HP-71B Minimax Polynomial Fit

Valentín Albillo (#1075, PPC #4747)

Curve fitting has always been a very popular application for every kind of computing device, from programmable calculators using **RPN** to large mainframe computers running **FORTRAN**, and understandably so. Besides the obvious uses in engineering, statistics and finance, where fitting laboriously gathered data to some appropriate mathematical function allows further analytical study and reasonably reliable forecasts, there are hundreds of real-life applications, from a programmer trying to minimize the time needed to compute a complicated special function, to a music lover trying to come up with a simple formula to convert the values in an old tape counter to actual times in seconds.

Now, in order to fit some mathematical function to a given set of data, we must select its *type* (polynomial, exponential, etc.) and some criteria to measure the *quality* of the fit. The type is very dependent on the characteristics of the given dataset but polynomials are the most popular of all: they're very easy and fast to compute, requiring nothing but a few additions and multiplications, and they're also extremely easy to integrate, differentiate, and solve for particular values.

As for measuring the quality of the fit, there are several options which minimize some well-defined error measure. If you decide to minimize the *average error*, you'll be doing a so called **Least Squares** fit, which minimizes the *squares* of the differences between the original values and those computed using your function. This is an extremely popular election since the coefficients of the least-squares polynomial are *really easy* to compute, and you'll end up with very *simple* code which will return the required coefficients *fast*, like this subprogram I wrote clearly demonstrates (Math ROM required) [1] :

```
SUB LSP(X(),Y(),A()) @ N=UBND(X,1) @ D=UBND(A,1) @ DIM U(N,D),V(D)
MAT U=CON @ FOR I=1 TO N @ FOR J=2 TO D @ U(I,J)=X(I)^(J-1) @ NEXT J
NEXT I @ MAT V=TRN(U)*Y @ MAT U=TRN(U)*U @ MAT A=SYS(U,V) @ END SUB
```

This 3-liner takes arrays **X**, **Y,** containing the coordinates of the data points, and returns the coefficients of the least-squares polynomial fit in array **A**. The number of data points is the size of **X**, while the degree of the polynomial is the size of **A**. Call it either from the keyboard or from some program, such as this simple tester:

```
DESTROY ALL @ OPTION BASE 1 @ INPUT "#Points, Degree=";N,D
DIM X(N),Y(N),A(D+1) @ MAT INPUT X,Y @ CALL LSP(X,Y,A) @ MAT DISP A
```

Let's test it. *First* key in the tester program, *then* the **LSP** subprogram (with line numbers of your choice), then try and fit a 2^{nd}-degree least-squares polynomial to the following data points:

[1] In all program snippets here line numbers are present only if referenced; you may use whatever line numbering suits you as long as it's sequential from lower to higher line numbers. Also, all code featured in this article extensively uses **Math ROM** keywords, mainly matrix operations.

X	0	1	2	3	4	5
Y	10.2	10.9	14.3	18.9	26.3	34.8

```
>FIX 3                                              [ENTER]
>RUN                                                [ENTER]
#Points, Degree= 6,2                                [ENTER]
X(1)?            0,1,2,3,4,5                        [ENTER]
Y(1)?            10.2,10.9,14.3,18.9,26.3,34.8      [ENTER]
  10.093, 0.055, 0.982
```

thus the sought-for fitting least-squares polynomial of degree 2 is:

$$\mathbf{P(x) = 10.093 + 0.055\ x + 0.982\ x^2}$$

Now let's fit a least-squares line to these data:

X	1	2	3	4
Y	2	2	4	4

```
>RUN                          [ENTER]
#Points, Degree= 4,1          [ENTER]
X(1)?            1,2,3,4      [ENTER]
Y(1)?            2,2,4,4      [ENTER]
  1.000, 0.800
```

so the fitting least-squares *line* (degree 1) is:

$$\mathbf{P(x) = 1.000 + 0.800\ x}$$

> **Note**: Data points need *not* be equally spaced, and further they need *not* be entered in any particular order.

However, Least-Squares polynomials, convenient as they are, do not distribute the errors *evenly* all over the range of approximation as they actually minimize the *average* error, which means there will be regions where the errors are *larger* than average while other regions will have *lower*-than-average errors. This might be acceptable in some applications but not in others, and is particularly non-optimal for the task of approximating *mathematical functions*, where you aren't interested in the average error but in making sure the *maximum* absolute error never *exceeds* a given tolerance for all arguments in some interval while using degrees as *low* as possible. Or, conversely, to achieve the *smallest maximum error* possible for a given degree. Enter **Minimax Polynomial Fit**.

Minimax Polynomial

By definition, the minimax polynomial is the approximating polynomial which has the *smallest maximum deviation* from the true function. Thus, we're minimizing ABS(P(x)-f(x)) instead of $(P(x)-f(x))^2$. The usual analytical approach is not readily applicable as the **ABS(x)** function is less amenable to analytic manipulations than the well-behaved $\mathbf{x^2}$ function; in particular the latter has a continuous derivative while the former does not. We need to use quite complex, iterative algorithms requiring raw power and processing time.

Introducing MMAXPOLY, a Minimax Polynomial Fitting program

MMAXPOLY is a 50-line (w/o comments) program I've written to compute minimax polynomial approximations to any given set of data points. You can enter the data points directly *from the keyboard*, you can *specify a generating function* which will be evaluated in a given range to automatically generate the dataset, or you can *read the dataset from a file*. In the first two cases, the whole dataset can be stored in a file, for later retrieval and possibly further fitting or processing.

MMAXPOLY allows the user to *either* specify a particular degree for the minimax polynomial, *or else* to give a maximum absolute error to be met, in which case it will iteratively compute a series of minimax polynomials for the given dataset, starting from degree 1 and incrementing it until either the maximum absolute error is equal or less than the one specified, or the degree is already **N-1** (where **N** is the number of points in the dataset), which, rounding errors notwithstanding, would necessarily result in an *exact* fit (maximum error = 0)

Here is the commented program listing[2]. For full details, see **Notes**, below:

'MMAXPOLY' *(3,085 Bytes)*

```
 1 ! MMAXPOLY - MINIMAX Polynomial Fitting (c) Valentin Albillo, 2005
 2 !
 3 ! Initialization
 4 !
 5 DESTROY ALL @ OPTION BASE 1 @ STD @ DIM F$[96],R$[96]
 6 DIM B,I,S,T,E,C,H,P,Q,L,M,N,X,J,K,D,W @ W=0 @ D=8 @ B=.001
 7 !
 8 ! Utility User-defined Functions
 9 !
10 DEF FNF(X) @ S=A(L) @ FOR J=N+1 TO 2 STEP -1 @ S=S*X+A(J) @ NEXT J
11 FNF=S @ END DEF
12 DEF FNS(I)=SGN(FNF(U(I))-V(I))
13 !
14 ! Ask for the data source (keyboard, defined function, file)
15 !
16 INPUT "Verbose ? (Y/N): ","N";R$ @ IF LEN(R$)#1 THEN 16
17 R$=UPRC$(R$) @ IF R$="Y" THEN W=1 ELSE IF R$#"N" THEN 16
18 INPUT "[K]bd,[D]ef,[F]ile=";R$
19 ON POS("KDF",UPRC$(R$))+1 GOTO 18,38,31,23
20 !
21 ! Ask for the name of the file and read in the data points
22 !
23 ON ERROR GOTO 23 @ INPUT "Filename=";R$ @ IF R$="" THEN 18
24 IF NOT POS(R$,":") THEN R$=R$&":MAIN"
25 ASSIGN #1 TO R$ @ READ #1;F$,P,Q,M @ DIM Z(M),Y(M) @ READ #1;Z,Y
26 ASSIGN #1 TO * @ OFF ERROR
27 DISP "f(X)=";F$;" (";P;",";Q;"),";M;"points" @ GOTO 48
28 !
29 ! Ask for the defined function, limits, and number of points
30 !
31 LINPUT "f(X)=";F$ @ F$=UPRC$(F$) @ IF F$="" OR POS(F$,"X")=0 THEN 18
32 INPUT "X1,X2,N=";P,Q,M @ IF Q<=P OR FP(M)#0 OR M<2 THEN 32
33 DIM Z(M),Y(M) @ J=(Q-P)/(M-1) @ ON ERROR GOTO 31
```

[2] No HP-71 ? No problem. Google the web for Emu71, a free emulator for Windows, and/or HP-71X, another outstanding emulator, this time for your HP48/49.

```
34 FOR I=1 TO M @ Z(I)=P+(I-1)*J @ X=RES @ Y(I)=VAL(F$) @ NEXT I @ GOTO 43
35 !
36 ! Ask for the number of data points and input them from the kbd
37 !
38 INPUT "# Points=";M @ DIM Z(M),Y(M) @ F$="Points" @ FOR I=1 TO M
39 DISP "#";STR$(I); @ INPUT ": X,Y=";Z(I),Y(I) @ NEXT I @ P=Z(1) @ Q=Z(M)
40 !
41 ! Allow the user to store the data points in a file
42 !
43 ON ERROR GOTO 43 @ INPUT "Store in ","*";R$
44 IF R$#"*" THEN ASSIGN #1 TO R$ @ PRINT #1;F$,P,Q,M,Z,Y @ ASSIGN #1 TO *
45 !
46 ! Ask the user for the degree/error of the minimax polynomial
47 !
48 ON ERROR GOTO 48 @ E=-1 @ INPUT "Degree=";R$ @ IF R$="" THEN 50
49 N=INT(ABS(VAL(R$))) @ IF N=0 OR N>=M THEN 48 ELSE 52
50 INPUT "Max. error=";E @ E=ABS(E) @ FOR N=1 TO M-1
51 STD @ DISP "Degree";N;"..."; @ IF W THEN DISP
52 L=N+2 @ DIM A(L),U(L),V(L) @ FIX D @ OFF ERROR @ P=0 @ Q=0
53 !
54 ! Choose an initial tuple
55 !
56 FOR I=1 TO L @ J=(M-1)*(I-1)/(L-1)+1 @ U(I)=Z(J) @ V(I)=Y(J) @ NEXT I
57 !
58 ! Compute mm-polynomial for this tuple and max. error for all data
59 !
60 CALL SPOLMM(U,V,A) @ H=ABS(A(1)) @ C=0 @ FOR I=1 TO M
61 R=FNF(Z(I))-Y(I) @ IF ABS(R)>ABS(C) THEN C=R @ T=I
62 NEXT I @ K=Z(T) @ IF W THEN DISP " h=";H;", H=";C;"in ";K
63 !
64 ! Check if the ending criteria are already met
65 !
66 IF N=M-1 OR C=0 THEN 81 ELSE IF ABS(ABS(H/C)-1)<B OR H=P AND C=Q THEN 81
67 !
68 ! Rebuild the current tuple to include the data point of max. error
69 !
70 P=H @ Q=C @ IF U(1)<K THEN 72 ELSE IF FNS(1)=SGN(C) THEN I=1 @ GOTO 77
71 FOR I=L TO 2 STEP -1 @ U(I)=U(I-1) @ V(I)=V(I-1) @ NEXT I @ GOTO 77
72 IF U(L)>K THEN 74 ELSE IF FNS(L)=SGN(C) THEN I=L @ GOTO 77
73 FOR I=1 TO L-1 @ U(I)=U(I+1) @ V(I)=V(I+1) @ NEXT I @ GOTO 77
74 FOR I=1 TO L @ IF U(I)>K THEN 76
75 NEXT I
76 IF FNS(I)#SGN(C) THEN I=I-1
77 U(I)=K @ V(I)=Y(T) @ GOTO 60
78 !
79 ! Process finished, display coefficients and max. error
80 !
81 FIX D@ DISP " H=";ABS(C)@ IF E>=0 AND ABS(C)>E AND N#M-1 AND NOT W THEN 83
82 FOR I=2 TO L @ STD @ DISP "A"&STR$(I-2)&"="; @ FIX D @ DISP A(I) @ NEXT I
83 IF E<0 OR ABS(C)<=E THEN END
84 NEXT N @ DISP "Max. degree reached" @ END
85 !
86 ! Optionally, scans an interval for the absolute maximum error
87 !
88 DEF FNE(X1,X2,S) @ M=-INF @ FOR X=X1 TO X2 STEP S @ J=VAL(F$)-FNF(X)
89 J=ABS(J) @ IF J>M THEN M=J @ I=X
90 NEXT X @ DISP I @ FNE=M @ END DEF
91 !
92 ! Subprogram to compute the mm-polynomial and max. error for a tuple
93 !
94 SUB SPOLMM(X(),Y(),A()) @ L=UBND(X,1) @ DIM M(L,L),Z @ MAT M=CON
95 FOR I=1 TO L @ M(I,1)=(-1)^I @ Z=X(I) @ FOR J=3 TO L
96 M(I,J)=Z^(J-2) @ NEXT J @ NEXT I @ MAT A=SYS(M,Y) @ END SUB
```

Notes:

- If a generating function is specified, the program uses **`VAL`** to evaluate it, so you may use most any BASIC expression which evaluates to a number (e.g. **`SIN(COS(X))`**). External function keywords present in LEX/ROMs are allowed too (e.g., **`GAMMA(X)+LOG2(X)+1`**) as well as so-called *'funny'* functions (i.e., **`FNROOT`** (Solve) and **`INTEGRAL`** (Integrate)). You must use **X** as the independent variable and *must ensure* that your expression is syntactically correct and is computable in the specified interval, else an error will eventually occur when **`VAL`** attempts to evaluate it and you'll be asked again for a correct function.
- The program computes the *exact* minimax polynomial fit of a given degree to a *discrete* set of data points. For *continuous* functions, a set of **N** data points is generated, where **N** is specified by the user, and the program fits a minimax polynomial to that. The resulting polynomial is not the mathematically exact minimax polynomial for the given function but rather a close approximation to it, which, for sufficiently large **N**, is close enough for all practical purposes.
- However, you should exercise care in not choosing *too large* a value for **N**, because the iterative algorithm used needs to construct and then solve a system of **N+1** linear equations per iteration, and so the memory requirements increase at least *quadratically* with **N** while the processing time increases *cubically*. Also, these large systems are inherently bad-conditioned, thus further impacting the solution's accuracy. Though the algorithm used is highly self-correcting and will cater for inaccuracies, they might well cause it to enter an indefinite loop. Reasonable maximum values for **N** would be from 20 (best) to 50.
- The iterative algorithm used selects an initial **M**-tuple of datapoints (where **M** depends on the specified degree), then exactly fits a minimax polynomial to it. This polynomial is then checked against the remaining datapoints. If some data point results in an error larger than the one computed for the **M**-tuple, it is exchanged by the one closest to it in the **M**-tuple (taking care to preserve error's sign alternance), and the whole process is repeated for the altered **M**-tuple. This is guaranteed to converge (and quite fast at that), resulting in a final minimax polynomial that has the same maximum error for the whole dataset, not just for datapoints belonging to its originating **M**-tuple.
- To avoid indefinite loops and unnecessary refinement, the program uses a comprehensive termination criterium (line **66**): if we're already at the maximum possible degree or if the maximum error is already zero, it terminates. Else, if during the last two iterations the error has decreased negligibly or the program is exchanging back and forth the same two datapoints, then terminate as well.
- There are three internal routines that are user-callable, see **Usage instructions**:

```
SUB SPOLMM(X(),Y(),A())   non-iteratively computes the coefficients (A)
                          of the minimax polynomial for an M-tuple (X,Y)
DEF FNE(X1,X2,S)          once done, scans interval [X1,X2] for maximum
                          error, in steps of S. Returns both X and MaxErr
DEF FNF(X)                once done, evaluates minimax polynomial at X
```

Usage instructions

Note: Most errors while entering data will result in the data being asked again.

Start the program: **`RUN [ENTER]`**

`Verbose ? (Y/N): N`

- Press **`Y [ENTER]`** if you want to see intermediate results every iteration, including the max. error for the **M**-tuple (**h**), the max. error for the whole datapoint set (**H**) and the **X** coordinate of the datapoint where it appears.
- Press **`N [ENTER]`** (or just **`[ENTER]`**) if you don't want intermediate results.

`[K]bd,[D]ef,[F]ile=`

- Press **`K [ENTER]`** if you intend to enter datapoints right from the *keyboard*
- Press **`D [ENTER]`** if you want to enter an expression defining a *function*
- Press **`F [ENTER]`** if you want to take datapoints from a *file*

Option K - Entering datapoints from the keyboard:

`# Points=`

- Enter the number of datapoints (X,Y) in your set

`#1: X,Y=`

- Enter each datapoint's coordinates X,Y in turn, separated by a ",". Datapoints ***must*** be entered in increasing **X**-order. Repeat until all datapoints are entered. Then, you'll be asked:

`Store in *`

- If you want to store the dataset in a file, enter its name, which can include a device suffix if it's going to reside in some external device.
- If you don't want to store the data in a file, simply press **`[ENTER]`**

Option D - Entering an expression defining a function:

`f(X)=`

- Enter the expression that computes **f(x)**. It must be syntactically valid and must evaluate without error for the range given below. You may use LEX/ROM functions and must use **X** as the independent variable.

`X1,X2,N=`

- Enter the interval of approximation and the number of points in the dataset that will be automatically generated by evaluating your function, separating all values by commas. The function will then be evaluated **N** times, which may take a long time depending on the nature of your function and the number of points specified. Then, you'll be asked:

`Store in *`

- See **Option K** above.

Option F – Taking datapoints from a file:

`Filename=`

- Enter the name of the file where your dataset is stored. It may be a RAM file or it may include a device suffix (i.e.: **`MYDATA:HDRIVE1`**).

`f(X)=EXP(X)+GAMMA(X) ( 1 , 2 ), 20 points`

- After entering the filename, the dataset is read and a message is shown, to confirm its origin, the interval of approximation, and the number of points in the dataset. **`Points`** appears if it was entered from the keyboard.

Specifying a given degree for the polynomial:

`Degree=`

- Enter the polynomial's degree, which must be a value from 1 to **N-1**, where **N** is the number of points in the dataset. After you press **`[ENTER]`**, the coefficients and maximum error for the polynomial are computed:

  `H= (maximum absolute error for the whole dataset)`

  `A0= (1st coefficient (a0))`

  `..`

  `An= (last coefficient (an))`

  And we have: $\mathbf{P(x) = a_0 + a_1 x + a_2 x^2 + .. + a_n x^n}$

Specifying a maximum absolute error for the polynomial:

If you prefer, you may specify a *maximum error* for the polynomial and let the program automatically find the required degree. To that effect, when asked:

`Degree=`

- Just press **`[ENTER]`**. It will immediately ask:

`Max. error=`

- Enter the maximum absolute error you want and press **`[ENTER]`**. The program will go on searching for the minimax polynomial which meets that criterium, starting from degree **1** upwards:

  `Degree 1 .. H= (maximum error for degree 1)`

  `..`

  `Degree n .. H= (maximum error for degree N)`

- It will stop and display the coefficients of the final polynomial as soon as the current degree meets the maximum error specified, or else the maximum possible degree has been reached, in which case it'll display:

  `Max. Degree reached`

Further options once the polynomial has been computed:

- You can evaluate the polynomial at any given argument from the keyboard using **`FNF(your argument)`**. For instance:

  `>FOR X=-1 TO 1 STEP 0.1 @ PRINT X, FNF(X) @ NEXT X  [ENTER]`

- You can scan the whole interval (using a given step) for its maximum error, right from the keyboard, with **`FNE(x1,x2,step)`**. It will return both the maximum error found and the **X** coordinate where it appears. For instance:

  `>FNE(-1,1,0.01) [ENTER]`

  `0.56000000` ***(x where the maximum error is)***

  `0.04525456` ***(maximum deviation found)***

You must have already computed the polynomial and must still be within the program's environment in order to use **`FNF`** and **`FNE`**. Additionally, scanning with **`FNE`** only makes sense when you've specified a function **f(x)** to fit.

Examples

1. For starters, try and fit a 2nd degree polynomial to y=e^x in the [-1,1] interval such that the absolute maximum error is minimized.

As we're asked to fit a minimax polynomial to a function, we'll have to do with an *approximation* to the exact minimax polynomial by selecting a sufficient number of datapoints in the given interval and fitting a minimax polynomial to that. We'll then check the resulting polynomial's maximum error over the *continuous* interval. Let's proceed, automatically generating and using 21 datapoints for the fit:

```
>RUN
Verbose ? (Y/N):    Y       [ENTER] (see maximum error's convergence)
[K]bd,[D]ef,[F]ile= D       [ENTER] (we'll give a function to fit)
f(X)=               EXP(X)  [ENTER] (the function to fit is y=e^x)
X1,X2,N=            -1,1,21 [ENTER] (we'll use 21 points in [-1,1])
Store in            *       [ENTER] (don't store the points in a file)
Degree=             2       [ENTER] (we want a 2nd degree polynomial)

 h= 0.03695390 , H= 0.05384982 in  0.60000000  (1st iteration)
 h= 0.04367387 , H=-0.04736857 in -0.40000000  (2nd iteration)
 h= 0.04472950 , H= 0.04472950 in -1.00000000  (final iteration)

  H = 0.04472950 (absolute maximum error over the 21 generated points)
  A0= 0.98915039 (minimax polynomial coefficients)
  A1= 1.13047170 (ditto)
  A2= 0.55393025 (ditto)
```

So we have:

$$P(x) = 0.98915039 + 1.13047170\,x + 0.55393025\,x^2$$

Let's check its maximum error by scanning the [-1,1] interval at 0.1 steps. From the keyboard, type:

```
  >FNE(-1,1,0.1)      [ENTER]   (scan [-1,1] using steps of 0.1)
    -1.00000000                 (x where the maximum error is)
     0.04472950                 (maximum deviation found)
```

which is as expected because the 21 datapoints used are spaced by exactly 0.1. Let's do a much finer scan, using 0.01 steps:

```
  >FNE(-1,1,0.01)     [ENTER]   (scan again using steps of 0.01)
     0.56000000                 (x where the maximum error is)
     0.04525456                 (maximum deviation found)
```

so our *discrete* approximation gives a maximum error that is within **1.17%** of the true maximum error over the whole *continuous* interval (you can compute this percentage by evaluating **(M-H)/H*100** right from the keyboard). That's good enough for us and certainly much better than what least-squares can achieve. Scanning the interval using 0.001 steps results in the same maximum error while using the ultra-fine (and ultra-time-consuming!!) 0.0001 step size results in a maximum deviation of **0.04525460 (**for x = **0.5603**), virtually the same as before.

2. Martial artist Gemma Saotome needs a polynomial approximation to the inverse of the Gamma function in [2,10]. He wants the predicted values' error to be less than 0.001 but first he's keen to try if a 5th degree P(x) would suffice.

```
>RUN
Verbose ? (Y/N):    N          [ENTER] (don't show convergence)
[K]bd,[D]ef,[F]ile= D          [ENTER] (we'll give a function to fit)
f(X)=               FNROOT(1,5,GAMMA(FVAR)-X) [ENTER](Gamma's inverse)
X1,X2,N=            2,10,20    [ENTER] (we'll use 20 points in [2,10])
Store in            IGAMDATA   [ENTER] (store the points for later use)
Degree=             5          [ENTER] (fit a 5th degree polynomial)
   H= 0.00223353(absolute maximum error over the 20 generated points)
   A0= 1.22884198, A1= ... (coefficients of P(x))
```

Regrettably, the resulting maximum error **H=0.002**+ exceeds 0.001 so he tries again, this time directly specifying the maximum error and having the program automatically find out the necessary degree and coefficients of the polynomial:

```
>RUN
Verbose ? (Y/N):    N          [ENTER] (don't show convergence)
[K]bd,[D]ef,[F]ile= F          [ENTER] (we'll specify the file ..)
Filename=           IGAMDATA   [ENTER] (.. where the points were stored)
  f(X)=FNROOT(1,5,GAMMA(FVAR)-X) (2, 10), 20 points (shows the info)
Degree=                        [ENTER] (we don't specify a degree ..)
Max. error=      0.001         [ENTER] (.. but max err must be <=0.001)
  Degree 1 ... H= 0.16574531           (maximum error for 1st degree)
  Degree 2 ... H= 0.05075812           (ditto for 2nd degree)
  Degree 3 ... H= 0.01708390           (..)
  Degree 4 ... H= 0.00601569           (..)
  Degree 5 ... H= 0.00223353           (..)
  Degree 6 ... H= 0.00082511           (error for 6th degree is <0.001!)
  A0= 0.92372552, A1= 1.82262391, A2=-0.56542641, A3= 0.10801104
  A4=-0.01201826, A5= 0.00071517, A6=-0.00001756  (coeffs of P(x))
```

So we have:

$$P(x) = 0.92372552 + 1.82262391\,x - 0.56542641\,x^2 + 0.10801104\,x^3 - 0.01201826\,x^4 + 0.00071517\,x^5 - 0.00001756\,x^6$$

which has a guaranteed maximum error **H=0.0008**+ over the 20 data points sampled. Let's check its maximum error when scanning the interval at 0.1 steps:

```
>FNE(2,10,0.1) [ENTER]
   2.30000000  (x of the point where the maximum error is)
   0.00098788  (maximum deviation)
```

which is still *less* than the required 0.001. An ultra-fine, ultra-slow scan using 0.01 steps gives **0.00098826** (at x=**2.29**), still within tolerance so we're done. Let's find out **Gamma^{-1}(5)** using our computed polynomial, then check the result:

```
>FNF(5)  [ENTER]
     3.85162159    (predicted value of Gamma^-1(5))
>FNROOT(3,4,GAMMA(FVAR)-5), RES-FNF(5)   [ENTER]
     3.85235546    (true value of Gamma^-1(5))
     0.00073387    (the absolute error is indeed less than 0.001)
```

3. Fledgling math genius F. Resnel is studying the function defined by this indefinite integral, which cannot be expressed in terms of elementary functions:

$$f(x) = \int_0^x \sin x^2 \, . \, dx$$

and he would like to find a polynomial approximation of the lowest possible degree yet guaranteing 4 correct decimal places for all arguments x in the interval [0,1]. He'll then print a comparison table from x=0 to 1 using 0.1 steps.

```
>RUN
Verbose ? (Y/N):    N      [ENTER]    (don't show convergence)
[K]bd,[D]ef,[F]ile= D      [ENTER]    (we'll define f(x) )
f(X)=               INTEGRAL(0,X,1E-5,SIN(IVAR*IVAR))  [ENTER]
X1,X2,N=            0,1,20 [ENTER]    (we'll use 20 points in [0,1])
Store in            *      [ENTER]    (no need to store the points)
Degree=                    [ENTER]    (we don't specify the degree ..)
Max. error=         5E-5   [ENTER]    (..but max.err must be <0.00005)
     Degree 1 ... H= 0.05773118
     Degree 2 ... H= 0.00768950
     Degree 3 ... H= 0.00103943
     Degree 4 ... H= 0.00023957
     Degree 5 ... H= 0.00003041      (degree 5 will suffice)
     A0= 0.00003041, A1=-0.00221685, A2= 0.02503280, A3= 0.23175179
     A4= 0.17790388, A5=-0.12220333   (polynomial's coefficients)
```

So we have:

$$P(x) = 0.00003041 - 0.00221685\,x + 0.02503280\,x^2 + 0.23175179\,x^3 + 0.17790388\,x^4 - 0.12220333\,x^5$$

which has a guaranteed maximum error **H=0.00003+** over the 20 data points sampled. Now, let's print the comparison table from x=0 to 1, using 0.1 steps:

```
>FIX 5    [ENTER]
>FOR X = 0 TO 1 STEP .1 @ X;FNF(X);INTEGRAL(0,X,1E-5,SIN(IVAR*IVAR));
 RES-FNF(X) @ NEXT X   [ENTER]
      {x}      {P(x)}      {f(x)}       {error}
  0.00000    0.00003     0.00000     -0.00003
  0.10000    0.00031     0.00033      0.00003
  0.20000    0.00269     0.00267     -0.00002
  0.30000    0.00902     0.00899     -0.00002
  0.40000    0.02128     0.02129      0.00001
  0.50000    0.04145     0.04148      0.00003
  0.60000    0.07132     0.07134      0.00001
  0.70000    0.11241     0.11239     -0.00002
  0.80000    0.16576     0.16574     -0.00002
  0.90000    0.23182     0.23185      0.00003
  1.00000    0.31030     0.31027     -0.00003
```

so F. Resnel does indeed get 4 correct decimal places (within 1 ulp) from P(x).

4. Freshman L. Oggsinn has been assigned the task of selecting a suitable polynomial to fit the following experimental data, rounded to 4 decimal places

X	0.0000	0.6931	1.0986	1.3863	1.6094	1.7918	1.9459	2.0794	2.1972	2.3026
Y	0.0000	0.6390	0.8906	0.9830	0.9993	0.9757	0.9305	0.8734	0.8101	0.7440

with the proviso that the degree must be selected so as to fit the data as closely as possible while filtering out rounding-induced noise, then compute P(Pi/2)

L's best strategy is to specify *neither* a degree for the polynomial *nor* a maximum error to be met but let the program compute *all* mínimax polynomials starting from degree **1** and going all the way up to **9th** degree, which is sure to be an *exact* fit (rounding errors aside) as there are 10 data points. Now, the proper degree will be the *last one* which results in a *markedly diminished* maximum error. As soon as the maximum error ceases to decrease significantly, then you know you're actually fitting also the errors, rather than just the data !

```
>RUN
Verbose ? (Y/N):      N              [ENTER] (no intermediates shown)
[K]bd,[D]ef,[F]ile=   K              [ENTER] (data from the keyboard)
# Points=            10             [ENTER] (number of data points)
#1: X,Y=             0,0            [ENTER] (1st data point)
#2: X,Y=             0.6931,0.6390  [ENTER]
#3: X,Y=             1.0986,0.8906  [ENTER]
#4: X,Y=             1.3863,0.9830  [ENTER]
#5: X,Y=             1.6094,0.9993  [ENTER]
#6: X,Y=             1.7918,0.9757  [ENTER]
#7: X,Y=             1.9459,0.9305  [ENTER]
#8: X,Y=             2.0794,0.8734  [ENTER]
#9: X,Y=             2.1972,0.8101  [ENTER]
#10: X,Y=            2.3026,0.7440  [ENTER] (last data point)
Store in             KDATA          [ENTER] (store them in "KDATA")
Degree=                             [ENTER] (no degree specified)
Max. error=          0              [ENTER] (compute all polynomials)
     Degree 1 ... H= 0.26781403  (error for polynomial of degree 1  )
     Degree 2 ... H= 0.02296048  (error reduced by a factor of 11+  )
     Degree 3 ... H= 0.00604656  (error reduced by a factor of 3.7+ )
     Degree 4 ... H= 0.00017079  (error reduced by a factor of 35+  )
     Degree 5 ... H= 0.00001993  (error reduced by a factor of 8.6+ )
     Degree 6 ... H= 0.00001823  (.. negligible error reduction ..)
     Degree 7 ... H= 0.00001676  (.. negligible error reduction ..)
     Degree 8 ... H= 0.00001089  (.. negligible error reduction ..)
     Degree 9 ... H= 0.00000001  (forced exact fit save rounding)

     A0=-1.53699813E-13, A1= ..  (coefficients of the 9th degree P(x))
     Max. degree reached
```

Notice how sharply the error *diminishes* at first, then after 5th degree you hardly get any improvements at all until, suddenly, 9th-degree necessarily achieves exact fit. Of course L's not interested in using an **(N-1)**-th degree P(x) to fit **N** data points, that's not minimaxing but plain *collocation*, mimicking both data and errors alike. L's best choice in this case is the 5th-degree polynomial, which we'll produce now:

```
>RUN
Verbose ? (Y/N):      N      [ENTER]    (don't show convergence)
```

```
[K]bd,[D]ef,[F]ile=   F     [ENTER]    (data points taken from a file)
Filename=             KDATA [ENTER]    (they were stored in "KDATA")
 f(X)=Points ( 0 , 2.3026 ), 10 points (details retrieved & shown)
Degree=               5     [ENTER]    (we want the 5th-degree P(x))
     H= 0.00001993
     A0= 0.00001993, A1= 0.99397302, A2= 0.02362975, A3=-0.20069600,
     A4= 0.02220127, A5= 0.00241094
```

So L's polynomial is:

$$P(x) = 0.00001993 + 0.99397302\,x + 0.02362975\,x^2 - 0.200696\,x^3 + 0.02220127\,x^4 + 0.00241094\,x^5$$

which has a maximum error **H=0.00002+** over all datapoints. As they are specified to 4 decimal places, this means P(x) agrees with them to that many places as well:

```
>FIX 4@ FOR I=1 TO 10@ DISP USING "Z.4D3X";Z(I),Y(I),FNF(Z(I))@NEXT I
   {x}      {y}      {P(x)}
 0.0000   0.0000   0.0000
 0.6931   0.6390   0.6390
 1.0986   0.8906   0.8906
 1.3863   0.9830   0.9830
 1.6094   0.9993   0.9993
 1.7918   0.9757   0.9757
 1.9459   0.9305   0.9305
 2.0794   0.8734   0.8734
 2.1972   0.8101   0.8101
 2.3026   0.7440   0.7440
```

Perfect agreement to 4 places, indeed. Now, let's compute P(Pi/2), as specified:

```
>FNF(PI/2)
  1.0000
```

which sure looks good.

Final remarks

Minimax polynomial fitting is an incredibly powerful technique to add to one's own data-fitting arsenal. Though the internal details are complex enough and it does require a lot of computing muscle when compared to other well-known strategies such as Least Squares, the final reward is the improved accuracy it provides for any given degree or, conversely, the least possible degree for any given accuracy. It may take longer to compute the minimax polynomial, but typically you do that exactly *once*, then evaluate the resulting polynomial *many* times, which, being of lower degree than the ones obtained by other methods, will then execute faster and thus result in time savings in the long run.

I hope you enjoyed this subject and perhaps even found it useful. As always, you're welcome and encouraged to port it to your favourite high-end HP model.