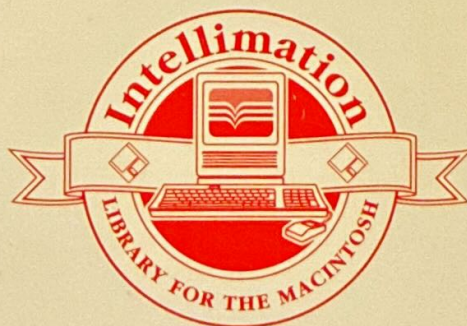


QA
90
.S35
1991
Man. 1

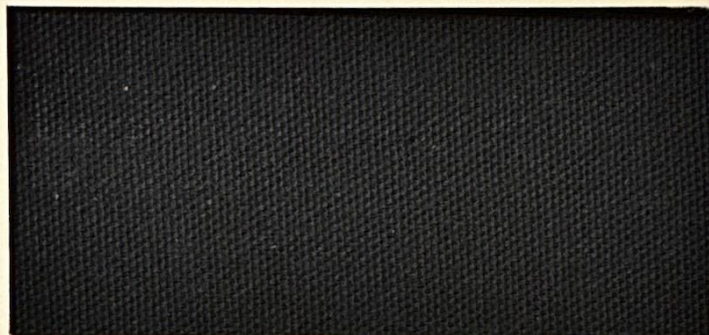
2 of 2

Scientific and Graphical Toolkit

Developed by
Robert Harding, Steve Lay
and Hugo Moule



BECAUSE LEARNING SHOULD KNOW NO BOUNDS™



LIBRARY
Michigan State
University

Scientific and Graphical Toolkit at a Glance

Program Description

Scientific and Graphical Toolkit

Developed by
Robert Harding, Steve Lay
and Hugo Moule

System Requirements

Intellimation

Library™ for the Macintosh®

P.O. Box 1530
Santa Barbara, CA 93116-1530

805-968-2291





Copyright © 1991 Robert Harding

All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language in any form by any means without the written permission of Intellimation, Inc.

SOFTWARE LICENSE NOTICE

Your license agreement with Intellimation, Inc., which is printed on the software envelope, specifies the permitted and prohibited uses of this product. Any unauthorized duplication or use of this product, in whole or in part, in print, or in any other storage and retrieval system is forbidden.

THE INTELLIMATION GUARANTEE

Intellimation stands behind all products in the Intellimation Library for the Macintosh. We're so certain you'll be pleased with your selection from the Library that all our products are unconditionally guaranteed for one full year from the date of purchase. All products may be returned for a full, 100 percent refund within 30 days of purchase, no questions asked. All we ask is that software and accompanying documentation must be in original condition and suitable for resale. You can order from us with confidence.

LICENSES AND TRADEMARKS

Macintosh®, LaserWriter®, ImageWriter®, and HyperCard® are registered trademarks of Apple Computer, Inc.

Intellimation
P.O. Box 1530
130 Cremona Drive
Santa Barbara, CA 93116-1530
805-968-2291
805-685-9610 (Customer Service Representative/Technical Support Line)

The Renaissance project is a
Multimedia in Education initiative
sponsored by Apple Computer and
brought to you by Intellimation. Renaissance



1.0 2.25.92

Scientific and Graphical Toolkit at a Glance

Program Description

The Scientific and Graphical Toolkit rapidly performs a variety of mathematical calculations and associated graph plotting. The Toolkit saves time by providing ready-made scripts for common requirements and by conducting many operations—including graph drawing—ten times faster than HyperTalk scripts alone. Several types of graphs can be produced, including simple function plots, contour plots, three-dimensional curves shown as perspective views, and functions of two variables plotted as wireframes or as surfaces. Data for graphs can be generated in a variety of ways, including taking data from user input into HyperCard fields, and calculating data in HyperTalk scripts. You can also have the Toolkit compile and evaluate a mathematical expression. Data can also come from various Toolkit XCMDs. With this program, you can create your own stacks, without having sophisticated programming expertise.

System Requirements

Minimum Memory	1Mb
Minimum System	Version 6.0.5
Hard Drive	Required
Other Requirements	HyperCard 2.0 minimum

Contents

1. Overview of The Scientific and Graphical Toolkit 1

- 1.1 What does it need? 1
- 1.2 What can it do? 1
- 1.3 Who can use it? 2
- 1.4 Why use it? 3
- 1.5 How to use the Toolkit? 3
- 1.6 How to use this manual 4

2. Principles of Operation of The Scientific and Graphical Toolkit 5

- 2.1 The Toolkit 5
- 2.2 Using the Toolkit from your stack 5
- 2.3 Using the Toolkit—the cookbook approach 6
- 2.4 More details of operation 9
- 2.5 Xblock types 11
- 2.6 Xblock names 11
- 2.7 Action names 12
- 2.8 Palettes 12
- 2.9 Handling external windows 12
- 2.10 How Errors are handled 13

3. The XCTRL Command 14

- 3.1 Actions “Init Toolkit” and “Dispose Toolkit” 14
- 3.2 Actions “Init Window” and “Dispose Window” 14
- 3.3 Various window actions 15
- 3.4 Action “Copy to Card” 16

4. The XCOMPILE and XEVAL Commands 17

- 4.1 XCOMPILE: Actions “Init Dictionary” and “Dispose Dictionary” 17
- 4.2 XCOMPILE: Action “Declare” 18
- 4.3 XCOMPILE: Actions “Compile” and “Dispose Expression” 18
- 4.4 XEVAL: Action “EvalExpr” 18
- 4.5 XEVAL: Action “EvalSimple” 19
- 4.6 XEVAL: Action “Eval At” 20
- 4.7 Actions “Preload” and “Unload” 20

5. The XARRAY Command 21

- 5.1 Action "BySteps" 21
- 5.2 Action "Dispose Array" 22
- 5.3 Actions "Declare" and "Get Info" 22
- 5.4 Actions "ArrayToList" and "ListToArray" 23
- 5.5 Actions "GetOne" and "PutOne" 23
- 5.6 Examples of use with XEVAL 24
- 5.7 Array utilities 25
- 5.8 Vector and matrix calculations 26
- 5.9 Actions "Preload" and "Unload" 27

6. The XGRAPH Command 28

- 6.1 Actions "Init Graph" and "Dispose Graph" 28
- 6.2 Actions "Draw Axes", "Label Axes", and "Draw Labelled Axes" 29
- 6.3 Action "Draw Graph" 30
- 6.4 Actions "SetGParam" and "GetGParam" 31
- 6.5 Actions "UserToLocal" and "LocalToUser" 33
- 6.6 Actions "Draw To", "Draw 2D" and "Draw 3D" 33
- 6.7 Actions "Local Text" and "User Text" 34
- 6.8 Actions "Copy" and "Cut" 34
- 6.9 Actions "Display Graph" and "Resize Graph" 35
- 6.10 Actions "Preload" and "Unload" 35

7. Further Graphics XCMDs 36

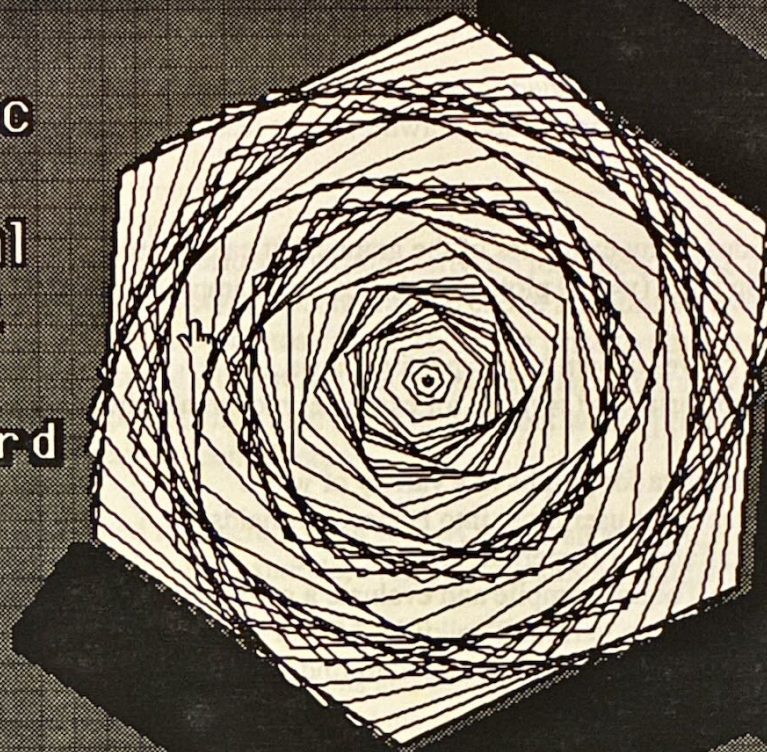
- 7.1 XCONTOURS 36
- 7.2 XREVO 37
- 7.3 XSURFACE 38
- 7.4 XBAR 39
- 7.5 XPIE 40

8. Mathematical XCMDs 42

- 8.1 XROOTS 42
- 8.2 XMIN 42
- 8.3 XINTEGRATE 43
- 8.4 XODE 43
- 8.5 XFFT 45

Solving Problems with the Software 47

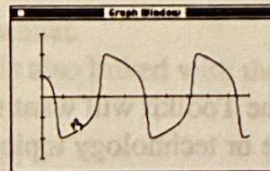
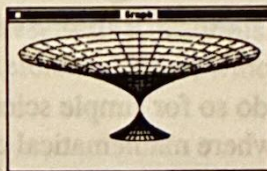
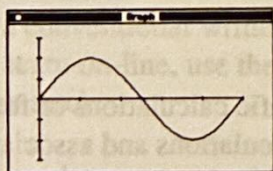
The Scientific and Graphical Toolkit for Hypercard



1. Overview of The Scientific and Graphical Toolkit

1.1 What Does it Need?

The toolkit is designed for use with HyperCard 2.0, for which the minimum recommended configuration is a Macintosh Plus with at least 1 Mb memory and a hard drive.



1.2 What Can it Do?

In a nutshell, the Toolkit can perform a variety of mathematical calculations and associated graph plotting very much more rapidly than HyperCard on its own.

Some of the operations which the Toolkit does *could* be done with Hypertalk scripts alone, but others would be impossible without the Toolkit. There are many scientific and mathematical operations for which until now it has just not been *practical* to use HyperCard. The Toolkit will save time in two ways:

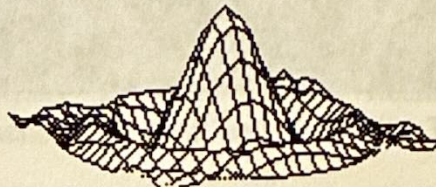
- Ready-made scripts are provided for common requirements
- Many operations, particularly graph drawing, are at least 10 times faster than Hypertalk scripts alone

This manual provides many examples of the graphs that can be produced. You will see:

- Simple function plots ($y=f(x)$) plotted against x , for example)
- Contour plots
- 3-Dimensional curves shown as perspective views
- Functions of two variables ($z=f(x,y)$ for example) plotted as wireframes or as surfaces

You can generate the data for graphs in a variety of ways:

- Data can be taken from user input into HyperCard fields.
- Data can be calculated in Hypertalk scripts.
- You can have the Toolkit compile and evaluate a mathematical expression.
- The data can come from various Toolkit XCMDs. For example, there are XCMDs to solve Ordinary Differential Equations and to find Fast Fourier Transforms.



1.3 Who Can Use It?

Even if you have never used HyperCard before, you will be able to use the Toolkit Demonstration Stack simply by editing the data which is visible on the screen and clicking on buttons.



Demonstration Stack

Most users of the Toolkit will want to do so for simple scientific calculations or for teaching and learning science or technology topics where mathematical calculations and associated graphics are important. We will assume that readers who have stayed with us this far are familiar with the simple mathematical ideas involved.

You do **not** need to be an expert programmer to use the Toolkit, but apart from direct use of the Demonstration Stack as mentioned above, the Toolkit is intended mainly for those who are creating their own stacks. The Toolkit achieves its full potential when used at the HyperCard scripting level,

and we will mostly assume that readers are familiar with Hypertalk, the HyperCard scripting language. For those who are in the process of becoming familiar with Hypertalk, we have tried to be helpful by providing some standard buttons, complete with their own script, which can be copied and pasted into users' own cards.

1.4 Why use it?

In most cases, the greatest amount of effort in writing software for others to use goes into writing the *user interface*. HyperCard makes it very quick and easy to create an intuitive user interface, and has become an established variant of the Macintosh style. There is a price to pay for these advantages, and that is a loss of processing speed. HyperCard has therefore been an attractive system for authors of educational software where processing speed is not critical, but where presentation and ease of control by the user are the uppermost consideration.

These characteristics have limited the usefulness of HyperCard as an authoring system for upper secondary science and mathematics. For example, even plotting simple mathematical functions is unacceptably slow.

The Toolkit can evaluate mathematical expressions and plot them almost as quickly as if you had programmed the Macintosh directly in a high-level language. It can do this without sacrificing the advantages of its authoring facilities. For example, it is now practical to include Fourier Transforms and the solution of differential equations in a HyperCard stack, yet allow users to type in the defining mathematical functions and the required parameters using standard HyperCard facilities.

1.5 How to use the Toolkit?

That is the \$64,000 question, and that is what this manual is all about. As well as the manual, we are providing information in stack form. There is a Demonstration Stack, which contains a number of cards that show various Toolkit facilities working, and there is a Reference Stack, which contains the same information as this manual, structured in HyperCard style.

So you have a choice!

- If you prefer looking at examples, start with the Demonstration Stack and examine the scripts that it contains.
- If you prefer a conventional written description, read this manual.
- If you like to learn on-line, use the Reference Stack, which is also linked with the Demonstration stack.

Or you can combine these sources of information as you please.



Reference Stack



Demonstration Stack

1.6 How to use this manual

As explained above, we will assume that users are reasonably familiar with HyperCard and have some experience of writing HyperCard scripts, but since this Toolkit makes use of features of HyperCard 2.0 which were not present in previous versions, we will start by explaining the general principles of operation of the Toolkit. This will include a description of how the Toolkit manages memory.

The remaining sections will then describe each Toolkit XCMD in turn.

This manual does not attempt to explain the principles behind the graphics transforms and mathematical operations used in the Toolkit. A more complete explanation will be found in *A Scientific Programmer's Toolkit*, by M.H. Beilby, R.D. Harding and M.R. Manning (Adam Hilger, 1991).

1.3 Who Can Use It?

This is the 264,000 question, and that is what this manual is all about. As we go on, we will provide information in stack form. There is a Demonstration Stack, which contains a number of cards that show various Toolkit facilities working, and there is a Reference Stack, which contains the same information as this manual, arranged in HyperCard style. We can get further by typing keys.

So you have a choice!

- If you prefer looking at examples, start with the Demonstration Stack and examine the scripts that it contains.
- If you prefer a conventional written explanation, read this manual.
- If you like to learn on-line, use the Reference Stack, which is also linked with the Demonstration Stack.

Components of the Toolkit are designed to be used in a number of ways. It is intended to be used as a learning science and mathematics, and as a reference stack. We will assume that you are familiar with the Toolkit and its components.

You do not need to be an expert programmer to use the Toolkit. It is designed to be used as a learning science and mathematics, and as a reference stack. We will assume that you are familiar with the Toolkit and its components.

2. Principles of Operation of The Scientific and Graphical Toolkit

2.1 The Toolkit

The *Scientific and Graphical Toolkit* is a systematic collection of software routines that enable the user to tackle a variety of mathematical and associated graphical operations. The basic concepts of the Toolkit are machine-independent and are described more fully in the book *A Scientific Programmer's Toolkit*, by M.H. Beilby, R.D. Harding and M.R. Manning (Adam Hilger, 1991). The *Scientific and Graphical Toolkit* is a new implementation of this concept for HyperCard 2.0, and it consists of several External Commands, known as XCMDs. The concept of XCMDs is well-supported by HyperCard.

XCMDs are made available to HyperCard in the form of a Macintosh Operating System object known as a Code Resource. A Code Resource can be thought of as a free-floating piece of compiled program. The architecture of the Macintosh is such that a Code Resource cannot contain more than 32K bytes of code.

If you have made use of XCMDs before, then you are most likely used to a situation where each XCMD that you use does a particular job, receiving appropriate data from HyperCard itself and supplying HyperCard with some kind of result, and then terminating its actions leaving no other trace that it has run apart from the results supplied. Each XCMD that you used could have been thought of as a self-contained entity.

The Toolkit XCMDs are not self-contained and are best thought of as a cooperating group that provides an interface to the underlying Toolkit library. This is inevitable given the restriction of a Code Resource to 32K, since even with the most cunning and devious programming, it would not be possible to perform all the functions of the Toolkit in a mere 32K of code.



Toolkit Resources

2.2 Using the Toolkit from Your Stack

In order to use the Toolkit XCMDs, HyperCard will need to know where to find the Toolkit code resources. These are supplied within a stack called "Toolkit Resources". The simplest way of telling HyperCard, and the one which is recommended, is to include the following line in the openStack script of your stack:

```
start using stack "Toolkit Resources"
```


After this script has been obeyed, HyperCard will recognise Toolkit XCMDs, and initialisation can then be done (see next section).

Notes for experts:

- (i) If you have used XCMDs with previous versions of HyperCard, you will be familiar with the idea of copying resources into the stack that is going to use them. To do this you will have used a utility such as ResEdit, or a special HyperCard XCMD like ResCopy. These techniques can still be used with the Toolkit but the "start using" technique seems better as it avoids having multiple copies of the Toolkit resources.
- (ii) For some purposes you may wish to have a reduced version of "Toolkit Resources" which omits unused XCMDs. Make sure you keep a complete copy of the resources supplied with the Toolkit, or have a backup, and then use ResEdit or an equivalent utility. Of course, the modified stack does not have to be called "Toolkit Resources". When you open up the resource stack with your editor, you will notice that there are several types of resource besides XCMDs. ONLY resources of type XCMD should be removed.



Simple example

2.3 Using the Toolkit—the Cookbook Approach

The way that the Toolkit XCMDs cooperate is rather like several workers in an office. For each project there will be a master file, and as the job progresses, the file is passed to each worker in turn together with information and a list of requirements. Important information is transcribed to a convenient form and kept in the file, where it can be used by any of the workers who needs it. A worker can be asked to extract data from the file or alter entries, or to dispose of data that is no longer required. HyperCard acts like the job supervisor, telling each worker when to start a task and ensuring that the necessary data is in the file.

Before work can start, the master file must be set up, and the following recipe will achieve this. The recipe also deals with closing down the Toolkit, and will work even when two or more stacks are to be open simultaneously. First, open the Toolkit Resources stack, and copy the scripts for the two handlers "initToolkit" and "closeToolkit". Then open the stack you are working on, paste in the scripts you have just copied, and add calls to the handlers as follows:

```
on openStack
  - ...
  - your own script as required
  - ...
  initToolkit
  - ...
end openStack

on closeStack
  - ...
```



```

- your own script as required
- ...
closeToolkit
- ...
end closeStack

on initToolkit
- script copied from "Toolkit Resources"
- ...
end initToolkit

on closeToolkit
- script copied from "Toolkit Resources"
- ...
end closeToolkit

```

Having edited your stack script, close your stack and open it again so that the handlers are obeyed. It is possible that there will be an error message when you close your stack since the `closeToolkit` handler will be called, and be unable to close the Toolkit because it has not been opened. This error message should not recur once the scripts are set up correctly.

Once the Toolkit is set up for use, the following scripts can be used as simple examples of how the Toolkit may be used to plot data. These are only simple examples, of course, to demonstrate the principles of use. The stack "Simple example" has already been set up with these scripts.

First, set up some data to plot. There are several ways of doing this, and first we show how the Toolkit itself can generate data. In our example these are just 11 points (= 10 intervals) on a parabola $y=x^2$.

```

on data1
- first calculate data to plot
XCOMPILE "Init Dictionary", "MyDict"
XCOMPILE "Declare", "MyDict", "x"
XCOMPILE "Compile", "ParabExpr", "MyDict", "x^2"
XARRAY "bySteps", "xVals", -1, 1, 10
XEVAL "EvalExpr", "ParabExpr", "xVals", "yVals"
XCOMPILE "Dispose Dictionary", "MyDict"
XCOMPILE "Dispose Expression", "ParabExpr"
end data1

```

Alternatively we might wish the data to be typed by hand, picked out of HyperCard fields, etc. The next example uses two HyperCard variables called `xvalues` and `yvalues`, which are set up to contain the same data as above. In this example the data are simply typed in as part of the script, but of course in practice the values would be taken from a HyperCard field with values typed by the user, or perhaps they would be calculated by another XCMD or HyperCard script.

```

on data2
put "-1, -0.8, -0.6, -0.4, -0.2, 0.0, 0.2, 0.4, 0.6, 0.8, 1.0" into xvalues
put "1.0, 0.64, 0.36, 0.16, 0.04, 0.0, 0.04, 0.16, 0.36, 0.64, 1.0" into yvalues
XARRAY "ListToArray", "xVals", xvalues
XARRAY "ListToArray", "yVals", yvalues
end data2

```


Whatever method is chosen, at this stage we now have two Toolkit objects, which we call *Xblocks*, containing the *x* and *y* values to be plotted. Their names are "xVals" and "yVals". Now we can set up the windows and graphs to do the plotting.

```
on plot
  XCTRL "Init Window", "window1", "Graph"
  XGRAPH "Init Graph", "Parabola", "-1,-1", "-1,0,1,1"
  XGRAPH "Draw Axes", "Parabola", "window1"
  XGRAPH "Draw Graph", "Parabola", "XVals", "YVals", "window1"
  XARRAY "Dispose Array", "xVals"
  XARRAY "Dispose Array", "yVals"
  XGRAPH "Dispose Graph", "Parabola"
  - <allow a pause to view graph, then .. >
  wait 2 secs
  XCTRL "Dispose Window", "window1"
end plot
```

There are several features here to be noted. The first is that Toolkit XCMD calls can have several arguments, but the first is always a quoted string which is called an *Action*. Each XCMD can perform several actions, and the remaining arguments depend on what that action is. Full details of each action are given in this manual and in the Reference Stack. You will also note that there is a great deal of initialising and eventually disposing of things. This may seem excessive for a simple example, but in the next section we explain why the Toolkit works like this. Here we only note briefly that the approach permits great flexibility, such as the ability to have several windows and several graphs and to draw whichever graph or graphs you want in any window. It has already been noted that the data itself is not passed directly to the graph drawing XCMD, but is first placed in *arrays*. Once again you may think that this needlessly complicates matters, but there are reasons which the next section goes into more fully.

With the above in mind, a narrative of the above scripts is as follows:

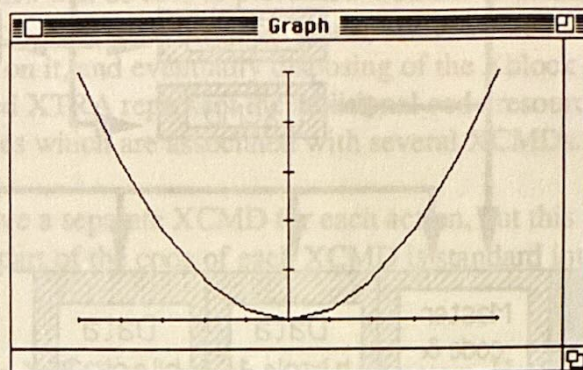
Before doing anything at all, initialise the Toolkit.

In the first approach to generating data, set up a dictionary for use in compiling expressions, declare just one variable *x*, compile the expression for the parabola, then set up a mesh of *x*-values, evaluate the formula at these values storing the resulting *y*-values, and finally tidy up by disposing of objects no longer needed.

In the second approach to generating data, set two local variables to hold the data to be plotted in string form. (This is not strictly necessary: the strings could be entered directly as XCMD arguments, but this way makes the scripts easier to read.) Next, convert the data into Toolkit arrays.

When it comes to plotting, set up a window called "window1" which will appear with a title "Graph". This window will have default size. Define a graph to be called "Parabola" which will have its *x*-axis from -1 to +1 and its *y*-axis from 0 to 1. The third argument allows you to define the screen size in which the graph is to fit, but setting this argument to "-1,-1" gives you the default size. Now draw the axes for the graph in "window1". Then draw the graph itself in "window1", taking the data from the data arrays previously created. Finally, dispose of the data, the graph, the window, and the Toolkit 'file' itself. Incidentally, if you were to close the stack "Simple Example" after having

obeyed one of the data scripts, but *before* obeying the plot script, you will be given an error message because the Toolkit does not like to be closed down whilst there are various undisposed objects still in existence.



2.4 More details of operation

At first sight the most convenient way to keep the 'file' and any data in it might seem to be to use HyperCard fields or global variables. The main reason why we do not do this is that *all* such HyperCard data is stored in character format. This is one of the attractions of HyperCard, making it so easy for users to input data, edit it, etc., but there is a considerable price to pay for these advantages in terms of speed when it comes to anything more than trivial numerical operations. Try for example (if you are familiar with HyperCard scripts) writing a script to plot $\cos(2\pi x)$ from $x=0$ to 2 using just 50 intermediate values. By contrast, an equivalent Toolkit script will work much faster, and part of the reason for the Toolkit's speed is that data is stored in the same way as a conventional application written in a high-level language.

Another reason for not using HyperCard fields to store data for the Toolkit is that it would complicate matters for users because the names of fields or globals would have to be fixed in advance. Users would need to know all the names that were likely to be used. It would also be difficult to link between stacks since there could be fields with the same name in different stacks. We have therefore used a method which is much simpler from the user's point of view. When the Toolkit is first referenced, a 'master block' is automatically set up. Note that although we have used the analogy of an office file to explain the purpose of the master block, it is not a disk file in the computing sense. It is in fact a section of the memory which was free before being allocated for this purpose. Unused memory in the Macintosh is known as 'the heap', and there is an operating system facility known as the 'memory manager' which is responsible for allocating such blocks of memory. None of these details need bother the normal user if the instructions for using the Toolkit are followed, but we give them, as many users like to know what's going on behind the scenes.

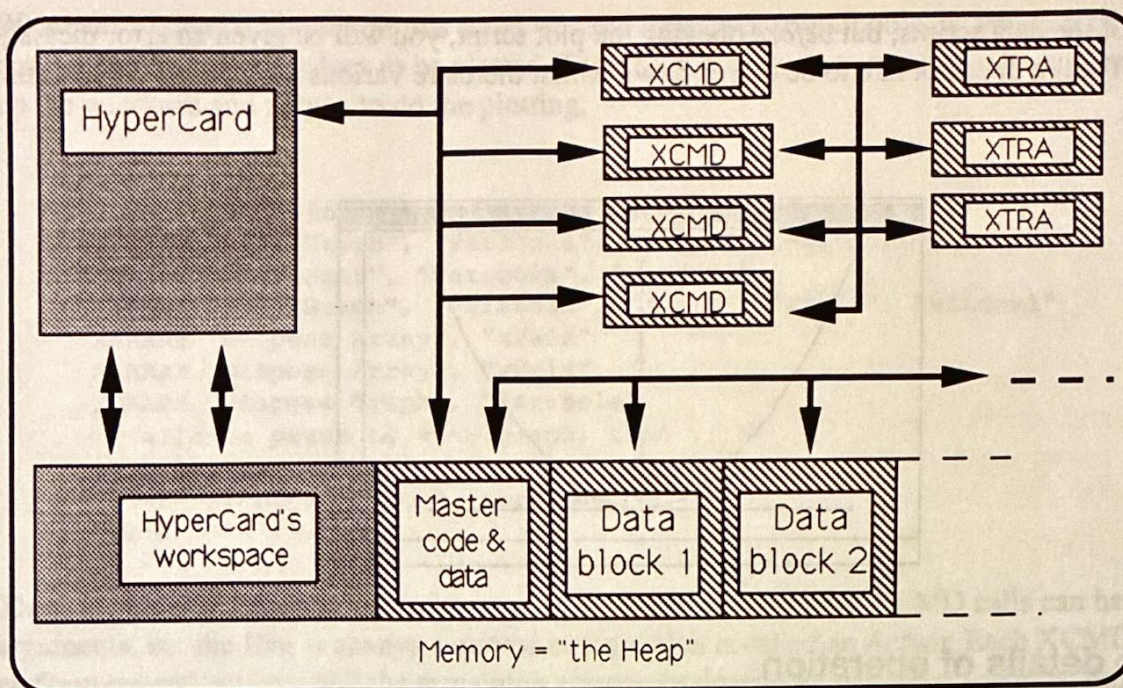


Figure 2.1

Figure 2.1 illustrates this system. The master block is the only part of memory outside HyperCard's own management that is directly relevant to both HyperCard and the Toolkit XCMDs. Any other blocks of memory which the Toolkit needs are allocated by the memory manager, and an index is kept in the master block; we will call such blocks Xblocks. When the user requests an XCMD to pass data back to HyperCard in HyperCard's character format, the conversion is done by one of HyperCard's service routines and the data stored in the part of the memory under HyperCard's own management. This conforms to HyperCard official protocols and so everything is in order and the integrity of the system is ensured.

In the example of the previous section, when "initToolkit" was called, the master Xblock was set up. When the "Init Window" action was performed, another Xblock was created to store information about the window, such as its title and size. Similarly, the action "Init Graph" creates an Xblock to remember the details of the graph called "Parabola". In passing, note again that a window and a graph are different objects, and this is emphasised by having separate blocks in which to keep information. In terms of our office 'file' analogy, an Xblock is like adding another piece of paper to the file, and updating the cover sheet accordingly. When the action "Draw Axes" is performed, two separate things happen: the axes are 'drawn' on the graph, but they are only visible because there is another argument "window1" present which means that the graph is simultaneously drawn in the window. Had the "window1" argument been omitted, the axes would have been constructed in nonvisible form for possible later display. The actions "ListToArray" can be thought of as transcribing the data in a customer's letter onto a standard office form. Two Xblocks are set up, called "xVals" and "yVals" to hold the converted data. Now at last the action "Draw Graph" can be called, adding a plot of the data to the graph and at the same time displaying the updated plot in "window1". Once the job is done, all the bits of paper in the 'file' are disposed of, and once the 'file' itself is empty, it too can be disposed.

As noted in the previous section, if an attempt is made to close the Toolkit before all the Xblocks have been removed, this is treated as a serious error because there would be blocks left in memory without any means of controlling them, which is a potentially damaging situation.

Usually each of our XCMDs will be able to perform a number of distinct (but related) actions. The concept of actions reinforces the relationship between various operations: for example, setting up a graph, plotting some data on it, and eventually disposing of the Xblock that defines the graph are all related. The blocks marked XTRA represent the additional code resources mentioned in 2.2. They contain a library of routines which are associated with several XCMDs.

It would be possible to have a separate XCMD for each action, but this would be an inefficient use of memory; since a large part of the code of each XCMD is standard interfacing code which must be repeated in each XCMD.

HyperCard allows any one XCMD to have up to 16 parameters. However, it is a feature of HyperCard that a single parameter can itself contain virtually unlimited data. A parameter can consist of a list of items. For example, HyperCard often uses a pair of values to indicate a point, as in "Get the MouseLoc" which would put something like "107,254" into the variable 'It'. Toolkit XCMDs make use of this powerful facility.

The movement of XCMD and XTRA blocks between disk and memory is automatic and need not be of concern in normal use. However, in certain demanding situations it may be desirable to reduce the amount of time that the system spends doing this task, and actions are provided for most XCMDs called "Preload" and "Unload" which give some control. "Preload" ensures that an XCMD is loaded into memory and locked there until "Unload"ed.

2.5 Xblock types

Xblocks are assigned to different types to ensure that (for example) data intended to define a window cannot be interpreted as a solution of a differential equation. Current types are:

WIND:	a window
GRAP:	a set of Graph parameters
ARAY:	an array of numbers
DICT:	dictionary for use by the compiler
PTRE:	a "parse tree" or compiled expression
PLET:	a palette (reserved for use in future versions)

The simple example discussed in 2.4 and 2.5 uses all these types.

2.6 Xblock names

Xblocks have names, as demonstrated by the example above. (For those familiar with programming jargon, they are actually referenced by handles, but as the user only sees names in terms of strings

appearing in XCMD arguments, the names can be thought of as names of the Xblocks themselves.) The names must be unique and conform to the following rules, but can otherwise be whatever the user wants. In the examples above, the names used were `window1`, `Graph`, `Parabola`, `XVals` and `YVals`.

- (i) No two XHANDLES may have the same name, even if they refer to Xblocks of different types.
- (ii) Case is ignored.
- (iii) Leading and trailing spaces are ignored, and word breaks are reduced to one space character.

e.g., The names “My Name”, “MY NAME” and “ my name” would all refer to the same Xblock.

2.7 Action names

The first parameter in each call to an XCMD of the toolkit always contains the ACTION. The rules of syntax for the ACTIONs are:

- (i) Case is ignored.
- (ii) All spaces are ignored.

e.g., “My action” and “ MYACTION” refer to the same ACTION.

N.B. Two actions may have the same name if they belong to different XCMDs.

2.8 Palettes

Palettes are usually associated with colour, but in the Toolkit the idea has been extended to include any drawing effect that is permitted by the Macintosh. There are three such effects: colour, pattern, and transfer mode. On black-and-white Macintoshes only the second two effects apply. The reason for grouping these effects is to simplify the user interface. When drawing a graph, for instance, there need only be one argument for palette instead of three separate arguments, one for each effect. Details are given in section 6.4.

2.9 Handling external windows

HyperCard 2.0 supports external windows, a concept that was not present in HyperCard 1.x. Users should naturally consult HyperCard 2.0 documentation for definitive details; the following notes are intended to indicate the principles behind the way that the Toolkit uses these windows.

The fundamental Macintosh graphics philosophy connects all screen graphics and text with a data structure called a *picture record*. In a normal Macintosh application, whenever a window (or part of a window) that was obscured becomes visible, then whatever regions have become exposed to view

require updating, and this is done by working through the picture record and redrawing as necessary. This is a totally different concept from that used in HyperCard's graphic layers, which contain bit-mapped graphics. External windows have many advantages for mathematical graph drawing, such as the ability to resize the window and move it around the screen as the user wishes.

In designing the Toolkit, we have taken the view that for many mathematical purposes it is the original Macintosh window techniques which users will prefer, and so we assume that mathematical graphics will normally be displayed in external windows. For those who do not wish to use external windows, it is simple to paste copies of such graphs into the HyperCard graphics layers, and indeed the external window need never be visible at all in order to do this. See section 3.5 "Copy to Card" and section 6.8 "Copy and cut". However, this process is slower than using external windows. If you are not going to use external windows, the rest of this section will not concern you.

There is just one problem associated with the Macintosh window philosophy when windows are used to show animated features. For example, a graph might contain a plot of a function against the usual axes, and on top of that there might be an animated construction showing how a certain mathematical process homes in on a certain feature, such as a maximum. The problem is, what happens if during this process the user (or some external event) brings the main card to the front layer, obscuring the graph or part of it? When the graph window resumes full visibility (which can be thought of as moving to the front layer) it will be updated by replaying all the preceding animated steps, which may not be what is wanted.

The Toolkit allows the HyperCard scriptwriter to deal with this situation by associating two types of picture record with each window, a *permanent* record and a *temporary* record. By default, everything drawn in a window is stored in the permanent record. The user may, however, specify that drawing should be to the temporary record. When a window needs to be updated, both the temporary and permanent records are redrawn: the distinction between them is that a separate action is provided to clear the temporary record alone—see section 3.3, action "Restore". When drawing animations, the sequence would be: draw background (permanent), draw first construction (temporary), clear temporary record, draw second construction (temporary), etc. It should be noted that the terms "permanent" and "temporary" have only relative meaning; the permanent record will be cleared on a "clear window" action.

2.10 How Errors are handled

A robust system needs an ERROR REPORTING MECHANISM. Toolkit XCMDs avoid bringing up dialogue boxes, since they reduce the freedom of scriptwriters to handle errors in their own fashion. Instead they use "the Result". If "the Result" is nil (an empty string), then the XCMD has worked normally, and no error has been detected.

XCMDs of The Scientific and Graphical Toolkit

3. The XCTRL Command

Control of Master Xblock and External Windows

The actions for XCTRL are:

- 3.1 "Init Toolkit", "Dispose Toolkit"
- 3.2 "Init Window", "Dispose Window"
- 3.3 "Clear Window", "Restore Window", "Clear Temp",
"Show Window", "Front Window",
- 3.4 "Copy to Card"

3.1 Actions "Init Toolkit" and "Dispose Toolkit"

The recommended methods for initialising and closing down the Toolkit were explained in sections 2.2 and 2.3. If the scripts "on initToolkit" and "on closeToolkit" are used, there will be no need ever to make direct use of the actions described in this section. We note here only that the "Init Toolkit" action causes various pieces of code and data to become resident in the heap, and that the "Dispose Toolkit" action reverses this process, releasing the space that was in use. The "Dispose Toolkit" action does not dispose any Xblocks that may have been created by the user, and the dispose action will fail if there are still other Xblocks in existence.

Syntax: XCTRL "Init Toolkit"
XCTRL "Dispose Toolkit"

e.g., on OpenStack
 XCTRL "Init Toolkit"
 end OpenStack

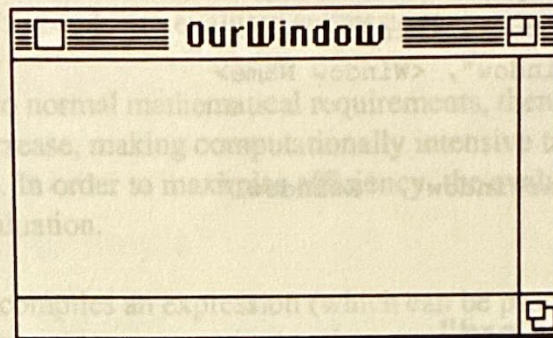
 on CloseStack
 XCTRL "Dispose Toolkit"
 end CloseStack

3.2 Actions "Init Window" and "Dispose Window"

In order to draw anything on a Mac screen there has to be a window to draw in (strictly speaking it's a PORT that's needed, but in practice they are windows that are visible). Windows need defining; for example, their dimensions, position, title and type. The HyperCard 2.0 XCMD Interface supports the creation and manipulation of external windows, known as XWINDOWS. (These are separate

from the windows in which a HyperCard card appears, which is why they are thought of as external.) The actions below control the toolkit's use of this interface.

Syntax: XCTRL "InitWindow", <Window Name> [,<title> [,<boundsRect>[,
 "<fixed|grow|shrink>[,noZoom,floating]"]]]
 XCTRL "DisposeWindow", <Window Name>



e.g., On MouseUp
 XCTRL "InitWindow", "Xwindow1", "OurWindow", "100,100,300,200"
 End MouseUp

 On MouseUp
 XCTRL "DisposeWindow", "XWindow1"
 End MouseUp

If the boundsRect is not supplied, a window of size 350 x 200 is defined in the centre of the screen. Note the order of values: boundsRect is the standard Hypertalk convention of left, top, right, bottom. If a title is not supplied or is null, then the default title is the name of the Xblock.

The final optional argument controls the type of window that is created. It may be one of three types (fixed, grow or shrink) and these may be modified by the qualifiers noZoom &/or floating. The default is grow. Notice that this is *one* argument although the string may contain more than one item. If there is more than one item, then the whole string must be enclosed in quotes. A shrink type window shows the complete picture associated with the window: any graphs drawn will be reduced to fit the window.

Note for experts:

Because of the way that Mac Window's work, "XWindow1" refers to a pointer, not a handle: this is the only occasion in this set of XCMDs where we use pointers rather than handles on a permanent basis.

3.3 Various window actions

"Clear Window" clears the named window and all picture records associated with it. "Restore Window" clears the temporary picture record (see section 2.9) and redraws the window from the permanent record alone. Note that if the window is displaying one or more graphs, then those graphs

are not cleared or disposed, but simply stop being displayed in the window. "Clear Temp" clears the temporary picture record but does not update the window. "Show Window" makes the window visible (clicking in the close box makes it invisible). "Front Window" makes the window the front window.

Syntax: XCTRL "ClearWindow", <Window Name>
 XCTRL "RestoreWindow", <Window Name>
 XCTRL "ClearTemp", <Window Name>
 XCTRL "ShowWindow", <Window Name>
 XCTRL "FrontWindow", <Window Name>

e.g., On MouseUp
 XCTRL "ClearWindow", "Xwindow1"
 End MouseUp

3.4 Action "Copy to Card"

This action copies the contents of a window directly onto the current card graphic layer. The offset defines where the top left corner of the window is aligned relative to the top left corner of the card. The default offset is (0,0). Only the permanent part of a window (see 2.9) is copied in this way.

Note that no scaling is performed; therefore, if you have drawn several graphs with differing "graphSize" parameters (see 6.4) into the same window, the copy may not achieve the desired effect.

Syntax: XCTRL "Copy to Card", <Window Name>[, <offset x,y>]

e.g., On MouseUp
 XCTRL "Copy to Card", "Xwindow1", "200,100"
 End MouseUp

Notes for experts:

Q. XCTRL doesn't seem to do a lot. Why do we have it?

A. Once a window is created, it needs maintaining with code to handle events, disposal, etc. The interface for HyperCard 2.0 is not necessarily unchanging, and so this split is used to keep the interface from the toolkit. In the future we may also want to draw graphs on a HyperCard card, have them printed out or save them to disk. These would be done in XCTRL as well.

3.2 Actions "Show Window" and "Dispose Window"

3.2.1 "Show Window" and "Dispose Window" actions

The HyperCard 2.0 interface

"Clear Window" clears the temporary picture record (see section 2.9) and redraws the window from the permanent record alone. Note that if the window is displaying one or more graphs, then those graphs

4. The XCOMPILE and XEVAL Commands

Compilation and Evaluation of Expressions

HyperCard does of course have it's own expression evaluator, but even under HyperCard 2.0, this is relatively slow, compared with evaluation under, say, Pascal. This is due to the way that HyperCard stores values and to the need to be able to evaluate expressions that reference fields, other cards, etc.

If expressions are restricted to normal mathematical requirements, then it is possible to obtain an order of magnitude speed increase, making computationally intensive tasks such as contour plotting and surface drawing feasible. In order to maximise efficiency, the evaluation is broken up into two stages: compilation, then evaluation.

The XCOMPILE command compiles an expression (which can be passed as a string or any valid HyperCard expression which evaluates to a string) and generates an Xblock, which contains a representation of the expression. The Xblock is evaluated by other XCMDs as needed. Xblocks produced by XCOMPILE are of type PTRE (see section 2.5).

The actions for XCOMPILE are:

- 4.1 "Init Dictionary"
- 4.2 "Dispose Dictionary"
- 4.3 "Declare"
- 4.4 "Compile"
- 4.5 "Dispose Expression"

The actions for XEVAL are:

- 4.4 "EvalExpr"

The actions for XEVAL are:

- 4.4 "EvalExpr"
- 4.5 "Evalsimple"
- 4.6 "Eval At"

Actions applying to both:

- 4.7 "Preload" and "Unload"

4.1 XCOMPILE: Actions "Init Dictionary" and "Dispose Dictionary"

In order to interpret an expression, a dictionary is needed to look up information such as the names of variables which are to be allowed in the expression. "Init Dictionary" creates a dictionary, and "Dispose Dictionary" disposes of it. On initialisation the dictionary contains definitions for the standard functions exp, log, ln, sqrt, sin, cos, tan, atan, acos, asin, cosh, sinh, tanh, abs, rnd, fnH and the constant pi.

[N.B. fnH is the Heaviside Function : $\text{fnH}(x) = 0$ if $x < 0$; $\text{fnH}(0) = 0.5$; $\text{fnH}(x) = 1$ if $x > 0$]

Syntax: XCOMPILE "Init Dictionary" <Dictionary Name>
XCOMPILE "Dispose Dictionary" <Dictionary Name>

4.2 XCOMPILE: Action "Declare"

Action "Declare" enables us to declare variables for use in the compilation process, up to a maximum of 10. The variables are declared as a list which is passed as the third parameter. These variables are added to the dictionary. The order of declaration is important, since an index number is associated with each variable according to its position in the list.

Syntax: XCOMPILE "Declare", <Dictionary Name>,
"var1[,var2[, var3[, ...var10]...]]"

e.g., on MouseUp
 XCOMPILE "Init Dictionary", "Shorter Oxford"
 XCOMPILE "Declare", "Shorter Oxford", "x,y,z,t"
 end MouseUp

4.3 XCOMPILE: Actions "Compile" and "Dispose Expression"

Action "Compile" creates an Xblock that contains information about the expression in a form that can be used to evaluate it with values substituted for the variables declared in the chosen dictionary. Action "Dispose Expression" deletes the Xblock. In an expression, all the usual operators $^$ $*$ $/$ $+$ $-$ and parentheses (...) may be used.

Syntax: XCOMPILE "Compile", <Compiled Name>, <dictionary name>, <expression>
XCOMPILE "Dispose Expression", <Compiled Name>

where 'compiled name' is the name of the Xblock to be used to store the compiled expression.

e.g., on MouseUp
 XCOMPILE "Init Dictionary", "Shorter Oxford"
 XCOMPILE "Declare", "Shorter Oxford", "x,y,z,t"
 XCOMPILE "Compile", "Modulus", "Shorter Oxford", "sqrt(x^2+y^2+z^2)"
 end MouseUp

Any syntax errors cause the HyperCard variable "the Result" to be set up with a suitable error message.

Note for experts:

An Xblock created by "Compile" has type PTRE.

4.4 XEVAL: Action "EvalExpr"

Action "EvalExpr" evaluates an expression which has been compiled using XCOMPILE above. The values used for the variables in the expression are taken from arrays. For details of how to create arrays, see the XARRAY command.

Syntax: XEVAL "EvalExpr", <Compiled Name>, <Variable Values>, <Results>

where <Variable Values> contains a list of names of arrays. The order of the names is important: the first array gives values for the variable number 1 (that is, the first variable in the list used when declaring variables), etc. <Results> is the name of an array which has dimension equal to the number of variables used (which must also therefore equal the number of items in the <Variable Values> list). The expression is evaluated for every combination of values. If one or more of the array names in the <Variable Values> list is replaced by a valid number (or omitted by using two commas with white space in between) then the corresponding variable is taken to be that value (or zero). For each array replaced by a constant, the dimension of the solution decreases by 1. Section 5.6 example (i) gives an example of a one-dimensional evaluation, and section 5.6 example (ii) shows how values can be created on a two-dimensional array.

4.5 XEVAL: Action "EvalSimple"

Action "EvalSimple" evaluates an expression which has been compiled using XCOMPILE above, but unlike action EvalExpr the result is always a one-dimensional array. The values used for the variables in the expression are taken from the arrays in the <Variable Values> list. For details of how to create arrays, see the XARRAY command. The values are used in the order first element of each array, then second element of each, etc. If there is only one array in the list, then the effect is exactly the same as for action EvalExpr.

Syntax: XEVAL "EvalSimple", <Compiled Name>, <Variable Values>, <Results>

where <Variable Values> contains a list of names of arrays. The expression is evaluated for each corresponding value in the arrays.

e.g., on MouseUp
 XCOMPILE "Init Dictionary", "Shorter Oxford"
 XCOMPILE "Declare", "Shorter Oxford", "x,y,z,t"
 XCOMPILE "Compile", "Modulus", "Shorter Oxford", "sqrt(x^2+y^2+z^2)"
 XEVAL "EvalSimple", "Modulus", "XValues,YValues,ZValues","RValues"
 end MouseUp

This will create an array in the Xblock called "RValues", which the moduli of the x,y,z values stored in the arrays "XValues,YValues,ZValues".

e.g., on MouseUp
 XCOMPILE "Init Dictionary", "Shorter Oxford"
 XCOMPILE "Declare", "Shorter Oxford", "x,y"
 XCOMPILE "Compile", "An expr", "Shorter Oxford", "x*y"
 XEVAL "EvalSimple", "An expr", "XValues,2","Double x"
 end MouseUp

The effect is to double each value in the array "XValues".

Note for experts:

The Xblocks used to store arrays of values are of type ARRAY. If the arrays have different definitions (that is, different sizes or starting indices) then index matching is performed. Evaluation is only done for the intersecting set of indices.

4.6 XEVAL: Action "Eval At"

Action "Eval At" evaluates an expression that has been compiled using XCOMPILE above, but unlike actions EvalExpr and EvalSimple, it performs the evaluation just once at values given in the <Variable consts> argument and returns a single result in "It".

Syntax: XEVAL "Eval At", <Compiled Name>,<Variable consts>

e.g., on MouseUp
 XCOMPILE "Init Dictionary", "Shorter Oxford"
 XCOMPILE "Declare","Shorter Oxford", "x,y"
 XCOMPILE "Compile","An expr","Shorter Oxford","x*y"
 XEVAL "Eval At", "An expr", "6,7"
 end MouseUp

The effect is to set "It" to the answer of the question of the meaning of Life, the Universe, and Everything.

4.7 Actions "Preload" and "Unload"

These actions apply to both XCOMPILE and XEVAL. "Preload" causes frequently used code to be made resident in memory (to reduce disk transfers). "Unload" reverses this effect.

Syntax: XEVAL "Preload|Unload"
 XCOMPILE "Preload|Unload"

e.g., On MouseUp
 XEVAL "Preload"
 End MouseUp

5. The XARRAY Command

Creation and manipulation of Xblock data

Numerical data used by the Toolkit is stored in Xblocks. There are three kinds relating to the three main numeric data types used in Pascal: extended, integer and longint. Each array is considered to be indexed, starting at index 0 by default. The starting index may be changed. In simple use with graph plotting, arrays will normally be indexed from 0 to N, so that

$$N = (\text{number of items of data} - 1).$$

Arrays may also be multi-dimensional, however in these cases the use of some actions are excluded.

The actions for XARRAY are:

5.1	"BySteps"	5.5	"GetOne"
5.2	"Dispose Array"		"PutOne"
5.3	"Declare"	5.6	More Examples
	"Get Info"	5.7	Array utilities
5.4	"ArrayToList"	5.8	Vector and matrix calculations
	"ListToArray"	5.9	"Preload" and "Unload"

Sections 5.1 to 5.5 contain information and examples that are needed for most purposes. In section 5.6 there are some more elaborate examples, which illustrate how the indexing works. Sections 5.7 on contain more array actions which extend the usefulness of XARRAY from being merely a data manipulator to being a vector and matrix calculator.

5.1 Action "BySteps"

The Action "BySteps" creates an array with the first index set to 0. It takes the values in the parameters and calculates the interval for the data to be stepped, putting the resulting set of numbers into the new Xblock with the specified name.

Syntax: XARRAY "BySteps" ,<Array Name>, <from>, <to>, <number of steps>

e.g., on MouseUp
 XARRAY "BySteps", "An array", -10, 10, 20
 end MouseUp

would create an array with elements, under an imposed indexing, with the values:

$$a[0] = -10, a[1] = -9, a[2] = -8, \dots, a[20] = 10.$$

If the array has already been declared, then the indices may be different; see action "Declare" below for details. It is not intended that arrays need be declared for most normal purposes.

5.2 Action "Dispose Array"

As usual Xblocks need disposing.

Syntax: XARRAY "Dispose Array", <Arrayname>

5.3 Actions "Declare" and "Get Info"

The Toolkit will usually be able to set up arrays which are appropriate for the job in hand but users will sometimes need to take control and impose dimensions and indexing of their own. A general array will have the structure $a[i, j, k, \dots]$ and the number of indices is called the *dimension* of the array a . Each index can run over a range of consecutive integers indicated by $i0 \dots imax$ etc. Action "Declare" only defines the number of indices and the starting value of each. $imax$, $jmax$, etc. are determined by the highest index for which a value is set in use after "Declare". Immediately after "Declare", the array has just one zero value and $imax=i0$, $jmax=j0$, etc.

Action "Get Info" puts information about an array into the HyperCard variable "It". The information consists of the minimum and maximum values stored in the array, the total number of elements, the dimension of the array, and then the index range for each dimension. The format is: min, max, No elements, dim, $i0$, $imax$, $j0$, $jmax$. . .

In this manual, when the word *array* is used it will usually refer to a one-dimensional array with an index range $0 \dots N$, but when it is necessary to be specific we will use the notation $array1$, $array2$, etc. to show the dimension. We will use the word *matrix* to refer to an $array2$ which is indexed from $i=1 \dots M$, $j=1 \dots N$. [Further, if *no* indices are declared, then the array will be treated as a *scalar* (see 5.8) in any vector and matrix calculations. This is a rather specialised use.]

There is a limit of 32K bytes per Xblock imposed by the Toolkit. Since the Toolkit uses the extended type for numeric data, which uses 10 bytes per value, that gives a limit of just over 3200 values per array.

Syntax: XARRAY "Declare", "<Array Name>", " $i0, j0, \dots$ "
XARRAY "GetInfo", "<Array Name>"

e.g., on MouseUp
 XARRAY "Declare", "my vector", "1"
 XARRAY "GetInfo", "my vector"
 put It
 end MouseUp

If an array is resized (declared for a second time), then the original Xblock is disposed automatically. Resizing therefore empties an array.

Notes for experts:

- (i) If an array is referenced which has not yet been declared, then it will be assumed that it is to be an $array1$ with the first index $i0=0$ unless the context of the XCMD being used requires other-

wise. Usually such a reference will occur because an array is being used to store some results. If such a results array already exists, it must be of the correct dimension (i.e., the same dimension as would be set up by default), and then its old contents are disposed and replaced with new values. In these circumstances the array will be implicitly resized if necessary; the first index *i0* is kept as found but *imax* may be changed.

(ii) To find out what dimensions, etc., an array has, use the "Get Info" action.

5.4 Actions "ArrayToList" and "ListToArray"

These are provided to help exchange data between HyperCard fields and variables on the one hand and the data stored in Xarrays on the other. You can think of "ListToArray" as an automated version of "BySteps".

Syntax: XARRAY "Array To List", <Array Name>, "<HC global>"
XARRAY "List To Array", <Array Name>, <data List>[, <sizes>]

e.g., on MouseUp
XARRAY "Declare", "my vector", "1"
XARRAY "List To Array", "my vector", "2.2,1.1,3.3"
XARRAY "Array To List", "my vector", "It"
put It
end MouseUp

The <sizes> argument in action "ListToArray" is only needed if the data is of dimension 2 or higher. For lower dimensions, the length of the data list determines the size uniquely. <sizes> is simply a list of integers giving the number of index values associated with each index. If you use "ListToArray" with the optional size parameter set greater than the number of items in the data list, then the data is filled out with zeros. This provides a simple way of setting up a dummy array.

5.5 Actions "GetOne" and "PutOne"

These actions allow specific values to be looked up or set.

Syntax: XARRAY "Get One", "<Array Name>", *i,j,...*
XARRAY "Put One", "<Array Name>", value, *i,j,...*

e.g., on MouseUp
XARRAY "Declare", "my vector", "1,3"
XARRAY "List To Array", "my vector", "2.2,1.1,3.3"
XARRAY "PutOne", "my vector", 4.4, 2
XARRAY "Array To List", "my vector", "It"
put It
XARRAY "GetOne", "my vector", 3
put It
end MouseUp

An attempt to “Get One” or “Put One” with an illegal value will result in an error. The number of indices must be the same as the dimension of the array, except that when dealing with a scalar (see 5.8) a single index of zero must be used.

5.6 Examples of use with XEVAL

A common use for arrays is to store values calculated by the XEVAL command. Here are some examples which will help clarify the rules about the dimension and indexing of arrays.

- (i) Evaluate $y = \exp(-x^2)$ for x from -5 to 5 with 10 intervals.

```
on MouseUp
  XCOMPILE "Init Dictionary", "MyDict"
  XCOMPILE "Declare", "MyDict", "x"
  XCOMPILE "Compile", "Gaussian", "MyDict", "exp(-x^2)"
  XARRAY "bySteps", "xvals", -5, 5, 10
  XEVAL "EvalExpr", "Gaussian", "xvals", "yvals"
  XARRAY "ArrayToList", "yvals", "It"
  put It
end MouseUp
```

Neither of the Xblocks `xvals` nor `yvals` has been declared explicitly, and they will both have `array1` type and be indexed from 0 to 10.

- (ii) As for (i) but with `xvals` declared explicitly.

```
on MouseUp
  XCOMPILE "Init Dictionary", "MyDict"
  XCOMPILE "Declare", "MyDict", "x"
  XCOMPILE "Compile", "Gaussian", "MyDict", "exp(-x^2)"
  XARRAY "Declare", "xvals", "1"
  XARRAY "BySteps", "xvals", -5, 5, 10
  XEVAL "EvalExpr", "Gaussian", "xvals", "yvals"
  XARRAY "ArrayToList", "yvals", "It"
  put It
end MouseUp
```

“BySteps” notes that `xvals` is already declared `array1` type, which is correct, and so `xvals` is re-used, but the starting index is kept as 1 rather than 0 as by default in example (i). “EvalExpr” creates `yvals` which inherits the starting index from `xvals`. The “ArrayToList” action should give exactly the same output as in (i) because the indexing does not affect the actual values in the Xblock.

- (iii) Evaluate $\exp(-(x^2 + y^2))$ on the square $-2 \leq x \leq 2$, $-1 \leq y \leq 1$ using 20 intervals for x and y .

```
on MouseUp
  XCOMPILE "Init Dictionary", "MyDict"
  XCOMPILE "Declare", "MyDict", "x,y"
  XCOMPILE "Compile", "expr", "MyDict", "exp(-(x^2+y^2))"
  XARRAY "bySteps", "xevals", -2, 2, 20
  XARRAY "bySteps", "yevals", -1, 1, 20
  XEVAL "EvalExpr", "expr", "xevals", "yevals", "fnvals"
```



```

XARRAY "ArrayToList", "fnvals", "It"
  put It into card field "results"
end MouseUp

```

Note that the results are displayed in *row order*. That is, the first line of the results display corresponds to fixed values of all the variables except the last, the order of the variables being defined by the "Declare" action. In this example, the first line of results corresponds to $x=-2$ with y -values running from -1 to 1.

Instead of using "ArrayToList" to inspect the results it is possible to inspect selected values; e.g.,

```

XARRAY "GetOne", "fnvals", i, j

```

This obtains the value $f(x_i, y_j)$ i.e., the value at the i th value of x , j th value of y . Although the values are stored as a 2D array, (i.e., a matrix) it should be noted that if displayed in normal matrix conventions, it is the row which corresponds to changing y and the column which corresponds to increasing x in the downward direction. This possible source of confusion is not the fault of the Toolkit, but is ingrained in mathematical notation, since matrix elements a_{ij} are by convention displayed as follows:

col no. $j \rightarrow$		1	2	3	
row no. i	1	a_{11}	a_{12}	a_{13}	
	$\downarrow$	2	a_{21}	a_{22}	a_{23}
		3	a_{31}	a_{32}	a_{33}
		:	:	:	

We have judged that it is less confusing to match the order of variables with the order of indices. Thus if we are using 3 x -values (= 2 intervals) ($x_0 = 0, x_1 = 0.1, x_2 = 0.2$), and 6 y -values (= 5 intervals) ($y_0 = 0, y_1 = 1, y_2 = 2, y_3 = 3, y_4 = 4, y_5 = 5$), then display using the actions using "ArrayToList" would give:

$f(x_0, y_0)$	$f(x_0, y_1)$	$f(x_0, y_2)$	 $f(x_0, y_5)$
$f(x_1, y_0)$	$f(x_1, y_1)$	$f(x_1, y_2)$	 $f(x_1, y_5)$
$f(x_2, y_0)$	$f(x_2, y_1)$	$f(x_2, y_2)$	 $f(x_2, y_5)$

5.7 Array utilities

The actions in this section provide further facilities for manipulating arrays.

Arrays can be scaled and have a constant added:

```

XARRAY "scale", "<Array>", "const"[, "<result>"]
XARRAY "add", "<Array>", "const"[, "<result>"]

```

e.g., on MouseUp

```

XARRAY "bySteps", "mesh", 0,10,10
XARRAY "scale", "mesh", 0.1
XARRAY "add", "mesh", 5
end MouseUp

```


1D arrays can be concatenated:

```
XARRAY "concat", "<ArrayList>", "<result>"
```

e.g., on MouseUp
 XARRAY "bySteps", "fine", 0,1,20
 XARRAY "bySteps", "course", 1.1,2,9
 XARRAY "concat", "fine, course", "varStep"
 end MouseUp

Subranges can be extracted:

```
XARRAY "subRange", "<extract>", "<source>", "i1,j1,...", "i2,j2, ..."
```

e.g., on MouseUp
 XARRAY "subRange", "last bit", "varStep", 20, 30
 end MouseUp

5.8 Vector and matrix calculations

All arrays using these actions must be of a special sort: they must be declared to have starting indices of 1 (to conform with normal mathematical notation). They can be inner and outer product according to the usual rules,

```
XARRAY "OuterProduct", "<Array>", "<Array>", "<result>"  
XARRAY "InnerProduct", "<Array>", "<Array>", "<result>"
```

and transposed (for 2D arrays (matrices))

```
XARRAY "transpose", "<matrixName>[", "<result>"]
```

and inverted (for nonsingular nxn matrices)

```
XARRAY "invert", "<matrixName>[", "<result>"]
```

e.g., on MouseUp
 XARRAY "Declare", "xvec", 1
 XARRAY "Declare", "yvec", 1
 XARRAY "ListToArray", "xvec", "1,2,1"
 XARRAY "ListToArray", "yvec", "1,-1,1"
 XARRAY "InnerProduct", "x dot y", "xvec", "yvec"
 XARRAY "ArrayToList", "x dot y", "It"
 put It
 XARRAY "OuterProduct", "matA", "xvec", "yvec"
 XARRAY "transpose", "matA"
 XARRAY "invert", "matA", "matB"
 XARRAY "InnerProduct", "matA", "matB", "unit"
 XARRAY "ArrayToList", "unit", "result"
 end MouseUp

Note that the "<result>" Xblock is dimensioned implicitly according to the input arrays. In the case of an inner product, one index from each array is contracted, which in the example above means that "x dot y" is a *scalar*. An array may be declared as a scalar (see 5.3), and then either the inner or the outer product may be used to scale all the values in an array. However the "scale" action (see 5.7) may be more conveniently used for this purpose.

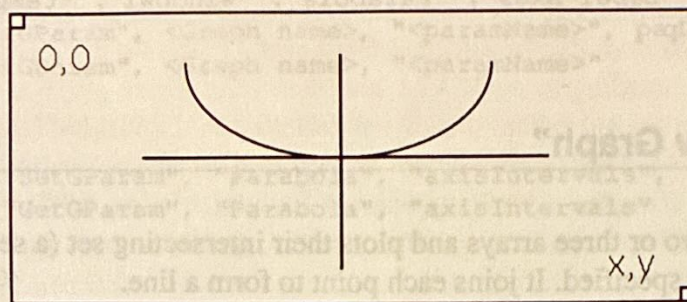
5.9 Actions "Preload" and "Unload"

"Preload" causes frequently used code to be made resident in memory (to reduce disk transfers).
"Unload" reverses this effect.

Syntax: XARRAY "Preload|Unload"

e.g., On MouseUp
 XARRAY "Preload"
 End MouseUp

e.g.,



Graph Size = "x,y"

If the `<ranges>` parameter is a list of 4 values, then it is taken as the ranges (in user coordinates) `xlo, ylo, xhi, yhi` and 2D axes will be set up. The `<ranges>` parameter can also be a list of 6 values `xlo,ylo,zlo,xhi,yhi,zhi` in which case 3D axes are set up. Note that the codes for ranges are given in the order low values, then high values; this results in an ordering: left bottom right top. This order is consistent with normal mathematical practice for coordinates, but different from the standard Mac order for specifying screen rectangles, which is left top right bottom (since the vertical screen coordinate on the Mac runs down rather than up). However, the XCMD will reorder values so that `xlo,yhi,xhi,ylo` has the same effect as `xlo, ylo, xhi, yhi`.

If the `<ranges>` parameter is not specified, then the ranges remain as defined in the default set of graphics parameters, which is -10 to +10 for all axes.

If the view angles are not specified then a default is chosen which will be dependent on whether the ranges were chosen to be 2D or 3D. The angles are expressed in degrees in the form "Theta, Phi" where Theta and Phi are defined in the usual way.

Note that a graph is independent of a window. This allows users to drag, shrink and otherwise manipulate the window in which a graph is displayed.

On successful conclusion, an Xblock of type GRAP is set up with i's handle stored in the master table under `<Graph Name>`.

6.2 Actions "Draw Axes", "Label Axes", and "Draw Labelled Axes"

"Draw Axes" uses the settings in graph Xblock to calculate an appropriate set of axes and draws them onto the graph, optionally into the specified window. If the window is not specified then the drawing is "invisible" until displayed using XCTRL's "Display Graph" action. "Label Axes" adds numerical labels to the axis ends and pips. "Draw Labelled Axes" combines the two actions. Note that "Perm" and "Temp" are permitted abbreviations for "Permanent" and "Temporary": see section 2.9 for their effect.

Syntax: `XGRAPH "Draw Axes", <Graph>[, <Window>[, "<Perm|Temp>"]]`
`XGRAPH "Label Axes", <Graph>[, <Window>[, "<Perm|Temp>"]]`
`XGRAPH "Draw Labelled Axes", <Graph>[, <Window>[, "<Perm|Temp>"]]`

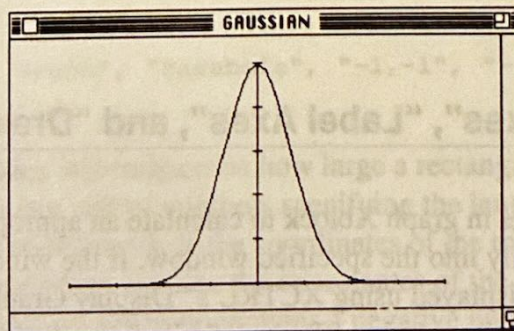

```
e.g.,      On MouseUp
           XGRAPH "Draw Axes", "Parabola", "window1"
           XGRAPH "Label Axes", "Parabola", "window1", "temp"
           End MouseUp
```

6.3 Action "Draw Graph"

"Draw Graph" takes two or three arrays and plots their intersecting set (a set of point pairs or triplets) onto the graph specified. It joins each point to form a line.

Syntax: XGRAPH "Draw Graph", <Graph name>, <XArray>,
 <YArray>[, <ZArray>...] [, <Window Name>]

```
e.g.,      On MouseUp
           - first calculate data to plot
           XCOMPILE "Init Dictionary", "MyDict"
           XCOMPILE "Declare", "MyDict", "x"
           XCOMPILE "Compile", "GaussExpr", "MyDict", "exp(-x^2)"
           XARRAY "bySteps", "xVals", -4, 4, 50
           XEVAL "EvalExpr", "GaussExpr", "xVals", "yVals"
           - now set up graph and plot data
           XCTRL "InitWindow", "Gaussian"
           XGRAPH "Init Graph", "GaussGraph", "-1,-1", "-4,0,4,1"
           XGRAPH "Draw Axes", "GaussGraph", "Gaussian"
           XGRAPH "Draw Graph", "GaussGraph", "xVals", "yVals", "Gaussian"
           - now tidy up unused Xblocks
           XARRAY "Dispose Array", "xVals"
           XARRAY "Dispose Array", "yVals"
           XCOMPILE "Dispose Expression", "GaussExpr"
           XCOMPILE "Dispose Dictionary", "MyDict"
           XGRAPH "Dispose Graph", "GaussGraph"
           - delay the next command until you've finished looking!
           XCTRL "Dispose Window", "Gaussian"
           End MouseUp
```



6.4 Actions "SetGParam" and "GetGParam"

Syntax: XGRAPH "SetGParam", <Graph name>, "<paramName>", paramvalue
XGRAPH "GetGParam", <Graph name>, "<paramName>"

e.g., On MouseUp
 XGRAPH "SetGParam", "Parabola", "axisIntervals", "7,4,4"
 XGRAPH "GetGParam", "Parabola", "axisIntervals"
 put It
 End MouseUp

The purpose of "SetGParam" is to allow the user to set the parameters that control the way graphs are set up and drawn. There are a great many attributes of graphs that can vary, and if they were all included in an XCMD parameter list, it would be a very unwieldy system. Instead, whenever a graph is initialised, the Xblock associated with the graph is set up to contain values of these parameters, known as *graphics parameters*, set to default values. As explained above in 6.1, the "Init Graph" action also makes it possible to set certain commonly used parameters like the ranges and view angles at the same time as initialisation, but it would be possible to start with the defaults and then use "SetGParam" later. These two courses of action have the same effect.

The purpose of "GetGParam" is to allow the user to find out what a particular parameter is set to. As with the Hypertalk "Get" command, the value obtained is placed in the Hypertalk variable "It".

Here is a list of graphics parameters and the template for the format in which they are presented. "Point2" means a pair of values like "x,y". "Point3" means a triplet like "x,y,z".

Name	Template	
range	point2, point2, or point3, point3	- sets/gets xlo, xhi, etc.
origin	xorig, yorig [, zorig]	- axes crossing
viewAngles	th, phi	- view angles (degrees)
ZViewplane	zvalu	- 2d view plane
axisIntervals	nxi, nyi [,nzi]	- number of intervals
axisFlags	xflag, yflag [,zflag]	- axis flags
pipModes	xpipm, ypipm [,zpipm]	- pip modes
pipLengths	xpipl, ypipl [,zpipl]	- pip length
graphSize	width,height	- graph size in screen coords
graphRect	left, top, right, bottom	- sub-rectangle for offset graphs
margins	left, top, right, bottom	- window margins
axisLabDisp	xdisp, ydisp [,zdisp]	- label displacement
graphLabDisp	labeldx, labeldy	- graph label displacement
labelField	xfield, yfield, zfield	- label field size
labelText	txFont, txSize, txFace, txMode	- label text specification
axisPalette	palette (see below for format)	- axis palette specification
pipPalette	palette	- palette for pips
labelPalette	palette	- palette for axis labels
linePalette	palette	- palette for lines and graphs

The Graphics Parameters control the way that the conversion from user coordinates to local screen coordinates is done. User coordinates over the *range* specified by the parameters will fit inside the bounding rectangle *graphRect* (local screen coordinates) which in turn fits inside a rectangle of size *graphSize* after margins have been allowed for. Note that the size of a graph is independent of the

size of a window. The axes will cross at the *origin* (user coordinates, default (0,0,0)) unless that is off screen in which case the crossing point is adjusted. The axes will have the nearest between-pip interval size to 1,2, or 5 times a power of 10 that gives approximately *axisIntervals* (default (7,7,7)) intervals. The pips on the axes can be either symmetrical (mode 0), displaced above the axis (mode 1), or below (mode 2) according to *pipModes*, and their lengths are given as percentages of the interval length along the axis to which they are parallel by *pipLengths*. When axes are drawn only the axes for which the *axisFlags* entries are True will actually be drawn (default (True, True, False)). The Toolkit treats all plotting as "false perspective" 3D projections; for 2D plotting the *viewAngles* are set to $\theta=0$ and $\phi=-90^\circ$, the z-axis flag is False so that only two axes appear, and the z-value used is *ZViewplane* (default 0). When labelling the axes, the font and other text style information is taken from *labelText* and the defaults are Geneva 9-point, plain text, patCopy mode. The labels are formatted using *labField* (default (5,5,5)) character places and the values *axisLabDisp* (screen coordinates) can be used to adjust the precise positioning of labels relative to the pips. *graphLabDisp* (screen coordinates) are used similarly to adjust the exact positioning of character strings.

The concept of a *palette* as used in the Toolkit requires a little explanation on its own. Users will know of course that the Macintosh allows a wide variety of graphic effects to be achieved, but specifying all the characteristics of such effects can be highly complex. For example, colour, pattern, and transfer mode might all need to be changed from their default settings. Individual graphics parameters could have been made available for each of these attributes, but it seems more natural to group them since together they control the appearance of whatever is about to be drawn. In XCMDS that allow palettes to be set (see for example Section 7 "Further Graphics XCMDS"), the palette concept permits just one argument to be required for this purpose instead of three or more which would be needed were each attribute to be set individually. A palette, as its name suggests, can specify a range of colour/effect combinations to choose from. This might be needed, for example, when drawing pie charts, or surfaces where the 'up' and 'down' sides are to be shaded differently, but for most purposes only the first group will be used.

A palette is specified simply by giving a list of attributes. The first three items in the list refer to the first group, the next three items refer to the second group, etc. Items may be left blank. Within each group the order is not important. For example:

```
dkGray, Xor, , yellow, ltGray
(first group)          (second group)
```

The basic attributes that may be used in palettes are:

```
patterns:      black, dkGray, gray, ltGray, white
colours:       blackColor, whiteColor, redColor, greenColor,
               blueColor, cyanColor, magentaColor, yellowColor
transfer modes: Copy, Or, Xor, Bic, notCopy, notOr, notXor, notBic.
```

It will be noted that normal Macintosh conventions are being followed, and that attributes like "white" are patterns, not colours.

In the palette string, if there is no mention of a particular type of attribute (in the above example there is no colour in the first group), then the current setting is preserved. Thus any one of the basic attributes on its own can be given wherever a palette is expected. When an XCMD uses more than one group from a palette, there is no guarantee that the palette will be used in the order in

which the groups are given, so in the above example again it is not defined whether the second group "yellow,ltGray" will use Xor mode or not. However, when "SetGParam" is used to set a palette, all the attributes in the first group of the palette are immediately made current, so that "SetGParam" can be used, for example, to ensure that the pen is in Copy mode.

In place of a palette definition string you may use the single word "default". This is treated as the name of an entire palette and will restore *Palette* or *axisPalette* to the default settings as used by the XGRAPH "Init Graph" call. In future Toolkit, implementations it is intended to allow named palettes, and to have palette editing facilities that will include the defining and storing of user-designed patterns.

We emphasise once again that for most purposes the system will set all graphics parameters to sensible defaults, and you will not need to worry about their precise effect. However, the Toolkit does allow you to alter any of the above parameters.

6.5 Actions "UserToLocal" and "LocalToUser"

Although the main purpose of XGRAPH is to take the work of converting to screen coordinates off the user's shoulders, for some purposes it may be useful to know the screen coordinates of points on a user's graph. Sometimes, it may also be required to know the user coordinates of a point on the screen whose screen coordinates are known. These two actions allow this to be done. The notation *usrpt* stands for (x,y) or (x,y,z) in user coordinates, and *locpt* stands for a point (X,Y) in local screen coordinates.

Syntax: XGRAPH "UserToLocal", <Graph name>, "<Usrpt>"
XGRAPH "LocalToUser", <Graph name>, "<Locpt>" [,ZViewplane]

e.g., On MouseUp
XGRAPH "UserToLocal", "Parabola", "0.6,0.36"
put It
End MouseUp

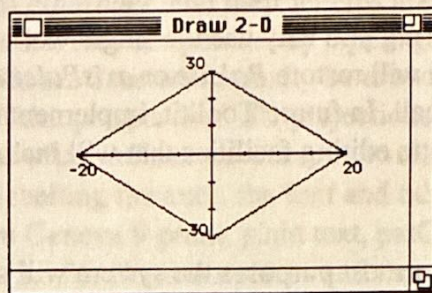
Action "UserToLocal" converts from user coordinates (2D or 3D) and puts the resulting screen coordinates (X,Y) into "It".

Action "LocalToUser" sets "It" to x,y user coordinates which would result in x,y, ZViewplane being converted to locpt. If ZViewplane is not given, ZViewplane is read from the graphics parameter *ZViewplane*.

6.6 Actions "Draw To", "Draw 2D", and "Draw 3D"

The "Draw Graph" action provides the fastest way of plotting graphs from data, but as explained it does require the data to be stored in Xblocks. Data in HyperCard fields or variables can always be converted using XARRAY's "ListToArray" action, but if there is no further use for the resulting array, it may simply be inconvenient to go through all these steps. Therefore we provide these two

actions which short-circuit the need for the conversion. It will be noted that the "DrawTo" action expects local screen coordinates in the arguments, whereas the "Draw nD" actions expect points in user coordinates.



Syntax: XGRAPH "DrawTo", <Graph>, "<Locpts list>" [, <Window>[, "<Perm|Temp>"]]]
 XGRAPH "Draw 2D", <Graph>, "<Usrpts list>" [, <Window>[, "<Perm|Temp>"]]]
 XGRAPH "Draw 3D", <Graph>, "<Usrpts list>" [, <Window>[, "<Perm|Temp>"]]]

e.g., On MouseUp — draw a lozenge on existing axes
 XGRAPH "Draw2D", "XGraph2", "0,30,-20,0,0,-30,20,0,0,30",
 "XWindow2"
 End MouseUp

6.7 Actions "Local Text" and "User Text"

These provide a means of writing text on a graph at points given by user coordinates or local screen coordinates, respectively.

Syntax: XGRAPH "LocalText", <Graph>, "<Locpt>", <string>
 [, <Window>[, "<Perm|Temp>"]]]
 XGRAPH "UserText", <Graph>, "<Usrpt>", <string>
 [, <Window>[, "<Perm|Temp>"]]]

e.g., On MouseUp — label the lozenge from previous example
 XGRAPH "UserText", "XGraph2", "10,30", "Lozenge", "XWindow2"
 End MouseUp

6.8 Actions "Copy" and "Cut"

These provide a means of copying images from Toolkit graphs to the Mac clipboard. The entire graph is copied or cut, and may then be pasted into any Mac application that accepts standard clipboard formats. There is a memory limitation imposed by the size of information that can be stored on the clipboard, which means that complicated graphs may not be easy to export in this way. For direct use within HyperCard (pasting a graph into the card, for example), use the XCTRL "Copy to Card" action.

Syntax: XGRAPH "Copy", "<Graph name>"
 XGRAPH "Cut", "<Graph name>"

e.g., On MouseUp
 XGRAPH "Copy", "Parabola"
 End MouseUp

6.9 Actions "Display Graph" and "Resize Graph"

It was explained in 2.3 and 2.4 that: (a) windows and graphs are distinct objects; and (b) it is possible to 'draw' a graph which is not visible whilst being drawn (this is done by omitting the window parameter from the arguments). Action "Display Graph" is the means by which 'invisible' graphs can be displayed, whereas "Resize Graph" is the means by which a graph and all its associated parameters are reset so that the graph size matches the window size. Note that the graph size will be smaller than the window size by 15 screen units in each direction to allow for margins.

Syntax: XGRAPH "Display Graph", <Graph name>, <Window Name>
 [, "<Permanent|Temporary>"]
 XGRAPH "Resize Graph", <Graph name>, <Window Name>

e.g., On MouseUp
 XGRAPH "Display Graph", "Parabola", "Xwindow1"
 End MouseUp

Note for experts:

In order to look at the contents of a graph, we must have a 'picture' (a PICT type resource). An XBLOCK of type GRAP will contain information pertaining to a picture record of that graph. Action "DisplayGraph" draws the PICT information into a specified window.

6.10 Actions "Preload" and "Unload"

"Preload" causes relevant code to be made resident in memory (to reduce disk transfers).
 "Unload" reverses this effect.

Syntax: XGRAPH "Preload"
 XGRAPH "Unload"

e.g., On MouseUp
 XGRAPH "Preload"
 End MouseUp

7. Further Graphics XCMDs

There are a number of separate XCMDs in this category:

- 7.1 XCONTOURS – draws a contour map for values supplied in an array
- 7.2 XREVO – draws a surface of revolution
- 7.3 XSURFACE – draws a surface for values supplied in an array
- 7.4 XBAR – draws a bar chart
- 7.5 XPIE – draws a pie chart

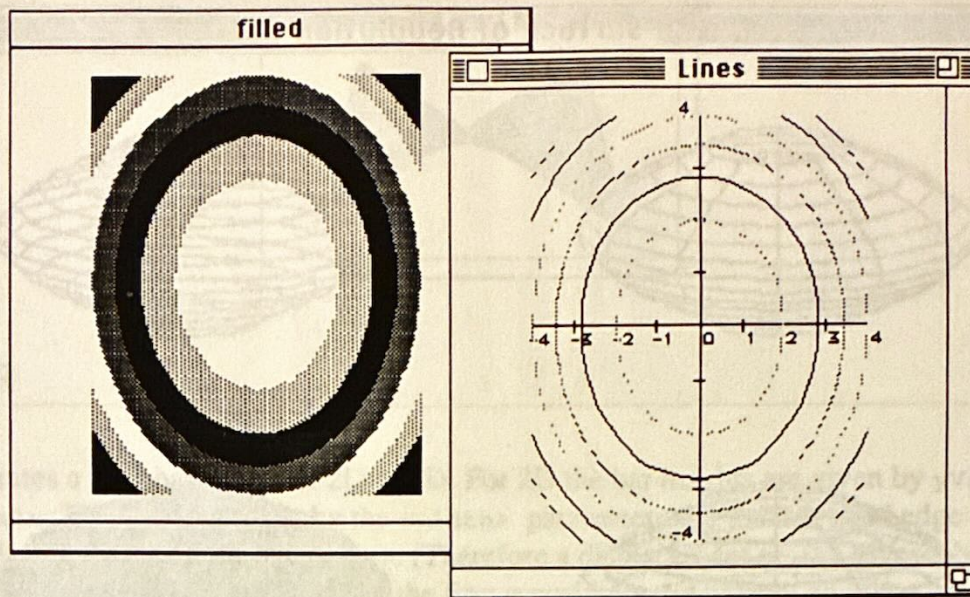
7.1 XCONTOURS

XCONTOURS draws contours on a 2D or 3D graph according to how the graph has been set up. The function to be contoured must previously have been calculated at all points of a rectangular mesh on x, y values. The x and y values used are passed as 1D arrays, and the function values must be stored in a 2D array. In accordance with the discussion in 5.7, if the function $f(x, y)$ were being contoured and the array was a_{ij} , then increasing i would correspond to increasing x , and increasing j would correspond to increasing y . Contours are drawn at the heights specified in a list using a palette supplied. (See 6.4 for a discussion of palettes.) If there is more than one group of attributes in the palette, then the groups are used cyclically. The first group will be used with the first height.

```
Syntax:  XCONTOURS "Lines|Filled", <graph>, <xVals>,
          <yVals>, <AVals>, "<heights>", "<palette>"
          [, <Window>[, "<Perm|Temp>"]]
```

```
e.g.,      On MouseUp
- first calculate data to plot
XCOMPILE "Init Dictionary", "MyDict"
XCOMPILE "Declare", "MyDict", "x,y"
XCOMPILE "Compile", "EllipseExpr", "MyDict", "x^2/4+y^2"
XARRAY "bySteps", "xVals", -4, 4, 20
XARRAY "bySteps", "yVals", -2, 2, 20
XEVAL "EvalExpr", "EllipseExpr", "xVals,yVals", "fnVals"
- now set up graph and plot data
XCTRL "InitWindow", "Contours"
XGRAPH "Init Graph", "EllipseGraph", "-1,-1", "-4,-2,4,2"
XGRAPH "Draw Axes", "EllipseGraph", "Contours"
XCONTOURS "Lines", "EllipseGraph", "xVals", "yVals", "fnVals",
    "1,2,3,4,5,6", "black,,,dkGray,,,ltGray", "Contours"
- now tidy up unused Xblocks
XARRAY "Dispose Array", "xVals"
XARRAY "Dispose Array", "yVals"
XCOMPILE "Dispose Expression", "EllipseExpr"
XCOMPILE "Dispose Dictionary", "MyDict"
XGRAPH "Dispose Graph", "EllipseGraph"
- delay the next command until you've finished looking!
XCTRL "Dispose Window", "Contours"

End MouseUp
```

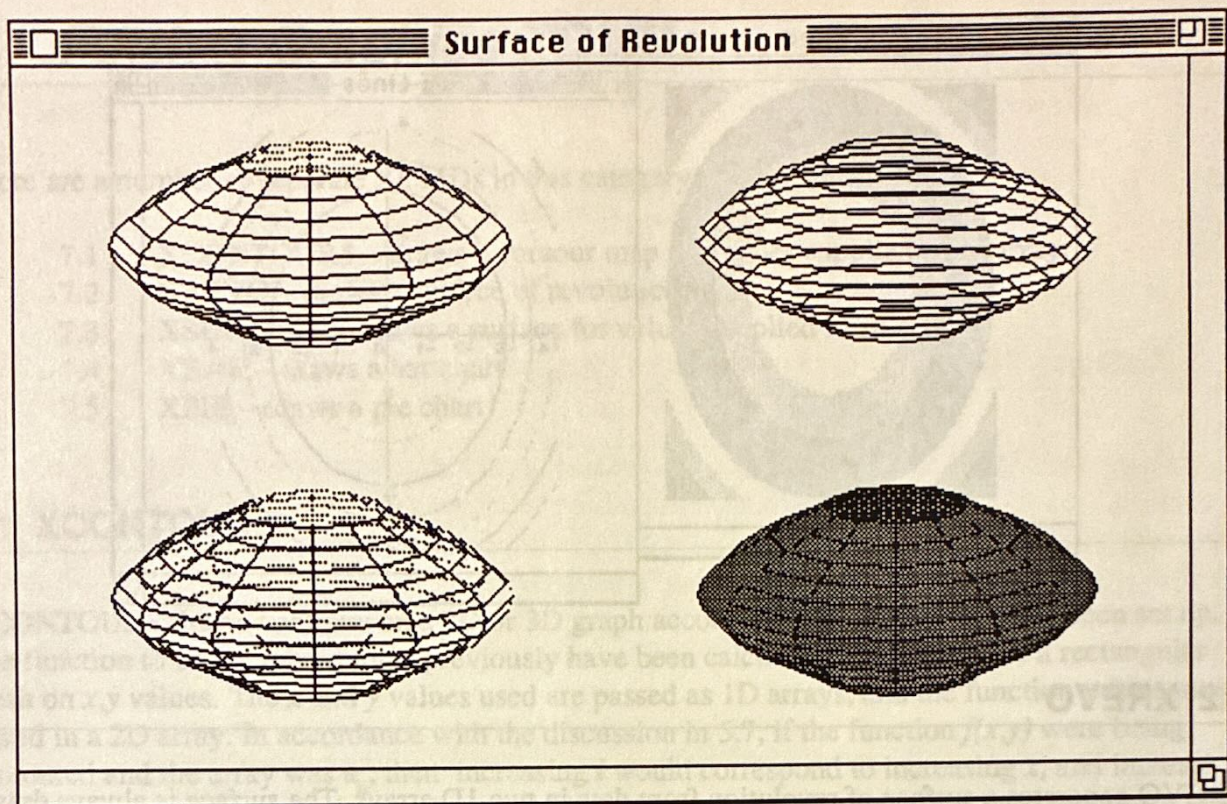



7.2 XREVO

XREVO generates a surface of revolution from data in two 1D arrays. The surface is always drawn rotated about the z-axis. One array gives z-values at which circles parallel to the x-y plane are drawn, with radii given the r-values array. There are several actions. "Contours" means that just the circles are drawn. "Wireframe" and "Surface" allow you to see inside the surface, with "Surface" performing hidden line removal. "Solid" paints the rectangles which "tile" the surface (rather than leaving them blank). There is an <nslices> parameter to control the number of facets around each circle. The first group in the palette controls the colour or style for the exterior of the surface, and the second group applies to the inside.

Syntax: XREVO "Contours|Wireframe|Surface|Solid", <graph>,
 <zVals>, <rVals>, <nslices>, "<palette>"
 [, <Window>[, "<Perm|Temp>"]]

e.g., On MouseUp
 XARRAY "BySteps" ,"TheArray" ,-1,1 ,20
 XCOMPILE "Init Dictionary" ,"Dictionary"
 XCOMPILE "Declare" ,"Dictionary" ,"x"
 XCOMPILE "Compile" ,"TheExpr" ,"Dictionary" ,"exp(-x^2)"
 XEVAL "EvalExpr" ,"TheExpr" ,"TheArray" ,"ResultArray"
 XCTRL "Init Window","DemoWind","Hello World","231,75,494,299"
 XGRAPH "Init Graph" ,"TheGraph" ,"248,209" ,—
 "-1 ,-1 ,-1, 1,1,1","60,0"
 XREVO "SURFACE" ,"TheGraph","TheArray","ResultArray" ,"24",—
 "Default","DemoWind"
 XCOMPILE "Dispose Dictionary" ,"Dictionary"
 XCOMPILE "Dispose Expression" ,"TheExpr"
 XARRAY "DisposeArray" ,"TheArray"
 XARRAY "DisposeArray" ,"ResultArray"
 XGRAPH "Dispose Graph" ,"TheGraph"
 end mouseUp



7.3 XSURFACE

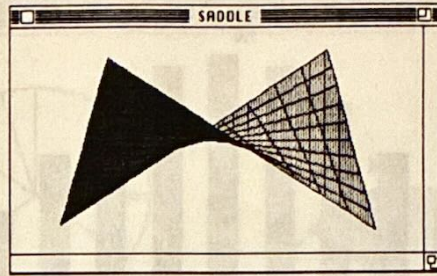
XSURFACE generates a representation of the surface defined by $z=f(x,y)$. The function must previously have been calculated at all points of a rectangular mesh on x,y values. The x and y values used are passed as 1D arrays, and the function values must be stored in a 2D array. In accordance with the discussion in 5.7, if the function was $f(x,y)$ and the array was a_{ij} , then increasing i would correspond to increasing x , and increasing j would correspond to increasing y . The outside of the surface is styled according to the first group in the palette, the inside with the second group.

Syntax: XSURFACE "Wireframe|Surface|Solid", <graph>, <xVals>, <yVals>, <zVals>, "<palette>" [,<Window>[, "<Perm|Temp>"]]

e.g.,

On mouseUp

```
XCOMPILE "Init Dictionary", "MyDict"
XCOMPILE "Declare", "MyDict", "x,y"
XCOMPILE "Compile", "expr", "MyDict", "x*y"
XARRAY "BySteps", "xValues", -10, 10, 10
XEVAL "EvalExpr", "expr", "xValues, xValues", "SurfaceArray"
XCTRL "InitWindow", "Saddle"
XGRAPH "InitGraph", "SurfaceGraph", "-1,-1", ~
    "-10,-10,-100,10,10,100", "70,80"
XSURFACE "SOLID", "SurfaceGraph", xValues, xValues, SurfaceArray, ~
    "ltGray, RedColor, Gray, BlueColor"
XGRAPH "DisplayGraph", "SurfaceGraph", "Saddle"
end mouseUp
```

7.4 XBAR

XBAR generates a bar chart in either 2D or 3D. For 2D the bar heights are given by `yVals` at positions `xVals`. The width is given by the `<width>` parameter, and the left-hand edge of each bar is displaced by `<displacement>*<width>`. (Therefore a displacement of -0.5 gives a bar centred on the required `x`-value.) The first bar uses the first group from the palette, and then each successive bar uses the next group, repeating cyclically. For 3D, the bar heights are given by `zVals` (which is a 2D array) at positions given by corresponding pairs from `xVals` and `yVals`, and the width and height are as specified.

Syntax: XBAR "2D", "<graph>", "<xVals>", "<yVals>", <width>,
<displacement>, "<palette>"[, <Window>[, "<Perm|Temp>"]]

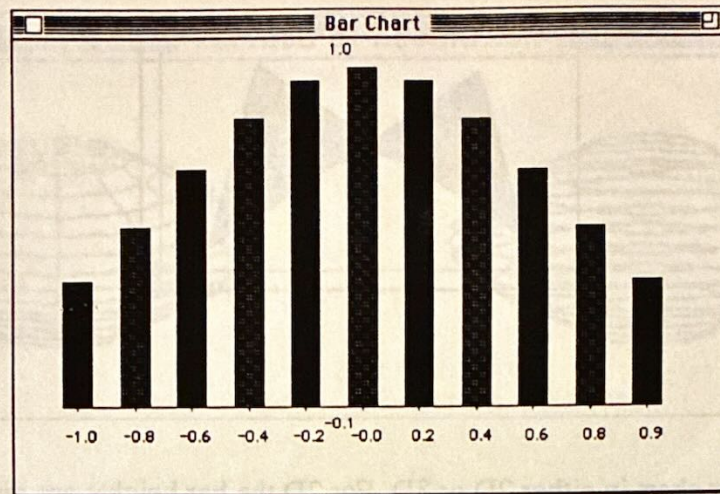
Syntax: XBAR "3D", "<graph>", "<xVals>", "<yVals>", "<zVals>",
<width>, <depth>, "<palette>"[, <Window>[, "<Perm|Temp>"]]

e.g., On mouseUp

```

- Calculate the points and store the results in an array
XARRAY "BySteps" , "TheArray" , -1, 1, 10
XCOMPILE "Init Dictionary" , "Dictionary"
XCOMPILE "Declare" , "Dictionary" , "x"
XCOMPILE "Compile" , "TheExpr" , "Dictionary" , "exp(-x^2)"
XEVAL "EvalExpr" , "TheExpr" , "TheArray" , "ResultArray"
- Set up window and draw axes
XCTRL "Init Window" , "DemoWind" , "Bar Chart" , "0,50,500,350"
XGRAPH "Init Graph" , "TheGraph" , "440,275" , "-1,-0.05 , 1 , 1"
XGRAPH "SetGParam" , "TheGraph" , "axisLabDisp" , "10,8,0"
XGRAPH "SetGParam" , "TheGraph" , "axisIntervals" , "11,11,0"
XGRAPH "Label Axes" , "TheGraph" , "DemoWind"
XGRAPH "SetGParam" , "TheGraph" , "axisFlags" , -
    "true,false,false"
XGRAPH "Draw Axes" , "TheGraph" , "DemoWind"
- Draw the bar chart
XBAR "2D" , "TheGraph" , "TheArray" , "ResultArray" , "0.1" , "-0.5" , -
    "black,,gray" , "DemoWind"
- Dispose of the dictionary, array, graph and expression
XCOMPILE "Dispose Dictionary" , "Dictionary"
XCOMPILE "Dispose Expression" , "TheExpr"
XARRAY "DisposeArray" , "TheArray"
XARRAY "DisposeArray" , "ResultArray"
XGRAPH "Dispose Graph" , "TheGraph"
end mouseUp

```

7.5 XPIE

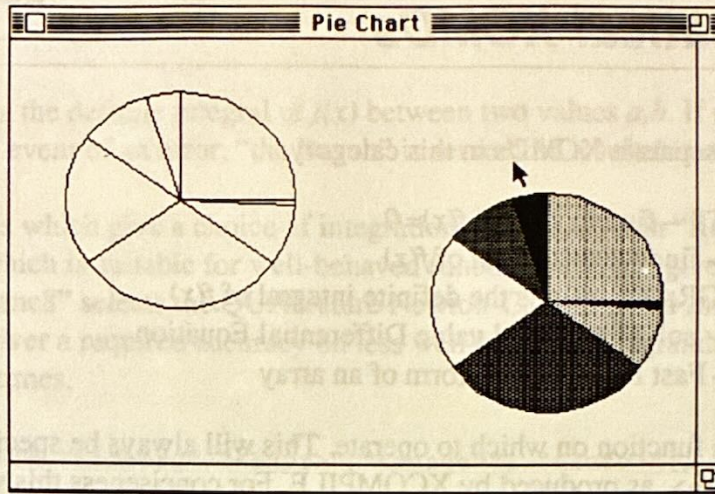
XPIE draws a pie chart according to the data passed in the Xblock <data>. The items in the data are in units in which the full circle = 1. Successive groups from the palette are used as the data is plotted. Plotting is done anti-clockwise for positive data, clockwise for negative data.

Syntax: XPIE "Outline|Filled", "<graph>", "<data>",
 "centre,radius,start angle", "<palette>"
 [,<Window>[, "<Perm|Temp>"]]

e.g., On mouseUp

- Store data in an array
- XARRAY "ListToArray" ,"xvals" ,"0.25,0.05,0.1,0.2,0.3,0.09,0.01"
- Set up window
- XCTRL "Init Window" ,"PieChart" ,"Pie Chart" ,"0,20,519,349"
- XGRAPH "Init Graph" ,"PieGraph" ,"505,330" ,"-4 ,-4,4,4"
- Draw the Pie Charts
- XGRAPH "SetGParam", "PieGraph","GraphRect", "0,0,250,250"
- XPIE "Lines" ,"PieGraph","xvals","0,0,4,0","black","PieChart"
- XGRAPH "SetGParam", "PieGraph","GraphRect", "270,80,520,330"
- XPIE "Filled" ,"PieGraph","xvals","0,0,4,0",-
 "ltgray,,,black,,,gray,,,white,,,dkgray","PieChart"
- Dispose of the array and graph
- XARRAY "DisposeArray" ,"xvals"
- XGRAPH "Dispose Graph" ,"PieGraph"

end mouseUp



8.1 XROOTS

XROOTS attempts to locate a zero of $f(x)$ between two values a and b which must be chosen so that $f(a)$ and $f(b)$ have opposite sign. The relative accuracy of the result is specified by an optional tolerance; maximum accuracy is used by default (about 10^{-10} for standard Apple extended type). If successful, the result is placed in "R". In the event of an error, the Result is set to hold a suitable message.

Syntax: XROOTS "Cerror", "Error", "<Error>", "<range>", "<tolerance>"
 The final argument is the number of times that the interval $[a, b]$ is halved until the function has changed sign. The default is 20. The range is specified by two values a and b in quotes. The tolerance is specified by a value in quotes. The error is specified by a value in quotes. The Cerror is specified by a value in quotes. The Error is specified by a value in quotes. The <Error> is specified by a value in quotes. The <range> is specified by two values in quotes. The <tolerance> is specified by a value in quotes.

For Quac8, the final arguments are the absolute and relative error tolerances, which are taken as 10^{-10} by default. <range> must be declared as for Romberg. The error is specified by a value in quotes. The Cerror is specified by a value in quotes. The Error is specified by a value in quotes. The <Error> is specified by a value in quotes. The <range> is specified by two values in quotes. The <tolerance> is specified by a value in quotes.

8.2 XMIN

XMIN will locate a minimum of $f(x)$ between two values a and b . The relative accuracy of the result is specified by an optional tolerance; maximum accuracy is used by default (about 10^{-10} for standard Apple extended type). If there is no internal minimum, then XMIN will return whichever of $f(a)$ or $f(b)$ is least. If successful, the result is placed in "R". In the event of an error, the Result is set to hold a suitable message.

Syntax: XMIN "Cerror", "Error", "<Error>", "<range>", "<tolerance>"
 The final argument is the number of times that the interval $[a, b]$ is halved until the function has changed sign. The default is 20. The range is specified by two values a and b in quotes. The tolerance is specified by a value in quotes. The error is specified by a value in quotes. The Cerror is specified by a value in quotes. The Error is specified by a value in quotes. The <Error> is specified by a value in quotes. The <range> is specified by two values in quotes. The <tolerance> is specified by a value in quotes.

For Quac8, the final arguments are the absolute and relative error tolerances, which are taken as 10^{-10} by default. <range> must be declared as for Romberg. The error is specified by a value in quotes. The Cerror is specified by a value in quotes. The Error is specified by a value in quotes. The <Error> is specified by a value in quotes. The <range> is specified by two values in quotes. The <tolerance> is specified by a value in quotes.

8. Mathematical XCMDs

There are a number of separate XCMDs in this category:

- 8.1 XROOTS – finds a root of $f(x)=0$
- 8.2 XMIN – finds a minimum of $f(x)$
- 8.3 XINTEGRATE – finds the definite integral of $f(x)$
- 8.4 XODE – solves an initial value Differential Equation
- 8.5 XFFT – Fast Fourier Transform of an array

Most of these require a function on which to operate. This will always be specified in the form of a `<compiled expression>` as produced by XCOMPILE. For conciseness this will be denoted by `<Expr>`.

8.1 XROOTS

XROOTS attempts to locate a zero of $f(x)$ between two values a, b which must be chosen so that $f(a)$ and $f(b)$ have opposite sign. The relative accuracy of the result is specified by an optional tolerance: maximum accuracy is used by default (about 10^{-22} for standard Apple extended type). If successful, the result is placed in "It". In the event of an error, "the Result" is set to hold a suitable message.

Syntax: XROOTS "GetRoot", "<Expr>", "<range>"[, tolerance]
e.g.,
on MouseUp
 XCOMPILE "Init Dictionary", "MyDict"
 XCOMPILE "Declare", "MyDict", "x"
 XCOMPILE "Compile", "quadratic", "MyDict", "x^2-3*x+2"
 XROOTS "GetRoot", "quadratic", "1.5,2.5"
 put It
end MouseUp

8.2 XMIN

XMIN will locate a minimum of $f(x)$ between two values a, b . The relative accuracy of the result is specified by an optional tolerance: by default the best possible accuracy is used, which for locating minima is about the $\sqrt{(\text{machine precision})} = 10^{-11}$. If there is no internal minimum, then XMIN will return whichever of $f(a)$ or $f(b)$ is least. If successful the result is placed in "It". In the event of an error, "the Result" is set to hold a suitable message.

Syntax: XMIN "GetMin", "<Expr>", "<range>"[, tolerance]
e.g.,
on MouseUp
 XCOMPILE "Init Dictionary", "MyDict"
 XCOMPILE "Declare", "MyDict", "x"
 XCOMPILE "Compile", "quadratic", "MyDict", "x^2-3*x+2"
 XMIN "GetMin", "quadratic", "1.1,2.5"
 put It
end MouseUp

8.3 XINTEGRATE

XINTEGRATE finds the definite integral of $f(x)$ between two values a, b . If successful the result is placed in "It". In the event of an error, "the Result" is set to hold a suitable message.

There are two actions which give a choice of integration method. Action "Romberg" selects the Romberg Method, which is suitable for well-behaved functions and can give an accurate result very quickly. Action "Quanc8" selects the QUAdrature Newton-Cotes 8-panel method, which is self-adaptive and can deliver a required accuracy on less well-behaved integrands at the expense of longer computation times.

Syntax: XINTEGRATE "Romberg", "<Expr>", "<range>", "<Info>"[, n]
XINTEGRATE "Quanc8", "<Expr>", "<range>", "<Info>"[, aber, reler]

e.g., on MouseUp
global RombInfo, Quanc8Info
XCOMPILE "Init Dictionary", "MyDict"
XCOMPILE "Declare", "MyDict", "t"
XCOMPILE "Compile", "Expr", "MyDict", "4/(1+t^2)"
XINTEGRATE "Romberg", "Expr", "1.5,2.5", "RombInfo", 4
put It into card field "Romberg result"
put RombInfo into card field "Romb info"
XINTEGRATE "Quanc8", "Expr", "1.5,2.5", "Quanc8Info"
put It into card field "Quanc8 result"
put Quanc8Info into card field "Quanc8 info"
end MouseUp

The final argument n for Romberg is the number of times that the interval will be halved, and is 5 by default. Thus Romberg by default performs $2^5 = 32$ function evaluations. <Info> is the name of a HyperCard global which must be declared in your script; it will be set to an error estimate of the result placed in "It".

For Quanc8, the final arguments are the absolute and relative error tolerances required, which are taken as 10^{-12} by default. <Info> must be declared as for Romberg, but it will be set to a list containing 4 values: flag, error estimate, number of function evaluations, xsnag. If flag is zero, then the error tolerances have been met. Otherwise flag gives the number of sub-intervals in which the required accuracy was not met, and xsnag gives the value of x (or whatever the independent variable is) where trouble was detected.

8.4 XODE

XODE provides a variety of methods for solving Ordinary Differential Equations in the form:

$$y_1' = f_1(x, y_1, y_2, \dots, y_n)$$

$$y_2' = f_2(x, y_1, y_2, \dots, y_n)$$

$$y_n' = f_n(x, y_1, y_2, \dots, y_n)$$

This is called an n^{th} order system. The values $(y_1, y_2, \dots, y_n)$ have to be specified at some initial value of the independent variable x , x_0 say. The variables $(x, y_1, y_2, \dots, y_n)$ can be named as you like and come in any order as long as the independent variable comes first. Each derivative expression $f_i(x, y_1, y_2, \dots, y_n)$, $i=1 \dots n$, must be compiled in advance with the same dictionary. The different actions give different methods.¹

Syntax: `XCODE "Euler", "<Expr list>", "<Init values>", "x0,x1", step,
 "<results list>", "<info>"
 [, "<plot list>", <graph>, <window>[, "<Perm|Temp>"]]`
`XCODE "RK4", "<Expr list>", "<Init values>", "x0,x1", step,
 "<results list>", "<info>"
 [, "<plot list>", <graph>, <window>[, "<Perm|Temp>"]]`
`XCODE "RKF", "<Expr list>", "<Init values>", "x0,x1", step, aber,
 reler, "<results list>", "<info>"
 [, "<plot list>", <graph>, <window>[, "<Perm|Temp>"]]`

e.g., on MouseUp
 XCOMPILE "Init Dictionary", "MyDict"
 XCOMPILE "Declare", "MyDict", "t,y,z"
 XCOMPILE "Compile", "f1", "MyDict", "z"
 XCOMPILE "Compile", "f2", "MyDict", "-4*pi^2*y - 0.2*z"
 XCODE "RK4", "f1,f2", "1,0", "0,5", 0.1, "tvals,yvals,zvals", "info"
 end MouseUp

e.g., on MouseUp
 - add declarations & compilations as above, and
 - definition and setting up of suitable window and graph
 XCODE "RK4", "f1,f2", "1,0", "0,5", 0.1, "tvals,yvals,zvals",
 "info", "yvals,zvals", "ODEgph", "ODE window"
 end MouseUp

For actions "Euler" and "RK4" (Runge-Kutta, 4th order), the arguments are the same. Both of these methods integrate over the interval x_0, x_1 with a fixed step size. Integration continues for the nearest whole number of steps that fit in the interval. The results list is a list of names of Xblocks which will store the solution for each variable from the initial to the final values. If the system is n^{th} order, then there must be n expressions in the expression list, and n initial values. There will be $n+1$ names in the results list because the first Xblock will hold the values of the independent variable. The "info" parameter is the name of a HyperCard global that will be set to a string containing two items: item 1 is the value of the independent variable reached, and item 2 is the number of steps taken.

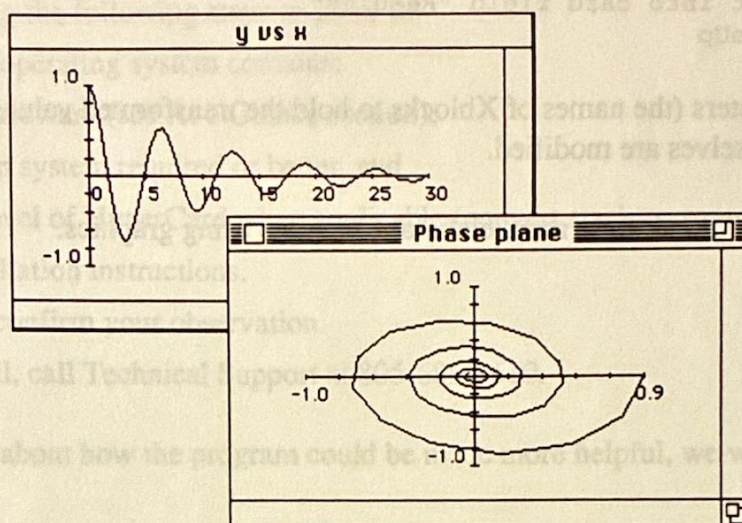
The plot list, graph, and window parameters are optional: if they are present, then plotting will be done dynamically using the window and graph, which must have been set up in advance. The plot list contains a selection of two or three names from the results list, and determines which variables correspond to the x , y , and z axes respectively. In the second example shown above, y will be plotted on the horizontal axis, z on the vertical axis.

For action "RKF", the adaptive Runge-Kutta-Fehlberg method is used. The method adjusts the step size as the calculation proceeds so that each step has local error below the absolute and

¹ See standard texts, such as *A Simple Introduction to Numerical Analysis* by Harding and Quinney (Adam Hilger).

relative error tolerances specified. The process halts if the specified accuracy cannot be achieved with a minimum step size of 1/512 of the specified size.

For further examples involving graphics see the Demonstration stack.



8.5 XFFT

XFFT finds the Discrete Fourier Transform of the values in an array. The array must be one dimensional and the number of values N which it contains must be a power of 2. The maximum is limited by the requirement that data segments cannot exceed 32K bytes, which means that $N=2^{11}$ values is the maximum. For full mathematical details, users are referred to standard texts or the *Scientific Programmer's Toolkit*, but briefly, if x_r with $r=0...(N-1)$ are the values to be transformed, then XFFT calculates

$$y_s = \sum_{r=0}^{(N-1)} x_r e^{(\pm 2\pi i r s / N)}.$$

The two actions correspond to the choice of sign in the exponential.

Syntax: XFFT "FFT +", "<x Real>", "<x Imag>"[, "<y Real>", "<y Imag>"]
 XFFT "FFT -", "<x Real>", "<x Imag>"[, "<y Real>", "<y Imag>"]

e.g., on MouseUp

```
XCOMPILE "Init Dictionary", "MyDict"
XCOMPILE "Declare", "MyDict", "x"
XCOMPILE "Compile", "decay", "MyDict", "exp(-abs(x))"
- note that the DFT assumes a "one-sided" range
XARRAY "bySteps", "mesh", 0,8,32
XEVAL "EvalExpr", "decay", "mesh", "xReal"
XARRAY "bySteps", "zero", 0,0,32
- to improve approximation to the continuous FT, replace first
- value with mean of values at sample end-points
XARRAY "GetOne", "XReal", 0
```


parameters (the names of Xblocks to hold the transformed values) themselves are modified.

Demonstration Stack for a more elaborate example using graphics.

Solving Problems with the Software

At Intellimation we care enough to make a difference, so if you have a problem with a program, help us help you by taking the following steps to solve it:

1. Make sure your operating system contains:
 - A. the proper hardware (see At a Glance section),
 - B. the minimum system required or better, and
 - C. the proper level of HyperCard when applicable (convert stacks upward when necessary).
2. Follow the installation instructions.
3. Ask a friend to confirm your observation.
4. If these steps fail, call Technical Support at 805-685-9610.

If you have an idea about how the program could be made more helpful, we would appreciate written suggestions.

The best way to help us solve a problem is to send a screen shot/print (command-shift-3) of the place where the failure appeared.

- What happened?
- What message appeared on the screen?
- Can you recreate the problem at will? (If you can, it will be easier for us to duplicate and solve the problem.)
- What kind of Macintosh were you using?
- Which version of the system and HyperCard (when applicable) were you using?
- How can we contact you (phone and address)?

Please send the above information to:

Intellimation Technical Support
P.O. Box 1530
130 Cremona Drive
Santa Barbara, CA 93116-1530
805-685-9610 FAX 805-968-8899

Thank you for your cooperation!



Library for the Macintosh
P.O. Box 1530, Santa Barbara, CA 93116-1530
Customers call 1-800-3-INTELL (1-800-346-8355)
Developers call 805-685-2100