicitly.

ROOT SOLVING DIFFICULTIES - A PRIMER

One of the most frequently occuring problems in scientific work is to find the values of x for an expression f(x) which will make f(x)=0. Those values of x are called the roots of the equation f(x)=0. The function may be given explicitly as, for example, a polynomial, or as a transcendental function. In rare caes it may be possible to obtain the exact roots by algebraic manipulations. In general, however, we can hope to obtain only approximations to the roots relying on some iterative computational procedure to produce those approximations.

In the year 1225 Leonardo of Pisa studied the equation:

$$f(x) = x^3 + 2*x^2 + 10*x - 20$$

and was able to produce the root of 1.368808107. Nobody knows by what method Leonardo found this value, but it was a remarkable achievement for his time.

Simply put, all we require are those values of x which will make f(x)=0. It should be easy, so why all the fuss? The basic difficulty stems from the fact that our root solving methods tend only to use the function expression to numerically evaluate f(x) and have no analytical knowledge of the function. It they did, better starting guesses and better exit criteria could be selected. More importantly, the best root solving method to use for a particular function could be selected.

Easy though root solving may seem, and when coupled with one of the popular methods, e.g., Newton's Method, Secant Method, Bisection Method, It may come as a surprise to know that the search for the perfect root solver is no less difficult than the search for the Holy Grail! For every root finding method put forward, a situation can be provided which will cause the method to fail and deny us the solution.

What can be done? A root solver is simply a computational process which uses known facts (data) to calculate a better approximation to the root. The basic difference among root solving methods is the way in which the known facts are used to calculate the improved estimate.

If we know situations that cause our root solver to fail to find the root, we can provide assistance by enhancing the basic method with strategies to detect these problem situations and allow an escape from them. How many difficulties may confront a root solver? How many strategies do we design into it? One, five or one hundred? If we limit our root solver to solve only a specific class of problems we may be able to implement some strategies to overcome the typical problems encountered by that class.

Starting Values

All root solvers require starting values whether they be entered manually as in SV or use some set of values, e.g., 1 and 10. If the starting value is not "close" to the root, our selected method may step away from the root, making the problem worse.

Therefore the accuracy of the starting values (guesses) can be seen to be as important as the selected root solving method and its inbuilt strategies. Do we have strategies for determining an approximation to the root we seek? In some cases the answer is yes.

Exit Criteria.

When the root solver has located a root, it exits the iteration process, gives us the answer and stops. How does the root solver know when it really has found a root? Our inbuilt exit criteria must apply certain tests to the known facts and assess if a root has been found. Again, situations can be provided to fool the exit tests, and cause execution to stop when really no root has been found (i.e., no root exists).

Should we have several exit tests built into the program? Under what conditions should each be invoked? If only the answers were simple!

Numerical instability and round off errors.

Some root solvers can exhibit instability in certain

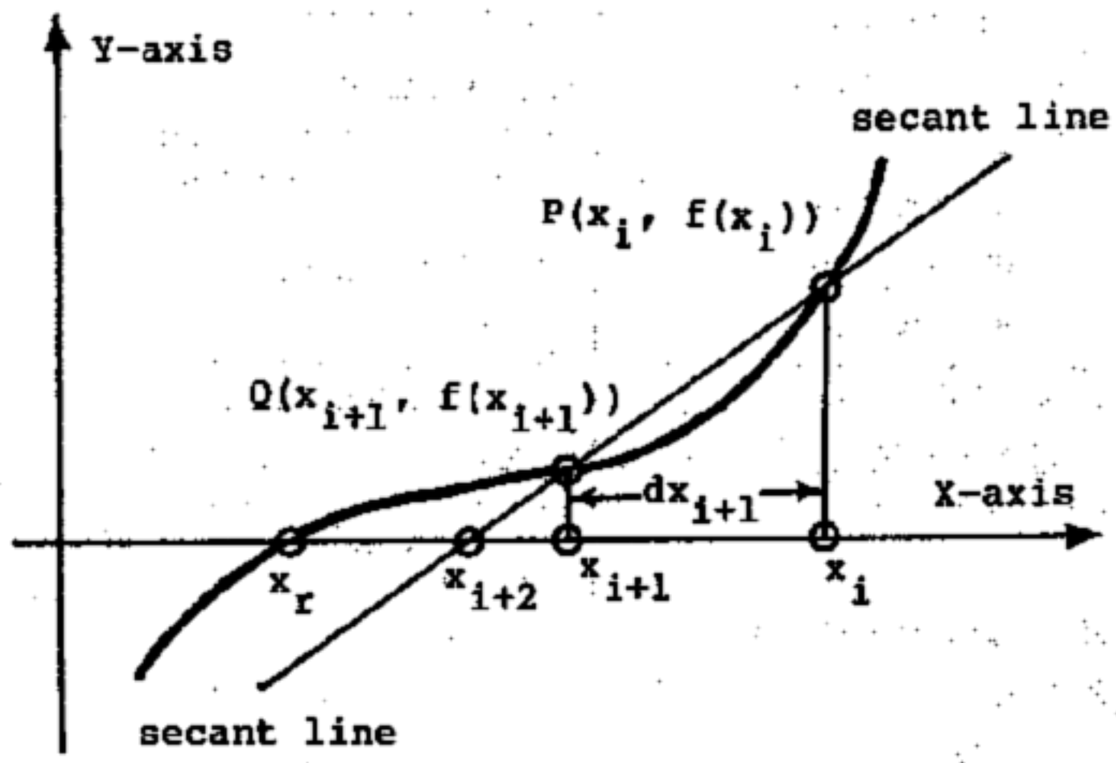
conditions. Do we know how many of these conditions exist for any one root solver? How do we overcome these problems? As all floating point calculations are rounded to ten digits by the calculator a further difficulty arises due the propogation of these errors.

Because of the enormous difficulties which may confront a root solving process, **SV** has been written as an elementary routine, with no refinements or strategies included. It is left to the user to provide such strategies, by observing the behavior of the approximations, and taking action should divergence occur. A little experience with particular problems will provide guidance. Readers are urged to consult Reference 5.

FORMULAS USED IN SV

Let f(x) be the function whose root x_r we desire.

x_{i+1} and x_i are the two previous approximations.



GRAPH OF GENERAL FUNCTION WITH SECANT LINE SHOWING APPROXIMATION X_{i+2} FOLLOWING X_i AND X_{i+1}

The slope of the secant line through points P and Q is given by:

$$[f(x_{i+1}) - f(x_i)]/[x_{i+1} - x_i]$$

Letting $dx_{i+1} = x_{i+1} - x_i$ we have as the equation of the secant line through points P and Q:

$$y - f(x_{i+1}) =$$

$$([f(x_{i+1}) - f(x_i)]/dx_{i+1})*(x - x_{i+1})$$

Hence,

(1) $$x_{i+2} = x_{i+1} + dx_{i+2}$$

where

(2) $$dx_{i+2} = \frac{(dx_{i+1})*f(x_{i+1})}{f(x_i) - f(x_{i+1})}$$

The initial value x_0 is input by the user and dx_0 is usually taken as a small fractional part of x_0

For example, if we assume $f(x_{-1})=0$ and $dx_0=(.01x_0)$ then $x_1 = (.99)x_0$

Execution halts when x_{i+2} and x_{i+1} are rounded and found to be equal.

Routine Listing For:	SV
91*LBL C	108 ST* 09
92*LBL "SV"	109 ST- 08
93 STO 07	110 RCL 09
94 .E	111 RCL 08
95 %	112 X≠0?
96 RCL Z	113 /
97 X=0?	114 STO 09
98 X<>Y	115 X<> 07
99 STO 09	116 ST+ 07
100 CLST	117 RND
101*LBL 04	118 RCL 07
102 RCL Z	119 RND
103 STO 08	120 X≠Y?
104 RCL 07	121 GTO 04
105 FS? 10	122 RCL 07
106 VIEW X	123 RTN
107 XEQ IND 06	

LINE BY LINE ANALYSIS OF SV

Lines 91-101 initialize the program by storing the initial guess X_0 in R07 and store the initial step size in R09. Note that if the user has input 0 as the initial step size then lines 94 & 95 and lines 97 & 98 calculate and select the value $0.01*X_0$ as the actual step size.

Lines 101-121 are the main loop in the program. At LBL 04 X, Y, and T are assumed to be scratch and $f(X_i)$ is assumed to be in Z. The next approximation is calculated via formula (1) and is stored in R07 (line 116). Next, the two most recent approximations are rounded and tested for equality in line 120. A branch is then made back to LBL 04 unless the rounded values are equal.

Lines 122 & 123 recall the final solution and end the routine.

REFERENCES FOR SV

1. Forman S. Action, NUMERICAL METHODS THAT WORK, Harper and Row, New York, 1970

2. S. D. Conte and C. de Boor, ELEMENTARY NUMERICAL ANALYSIS, McGraw-Hill, 1972

3. John Kennedy (918) PPC JOURNAL "Method of Successive Bisections" V5N8P19. See also V6N5P10

4. Chris Stevens, PPC JOURNAL, V5N8P45, Sept.-Oct. 1978

5. William M. Kahan, "Personal Calculator Has Key To Solve any Equation f(x)=0", Hewlett-Packard Journal, December 1979.

CONTRIBUTORS HISTORY FOR SV

John Kennedy (918) wrote the SV program for the HP-41C from a previous HP-25 program. Graeme Dennes (1757) and Richard Schwartz (2289) made suggestions for improvements in the accuracy and overall program operation. Harry Bertucelli (3994) suggested register usage to allow SV to be used with IG. Graeme Dennes(1757) and Iram Weinstein (6051) contributed to the documentation of SV.

FINAL REMARKS FOR SV

SV needs improvement in almost all areas. SV is only a basic routine designed to be used primarily from the keyboard where the user may watch the convergence (or lack thereof) and take action to halt SV and make a new guess. SV lacks all of the sophistication of the SOLVE function on the HP-34C calculator.

FURTHER ASSISTANCE ON SV

John Kennedy (918) phone: (213) 472-3110 evenings

Graeme Dennes (1757) phone (415) 592-2957 evenings

NOTES

TECHNICAL DETAILS

XROM: 20, 10 | SV | SIZE: 010 minimum

Stack Usage:

- 0 T: used
- 1 Z: used
- 2 Y: used
- 3 X: used
- 4 L: used

Alpha Register Usage:

- 5 M: not used
- 6 N: not used
- 7 O: not used
- 8 P: not used

Other Status Registers:

- 9 Q: not used
- 10 ⊢: not used
- 11 a: not used
- 12 b: not used
- 13 c: not used
- 14 d: not used
- 15 e: not used

ΣREG: not used

Data Registers:

R00: not used

R06: function LBL name

R07: point x_1

R08: $f(x_1)$

R09: dx_1

R10: not used

R11: not used

R12: not used

Flag Usage:

04: not used

05: not used

06: not used

07: not used

08: not used

09: not used

10: displays successive approximations when F10 is set

25: not used

Display Mode:

SCI n recommended

controls accuracy

Angular Mode:

not used, but may be required by function

Unused Subroutine Levels:

4

Global Labels Called:

Direct	Secondary
function LBL in R06	none

Local Labels In This Routine:

C, 04

Execution Time: variable, depends on function, display setting, and initial guess.

Peripherals Required: none

Interruptible? yes

Execute Anytime? no

Program File: IG

Bytes In RAM: 51

Registers To Copy: 43

Other Comments:

IG - INTEGRATE

This routine uses the Romberg algorithm to calculate a numerical approximation of the definite integral of a function. The routine is iterative in that increasingly accurate approximations are calculated until two rounded consecutive approximations are equal. The routine is automatic in that no step-size information has to be provided. The desired accuracy of the final approximation is determined by the display setting. The consecutive approximations may be viewed by setting a flag.

Example 1: Use IG to approximate $\int_0^1 4/(x^2 + 1)\,dx$,

to five significant digits.

1. SIZE 030 minimum.
2. Select a display setting of SCI 4.
3. Set flag F10 to view the successive approximations
4. Key the integrand of the above integral as a function in program memory starting with a global label and ending with a RTN. Assume x is in the X register and leave f(x) in the X register. Key in the following steps for this example.

```
01*LBL "FX1"
02 X↑2
03 1
04 +
05 4
06 X<>Y
07 /
08 RTN
```

5. Alpha-store the global label name into register R10.
   Key "FX1" ASTO 10.
6. Enter the limits of integration into the stack as 0 ENTER↑ 1.
7. XEQ " IG ".

The following sequence of numbers will be displayed.

```
3.2000+00
3.1405+00
3.1413+00
3.1416+00
3.1416+00
```

This example takes about 49 seconds to run. The true answer is pi, and the displayed result is accurate to five significant digits. Switching to FIX 9 we see the last value returned is 3.141592651 but because we were in SCI 4 mode we should not expect more than 4 decimal places of accuracy. Displaying more digits will cause the program to run longer.

Calculate the same integral a second time but change to SCI 6 display mode. Since the function subroutine and the global label name in R10 are not changed, simply key 0 ENTER↑ 1 and XEQ " IG " again. The following sequence of numbers will be displayed.

```
3.200000+00
3.140464+00
3.141329+00
3.141598+00
3.141593+00
3.141593+00
```

The last value is returned after about 89 seconds.

COMPLETE INSTRUCTIONS FOR IG

(Keyboard Operation):

To calculate $\int_a^b f(x)dx$

1) Select SIZE. SIZE 030 is the recommended minimum. A few integrals may require a larger size.

2) Set display mode. The display setting will control the accuracy of the final approximation. In general, a display mode of SCI n will return a value correctly rounded to n+1 significant digits. Larger values of n will cause the program to run longer so it is best to select the minimum value of n that is acceptable. The use of SCI or ENG display modes are generally preferable to the FIX mode.

3) Select display view option. Flag 10 controls a display option. If F10 is set then the successive approximations that the program calculates will be displayed. In this manner the user may view the progress of the iterations. If F10 is set and a printer is connected the approximations will be printed. If F10 is clear only the final approximation is returned in the X-register.

4) Specify the integrand. The integrand represented by the function f(x) must be programmed as a subroutine in program memory which starts with a global label name and ends with a RTN or END instruction. This label name should be of six or less characters and will be stored in R10. The input x and the output f(x) are both assumed to be in the X-register. Since global label search begins at the bottom of program memory, when the f(x) subroutine is in RAM it should be located at the bottom of RAM. Using a single character global label name will reduce execution time. The f(x) program should not use registers R10-R29 or use flags F09 and F10.

5) Alpha-store the global label name from step 4 (six or less alpha characters) in R10.

6) Enter limits of integration. The lower and upper limits of integration, a and b, respectively, are to be keyed in as a ENTER↑ b so that a is keyed into the Y-register and b is keyed in the X-register.

7) Execute IG program. Key XEQ " IG ". Program execution will commence, and if F10 is set, the consecutive approximations will be displayed. If a printer is connected and turned on the approximations will be printed. The final approximation will be left in the X-register when the program ends.

MORE EXAMPLES OF IG

Example 2: Calculate $\int_0^1 x^{1/2}dx$

1. Select SIZE 030.
2. Select a display mode of SCI 4.
3. Set flag F10 to VIEW the approximations.
4. Key in the following routine for f(x).

```
LBL*FX2
SQRT
RTN
```

5. Key "FX2" in the alpha register and ASTO 10.
6. Key in the limits of integration as 0 ENTER↑ 1.
7. XEQ " IG ".

The following approximations will be displayed.

```
7.0711-01
6.6947-01
6.6667-01
6.6667-01
```

The final answer is returned after about 23 seconds.
The true answer is 2/3.

Example 3: Calculate $\int_0^1 \sin(\pi x)\,dx$

1. Select SIZE 030.
2. Select a display mode of SCI 4 and select RADIANS angle mode.
3. Set flag F10 to VIEW the approximations.
4. Key in the following routine for f(x).

```
LBL*FX3
PI
*
SIN
RTN
```

5. Key "FX3" in alpha and ASTO 10.
6. Key in the limits of integration as 0 ENTER↑ 1.
7. XEQ " IG ".

The following approximations will be displayed.

```
1.0000+00
6.0355-01
6.3789-01
6.3660-01
6.3662-01
6.3662-01
```

The final answer is returned after about 102 seconds.
The true answer is $2/\pi$.

In the following examples only the original problem and the numbers output are given.

Example 4: Calculate $\int_0^1 LN(x)\,dx$

```
-6.931-01
-9.331-01
-9.879-01
-9.972-01
-9.993-01
-9.998-01
-1.000+00
-1.000+00
```

The approximate time is 301 seconds in SCI 3.
The true answer is exactly -1.

Example 5: Calculate $\int_0^1 \frac{x^{1/2}}{x-1} - \frac{1}{LN(x)}\,dx$

```
2.8481-02
3.6106-02
3.6618-02
3.6519-02
3.6496-02
3.6491-02
3.6490-02
3.6490-02
```

The approximate time is 402 seconds in SCI 4.
The true answer to 7 decimals is 0.0364900.

Example 6: Calculate $\int_0^2 [x(4-x)]^{1/2}\,dx$

```
3.46410162
3.15270628
3.14152977
3.14159373
3.14159265
3.14159265
```

The approximate time is 85 seconds in FIX 8.
The true answer is π.

Example 7: Calculate $\int_0^{\pi} \frac{600*\sin^2(x)}{x^{1/2} + (x+600\,\pi)^{1/2}}\,dx$

Remember to use RADIANS angle mode.

```
4.21808+01
1.75899+01
2.13355+01
2.10986+01
2.11020+01
2.11020+01
```

The approximate time is 146 seconds in SCI 5.
The correct answer to 5 decimals is 21.10204

Example 8: Calculate $\int_0^1 COS(LN(x))\,dx$

Use RADIANS angle mode.

```
7.692-01
4.563-01
4.765-01
5.035-01
5.018-01
4.999-01
4.999-01
```

The approximate time is 213 seconds in SCI 3.
The true answer is exactly 0.5

Example 9: Calculate $\int_0^1 x^{-1/2}\,dx$

1.414+00
1.710+00
1.865+00
1.934+00
1.967+00
1.984+00
1.992+00
1.996+00
1.998+00
1.999+00
1.999+00

The approximate time is 33 minutes and 52 seconds in SCI 3. The true answer is exactly 2.

Example 10: Calculate $\int_0^1 (1-x^2)^{1/2}\,dx$

8.6602540-01
7.8817657-01
7.8538244-01
7.8539843-01
7.8539816-01
7.8539816-01

The approximate time is 85 seconds in SCI 7. The true answer is $\pi/4$.

Example 11: Calculate: $\int_{-1}^{1} \frac{x^7(1-x^2)^{1/2}}{(2-x)^{13/2}}\,dx$

0.0000+00
6.6870-03
3.0827-02
2.3585-02
2.3850-02
2.3857-02
2.3857-02

The approximate time is 342 seconds in SCI 4. The true answer to 7 decimal places is 0.0238566

Example 12: Calculate: $\int_{-1}^{1} [(1-x^2)(2-x)]^{1/2}\,dx$

2.8284+00
2.2239+00
2.2033+00
2.2033+00

The approximate time is 29 seconds in SCI 4. The true answer to 6 decimal places is 2.203345

FURTHER DISCUSSION OF IG

The ability of any numerical integrator to determine the definite integral of a function is basically determined by:

1. The behavior of the function over the interval of integration.
2. The selected numerical method.
3. The required accuracy of the solution.

Due to the above it is not possible to fully expand in this documentation on the applications of IG or to fully discuss the difficulties that may arise. This type of information must be obtained from books and publications on numerical methods such as those given in the References.

However, References 5 and 8 would probably be the most informative on the Romberg numerical integration method. It was from those two References that the IG routine was born. In Reference 8, the theory underlying the integrate function key on the HP-34C calculator is presented in quite some detail. Reference 5 describes the Romberg integration procedure, along with the theory of many other methods. An extensive bibliography on the subject is also provided in Reference 5.

Reference 11 provides highly valuable practical insight into the HP-34C integration key function and its applications.

Mathematical Background of IG

The method underlying the program is due to Romberg (Reference 1) and is essentially an application of Richardson's extrapolation procedure to the Euler-Maclaurin sum formula. Romberg was first to describe the method in recursive form. Commencing with improved midpoint rule estimates, the continued application of extrapolation to the limit produces a lower triangular matrix. Although the columns of the matrix converge to the solution, the diagonal elements converge asymptotically faster than any geometric series, or, superlinearly. Program shut-off occurs when two rounded consecutive diagonal elements are equal. The diagonal elements M(k,k) are the values displayed when flag F10 is set.

Assuming that the number of divisions of the interval of integration is increased by improved midpoint rule estimates, one would assume that convergence would occur when we made the number of intervals high enough. However, at some point, roundoff errors eventually dominate and our effective accuracy decreases. The Romberg method allows the simulation of a high number of sub-intervals or divisions which decreases the error, without actually increasing the number of sub-intervals. This process is called extrapolation to the limit.

Romberg integration is iterative, automatic, and non-adaptive. It is iterative in that it produces increasingly accurate estimates of the solution until the convergence criterion is satisfied. It is automatic in that the number of function evaluations depends upon the behavior of that function over the interval of integration. It is non-adaptive in that function evaluations occur at a fixed set of points, independent of the function.

As the Romberg method successively halves the interval of integration to produce improved midpoint rule estimates, it uses all previously computed functional evaluations at each stage. The retention of all previously calculated functional evaluations is a significant aspect of the Romberg algorithm. As the function is evaluated at the center of each interval, the end points of the intervals are not used as sample points. Hence the endpoints of the interval of integration, a and b are also not used as sample points. This allows certain improper integrals to be approximated. For example, LN(x) can be integrated over the interval (0, 1] even though LN(0) is undefined.

A refinement was implemented in the basic Romberg scheme. If uniformly spaced sample points are taken, periodic integrands may sometimes cause a problem due to resonance phenomena. In this program the sampling has been made non-uniform by a non-linear substitution. Such a substitution was implemented in the HP-34C calculator integration key routine.

A complete discussion of the theory of Romberg integration is given in References 2 and 3. Reference 8 provided the starting point for IG and users are urged to consult that work.

FORMULAS USED IN IG

(1) $$I = \int_a^b f(x)\,dx$$

There are three steps to the solution of (1) using the Romberg method.

A. Change of limits:

The interval of integration [a,b] is changed to the interval [-1,1] by the change of variable:

Let $x = [(b-a)/2]*t + (b+a)/2$, then $dx = [(b-a)/2]dt$

After substitution into the right side of (1) and simplifying we have:

(2) $$I = [(b-a)/2]*\int_{-1}^{1} f([(b-a)/2]*t + (b+a)/2)\,dt$$

B. Introduce non-uniform sample points:

Equation (2) is further refined by another change of variable which causes the sample points to be non-uniform over the original interval of integration.

Let $t = (3/2)*u - (1/2)*u^3$,

then $dt = (3/2)(1-u^2)du$

After substitution into the right side of equation (2) and some simplification we have:

(3) $I =$

$$3(b-a)/4*\int_{-1}^{1} f([(b-a)/4]u(3-u^2) + (b+a)/2)(1-u^2)du$$

Now a uniform distribution of sample points in u over the interval [-1,1] will be transformed to a non-uniform distribution of sample points x over the original interval of integration [a,b]. The Romberg extrapolation procedure is applied next to produce elements of a matrix M(k,k).

C. Generating the Romberg Matrix

(4) $$I = \lim_{k\to\infty} M(k,k) \qquad k=0,1,2,3,\ldots$$

where:

(5) $u_0 = -1 + 2^{-k}$

(6) $u_i = u_{i-1} + 2^{1-k}$

(7) $x_i = [(b-a)/4]u_i(3-u_i^2) + (b+a)/2$

(8) $S_0 = f((a+b)/2)$

(9) $$S_k = \sum_{i=0}^{2^k-1} f(x_i)(1-u_i^2) + S_{k-1}$$

(10) $M(k,0) = [3(b-a)/4]*2^{-k}*S_k$

and finally

(11) $$M(k,j) = M(k,j-1) + \frac{[M(k,j-1) - M(k-1,j-1)]}{4^j - 1}$$

The elements M(k,j) form a lower triangular matrix as follows:

M(0,0)					
M(1,0)	M(1,1)				
M(2,0)	M(2,1)	M(2,2)			
M(3,0)	M(3,1)	M(3,2)	M(3,3)		
M(4,0)	M(4,1)	M(4,2)	M(4,3)	M(4,4)	.
.	.	.	.	.	.
.	.	.	.	.	.

Equation (11) indicates that each element M(k,j) depends on the element immediately to its left, and on the element above the one immediately to its left. Only the most recent row M(k,0), M(k,1), M(k,2),... M(k,k) is stored in data registers R18 and up.

The program halts when two rounded consecutive diagonal elements M(k-1,k-1) and M(k,k) are equal.

When flag F10 is set, the first result to be displayed is (4/3)*M(0,0) which in fact is the element M(0,1). Subsequent displays show M(1,1), M(2,2), M(3,3), etc., until convergence occurs. M(0,0) is not displayed. The final display is M(k,k).

Analysis of a numerical example.

To further illustrate the Romberg method used in IG, the evaluation of the following function will be described in detail.

$$I = \int_0^1 (1-x^2)^{1/2}\,dx = \pi/4$$

$$= 7.853981635\text{-}01$$

The transformation from u to x is shown in Table 1, where linear samples in u, over the interval (-1,1), are transformed to non-linear samples in x over the interval (0,1). For k=0, the first point is u=0, at the center of the interval (-1,1). This then divides the interval in u into two new subintervals, (-1,0) and (0,1). The next iteration for k=1 samples u at the center of the new intervals at -1/2 and 1/2.

k	u	x
0	0	0.5
1	-1/2, 1/2	0.15625, 0.84375
2	-3/4, -1/4 1/4, 3/4	0.04297, 0.31641 0.68359, 0.95703
3	-7/8, -5/8, -3/8, -1/8, 1/8, 3/8, 5/8, 7/8	0.01123, 0.09229, 0.23193, 0.40674, 0.59326, 0.76807, 0.90771, 0.98877

TABLE 1
Transformation of u into x

k	M(k,0)	M(k,1)	M(k,2)	M(k,3)
0	0.64951905			
1	0.75351219	0.78817657		
2	0.77754585	0.78555708	0.78538244	
3	0.78344255	0.78540811	0.78539818	0.78539843
4	0.78490973	0.78539879	0.78539816	0.78539816
	M(4,4)=0.78539816			

TABLE 2
M Matrix Generation

This process is continued, with u being sampled at the center of each new interval. The sample point in u is then transformed to a sample point in x by equation (7) as described previously.

The evaluation of the M matrix is shown in TABLE 2. The elements M(k,0) are the improved midpoint rule estimates; the elements M(k,1) represent the first extrapolation, and in fact turn out to be the values obtained using Simpson's Rule improvements; the elements M(k,2) represent the second extrapolation, and in fact turn out to be the values obtained using the closed form, Newton-Cotes formula for five points.

In this program the element M(0,1) is the first element displayed if flag F10 is set, but is not used for any later calculations. M(0,0) is not displayed.

When FIX 6, SCI 5, or ENG 5 display modes are used in this example, the program halts after iteration k=4 since M(3,3) and M(4,4) agree to six significant digits. The element M(4,4) is returned to the X-register as the final solution.

Convergence of IG.

Convergence occurs, and execution halts, when two consecutive rounded diagonal elements, M(k,k), are equal. An advantage of the automatic Romberg Integrator is that no decision has to be made in advance concerning the optimum step size. The convergence criterion of IG is not as strict as that implemented in the HP-34C. In the HP-34C the program does not halt until three consecutive diagonal elements agree to the desired accuracy. Due to limited space, this criterion could not be implemented in the PPC ROM. For most integrals this difference will not be noticed, but it is possible that a few integrals evaluated by IG will halt prematurely.

Timing Data

IG execution time is mainly proportional to the number of times the f(x) subroutine is called. However, it is also proportional to the execution time of the f(x) routine, and the number of characters in the global label name of the f(x) routine. It is also governed by the location of the f(x) routine global label in program memory. The further the global label is from the bottom end of program memroy, the longer the execution time. At the end of the kth iteration, the function f(x) will have been called

$$2^{k+1} - 1$$

times. The time for k iterations in seconds is given approximately by:

$$T_k = 1 + 3.4K + (0.735 + t_l + t_f)*(2^{k+1} - 1)$$

where:

t_l = time to search and locate the f(x) global label.

t_f = time to execute the f(x) routine.

Each new iteration takes as long as all previous iterations. SIZE 030 will allow iterations up to k=11. The minimum time to complete this many iterations is approximately one hour.

The execution times given for all the previous numerical examples were obtained using a calculator which executed 500 "+" instructions in 15 seconds.

Routine Listing For: **IG**

```
01♦LBL "IG"
02♦LBL B
03 STO 17
04 X<>Y
05 -
06 4
07 /
08 STO 16
09 ST- 17
10 ST- 17
11 .
12 STO 15
13 STO 11
14 STO 18
15 SF 09
16♦LBL 01
17  E
18 2
19 STO 14
20 RCL 11
21 CHS
22 Y↑X
23 ST♦ 14
24  E
25 -
26♦LBL 02
27 STO 12
28 X↑2
29 -
30 STO 13
31 2
32 +
33 RCL 12
34 *
35 RCL 16
36 *
37 RCL 17
38 +
39 XEQ IND 10
40 RCL 13
41 *
42 ST+ 15
43  E
44 RCL 12
45 RCL 14
46 +
47 X<Y?
48 GTO 02
49 RCL 11
50 STO 13
51 18
52 STO 12
53  E
54 ST+ 11
55 RCL 15
56 RCL 16
57 1.5
58 *
59 *
60 RCL 14
61 *
62♦LBL 03
63 R↑
64 4
65 *
66 ENTER↑
67 DSE Y
68 X<> Z
69 ENTER↑
70 X<> IND 12
71 ST- Y
72 RND
73 X<> Z
74 /
75 RCL IND 12
76 +
77 ISG 12
78 STOP
79 DSE 13
80 GTO 03
81 STO IND 12
82 FS? 10
83 VIEW X
84 FS?C 09
85 GTO 01
86 RND
87 X≠Y?
88 GTO 01
89 LASTX
90 RTN
```

LINE BY LINE ANALYSIS OF IG

Lines 01-10 intialize the program by storing the constants (b-a)/4 and (b+a)/2 in R16 and R17 respectively.

Lines 11-14 initialize S_k, k, and M(k,k) for k=0.

Line 15 is used to force the program to run through at least two iterations. See also lines 84 and 85.

Lines 16-25 calculate u_0 and the step size 2^{k-1}.

Lines 26-48 calculate x_i (line 38), $f(x_i)$, (line 39), and S_k (line 42).

Lines 49-61 calculate M(k,0).

Lines 62-80 calculate M(k,j).

Lines 86-88 are the exit test. The routine ends when two consecutive rounded approximations are equal.

Lines 89-90 recall the final approximation and halt.

REFERENCES FOR IG

1. W. Romberg, "Vereinfachte Numerische Integration," NORSKE VIDENSKAB. SELSKAB., FORH. (TRONDHEIM) 28, No. 7, 1955.

2. F.L. Bauer, H Rutishauser, E.L. Stiefel, "New Aspects in Numerical Quadrature," in EXPERIMENTAL ARITHMETIC, HIGH SPEED COMPUTING AND MATHEMATICS, pp. 199-218, American Mathematical Society, Providence, Rhode Island, 1963.

3. E.L. Stiefel, AN INTRODUCTION TO NUMERICAL MATHEMATICS, Academic Press, New York, 1963.

4. Abramowitz and Stegun, HANDBOOK OF MATHEMATICAL FUNCTIONS, National Bureau of Standards, Applied Mathematics Series, No. 55, 1964

5. P. J. Davis and P. Rabinowitz, METHODS OF NUMERICAL INTEGRATION, Academic Press, New York, 1975.

6. John Kennedy, PPC Journal, V6N5P18, August 1977.

7. Irving Allen Dodes, NUMERICAL ANALYSIS FOR COMPUTER SCIENCE, North-Holland, New York, 1978.

8. William M. Kahan, "Handheld Calculator Evaluates Integrals," Hewlett-Packard Journal, August 1980.

9. Read Predmore, PPC CALCULATOR JOURNAL, V7N6P4, July-August 1980.

10. Read Predmore, PPC CALCULATOR JOURNAL, V7N9P13, November 1980.

11. OWNER'S HANDBOOK AND PROGRAMMING GUIDE, Hewlett-Packard HP-34C Calculator, Hewlett-Packard Company.

CONTRIBUTORS HISTORY FOR IG

Several numerical integration methods are in popular use today. References 5 and 8 provide insight into the performance limitations of some of these methods. Other methods may be faster than the iterative Romberg method, but the Romberg method is very efficient in that it uses all previously calculated estimates and it is automatic in the sense that step-size information does not have to be provided. Also, error propogation is reduced by the matrix calculation procedure which in turn greatly speeds convergence.

Using References 5 and 8 Read Predmore (5184) produced a very efficient, compact, and elegant HP-41C program (Reference 10) using the iterative Romberg method. That program was almost identical in function to the routine underlying the integrate key on the HP-34C calculator as described in Reference 8. The Reference 10 program formed the basis for IG.

John Kennedy (918) reduced the program size to its final form. Harry Bertucelli (3994) suggested register usage to allow IG to be used with SV. The documentation on IG was written by Graeme Dennes (1757) after proof reading by Read Predmore (5184). Thanks to the Hewlett-Packard Company for allowing the reproduction of numerical examples from Reference 8.

FINAL REMARKS FOR IG

By adding some enhancements IG could be improved to make the procedure more closely fit the complete method used in the HP-34C, the first calculator to have an integration routine as a built-in function. Speed is also an area of needed improvement.

FURTHER ASSISTANCE ON IG

Read Predmore (5184) phone: (413) 367-9513

Graeme Dennes (1757) phone (415) 592-2957 evenings

NOTES

TECHNICAL DETAILS

XROM: 20, 09 | IG | SIZE: 030 minimum

Stack Usage:

0 T: used
1 Z: used
2 Y: used
3 X: used
4 L: used

Alpha Register Usage:

5 M: not used
6 N: not used
7 O: not used
8 P: not used

Other Status Registers:

9 Q: not used
10 F: not used
11 a: not used
12 b: not used
13 c: not used
14 d: not used
15 e: not used

ΣREG: not used

Data Registers:

R10: function LBL name
R11: k = counter
R12: u_1
R13: $1 - u_i^2$
R14: delta u = 2^{1-k}
R15: S_k
R16: (b-a)/4
R17: (b+a)/2
R18: M(k,0)
R19: M(k,1)
R20: M(k,2)
⋮ ⋮

Flag Usage:

04: not used
05: not used
06: not used
07: not used
08: not used
09: used to force two iterations
10: set to display approximations
25: not used

Display Mode:

SCI n recommended

Angular Mode:

not used, but may be required by function

Unused Subroutine Levels:

4

Global Labels Called:

Direct	Secondary
function LBL in R10	

Local Labels In This Routine:

B, 01, 02, 03

Execution Time: see IG documentation for detailed timing information.

Peripherals Required: none (printer recommended)

Interruptible? yes

Execute Anytime? no

Program File: IG

Bytes In RAM: 131

Registers To Copy: 43

Other Comments:

X<>O, X<>N, X<>M, RDN, RTN uses only 17 bytes... Any advance? (Notice that these won't replace my original alpha slicers, which were designed for use in right-justification for printing, and placed an alpha space in Y, with the removed RH alpha character in X ready for a space detect conditional. See PPCTNV1N3P36, and ditto, N5P48.) How about one of these for 24 - or even 28 characters? For that one may, in the manner of a compact with the devil, not so much sell one's soul, as blow one's stack - and on that note, forget the stack in this last routine, sacrifice X, and reduce the bytes to 15. Better still, don't call it as a subroutine at all, and then we are down to 13! (When I was quite small, SIZEd about ØØ8, the fad was to chew every byte 32 times. The compulsion comes through in one's dotage.)

One last observation, helping to fill this page, amongst other things. Using this method of shunting the alpha register contents to the left leaves the way open to control of the number of characters sliced from the right, and also, possibly, alters the number that will be lost from the left. (Two of the normally usable 24, remembering that one can use 28 with care - see PPCTNV1N3Pp.36-40.) If lines 03 and 04 in the second revision above are reversed, and the FIX 4 is changed to FIX IND X, entry with 3 in X will slice two characters of 23 in alpha (only one of 24 would be lost from the left), entry with 2 in X will slice 3 of 24, 1 in X slices 4 (but may add a junk byte from P on the left), Ø slices 5. Add then a CF 29, and a zero will slice 6 (and add two junkies with pinky leanings). 7, of course, is easy: have a flag test before the ARCL X. When false, 7 characters, perhaps shady, disappear from the face of the 41c...

Alpha McGechie (CFC)

A small town country newspaper had the (bad) habit of filling up the foot of its columns with things like: This line fills this space. We know rather better: we say "This space fills this line".

**Fortunately for the State of our Art, this is wrong! Figure out what, what is the truth, and then find a use for what you then know. Solutions to be found in TN#7.*

R/S

BUG 9 COMMENTS

COMMENTS ON BUG 9 — *Charles Close (3878)*

Sometimes one of the things we are trying to do here in Melbourne, really works. For the last few days, I have been bombarded with letters (3, actually) from Charles. Yesterday's arrival, mailed on March 12th, contained this valuable increment to our collective knowledge. Keep up this way, and we will even be able to use bug 9 for some useful purpose! To Charles' opening remark you should add congratulations to Valentin Albillo (remember him?!), whose independent discovery and exploitation recipes for bug 9 appeared in TN #5. (Some of that had to be PRIVATE, for reasons given there.) I have a note from someone else, but have for the moment lost the reference (I don't have CONTINUOUS MEMORY, and sometimes suffer from MEMORY LOST when I emerge from SLEEP MODE), that placing a global label at the top of the first file of program memory, stopping at that, and executing DEL ØØ1 will delete it and achieve the same END as Charles describes below. This is implicit in Valentin's article in TN #5. (See p.36 near the foot.) Some of Charles' observations are quite new, and even if there was nothing here not already in print, it would still be worthy of appearing, if only for his characteristic, individual way of seeing and presenting these things.

Congratulations to Mark MacLean, #5519! His discovery of bug 9 provides a excellent solution to the problem of gaining access to the H-P 41C user key registers and status registers with a bug free (?) calculator. Further examination of this bug reveals that Master Clear is not necessary to gain access to the user key and status registers. The critical parameters are the address of register zero (ØØ) and the location of the permanent ".END.".

To gain access to the user key and status registers, place a "END" at the beginning of the program registers, just before the first step of the first program in memory. Next, execute "CAT 1", stop the listing at the previously inserted "END" with PRGM mode on and proceed with the instructions in PPCCJ V7N9P25 starting with DEL ØØ1. Access cannot be gained if any part of the registers allocated to program, the space between the data register zero (ØØ) and the permanent ".END.", occupies machine register hex 111. In this case the program pointer goes to RAM address Ø112 (Displays step at RAM address 6111) after the first DEL ØØ1. If the permanent ".END." is at a address greater than register hex 111, the program pointer will go to the first byte in the user key registers after the first DEL ØØ1 (GTO .ØØ1 not required). If the address of register zero (ØØ) is less than hex 112, then the remaining instructions in PPCCJ V7N9P25 apply, i.e., DEL ØØ1 twice and GTO .ØØ1.

Now users without the Card Reader and/or Wand need not lose programs in order to gain access to the user key and status registers if no synthetic instructions are loaded. Users without memory modules need not lose programs while accessing these registers since the address of register zero (ØØ) cannot exceed hex ØFF.

Congratulations again Mark, you did what I would like to have done. CHARLES CLOSE #3878

We now have several bug 9 ways of gaining access to status from scratch. A year ago bug 2 was the only one known. Bill Wickes found RAM module pulling, Keith Jarett above adds ROM module pulling, and below (I hope), there is the wand method discovered (where else?) here in January by our youngest member, Tony Burton, who seems well on the way to becoming our local wand expert. Jim Trainor, of course, can do it in a twinkling flash, but it is not certain that all 41c's respond in the same way to his kind of handling.

R/S

HP-41 DOUBLE INTRGRALS

HP-41 **DOUBLE INTEGRALS**

Ernest (4610) Gibbs

This routine numerically integrates the double integral

$$I = \int_{x_0}^{x_1} \int_{y_0}^{y_1} F(x,y)\,dx\,dy$$

using a method of adaptive quadrature. The routine is suitable for peaked or unbounded integrands, but remains efficient for well-behaved functions.

Method

To evaluate I, the limits of integration are first changed to 0 and 1 by changing variables,

$$I = \int_0^1 \int_0^1 g(x,y)\,dx\,dy$$

with $\Delta x = x_1 - x_0$, $\Delta y = y_1 - y_0$,

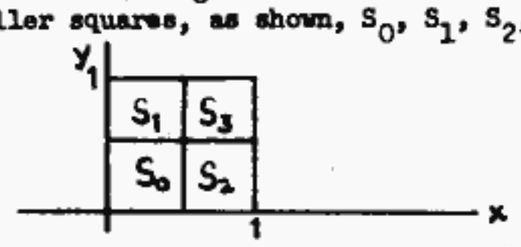
and $g(x,y) = F(x_0 + x.\Delta x\ ,\ y_0 + y.\Delta y)$.

The integration is then accomplished by application of the formula,

$$\int_{a-h}^{a+h} \int_{b-h}^{b+h} g(x,y)\,dx\,dy \approx \frac{h^2}{3}\Big[8g(a,b) + g(a+h,b+h) + g(a+h,b-h) + g(a-h,b-h) + g(a-h,b+h)\Big] \quad \ldots(1)$$

The formula is exact if g(x,y) is any polynomial of degree less than four.

First apply (1) with a=b=h=0.5 . This gives an approximate value I_1 for the integral. Then subdivide the unit square into four smaller squares, as shown, S_0, S_1, S_2, and S_3.



Then apply (1) to each of the smaller squares with h=¼ , add the results and substitute for the original contribution from just S_i to obtain a new approximate integral I_2. If the change from I_1 is too large, subdivide S_i into four parts and repeat the process. This process is continued until either the desired accuracy is obtained, or a preset number of subdivisions, called LEVEL, is reached. At this point we subdivide no further, but return to the remaining squares of varying size, on each of which the whole process is repeated.

This "adaptive" process means that many subdivisions are used in regions of singularities in F(x,y) , and fewer subdivisions in regions of good behaviour of F.

The desired accuracy should be able to be selected from either of absolute ($|I_i - I_{i-1}|$) or relative ($|(I_i - I_{i-1})/I_{i-1}|$) comparisons with the previously calculated integral value. Note that the desired accuracy may not be obtained if the value of LEVEL is too small. On the other hand, if LEVEL is too large, computation time may be exceedingly long.

$$\int_a^{a+h'} \int_b^{b+h'} g(x,y)\,dx\,dy \quad .$$

for a singularity about the point (a,b) . The examples which follow should illustrate this.

Examples

Three examples illustrate the accuracy of the method, and give an indication of the times for execution for various values of LEVEL and FIX. Examples 1. and 2. are well-behaved, whilst 3. has an infinite singularity at x=y=0 .

1. $F(x,y) = x+y$ $\quad x_0 = 1$
   $= 9$ (exactly) $\quad y_0 = 2$
   $x_1 = y_1 = 3$
2. $F(x,y) = y.\cos(\pi xy)$
   $= 2/\pi^2$ $\quad x_0 = y_0 = 0$
   $x_1 = y_1 = 1$

3. $F(x,y) = \begin{cases} (x+y)^{-1}, & x \neq 0, y \neq 0 \\ 0, & x=y=0 \end{cases}$

$= \ln 4 \qquad x_0 = y_0 = 0$

$x_1 = y_1 = 1$

The table below lists the results.

Example	Known value	FIX	error type	LEVEL	Calc. value	Time (mins)
1	9	3	abs	5	9.000000000	1.0
2	0.202642367	3	rel	5	0.202662	17
		3	abs	5	0.2027	13
		4	abs	5	0.20265	24
3	1.38629436	3	abs	5	1.371	24
		2	abs	6	1.379	12
		3	abs	8	1.3845	33

Description of program

Lines 01 to 09 and LBL 14 account for the change in variable. Register R_{11} holds the LEVEL index, and this is computed from the input maximum LEVEL (stored in R_{11} before execution) at lines 11 to 15. The initial integral I_1 is calculated from lines 16 to 21 and stored in R_{05} at line 22. LBL 10 (lines 114 - 145) computes the integral function (1).

For subsequent subdivisions the integrals are computed at lines 25 - 65 , with the accuracy comparison being made at line 49. LBL 08 at lines 73-113 calculate the appropriate h, a, and b for the current subdivision. The information concerning the current state of subdivision is stored in a cascade of indices in registers R_{12} and upwards. Each index varies over the range 0 to 3. Although this seems wasteful at the moment, it is intended to expand the program to evaluate triple integrals, and this approach will be useful then,

The highest register used by this program is dependant upon the value of LEVEL selected. R_{max} = 11 + LEVEL , and so

SIZE = 12 + LEVEL

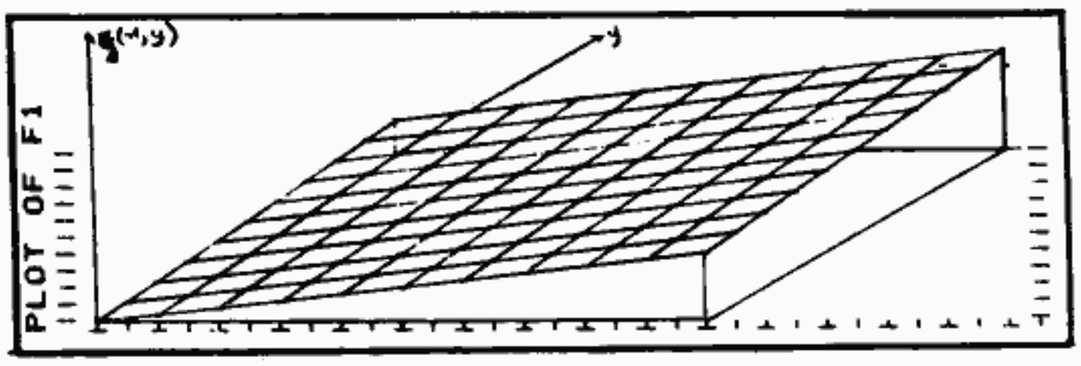
Using the program

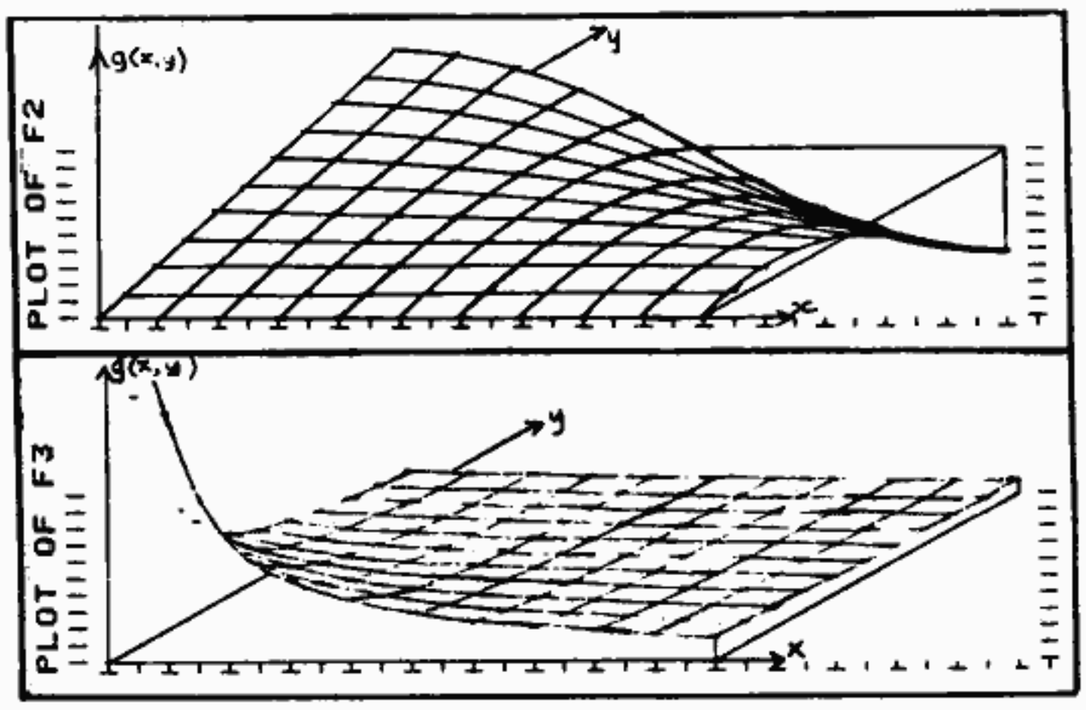
1) The integrand function must have been previously written and stored under a global label. It is executed from "DI" with x in X register and y in Y. These are also available in R_{00} and R_{01} respectively. If the function requires intermediate storage, the registers above R_{max} are available, as well as the synthetic alpha register R_0.
2) SIZE as required ($\geq$ 12 + LEVEL)
3) Within the calling program, or before execution, store the following information:
   i) set/clear flag 00 according as whether absolute/relative accuracy required;
   ii) ASTO function name in R_{06};
   iii) FIX to appropriate number of decimal places. "DI" attempts to achieve the integral correct to the FIX setting;
   iv) store LEVEL, the maximum number of subdivisions, in register R_{11};
   v) place in the stack the limits of integration, in the order x_0, x_1, y_0, y_1;
   vi) XEQ "DI".
4) The evaluated integral is returned in the X register, as well as in the user register R_{05}.

The selection of an appropriate FIX and level can alter the execution time enormously, as well as affecting the accuracy of the result. Note that the smallest value of h which will be used is

$$h' = 1 / {}_2(\text{LEVEL} + 1)$$

and that this will result in the iteration effectively throwing away an amount given by





"DI" Double Integrals by adaptive quadrature — Ernest Gibbs (4610)

```
01+LBL "DI"
02 X<>Y
03 STO 00
04 -
05 STO 10
06 X<> Z
07 STO 07
08 -
09 STO 09
10 RCL 11
11  E3
12 /
13 12.011
14 +
15 STO 11
16 .5
17 STO 02
18 STO 03
19 STO 04
20 ENTER↑
21 XEQ 10
22 STO 05
23 CLA
24 SF 01
25+LBL 00
26 .003
27 STO IND 11
28 FS?C 01
29 GTO 01
30 DSE 11
31 --
32 XEQ 08
33 ST- [
34 ISG 11
35+LBL 01
36 XEQ 08
37 ST+ [
38 ISG IND 11
39 GTO 01
40 RCL [
41 ENTER↑
42 X<> 05
43 -
44 LASTX
45 FC? 00
46 ST/ Y
47 RDN
48 RND
49 X=0?
50 GTO 02
51 .003
52 STO IND 11
53 ISG 11
54 GTO 00
55 DSE 11
56+LBL 02
57 RCL 11
58  E
59 -
60 ISG IND X
61 GTO 00
62 STO 11
63 13
64 X<=Y?
65 GTO 02
66 RCL 05
67 RCL 09
68 RCL 10
69 *
70 *
71 CLA
72 RTN
73+LBL 08
74 RCL 11
75 INT
76 .011
77 +
78 STO \
79 RCL d
80 STO ]
81  E
82 ENTER↑
83 ENTER↑
84+LBL 09
85 RCL IND \
86 INT
87 STO d
88 RDN
89 ST+ X
90 FS? 06
91 ST+ Z
92 FS? 07
93 ST+ Y
94 DSE \
95 GTO 09
96 RDN
97 RCL ]
98 STO d
99 RDN
100 RCL 11
101 INT
102  E1
103 -
104 2
105 X<>Y
106 Y↑X
107 1/X
108 STO 02
109 ST+ Z
110 *
111 STO 04
112 X<>Y
113 STO 03
114+LBL 10
115 XEQ 14
116 0
117 *
118 STO \
119 SF 03
120+LBL 11
121 SF 04
122+LBL 12
123 RCL 04
124 RCL 02
125 FC? 03
126 CHS
127 +
128 RCL 03
129 RCL 02
130 FC? 04
131 CHS
132 +
133 XEQ 14
134 ST+ \
135 FS?C 04
136 GTO 12
137 FS?C 03
138 GTO 11
139 RCL 02
140 X↑2
141 3
142 /
143 RCL \
144 *
145 RTN
146+LBL 14
147 RCL 09
148 *
149 RCL 07
150 +
151 STO 00
152 X<>Y
153 RCL 10
154 *
155 RCL 00
156 +
157 STO 01
158 X<>Y
159 XEQ IND 06
160 END
```

LBL"DI
END
245 BYTES

R/S

TIP

2 Way Metric Conversion Chart Program
by: Richard Collett (4523) - the Whizz Kid & Phill Jury (5484) - The Synthetic Text Labourer.

This program will produce any conversion chart starting at any point & finishing at any point, with an increment from .1 up. Furthermore you can change the increment at any point in your required chart. To this point we have made up subroutines to produce the following charts:

1. Inches - milimetres
2. Feet - Metres
3. Yards - Metres
4. Miles - Kilometres
5. Celsius - Fahrenheit
6. Celsius - Kelvin
7. Ounces - grams
8. Pounds - Kilograms
9. Tons - Tonnes
10. Stone - Kilograms
11. Fluid Ounces - mililitres
12. Pints - Litres
13. U.S. Gals - Litres
14. Imp. Gals - Litres

Any member wanting a copy of the above program & subroutines, please contact: Phill Jury on 509 - 0155 A.H.

Celsius		Fahrenheit
-18.9	-2.0	28.4
-18.3	-1.0	30.2
-17.8	0.0	32.0
-17.2	1.0	33.8
-16.7	2.0	35.6
-16.1	3.0	37.4
-15.6	4.0	39.2
-15.0	5.0	41.0
-14.4	6.0	42.8
-13.9	7.0	[illegible]
-13.3	8.0	46.4
-12.8	9.0	48.2
-12.2	10.0	50.0
-6.7	20.0	68.0
-1.1	30.0	86.0
4.4	40.0	104.0
10.0	50.0	122.0
15.6	60.0	140.0
21.1	70.0	158.0
26.7	80.0	176.0
32.2	90.0	194.0
37.8	100.0	212.0

BY
P.JURY, [illegible]
AND
R.COLLETT, [illegible]