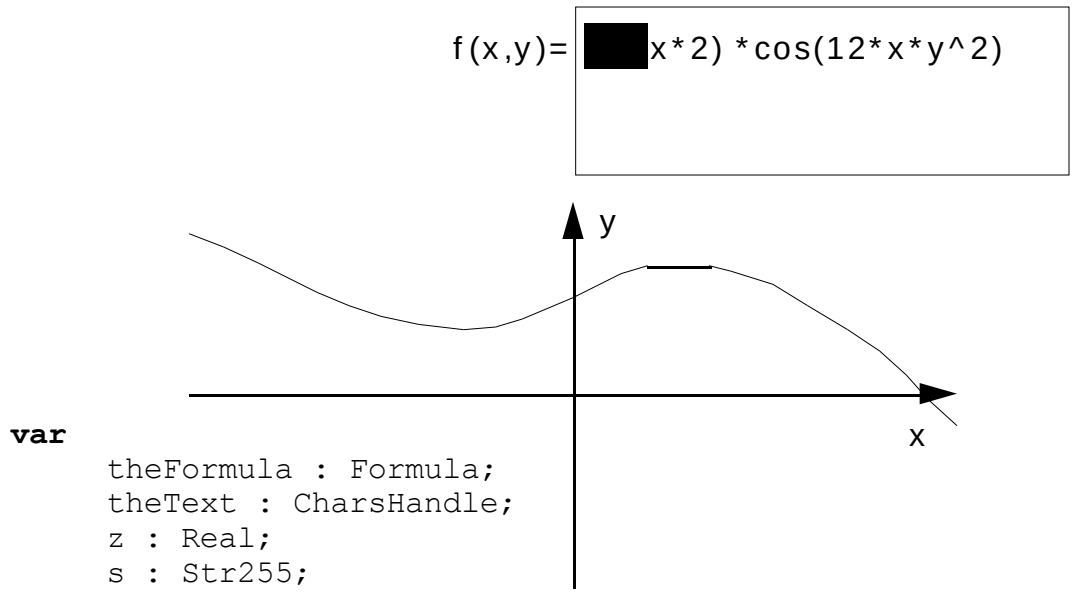
Christian Franz

Dynamic Math

for programmers

Version 1.0.1

```
Expr = ["+" | "-"] Term {("+" | "-") Term}.
Term = Factor { Op Factor}.
```




```
var
      theFormula : Formula;
      theText : CharsHandle;
      z : Real;
      s : Str255;

begin
      s := 'Sin(x^2)*Cos(y)^2';
      Str2Text(s,theText);
      theText := NewHandle(SizeOf(Chars));
      New(theFormula);
      theErr := Parse(theText,theFormula);
      for x := 1 to 20 do begin
            z := theFormula.evaluate(x,y);
            ...
      end;
end;
```

© 1993 by Christian Franz

Notice:

You may use this software and its documentation free of charge for any non-commercial use. This includes using it for writing public-domain or other *freeware* programs as well as in-house development. If you use this software in your programs you *must* include the line

"**uses Christian Franz Dynamic Math ©1993 by Christian Franz**"

in both the program's documentation and 'About...' dialog.
You should also write me an email or postcard telling me if you like the library.

Permission is granted to freely distribute this package and its accompanying documentation as long as neither is modified in any way and no fees are charged other than the usual downloading fees on commercial bulletin boards.

For commercial use of this software (for shareware programs and any other purpose) or its documentation you must contact me and have my written consent. Usually all I want in return is a free registered copy of your finished work.

My address is

Christian Franz
Sonneggstrasse 61
CH-8006 Zurich

Switzerland

email cfranz@iiic.ethz.ch
tel. +1-261 26 96 (+ = your code for Switzerland)

If you have any questions or bug reports or would like to see other features implemented, please feel free to contact me at above address.

Note:
As you will notice throughout the documentation, English is not my primary language. There are bound to be many mistakes. If you find some, please take the time to write them down and (e)mail them to me so I can correct them.

Blatant Plug

Other programs/libraries written by me:

Galaxis
A space strategy game where you try to recover four capsules by triangulation. Features full background stereo music. Multiplayer version coming soon. Postcardware.

Programmers 3D GrafSys
Full 3D graphics and animation for the game programmer. No hassle with 3D transformation. The GrafSys does all this for you. Designed with game programmers in mind, the GrafSys trades accuracy for performance and should not be used for engineering projects. On the other hand, set AutoErase to true, rotate a few objects around, move the eye, call Draw and the objects erase and draw themselves as the eye would see them...
GrafSys supports full translation, independent rotation, clipping and many more features. Upcoming versions support better hidden-line/surface animation. Written for THINK Pascal or THINK C.
Postcardware for non-commercial use.

FastPerf Trigs
For those of you who have no FPU but crave for speed, here it is. A library that uses ultra-fast look-up tables for sine, cosine and tangens, this boosts trigonometric operation by a minimum factor of two to six.
Postcardware for non-commercial use.

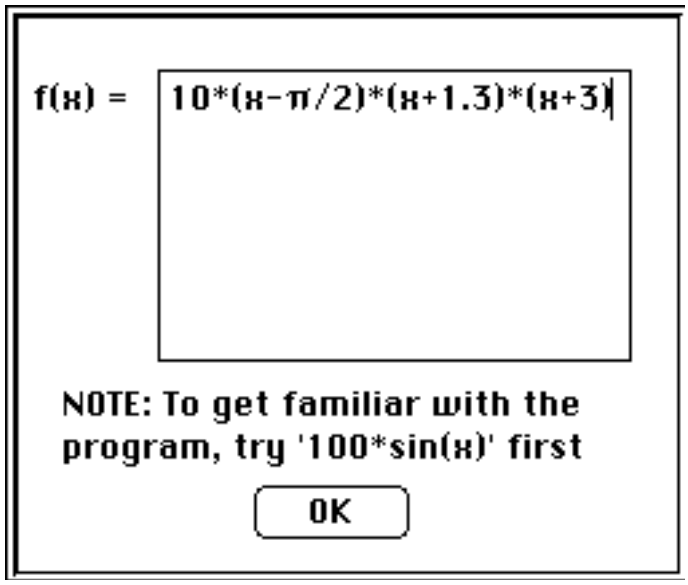
All programs/libraries are available via anonymous ftp from sumex-aim or mac.archive.umich.edu or better BBS.

What it is:

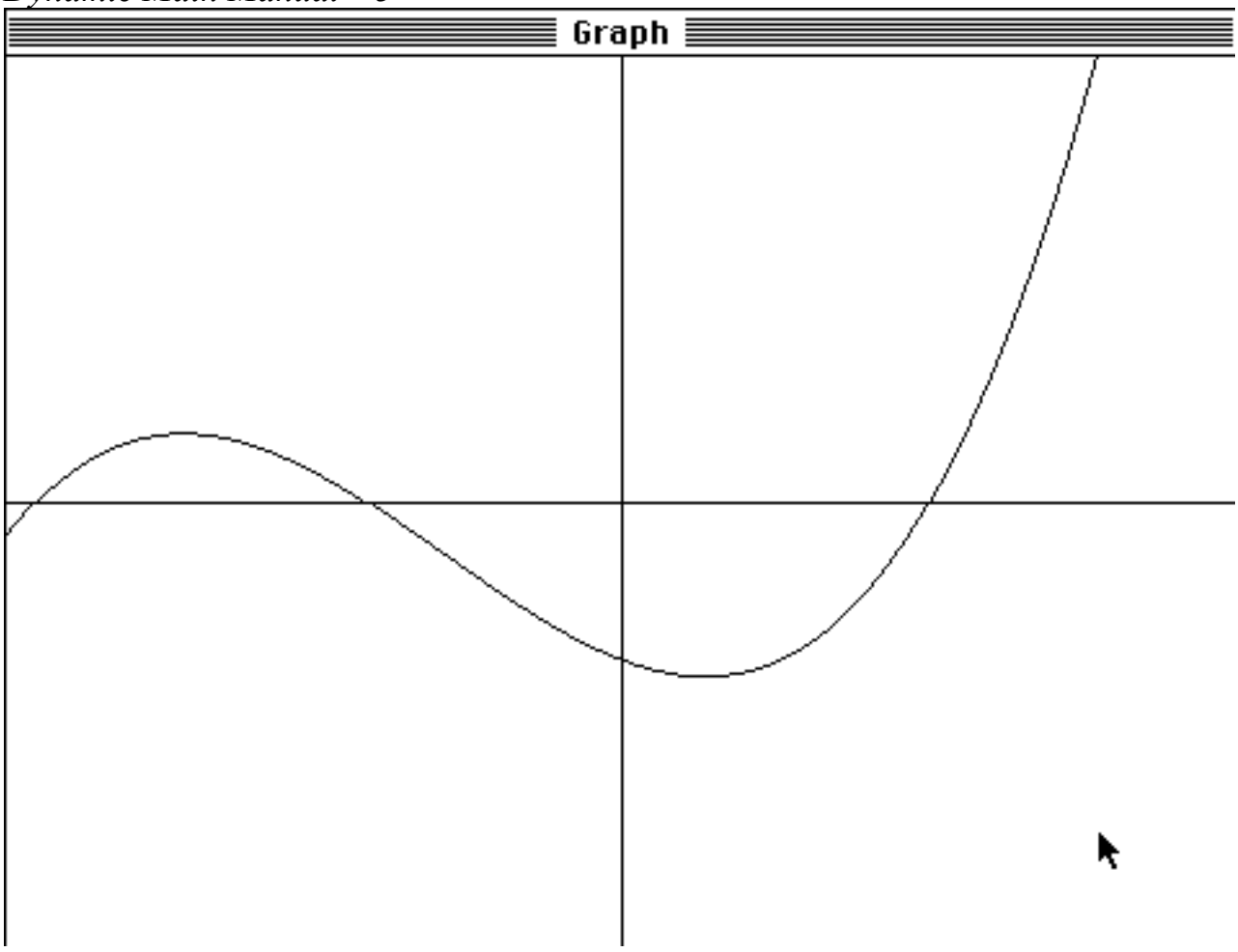
Dynamic Math is a combined mathematical Parser/Interpreter to include into your programs. With it you can enter and evaluate formulas while the program is running (i.e. on-the-fly). The formula is entered as normal Text and then parsed into an object (or procedure for you non-OOP folks). This object can be used like any normal object. Calling the 'evaluate' procedure will evaluate the formula with the given parameters. Obvious uses would be for testing or visualizing formulas. Instead of compiling the program every time you change the formula you now would get the formula from a dialog and then interpret it. Dynamic Math is written for use with THINK Pascal or THINK C.

Example:

Try out the DM Demo program. After doubleclicking on the Application the following dialog should appear:



After pressing 'Enter' or clicking on the OK button, the program will convert (parse) the formula and draw the function from -π to +π:



Note that the scales for x and y are different. While from left to right from -π to +π, the y scale is from -150 to +150. Take a look at the source code provided.

Version History:

1.0.1 (the euler/unary bug)

Murphy's Law at it's best. Version 1.0 was posted to sumex for 5 minutes and I proudly demonstrated DM to a friend when I discovered that DM parser didn't recognize the 'e' constant. Version 1.0.1 fixes this bug.

Another little bug affected the parser: if you incorrectly wrote an expression like 'e^-x' instead of 'e^(-x)', DM would crash. 1.0.1 catches this and now returns a syntax error code (Note that violation of the EBNF includes passing an empty text. This situation now also generates eSyntaxError).

Related to this was a bug that occasionally woudn't pass on the minus sign. Since the minus sign can be unary as in '-2' or binary as in 'a-b', the DM parser treats the minus sign *always* as unary. The expression 'a-b' gets internally resolved to 'a + (-b)'. Well, **rrwood@unixbox.canrem.com (Roy Wood)** asked me how I handled the binary/unary problem and when trying to demonstrate, it showed that the parser goofed up on certain occasions. This is fixed now.
This is yet another proof for the "Never trust a 'dot zero'" axiom. Heck, I even prefer betas to 'dot zeros'. At least you *know* there are bugs.

1.0 (dot zero)

Original Version. Yeah, it had bugs.

How to use DM:
Include the DynamicMath.lib and DynamicMath.Int into your project. This library contains the parser and interpreter for the formulas. In addidtion to that, you must include the SANE.lib. C programmers will also have to include the µRuntime.lib that comes with THINK Pascal to avoid link errors. There is a second version of the DM library, called DM881.lib. As you have guessed, it's the same library, this time compiled to use direct FPU calls and 68030 code. Note that with library you should use the SANE881.lib.

Dynamic Math and THINK C
C programmers will also have to include the µRuntime.lib that comes with THINK Pascal to avoid link errors. Before including the libraries, you have to use the oConv program with the '-v' option set to convert the library. You will also have to translate the DynamicMath.Int file to the standard C header file. If you did so, I'd be greatful if you could email a copy to me so I can include it into upcoming releases.

Programming with Dynamic Math (DM):

How Dynamic Math works:

The general principle is pretty straightforward. You type in the formula as you would normally. Then you pass this Text to DMs parser (i.e. converter). After succesfully converting the program you receive an object (or, if you prefer, a pointer to a procedure). Now, every time you call the objects evaluate procedure the formula will be evaluated. Note that you have to convert the formula only once.

```
var
     theFormula : Formula;
     theText : CharsHandle;
     z : Real;
     s : Str255;

begin
     s := 'Sin(x^2)*Cos(y)^2';
     Str2Text(s,theText);
     theText := NewHandle(SizeOf(Chars));
     New(theFormula); (* allocate mem *)
     theErr := Parse(theText,theFormula);
     for x := 1 to 20 do begin
          z := theFormula.evaluate(x,y);
          ...
     end;
end;
```

In the Example above the string s containing the text 'Sin(x^2) * Cos(y)^2' is converted into the formula object theFormula. Inside the loop the formula gets evaluated for the two variables x and y.

Dynamic Math Syntax:

Dynamic Math supports the standard mathematical notation. If in doubt, refer to the EBNF notation below. DM is *not* case-sensitive.

```
Formula = Expression.
Expression = ["+" | "-"] Term {("+" | "-") Term}.
Term = Factor {Op Factor}.
Factor = Function | Number | Ident | ( "(" Expression ")" ).
Op = "*" | "/" | "^".
Function = FuncIdent "(" Expression ")".
FuncIdent = "Sqrt" | "Sin" | "Cos" | "Tan" | "ATan" | "Rnd" | "Fact" |
                "Exp" | "Abs" | "Sgn" | "Trunc" | "Round" | "Ln".
ident = "x" | "y" | "e" | "π".
Number = Digit {Digit} ["." Digit {Digit}].
```

Standard Functions in Dynamic Math:

Sqrt : Square root. If argument < 0 then Sqrt returns 0.
Sin: Sine of argument. Argument in radiants
Cos: Cosine of argument. Argument in radiants
Tan: Tangent of argument. Argument in radiants
ATan: Arcus tangent of argument. Returns radiants.
Rnd: Random. Returns real in range 0 to argument.
Fact: Factorial of argument. Negative argument yields 1
Version 1 of DM rounds argument to integer
Exp: Raises e to the power of argument
Abs: Returns absolute value of argument
Sgn: Returns sign of argument: -1 if argument < 0, 0 if argument = 0 and 1 if argument > 0
Trunc: Yields the integer part of argument
Round: Rounds argument to nearest integer
Ln: Returns logarithmus naturalis (base e) of argument

Variables passed to the Function:

In version 1.0 of Dynamic Math you can only have two independent variables. They are called 'x' and 'y' and correspond to the first and second parameter you pass the evaluate procedure, respectively.

Constants in Dynamic Math:

DM recognizes two named constants: e and π. They are automatically converted into their corresponding values at maximum precision. e is the euler constant (=Exp(1)), π the circle constant (3.14...).Other constants (e.g. γ = 0.5772...) have to be defined as numbers.

Known Bugs:

DM has one bug that can be read directly from the EBNF syntax: The '^' operator has no precendence over the '*' and '/' operators. Although this should be no problem, there are some cases where you

might get unexpected results. To avoid this, you should always put parentheses around exponent and base. So instead of writing 3^x you should write (3^x).

There is another bug with the '^' operator, this time it is intentional. Since DM allows you to use reals with it, you can easily pass 0.5 as exponent. This is of course the square root function. Now, what if you tried to calculate -1^0.5 ? The result is imaginary. To circumvent this problem, DM takes a pragmatic aproach.

The '^' operator is simulated after the well known definition :

$$a^x = e^{x \ln(a)}$$

allowing for real exponents. If a happens to be negative, the following algorithmus is used: The exponent x is truncated to its integer (i.e. 1.9 is truncated to 1). If this is even, the result will have a positive, otherwise a negative sign. Then a is converted to its absolute value and a^x is computed and the sign applied.

This algorithmus works fine for all integer exponents. Otherwise it will return the negative absolute value of the imaginary number generated.

Example:

$8^{2.1} = 78.7932424...$

$-8^{2.1} = 74.937... + i * 24.3485...$ (correct result)

$-8^{2.1} = +78.793242...$ (DM returned result. **Sign** is positive.

$|-8^{2.1}| = 78.793242$)

Coming Versions:

The next release will support more independent variables. The Factorial function will support real arguments (i.e Fact(3.12) will yield 6.99023687457 instead of 6 (= 3*2*1).

Using Dynamic Math

Include the DynamicMath.lib and DynamicMath.Int into your project. If not already included, add the SANE.lib as well. Dynamic Math can convert any zero-delimited Text. Since you would normally use Pascal-Strings, DM includes a string-to-text conversion routine.

DMs two main routines are `Parse` and `evaluate`. Parse will convert the Text you pass it to a structure returned in the formula object. Calling this objects `evaluate` procedure will evaluate the formula for the passed parameters x and y.

Dynamic Math Routines

function Parse (text : CharsHandle; var theFormula : Formula) : integer

Parse converts the zero delimited text pointed to by text to a structure in theFormula. Note that you must allocate memory for theFormula before you call Parse. Passing the evaluate message to theFormula will evaluate the formula for the passed parameters (see below).
Parse returns noErr if the parsing is successful. Otherwise, the following errors are defined:

```
eClsExpected = -1; (* ")" expected *)
eOpnExpected = -2; (* "(" expected *)
eUnknownSymbol = -3; (* unknown function *)
eSyntaxError = -4; (* EBNF syntax violation *)
```

The variable gPos contains the position in the text where the error occured. If you converted a string to text, add 1 to gPos to get the position in the string.

function Formula.evaluate(x,y : real) : real

Evaluate interprets the structure the parser put into the Formula object for the two parameters x and y. The result is returned.

procedure Str2Text (s: Str255; var t: CharsHandle)

This procedure converts a Pascal string to a zero-delimited text that can be parsed with the Parse procedure. Note that you have to allocate memory for the zero-delimited text before you call Str2Text.

Routines

```
const

     eClsExpected = -1; (* ")" expected *)
     eOpnExpected = -2; (* "(" expected *)
     eUnknownSymbol = -3; (* unknown function *)
     eSyntaxError = -4; (* EBNF syntax violation *)

type
     (* The Item object is just defined so your  *)
     (* compiler doesn't gag on the Formula      *)
     (* definition. Don't change or even use it! *)
     Item = object (* don't ever mess with me *)
          thevalue: real;
          negate: boolean;
          function evaluate: real; (* no you don't *)
     end;

     Formula = object
          structure: Item;
          function evaluate (x, y: real): real;
          (* call me to evaluate parsed function *)
     end;

var
     gPos : integer;

procedure Str2Text (s: Str255; var t: CharsHandle);

function Parse (text: CharsHandle;
     var theFormula: Formula): Integer;
```