HP-41 Module: Non-Linear Systems

Overview

This module includes a selection of programs from Jean-Marc Baillard's collection with the "non-linear" common theme as selection criteria. There are basically three categories in the module:

1. Non-Linear Systems of equations
2. Nth. Degree Differential Equations
3. Laplace, Poisson and Diffusion Equations.

It comes without saying that this module wouldn't exist without Jean-Marc's excellent core routines. The topics are wide and the selection is necessarily incomplete but the goal was to make it fit into a single ROM with a 4k footprint. Every effort has been made to make the operation as automated and simplified as possible. Many driver programs are included for a more convenient access, with automated data entry and output. Most of the chapters also include examples of utilization pre-programmed in the module to quickly familiarize the user in the techniques. All in all, this module is as close to a "plug and play" solution as it gets.

So here you have it, an interesting foray into subjects not typically treated by calculators but that so well showcases the power and versatility of the HP-41 with some masterful programming behind it.

Module Programs by functional area.

Rather than providing a linear list of functions (pun intended), the following tables show a summary of all programs in the module, grouped by the functional area they belong to. This should provide some clarification as to the scope and role of each function.

1. *non-linear systems routines,* - http://hp41programs.yolasite.com/system-eq.php

System	Driver Prog	Math Routine	Aux Routines	Examples
2 Equations	**2NLS+**	**2NL**	n/a	**FG**
3 Equations	**3NLS+**	**3NL**	n/a	**FGH**
n-Equations	**NNLS+**	**NLS, LS**	**FIN, XIN, XOUT**	**F1, F2, F3**
n-Equations (Advtg)	**NLSYS**	**MSYS**	**FIN, XIN, XOUT**	**F1, F2, F3**

2. *numerical analysis routines,* - http://hp41programs.yolasite.com/poisson.php

System	Driver Prog	Math Routine	Examples
Laplace Eq.	**LAP+**	**LAP**	**VLX, VX0, CLX**
Poisson Eq.	**POIS+**	**POIS**	**UX0, UY0, UXL, ULY, FXY**
Diffusion Eq.	**DFSN+**	**DFSN**	**aXT, bXT, cXT, FX0, F0T, FLT**

3. *differential equations routines*, - http://hp41programs.yolasite.com/n-thorderdifeq.php

System	Driver Prog	Math Routine	Examples
Second Order	**2DFEQ**	**2RGK**	**d2/dX2** (Lane-Emden)
Third Order	**3DFEQ**	**3RGK**	**d3/dX3**
Nth. Order	**4DFEQ**	**NRGK**	**d5/dX5**

4.*Successive Approximation routines*, - http://hp41programs.yolasite.com/approx.php

System	Driver Prog	Math Routine	Examples
Real	**XSAM**	**FXN**	**X=, Y=, Z=**
Complex	**ZSAM**	**FZN**	**Z1=, Z2=**

List of functions from CAT 2:

Driver programs are shown in bold, underlined font. Section headers have no functionality. The Library#4 is not required, but some dependencies exist, like function **PMTA** (from the OS/X Module) is used by the Laplace and Poisson driver programs.

-NON LINEAR

"2LS"
"**2NLS+**"
"3NL"
"**3NLS+**"
"LS"
"NLS"
"**NNLS+**"
"FIN"
"XIN"
"XOUT"
"FXN"
"**XSAM**"
"FZN"
"**ZSAM**"
"2RK4"
"**2DFEQ**"
"3RK4"
"**3DFEQ**"
"NRK4"
"**NDFEQ**"

-NMRCL MTH

"**DFSN+**"
"DFSN"
"3DLS"
"aXT"
"bXT"
"cXT"
"FX0"
"F0T"
"FLT"
"**LAP+**"
"L"
"**POIS+**"
"POIS"
"FXY"
"OUT"
"UX0"
"UY0"
"UXL"
"ULY"

-TOOLTIPS

AIP
E3/
E3/E+
"CLX"
"F1"
"F2"
"F3"
"FG"
"FGH"
"VLX"
"VX0"
"X="
"Y="
"Z="
"Z1="
"Z2="
"d2/dX2"
"d3/dX3"
"d5/dX5"

1.- 2 Equations in 2 Unknowns: f(x,y) = 0 ; g(x,y) = 0

"**2NL**" uses the Newton's iterative method - more exactly a generalization of the secant method to approximate the partial derivatives. It requires 2 initial guesses (x , y) and (x' , y') which are to be stored in R01 , R02 and R03 , R04 respectively. You must choose x # x' and y # y' (very important). Registers R00 thru R11 are used by the program.

You also have to load a subroutine that takes y in Y-register and x in X-register and calculates f(x;y) in Y-register and g(x;y) in X-register. Registers from R11 and greater are available for this job. Thus, your subroutine changes the stack as follows:

Stack	From	To
Y	y	f(x,y)
X	x	g(x,y)

To call "**2NL**" you store the name of this subroutine into R00 (global label of 6 characters maximum) and XEQ "2NLS". The successive x-values are displayed and when the program stops the results are arranged as follows:

- x is in X-register and in R01
- y is in Y-register and in R02
- Z-register contains | f(x,y) | + | g(x,y) |
- f(x,y) is in R05
- g(x,y) is in R06

Stack	Inputs	Outputs
Z	/	|f|+|g|
Y	/	y
X	/	x
ALPHA	FNAME	FNAME

To find another solution, re-initialize R01 thru R04 and R/S .

Warning: The program stops when the approximate Jacobian determinant equals zero. This happens when x =x' or y = y' but it may also happen by a stroke of bad luck, for instance if x converges much more quickly than y to the solution. That's why it's always wise to check the value of ε =|f| +|g| .

For your convenience, the module also includes "**2NLS+**", a driver program that prompts you through all the data entry process (including the name for the function subroutine), and then redirects the execution to "**2NL**" . Furthermore, the example below is also pre-programmed under the label "**FG**", so you can check the results and get familiar with the operation.

Example: Find x and y such that $x.y = 7$ and $x^2 + y^4 = 30$ with (2; 2) (3; 3) as initial guesses

XEQ "2NLS+"	"FG(X,Y)?"	prompts for function label
"FG", R/S	"G1=? Y^X"	first guess pair
2, ENTER^, 2, R/S	"G2=? Y^X"	second guess pair
3, ENTER^, 3, R/S	*successive approximations, then convergence:*	
	X=3.368200265,	also stored in R01
R/S	Y=2.078261222,	also stored in R02

And this is how the function has been programmed:

```
01 LBL "FG"
02  RCL Y
03  RCL Y
04  *
05  7
06  -
07  X<>Y
08  X^2
09  R^
10  X^2
11  X^2
12  +
13  30
14  -
15  RTN
```

2.- 3 Equations in 3 Unknowns: $f(x,y,z)= g(x,y,z)= h(x,y,z)= 0$

"**3NL**" uses the same method for solving a system of 3 non-linear equations. Registers R00 thru R20 are used by this program. It also requires two initial guesses (x , y , z) and (x' , y' , z') which are to be stored into {R01 , R02 , R03} and {R04 , R05 , R06} respectively (ensuring that x#x' , y#y' , z#z').

You must write a subroutine that takes x from the X-register; y from the Y-register; and z from the Z-register to compute f(x,y,z) in Z-register, g(x,y,z) in Y-register, h(x,y,z) in X-register:

Stack	From	To
Z	z	f(x,y,z)
Y	y	g(x,y,z)
X	x	h(x,y,z)

To call "**3NL**" you store the name of this subroutine into R00 (global label of 6 characters maximum) and XEQ "3LS". The successive x-values are displayed and when the program stops the results are arranged as follows:

- x is in X-register = R01
- y is in Y-register = R02
- zis in Z-register = R03
- T-register contains |f(x,y,z)| + |g(x,y,z)| + |h(x,y,z)|
- f(x,y,z) = R07
- g(x,y,z) = R08
- h(x,y,z) = R09

Stack	Inputs	Outputs
T	/	|f|+|g|+|h|
Z	/	z
Y	/	y
X	/	x
ALPHA	FNAME	FNAME

For your convenience, the module also includes "**3NLS+**", a driver program that prompts you through all the data entry process (including the name for the function subroutine), and then redirects the execution to "**3NL"** . Furthermore, the example below is also pre-programmed under the label "**FGH**", so you can check the results and get familiar with the operation.

Good guesses are not always easy to find but in any case, always check the values of f , g , h – or at the very least the value in the T-register!

Example: Find a solution of the system below, with (2,2,2) and (1,1,1) as initial approximations:

$$x.y^2 - z/y = 0$$
$$x - y - z = 0$$
$$\ln x + y.z = 0$$

XEQ "3NLS+"	"FG(X,Y,Z)?"	prompts for function label
"FGH", R/S	"G1=? Z^Y^X"	first guess pair
2, ENTER^, ENTER^, R/S	"G2=? Y^X"	second guess pair
3, ENTER^, ENTER^, R/S	*successive approximations, then convergence:*	
	X=0.865408832,	also stored in R01
R/S	Y=0.639295476,	also stored in R02
R/S	Z=0.226113356,	also stored in R03

3.- N Equations in N Unknowns: $fk(x_1,... ,x_n) = 0;\ k=1...n$

In this case we deal with the general case of a system of (n x n) non-linear equations. This task will need to call "**LS**" as a subroutine – which therefore has been added to the module as well. The main routine is "**NLS**".

Once again, the same (quasi-) Newton's method is used: each iteration solves a linear system of n equations in n unknowns and that's why "**LS**" is needed. Unlike "**2NL**" and "**3NL**", it's of course impossible to take the n variables from the stack and calculate the n functions in the stack if n > 4, therefore you'll have to key in n different subroutines for computing the f_i in the X-register with x_1 in R01, x_2 in R02, ... , and x_n in Rnn

Synthetic registers {M N O} and data registers R00 thru Rn^2+4n are used by the program. It also requires two initial guess-vectors (x1, x2,..xn) and (x1', x2',… xn') which components are to be stored into {R01- Rnn} and {Rnn+1 to R2n} respectively (and ensuring that $x_i \# x'_i$ for i = 1 , 2 , ... , n).

The successive x_1-values are displayed during the calculations, and when the program stops, $| f_1 | + ... + | f_n |$ is in the X-register; and the solution (x_1 , ... , x_n) is in { R01 , ... , Rnn }.

The table below summarizes the data input requirements for "**NLS**":

Register	*Value*		*Register*	*Value*		*Register*	*Value*
R00	n						
R01	x1		Rnn+1	x1'		R2n+1	F1 Name
R02	x2		Rnn+2	x2'		R2n+2	F2 Name
…..	…..		…..	…..		…..	…..
Rnn	xn		R2n	xn'		R3n	Fn Name

The module includes several auxiliary routines to make the complete process more convenient. First off, the driver program "**NNLS+**" will present all needed prompts for the input parameters automatically, storing them in the appropriate data registers. Within the driver program there are calls to other utilities to input the initial guesses ("**XIN**") and the function names ("**FIN**"). Finally, after the system has been resolved, the driver program will invoke a data-output routine ("**XOUT**") to show the results. All this will happen transparently to the user.

Also for your convenience the module includes a practical example, programmed with the routines "**F1**", "**F2**", and "**F3**", defined and programmed as follows:

f1(x,y,z) = x^2 + y – 3
f2(x,y,z) = y^2 - z -1
f3(x,y,z) = x - z^2 + 8

The three solutions are:
x3 = 3, x2 = 2, x1 = 1

```
LBL "F1"      LBL "F2"      LBL "F3"
RCL 01        RCL 02        RCL 01
X^2           X^2           RCL 03
RCL 02        RCL 03        X^2
+             -             -
3             E             8
-             -             +
RTN           RTN           END
```

Go ahead and execute **NLSN+**, using (1, 1, 1) and (2, 2, 2) for the initial guesses for the values.

*(*) Note that you cannot use **FG** of **FGH** as examples, because their data input/output conventions are different and not compatible with **NLNS**.*

4.- N Equations in N Unknowns w/ Advantage Pac

Another version of the same program is available in the "Advantage Math" Module. This alternative version was written by Greg McClure, and uses the Advantage Pack's (or SandMatrix') **MSYS** to solve the linear systems involved in the resolution of the non-linear problem. Refer to the module documentation for details and examples.

5.- Successive Approximations Method.

A change in the paradigm, the Successive approximations method can also be used to solve systems of non-linear equations, as it's described in this section. The basis of the method is the following theorem:

If F is a contraction mapping on a complete metric space, then the equation F (X) = X has a unique solution, which is the limit of the sequence: $X^{(k+1)} = F(X^{(k)})$ *where* $X^{(0)}$ *is arbitrary.*

This theorem is used in the following program to solve a system of n equations in n unknowns written in the form:

$$
\begin{aligned}
x_1 &= f_1 (x_1 , \ldots , x_n) \\
x_2 &= f_2 (x_1 , \ldots , x_n) \\
&\ldots\ldots\ldots\ldots\ldots \\
x_n &= f_n (x_1 , \ldots , x_n)
\end{aligned}
$$

The advantage of this method is its simplicity: there is no need to solve a n x n linear system like with "**NLS**" (see previous sections), but unfortunately, it's not always easy to find the good function F that leads to convergence. When it does, it can be used to solve large linear or non-linear systems, with n > 16 , which otherwise would be impossible to solve on an HP-41.

The routines "**XSAM**" and "**ZSAM**" are driving programs for the real and complex cases of the successive approximation method, **"FXN"** and "**ZXN**" respectively. As usual, the driver programs automate the data entry process, prompting for the needed initial conditions and guesses.

Both programs require an initial guess, i.e. a vector of n components with initial estimations for the solution. Besides, the user must program the n functions linking the variables in the form shown above. These will assume the variables X1,… Xn are to be taken from data registers R01, … Rn.

The program calculates a new estimation $(x'_1 , \ldots , x'_n)$ from the precedent$(x_1 , \ldots , x_n)$ by using the formulae:

$$
\begin{aligned}
x'_1 &= f_1 (x_1 , \; x'_2 , \ldots , x'_{n-1} \; , x'_n) \\
x'_2 &= f_2 (x_1 , \; x_2 , \ldots , x'_{n-1} \; , x'_n) \\
&\ldots\ldots\ldots\ldots\ldots \\
x'_{n-2} &= f_{n-1} (x_1 , \; x_2 , \ldots , x'_{n-1} \; , x'_n) \\
x'_{n-1} &= f_{n-1} (x_1 , \; x_2 , \ldots , x_{n-1} \; , x'_n) \\
x'_n &= f_n \; (x_1 , \; x_2 , \ldots , x_{n-1} , x_n)
\end{aligned}
$$

In other words, every new estimation of x_i replaces the previous one as soon as it is computed. Therefore, the program is shorter, it requires less registers and convergence is improved. This however comes at a price: not surprisingly the execution takes longer than the explicit cases.

Example 1. Solve the system of three real equations shown below, re-writing it as a successive approximation case getting each variable in the first term. Use the vector (2, 2, 2) as initial guess.

$$\begin{aligned} x^2 / y + y / z - z^2 &= 0 \\ x.y.z - x - y^2 &= 0 \\ x.\ln y - z &= 0 \end{aligned} \quad \Rightarrow \quad \begin{aligned} x &= z / \ln y \\ y &= (x.y.z - x)^{1/2} \\ z &= (x^2 / y + y / z)^{1/2} \end{aligned}$$

For your convenience, these three functions are already pre-programmed in the module as follows:

```
01  LBL "X="        10  *             19  RCL 02
02  RCL 03          11  RCL 01        20  ST/ Y
03  RCL 02          12  ST* Y         21  RCL 03
04  LN              13  -             22  /
05  /               14  SQRT          23  +
06  RTN             15  RTN           24  SQRT
07  LBL "Y="        16  LBL "Z="      25  END
08  RCL 02          17  RCL 01
09  RCL 03          18  X^2
```

And here's the sequence of instructions using "**XSAM**". Note that each prompt already suggests the current content in the relevant register. If this is already correct, all you need to do is press R/S to continue with the data entry process.

```
XEQ "XSAM"          "N=?"
3, R/S              "F#1? 0.0000",  it shows the current value in R04, ALPHA is ON.
"X=", R/S           "F#2? 0.0000".  ditto as above, with R05
"Y=", R/S           "F#3? 0.0000",  same as above, with R06
"Z=", R/S           "X0(1)=4.0030?"
2, R/S              "X0(2)=7.1000?"
2, R/S              "X0(3)=0.0000?"
2, R/S              successive iterations shown, then convergence:

                    X3= 1.703912160  in R03
R/S                 X2=2.457696043   in R02
R/S                 X1=1.894868809   in R01
```

Example 2. Solve the system of two complex equations shown below, which is already expressed as a successive approximation case with each variable in the first term. Use the vector (1+i; 1+i) as initial guess.

$$z1 = (z1^2 - z2)^{1/3} \quad , \quad z2 = [(z2)^2 - z1]^{1/4}$$

The equations are programmed using functions from the 41Z Module, as shown below. Note the expected order of registers for the real and imaginary parts of each variable, starting in (R01 + j R02)

```
01  LB "Z1="
02  RCL 02
03  RCL 01
04  Z^2
05  ZENTER^
06  RCL 04
07  RCL 03
08  Z-
09  3
10  1/X
11  Z^X
12  RTN
```

```
13  LBL "Z2="
14  RCL 04
15  RCL 03
16  Z^2
17  ZENTER^
18  RCL 02
19  RCL 01
20  Z-
21  4
22  1/X
23  Z^X
24  END
```

Note that the pre-imposed order of variable's real and imaginary parts is not compatible with the 41Z function ZRCL, and thus the native RCL function has to be used instead.

Here's the sequence of execution using the driver program "**ZSAM**"- don't forget to plug in your 41Z module!

XEQ "ZSAM"	"N=?"	
2, R/S	"F#1? Y="	the previous function is still in R04
"Z1=", R/S	"F#2? Z="	ditto as above
"Z2=", R/S	"Z(1)=? Y^X"	enter first initial guess, imaginary part in Y
1, ENTER^, R/S	"Z(2)=? Y^X"	enter second initial guess
1, ENTER^, R/S	*successive iterations shown, then convergence*:	
	1.041713087-J0.462002407, in R01 + i R02	
R/S	1.038322757 +J0.715596478, in R03 + i R03	

More examples are shown in Jean-Marc's web pages :http://hp41programs.yolasite.com/approx.php you're encourages to check those for additional insight into this method.

Numerical methods: 2-D Laplace & Poisson Equations.

The programs hereafter use finite differencing to solve these partial differential equations with boundary conditions defined on a 2x2 rectangle [0,L]x[0,H]. They employ a method of order 2.

1. ***Laplace Equation:*** $\partial^2u/\partial x^2 + \partial^2u/\partial y^2 = u_{xx} + u_{yy} = 0$

This program solves numerically the partial differential equation: $\partial^2u/\partial x^2 + \partial^2u/\partial y^2 = u_{xx} + u_{yy} = 0$

```
 y                                        with the boundary conditions:
 H| --------------------------|-               u(x,0)  = f1(x)     u(x,L') = f2(x)     0 <= x <= L
   |                          |  u(0,y) =  g1(y)    u(L,y) =  g2(y)     0 <= y <= L'
   |     u = u(x,y)           |
   |                          |
 --|--------------------------|--------x
  0                        L
```

The interval [0,L] is divided into M parts, and the interval [0,H] is divided into N parts. With this arrangement, the partial derivatives are approximated by

$$\partial^2u/\partial x^2 = u_{xx} \sim (u_{i-1,j} - 2.u_{i,j} + u_{i+1,j})/h^2 \quad \text{where } h = L/M \quad , \quad u_{i,j} = u(i.h,j.k)$$
$$\partial^2u/\partial y^2 = u_{yy} \sim (u_{i,j-1} - 2.u_{i,j} + u_{i,j+1})/k^2 \quad \text{where } k = H/N$$

which yields: $\mathbf{u_{i-1,j} + u_{i+1,j} + h^2/k^2 (u_{i,j-1} + u_{i,j+1}) = 2.(1 + h^2/k^2).u_{i,j}}$

So, we have to solve a linear system of (M-1).(N-1) equations in (M-1).(N-1) unknowns. Fortunately, this is a sparse system and we can use an over relaxation method. The over relaxation parameter (usually between 1 and 2) is chosen to be 1.2, but it may not always be the best choice.

The program shows successive iterations, which stop when the last correction is rounded to 0; therefore, the accuracy is controlled by the display format. However, say a 4-place accuracy in the solution of the linear system doesn't necessarily mean a 4-place accuracy in u(x,y).

From the boundary conditions we see we'll need to program up to four functions (which may be the same in some instances but not necessarily so), to describe the behavior in the boundary regions, as follows:

- f1(x) in the "lower| boundary, {y=0 }
- f2(x) in the "upper" boundary, {y=L}
- g1(y) in the "left" boundary, {x=0} and
- g2(y) in the "right" boundary, {x=H}

Obviously we'll need to input the geometric parameters that define the region, i.e. L, H , M, and N. The step sizes are derived from the M,N values, as h = L/M and k = H/N.

The program uses data registers R00 to R(16+(m+1)(N+1) – as you can expect the calculator SIZE will be checked and adjusted accordingly in case it's needed (and enough data registers exist). No user flags are used at all.

The results are the values of the function U(x,y) in the points defined by the MxN grid, i.e. the intersection of the grid coordinates. They are stored in contiguous data registers, and the control word of the data range "bbb.eee" is left in X upon completion of the calculation. It is therefore a parameterized solution of the problem.

For your convenience the module includes a driver program "**LAP+**" to handle the data input automatically, then it transfers the execution to a main routine "**LAP**", and to finally use a data output routine "**OUT**" that sequentially shows the data registers values.

- Data input includes the geometric dimensions, the grid dimensions and the name of the functions defining the boundary conditions.

- You can use "OUT" independently for any general register visualization. It uses the control word in X as input.

Example 1. The following example should be used for familiarization. In this case the function is defined in the rectangle $0 <= x <= 1$, $0 <= y <= \pi/2$ (i.e. $L = 1$, $H = \pi/2$) and subject to the boundary conditions:

$u(x,0) = \sinh x$	$u(0,y) = 0$
$u(x,\pi/2) = 0$	$u(1,y) = (\sinh 1) \cos y$

For your convenience, the four boundary conditions are already included in the module, under the program labels {"**VX0**", "**VLY**", and "**CLX**"}, programmed as follows:

```
 1    LBL "VX0"     12    LBL "VLY"
 2    E^X           13    COS
 3    ENTER^        14    1
 4    1/X           15    E^X
 5    -             16    ENTER^
 6    2             17    1/X
 7    /             18    -
 8    RTN           19    2
 9    LBL "CLX"     20    /
10    CLX           21    *
11    RTN           22    END
```

Note how two boundary conditions can be "shared" by the same function, so we save one global label. "**CLX**" can be used in your own programs to denote null boundary conditions. Also note that on all cases the independent variable is expected to be in the X register – as these conditions are functions of a single variable, *not two*.

Set FIX 3, and RAD mode, then we proceed to execute the driver routine:

XEQ "LAP+"	"U(x,0)? _"	with ALPHA mode ON
"VX0", R/S	"U(X,L)? _"	with ALPHA mode ON
"CLX", R/S	"U(0,Y)? _"	with ALPHA mode ON
"CLX", R/S	"U(1,Y)? _"	with ALPHA mode ON
"VLY", R/S	"DIM(LXL')=?"	use ENTER to separate the values
1, PI, 2, / R/S	"GRID(MXN)=?"	use ENTER^ to separate the values
4, ENTER^, 6, R/S	17.05105 in X after 3 min 45s, then results shown:	

The results are arranged in an (m+1)x(n+1) array, with registers in ROW order as shown below:

R17	R22	R27	R32	R37	R42	R47
R18	R23	R28	R33	R38	R43	R48
R19	R24	R29	R34	R39	R44	R49
R20	R25	R30	R35	R40	R45	R50
R21	R26	R31	R36	R41	R46	R51

The numbers in black are the boundary conditions; the numbers in red are obtained by the finite differences method – and they approximate the solution of a linear system of 15 equations in 15 unknowns. The numeric values are tabulated below:

x \ y	0	π/12	2π/12	3π/12	4π/12	5π/12	π/2
0	0	0	0	0	0	0	0
1/4	0.2526	**0.2441**	**0.2189**	**0.1788**	**0.1264**	**0.0655**	0
2/4	0.5211	**0.5036**	**0.4516**	**0.3688**	**0.2608**	**0.1350**	0
3/4	0.8223	**0.7946**	**0.7125**	**0.5818**	**0.4114**	**0.2130**	0
1	1.1752	1.1352	1.0178	0.8310	0.5876	0.3042	0

For instance, u(3/4; π/12) ~ 0.7946 whereas the correct value is 0.794297. .. We can get a better accuracy with larger M- and N-values and if we execute with FIX 6 or greater. For instance, with M = 8 , N = 12 it gives u(3/4; π/12) ~ 0.794380 = R41 (albeit the execution time is considerably longer).

The exact solution is $u(x,y) = \sinh x \cos y$

2. ***Poisson Equation:*** $\partial^2 u/\partial x^2 + \partial^2 u/\partial y^2 = u_{xx} + u_{yy} = F(x,y)$

This case adds to the previous one the source term F(x, y) as a non-zero member of the differential equation. The methodology and resolution is very similar, thus we'll just mention the differences from the previous case.

The system of equations is in this instance:

$$-h^2.F_{i,j} + u_{i-1,j} + u_{i+1,j} + h^2/k^2 \,(\, u_{i,j-1} + u_{i,j+1}\,) = 2.(\,1 + h^2/k^2\,).u_{i,j} \quad ; \quad \text{where} \quad F_{i,j} = F(i.h,j.k)$$

This will also be resolved using an over-relaxation method with parameter 1.2.Also here the same considerations to accuracy and methodology apply, as mentioned in the Laplace equation paragraphs.

Obviously now we'll need a fifth program to define the source term, F(x,y) under its own global label. This will assume the variables to be in the X- and Y- registers on entry, and the result should be left in the X register upon completion.

Besides the longer execution times, another consideration of using denser grids is that increased number of data registers needed to store all the result values. This can be partially addressed by using data files in X-Memory instead of main memory for the storage – and this approach has been used in the program described below.

Example 2. Solve the Poisson equation defined below in the rectangle with both base and height equal to one, and on a (4 x 5) grid:

$$\partial^2u/\partial x^2 + \partial^2u/\partial y^2 = u_{xx} + u_{yy} = 2.\exp(-x-y) \quad ; \ 0 <= x <= 1 , 0 <= y <= 1 \ ; \ (L = H = 1)$$

with boundary conditions: $u(x,0) = \exp(-x)$ $u(0,y) = \exp(-y)$
$u(x,1) = \exp(-1-x)$ $u(1,y) = \exp(-1-y)$

The exact solution is $u(x,y) = \exp(-x-y)$

For your convenience, both the source term and the four boundary conditions are already included in the module, under the program labels "**FXY**", and {"**UX0**", "**UY0**", "**UXL**", "**UYL**"}, programmed as follows:

1	**LBL "UX0"**	7	**LBL "UXL"**	13	**LBL "FXY"**
2	**LBL "U0Y"**	8	**LBL "ULY"**	14	+
3	LBL 00	9	E	15	CHS
4	CHS	10	+	16	E^X
5	E^X	11	GTO 00	17	ST+ X
6	RTN	12	RTN	18	END

Note how the functions can be re-used, as they have the same expressions even if they apply to different variables. Here's the sequence of operation using the driver program "POIS+":

XEQ "POIS+"	"F(X,Y)? _",	with ALPHA mode ON
"FXY", R/S	"U(X,0)? _"	with ALPHA mode ON
"UX0", R/S	"U(X,L)? _"	with ALPHA mode ON
"UXL", R/S	"U(0,Y)? _"	with ALPHA mode ON
"U0Y", R/S	"U(L,Y)? _"	with ALPHA mode ON
"ULY", R/S	"DIM(LXL')=?"	use ENTER to separate the values
1, ENTER^, R/S	"GRID(MXN)=?"	use ENTER^ to separate the values
4, ENTER^, 5, R/S	18.047 in X after 3 min 45s, then results shown:	

The results are arranged in an (m+1)x(n+1) array, with registers in ROW order as shown below:

R22	R27	R32	R37	R42	R47
R23	R28	R33	R38	R43	R48
R24	R29	R34	R39	R44	R49
R25	R30	R35	R40	R45	R50
R26	R31	R36	R41	R46	R51

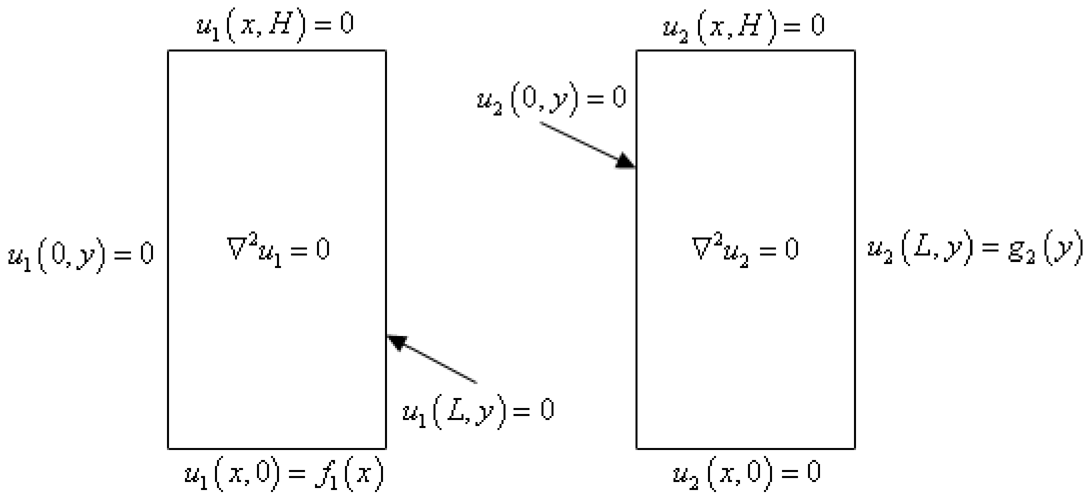
The numbers in black are the boundary conditions; the numbers in red are obtained by the finite differences method – and they approximate the solution of a linear system of 12 equations in 12 unknowns. The numeric values are tabulated below:

x \ y	0	1/5	2/5	3/5	4/5	1
0	1	0.8187	0.6703	0.5488	0.4493	0.3679
1/4	0.7788	**0.6377**	**0.5222**	**0.4276**	**0.3500**	0.2865
2/4	0.6065	**0.4967**	**0.4067**	**0.3330**	**0.2727**	0.2231
3/4	0.4724	**0.3868**	**0.3168**	**0.2594**	**0.2123**	0.1738
1	0.3679	0.3012	0.2466	0.2019	0.1653	0.1353

For instance, u(3/4; 2/5) ~ 0.3168 whereas the correct value is 0.316637 –however the results are more accurate than what one could expect! We can get a better accuracy with larger M- and N-values and if we execute it in FIX 6, for example with M = 8 and N = 10 we find that register R64 = u(3/4,2/5) ~ 0.316677.

With M = 8 and N = 10, we solve a system of 63 equations so a good emulator - like Warren Furlow's V41 in turbo mode - is quite useful if you want to execute this routine for (relatively) large M- & N-values.

The programs use the function **PMTA** for the alphabetical entry of the labels used for the boundary conditions and source terms. This is more convenient that the default functions as it lets you see the input while you enter its values. This also introduces a dependency with the AMC_OS/X module that is required as well (it should always be present on your machine anyway).



A good tutorial can be found at: http://tutorial.math.lamar.edu/Classes/DE/LaplacesEqn.aspx

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

$$u(0,y) = g_1(y) \qquad u(L,y) = g_2(y)$$

$$u(x,0) = f_1(x) \qquad u(x,H) = f_2(x)$$

Numerical methods: Linear Diffusion Equation.

3. *Diffusion Equation:* $\partial T/\partial t = a(x,t)\ \partial^2 T/\partial x^2 + b(x,t)\ \partial T/\partial x + c(x,t)\ T$

The following program solves the partial differential equation (in one spatial dimension, X):

$$\partial T/\partial t = a(x,t)\ \partial^2 T/\partial x^2 + b(x,t)\ \partial T/\partial x + c(x,t)\ T$$

with the initial values: $T(x,0) = F(x)$ and with the boundary conditions:

$T(0,t) = f(t)$ $\quad$ a, b, c, F, f, g are known functions
$T(L,t) = g(t)$

Using a simplified notations, we write: $T_t = a(x,t)\ T_{xx} + b(x,t)\ T_x + c(x,t)\ T$;
where T , a , b , c are functions of x and t

For parameterization purposes, the interval [0,L] is sub-divided into M parts, and h = L/M is the spatial step-size for the characterization.

```
    x
 L|---------------------------------------------------
  |
  |
  |
--|---------------------------------------------------- t
  0
```

This program uses the Crank-Nicholson Scheme, an implicit - stable - method of order 2 in both space and time, thus producing more accurate results than other approaches – even if it may be slightly slower.

The diffusion coefficient $a(x,t)$ is assumed to be *non-negative*. Otherwise, explicit methods may be stable whereas the implicit methods are not! The routine uses the **REGMOVE** function from the X-Functions module.

In an implicit method, the partial derivative with respect to t is approximated by

$$\partial T/\partial t = T_t \sim [\ T(x,t) - T(x,t-k)\ ]/k \qquad ;\ \text{with } k = \text{time step-size}$$

Denoting by$T_{m,n} = T(\ m.h\ ,\ n.k\ )$, the equation becomes :

$$\mathbf{T_{m-1,n}\ (\ a.k/h^2 - b.k/(2h)\ ) + T_{m,n}\ (\ -1 - 2.a.k/h^2 + c.k\ ) + T_{m+1,n}\ (\ a.k/h^2 + b.k\ )\ =\ T_{m,n-1}}$$

So, the new values $T_{m-1,n}$,$T_{m,n}$,$T_{m+1,n}$ are the solutions of a tri-diagonal system of M-1 equations

The approximation $\partial T/\partial t = T_t \sim [\ T(x,t+k) - T(x,t)\ ]/k$ is of order 1, but this formula becomes second-order if it approximates the derivative at $t + k/2$ .Therefore using the averages:

$\partial T/\partial x = T_x \sim \{ [T(x+h,t) - T(x-h,t)]/(2h) + [T(x+h,t+k) - T(x-h,t+k)]/(2h) \} / 2$

$\partial^2 T/\partial x^2 = T_{xx} \sim \{[T(x+h,t) - 2.T(x,t) + T(x+h,t)]/h^2 + [T(x+h,t+k) - 2.T(x,t+k) +$
$+ T(x+h,t+k)]/h^2 \} /2$

Where all these approximations are centered at: $(t + k/2)$ and the formulas are of order 2 both in space and time. Denoting now$T_{m,n} = T(m.h, n.k)$ the diffusion equation becomes:

$$\mathbf{T_{m-1,n+1} (a/h^2 - b/(2h)) + T_{m,n+1} (-2/k - 2a/h^2 + c) + T_{m+1,n+1} (a/h^2 + b/(2h)) =}$$
$$\mathbf{= -T_{m-1,n} (a/h^2 - b/(2h)) - T_{m,n} (2/k - 2a/h^2 + c) - T_{m+1,n} (a/h^2 + b/(2h))}$$

And the functions $a(x,t)$, $b(x,t)$, $c(x,t)$ are to be evaluated at $(x , t + k/2)$;This will be done in the following auxiliary routines written by the user:

- one for $a(x,t)$, assuming "x" and "t" are in the X- and Y-registers upon entry
- another for $b(x,t)$, assuming "x" and "t" are in the X- and Y-registers upon entry
- another for $c(x,t)$, assuming"x" and "t" are in the X- and Y-registers upon entry
- another for $F(x)$, assuming "x" is in X-register upon entry
- another for $f(t)$, assuming "t" is in X-register upon entry, and
- last one for $g(t)$, assuming "t" is in X-register upon entry

A modified version of the "**3DLS**" routine is also included in the module – as it'll be used as a subroutine by "**DFSN+**".

<u>Example 3:</u> $\partial T/\partial t = (x^2/2) \partial^2 T/\partial x^2 - t.x \partial T/\partial x - T$ [or: $T_t = (x^2/2) T_{xx} - t.x T_x - T$]

With <u>initial values:</u> $T(x,0) = 1 + x^2$ ($t_0 = 0$)
Characterized by:L = 1;M=4; k= 1/16, and N = 1
and <u>boundary conditions:</u> $T(0,t) = \exp(-t)$
$T(1,t) = \exp(-t) + \exp(-t^2)$

For your convenience, the module already has the needed routines programmed under the labels "**aXT**", "**bXT**", "**cXT**", "**FX0**", "**F0T**", and "**FLT**". You'll enter those names in the data register either manually of answering the prompts in the driver program.

We proceed to execute the driver routine "**DFSN+**", which will automatically present the prompts for the data entry as follows:

XEQ "DFSN+"	"a(X,T)? _"	with ALPHA mode ON
"aXT", R/S	"b(X,T)? _"	with ALPHA mode ON
"bXT", R/S	"c(X,T)? _"	with ALPHA mode ON
"cXT", R/S	"F(X,0)? _"	with ALPHA mode ON
"FX0", R/S	"F(0,T)? _"	with ALPHA mode ON
"F0T". R/S	"F(L,T)? _"	with ALPHA mode ON
"FLT", R/S	"T0=?"	initial time
0, R/S	"L=?"	length
1, R/S	"X-STRIPS=?"	value for M
8, R/S	"T-STEP SZE.=?"	value for k
16, 1/X, R/S	"#. STEPS=?"	value for N
1, R/S	*calculation… and convergence in about 82 sec*	

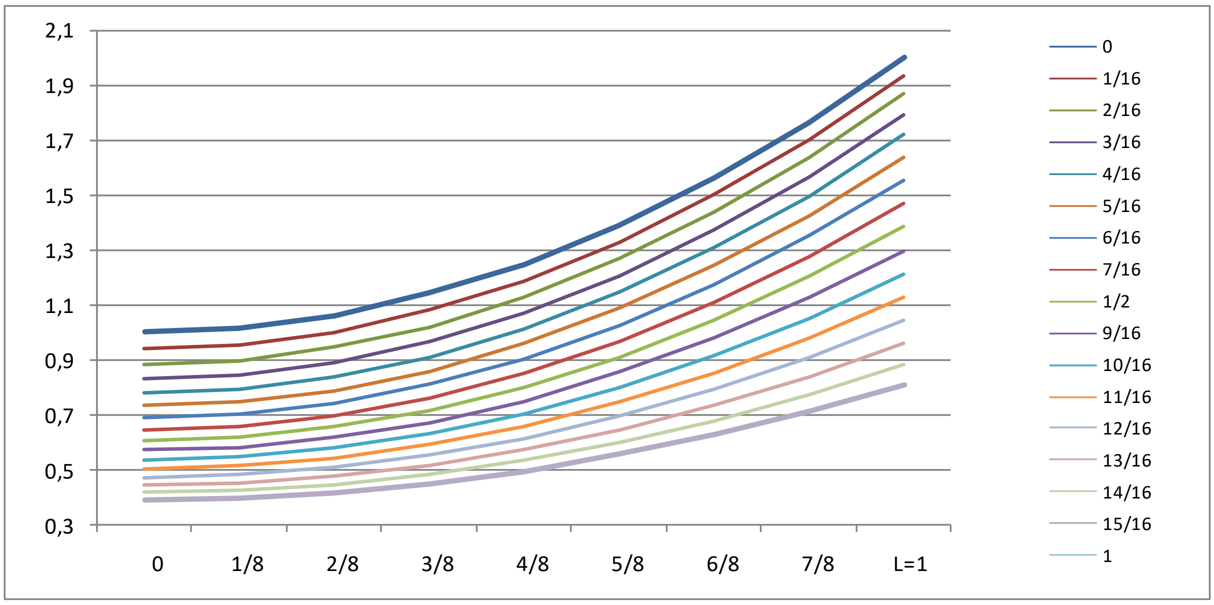
What follows is the parametric listing of the values of the function in the partitions of the interval, at the time corresponding to the current iteration, starting at t0. For example:

T=0.0625; R20=0.9394; R21=0.9550; R22=1.0017; R23=1.0795;
R24=1.1884; R25=1.3285; R26=1.4997; R27=1.7020; R28=1.9355

The table below summarizes the results for several iterations in time. Note that this version of the program ***also stores all the results in an X-Memory data file***, titled "**DFGRID**", which in this instance it has up to 140 registers in size and contains all data values for the distribution.

t\x	0	1/8	2/8	3/8	4/8	5/8	6/8	7/8	L=1
0	1	1.0156	1.0625	1.1406	1.2500	1.3906	1.5625	1.7656	2
1/16	0.9394	0.9550	1.0017	1.0795	1.1884	1.3285	1.4997	1.7020	1.9355
2/16	0.8825	0.8978	0.9440	1.0209	1.1286	1.2670	1.4362	1.6362	1.8670
3/16	0.8290	0.8441	0.8893	0.9647	1.0703	1.2061	1.3721	1.5682	1.7945
4/16	0.7788	0.7934	0.8375	0.9108	1.0136	1.1457	1.3072	1.4980	1.7182
5/16	0.7316	0.7457	0.7882	0.8591	0.9583	1.0858	1.2417	1.4260	1.6386
6/16	0.6873	0.7008	0.7415	0.8094	0.9044	1.0266	1.1759	1.3524	1.5561
7/16	0.6456	0.6585	0.6972	0.7617	0.8520	0.9681	1.1101	1.2779	1.4714
1/2	0.6065	0.6186	0.6551	0.7160	0.8011	0.9107	1.0445	1.2028	1.3853
9/16	0.5698	0.5811	0.6152	0.6722	0.6722	0.8544	0.9796	1.1277	1.2985
10/16	0.5353	0.5457	0.5774	0.6303	0.7043	0.7995	0.9158	1.0533	1.2119
11/16	0.5028	0.5125	0.5417	0.5904	0.6585	0.7462	0.8534	0.9800	1.1262
12/16	0.4724	0.4812	0.5079	0.5524	0.6147	0.6948	0.7928	0.9085	1.0421
13/16	0.4437	0.4517	0.4759	0.5163	0.5728	0.6455	0.7343	0.8393	0.9605
14/16	0.4169	0.4240	0.4458	0.4821	0.5330	0.5984	0.6783	0.7728	0.8819
15/16	0.3916	0.3980	0.4174	0.4499	0.4953	0.5537	0.6250	0.7094	0.8068
1	0.3679	0.3735	0.3908	0.4195	0.4597	0.5114	0.5747	0.6494	0.7358
in:	R20	R21	R22	R23	R24	R25	R26	R27	R28

This can also be plotted graphically, with each curve showing the function values on the data points within the interval, at a given time. Observe how their values decrease as the time progresses:



Runge-Kutta 4th-Order method.

Even if a differential equation of n-th order can be replaced by a system of n first-order equations, the same order of accuracy may be obtained with less evaluations of the function using formulas specially devised for these problems.

1. *Second-order differential equations:* $y'' = f(x,y,y')$

Here we have to solve $y'' = f(x,y,y')$ with the initial 2 values: $y(x_0) = y_0$ and $y'(x_0) = y'_0$

This program uses the 4th-order Runge-Kutta formula, which assumes a step-size small enough to have a representative characterization of the equation along the independent variable from the initial value to the evaluation point:

$$y(x+h) = y(x) + h\,(\,y'(x) + k_1/6 + 2k_2/3\,)$$
$$y'(x+h) = y'(x) + k_1/6 + 2k_2/3 + k_3/6$$

where $k_1 = h.f(x,y)$; $k_2 = h.f(x+h/2, y+h.y'/2+h.k_1/8)$; $k_3 = h.f(x+h, y+h.y'+h.k_2/2)$

Note that only 3 evaluations of the function are needed for each step. This is done in a user-written program to compute $y'' = f(x,y,y')$, assuming x , y , y' are in registers X , Y , Z (respectively) upon entry.

The driver program "**2DFEQ**" is the convenient way to solve this problem. Not only it takes care of setting the calculator's SIZE to the number of registers required, but consistent with the other sections in the module, the data entry section of the program will prompt for the required parameters, and then it'll call the main routine "**2RK4**". Let's see an example of utilization next.

Example. Let's consider the Lane-Emden equation (LEE) of index 3: $y'' = -(2/x)\,y' - y^3$ with the initial values $y(0) = 1$, $y'(0) = 0$

There is an initial difficulty because x = 0 is a singular point, but if we use a series expansion, we find after substitution that $y = 1 + a.x^2 + \ldots.$ will satisfy the LEE if $a = -1/6$, whence $y''(0) = -1/3$, which can be used as a singular point in the following subroutine – already programmed in the module with the global label "**d2X/dX2**":

```
LBL "LEE"
X=0?
GTO 00
RCL Z
ST+ X
X<>Y
/
X<>Y
3
Y^X
+
CHS
RTN
LBL 00
3
1/X
CHS
END
```

Here's the sequence using the driver program "**2DFEQ**""

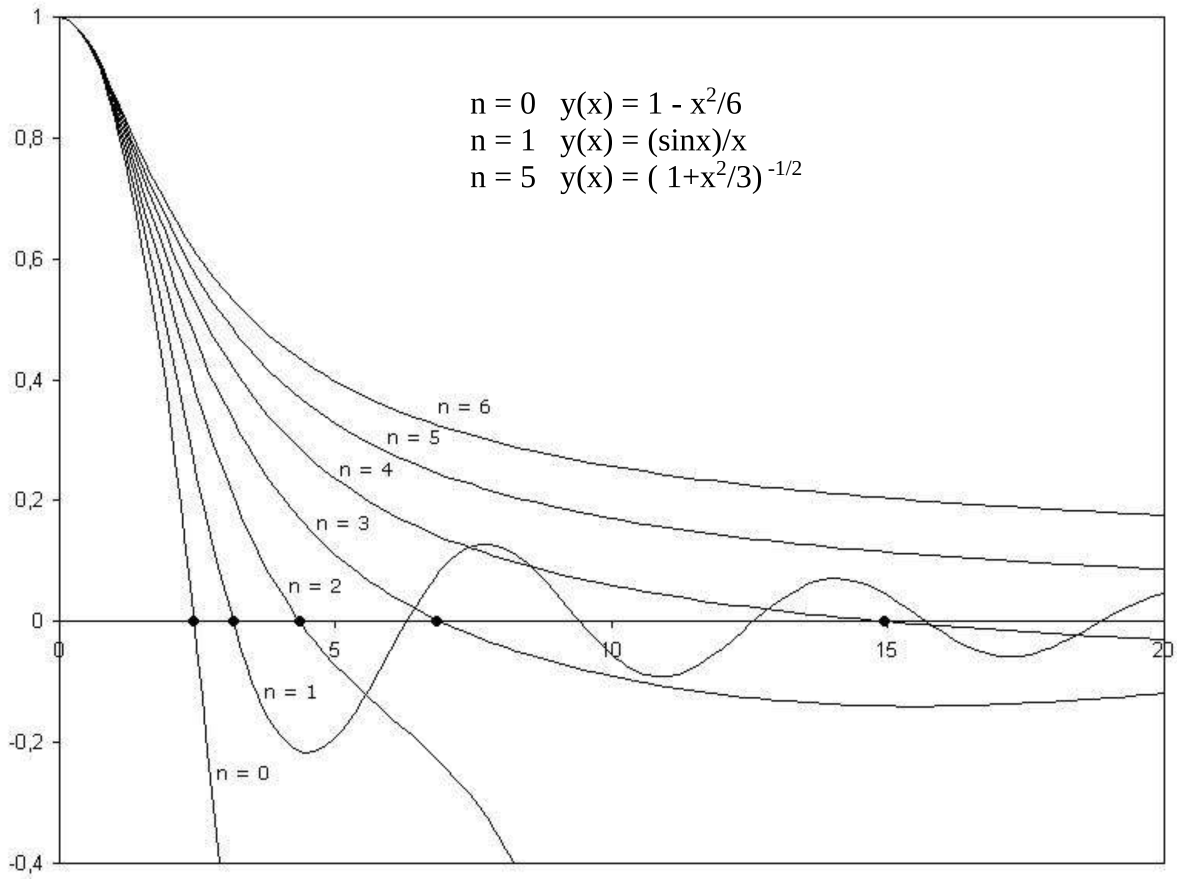
XEQ "2DFEQ"	"F.NAME?"	with ALPHA mode ON
"d2X/dX2", R/S	"X0=?"	
0, R/S	"Y0=?"	
1, R/S	"Y0'=?"	
0, R/S	"#STEPS=?"	i.e. evaluation points
10, R/S	"STPSZE=?"	i.e. divisions between each step
0.1, R/S	*calculation, and convergence:*	

With the results obtained tabulated below:

X=1.0000		X=2.0000		X=3.0000	
R/S	Y= 0.855057170	R/S	Y=0.5829	R/S	Y=0.3592
R/S	Y'= -0.252129561	R/S	Y'=-0.2615	R/S	Y'=-0.1840

$y(x_1) = 0$ for $x_1 = 6.896848619$ and $y'(x_1) = -0.0424297576$; and there is an inflexion point I with $x_I = 1.495999168$, $y_I = 0.720621687$ and $y'(x_I) = -0.279913175$

The solutions of the Lane-Emden Equations of index n can be expressed by elementary functions for only 3 values of n: 0,1, and 5: $y'' + (2/x).y' + y^n = 0$; $y(0)= 1$, $y'(0) = 0$



2. *Third-order differential equations:* y''' = f(x,y,y',y'')

Here we have to solve y''' = f(x,y,y',y'') with 3 initial values: y(x0) = y0; y'(x0) = y'0; y''(x0) = y''0

The main routine is "**3RK4**", which expects all parameters to be in the data registers before starting. This also includes the name of the global label used to program the differential equation, y'' = f(x,y,y') assuming x , y , y' are in registers X , Y , Z (respectively) upon entry.

For your convenience, the module includes "**3DFEQ**", a driver program that does all the data entry automatically for you, prompting for all values sequentially. Let's see an example of utilization next.

Example: $y''' = 2xy'' - x^2y' + y^2$ with y(0) = 1 , y'(0) = 0 , y''(0) = -1

As in previous sections, this example (and the example for next section) is already pre-programmed in the module, under the global label "**d3/dX3**"(and "**d5/dX5**") as follows:

```
1   LBL "d3/dx3"     12  LBL "d5/dX5"
2   X^2              13  RCL 14
3   ST* Z            14  RCL 09
4   X<> L            15  ST+ X
5   ST+ X            16  RCL 13
6   ST* T            17  *
7   RDN              18  -
8   X^2              19  RCL 12
9   -                20  +
10  -                21  RCL 10
11  RTN              22  RCL 11
                     23  *
                     24  -
                     25  END
```

Here's the sequence to solve the example:

XEQ "3DFEQ"	"F.NAME?"	with ALPHA mode ON
"d3/dX3", R/S	"X0=?"	initial point
0, R/S	"Y0=?"	initial value
1, R/S	"Y0'=?"	initial first derivative
0, R/S	"Y0"=?"	initial second derivative
1, CHS, R/S	"#.STEPS=?"	increments of variable
1, R/S	"STPSZE=?"	internal slice size
0.1, R/S	*calculation, and convergence:*	

With the results obtained tabulated below:

```
X=1.0000                    X=2.0000                    X=3.000000000
R/S,    Y=0.595434736       R/S     Y=-0.655723250      R/S     Y=2.790989383
R/S     Y '=-0.776441445    R/S     Y '=-1.708065832    R/S     Y'=2.790989383
R/S     Y "=-0.776441445    R/S     Y "=-1.708065832    R/S     Y"=2.790989383
```

3.- Nth-order differential equations: $y^{(n)} = f(x,y,y',y'',.....,y^{(n-1)})$

The differential equation is now $y^{(n)} = f(x,y,y',y'',.....,y^{(n-1)})$ with the "n" initial values:
$y(x_0) = y_0$, $y'(x_0) = y'_0$, ... , $y^{(n-1)}(x_0) = y^{(n-1)}_0$

Now for the user program describing the equation, the values are to be taken from data registers R09, to Rn+9 (obviously the stack won't hold more than four, so it's discarded altogether). Also the initial values should be store in the same registers prior to the execution of the "**NRK4"** routine. As usual, R00 needs to have the global label used to program the equation.

Example: $y^{(5)} = y^{(4)} - 2x.y''' + y'' - y.y'$ with: $y(0) = 1; y'(0) = y'''(0) = y^{(4)}(0) = 0 ; y''(0) = -1$

As mentioned before, this example is pre-programmed in the module with the global label "**d5/dX5**" – so there's no need to create a program in RAM for it.

The most convenient way to solve this is using the driver program "**NDFEQ**". This will set the required SIZE in the calculator, and will go through the data entry process prompting all parameters sequentially, as follows:

XEQ "NDFEQ"	"N=?"	order of equation
5, R/S	"F.NAME?"	with ALPHA mode ON
"d5/Dx5", R/S	"X0=?"	initial point
0, R/S	"Y0=?"	initial function
1, R/S	"Y0(1)'=?"	initial first derivative
0, R/S	"Y0(2)=?"	initial second derivative
-1, R/S	"Y0(3)=?"	initial third derivative
0, R/S	"Y0(4)=?"	initial fourth derivative
1, CHS, R/S	"#.STEPS=?"	increments of variable
1, R/S	"STPSZE=?"	internal slice size
0.1, R/S	*calculation, and convergence:*	

With the results obtained tabulated below:

```
X=1.0000                      X=2.0000                      X=3.000000000
Y(0)=0.481580647              Y(0)=-1.433907352             Y(0)=-6.123554195
Y(1)=-1.083569461             Y(1)=-2.979546824             Y(1)=-6.867842146
Y(2)=-1.298903838             Y(2)=-2.681382961             Y(2)=-5.643138017
Y(3)=-0.779951410             Y(3)=-1.891414056             Y(3)=-5.603544659
Y(4)=-1.248671208             Y(4)=-0.832432414             Y(4)=-10.79708927
```

With: $h/2 = 0.025$ and: $m = 20$, it yields:

$y(1) = 0.491724223$, $y'(1) = -1.041200398$,
$y''(1) = -1.163353549$, $y'''(1) = -0.479804004$,
$y^{(4)}(1) = -0.897594479$

End of the manual.