Guidelines for *Mathematica* Documentation, Packages, and Notebooks

April 1993

General Information

This pamphlet gives instructions for authors who wish to create *Mathematica* documentation, packages, and Notebooks in accordance with the standards and conventions followed at Wolfram Research, Inc. Packages are programs written in *Mathematica*, to be used on their own or integrated with the user's code. Notebooks are interactive documents integrating *Mathematica* code, formatted text, and graphics. Following the conventions described here will make it easier for users familiar with the "look and feel" of *Mathematica* to use your materials. However, it is recognized that certain applications may require a format that deviates significantly from these guidelines.

This document and other development tools may be found on *MathSource*, an electronic archive of *Mathematica*-related materials accessible by email (`mathsource@wri.com`), anonymous ftp or direct dialup. Outside developers interested in the free distribution of their *Mathematica* code and documentation are invited to upload their materials to *MathSource*. Developers interested in pursuing joint projects with Wolfram Research, Inc. may submit proposals to Leszek Sczaniecki, or email `devproposal@wri.com`. Developers seeking technical, marketing, and distribution support should submit requests to Ben Friedman, or email `devsupport@wri.com`.

Guidelines for *Mathematica* Documentation

Useful documentation for *Mathematica* packages may take a number of forms; in particular, a *Mathematica* Notebook is a convenient way to document a package electronically. What follows is a description of how the WRI Technical Report *Guide to Standard Mathematica Packages* documents the packages distributed with *Mathematica*. This sort of hardcopy documentation may not be the best way to document a complex package; the description of each package in the *Guide* is necessarily brief, due to the number of packages in the standard library. However, documenting a package in this manner may be helpful in fine-tuning a package's design. The result can serve as final documentation, or be expanded into a complete indexed textbook, or put in the form of an interactive *Mathematica* Notebook.

The system used to generate both the *Guide to Standard Mathematica Packages* and Stephen Wolfram's book *Mathematica: A System for Doing Mathematics by Computer*, is described in the WRI Technical Report *The WRILaTeX Document System*. This system includes the TeXprep preprocessor, which allows you to keep only *Mathematica* inputs in a documentation source file. TeXprep sends the input to *Mathematica* via *MathLink* and returns coded *Mathematica* output along with any PostScript graphics you have produced. Technical details concerning the use of the WRILaTeX Document System and the macros for creating the different boxes mentioned below are given in the report. Of course, the documentation format followed by the *Guide to Standard Mathematica Packages* may be imitated without using the WRILaTeX Document System.

The general approach to documentation in the *Guide* is to start with the simplest ways to use the package and then go on to increasingly complex examples. The following outline gives the preferred format.

- Title: for example, `NumberTheory‘ContinuedFractions‘`

- Introduction: a background paragraph discussing such things as the field of math or science in which the package would be used, history, definitions, etc. If the package defines mathematical functions, this paragraph could include the standard mathematical notation for the functions. The introduction should point out how the package differs from or enhances the kernel. It should also place this package in context by pointing out similar things that can be done using the kernel or other packages. If nothing interesting can be said about the package without first presenting the syntax of the functions, then this paragraph can be skipped or placed after the first box.

- First Box: the simplest ways of using the functions. If the same function takes different types of arguments, the arguments are given in the order of increasing complexity. Some verbs used in the description of what a function does are: give, generate, display, represent, and compute. The function descriptions do not end in a period. The title of the box starts with a capital letter and ends in a period.

- Explanation: an explanation of how to use the functions described in the first box. A good format is to explain the general case and then give an example. This paragraph could also be an explanation of the *Mathematica* examples to follow, in cases where the comments to the left of the examples are insufficient. Short sentences and active verbs are best.

- *Mathematica* Examples: examples presented in a two-column format with a comment to the left of each *Mathematica* input. Start by loading the package, and follow the simplest examples with the more complicated. In some cases, the sequence of examples may tell a story or illustrate a point. The examples should not require an extensive knowledge of *Mathematica* and, if possible, should be chosen to have a reasonably small output. The dialog of comments should contain complete sentences with standard punctuation. Since package documentation is not meant to be read cover-to-cover, it is not necessary to avoid repetitive comments. Repetition may help a user see the similarity between related functions.

- Other Boxes: The "box-explanation-examples" cycle may be repeated, with each box containing a subset of related functions defined in the package. If there is an option to explain, a box may be created listing the relevant function(s) with the option as an argument. If there are several options to explain, it may be better to create a box giving a table of option names, default option values, and option descriptions. If the package exports a large number of symbols with identical usage (*e.g.*, colors or coordinate systems), it is appropriate to simply list these symbols in a box and describe their use in the text.

If the function you are documenting makes use of built-in options that are described in The *Mathematica* Book, you might repeat the option documentation so that users do not need to refer to The *Mathematica* Book. Similarly, if your package serves to enhance the functionality of a built-in function, you should reproduce the function's usage box as it appears in The *Mathematica* Book.

Typically, the documentation for a package will end with the last sequence of *Mathematica* examples. In some cases, the documentation will end with a note or caveat regarding function usage.

Guidelines for *Mathematica* Packages

The single most important consideration in designing a *Mathematica* package for distribution is that it must be readily usable by others. If your package follows the design elements adhered to in *Mathematica* itself, users will find it easier to learn. And since the reuse of code in the distributed package library (with acknowledgment) is encouraged, attention to precedent can both enhance the usability of your package and speed package development.

The next few paragraphs describe some of the conventions that are followed by the built-in and standard library package functions. More on the subject, including pointers on programming style, may be found in the book *Programming in Mathematica*, by Roman Maeder (Addison-Wesley 1989). Evaluating the design of a package is best accomplished by attempting to document it. That is, a description of function usage, including options and examples, should be completed before the code is written. If the description is lengthy and overly complex, there is probably something wrong with the design. On the other hand, such a description may reveal that a proposed function is only a minor perturbation of an existing function, and does not warrant a new package.

Naming

Follow the general naming convention used in *Mathematica*: names should be full English words or sequences of words, with the first letter of each word capitalized. When in doubt, use the longer or more explicit term; avoid abbreviations and names that are not descriptive. If there exists in *Mathematica* a function with similar functionality, then the function you are defining should be named similarly. Conversely, functions with dissimilar functionality should have dissimilar names.

Function names should be unique. With a system of about a thousand built-in symbols and over a thousand additional symbols defined in the standard package library, this is not always possible. However, users count on the built-in symbols (and, to a lesser extent, library symbols) behaving in a particular way. In addition to confusing the user, the unprotecting of basic arithmetic objects (*e.g.*, `Plus`, `Times`, `Power`, `Integer`) almost always causes the unnecessary re-evaluation of expressions.

Programming style

Primarily use transformation rules and functional programming; procedural programming is less efficient, longer and more difficult to understand in *Mathematica*. Try to use list-manipulating functions instead of loop functions that require indexing. Note that lists are implemented internally in *Mathematica* as arrays; operations that change the size of lists thus tend to be expensive in time and memory (*e.g.*, `Append` and `Prepend`).

Minimize or eliminate argument checking done within a function by writing specific patterns that only desired arguments will match. Attach rules to those symbols that will make search most efficient. For example, if you have a rule for the pattern `f[Cos[x_]] + f[Sin[x_]]`, then attach that rule to `f` rather than `Plus`.

Modularity

Avoid using global variables; use options instead. A function that makes use of options should be written so that the option can be passed as an argument, *i.e.*, it should not be necessary to execute `SetOptions` in order to pass an option to the function. Options can be parsed using the standard method employed in numerous distributed packages. A useful utility for dealing with options that are passed from one function to subsidiary functions can be found in the package ``Utilities`FilterOptions` ``. A user should be able to discover the default values of a function's options by applying `Options` to the function's name.

Define all symbols—functions, options, variables, etc.—in appropriate *Mathematica* contexts. For details, see Section 2.6.9 of The *Mathematica* Book. Think about how your functions will fit with other functions a person will be using. Will the input for your functions be easy to obtain from other *Mathematica* functions? Will their output be readily amenable to further processing? Try to make sure that your package can be used as part of a "pipeline," rather than only as a stand-alone application. If the output of a function is sufficiently complex, it may make sense to create a special object that hides most of that information (*e.g.*, `InterpolatingFunction` and `CompiledFunction`).

The state of a user's *Mathematica* session should not be changed unnecessarily by loading a package or evaluating a function. For example, it is frequently desirable to turn off messages within a function, so that the user is not bothered by messages associated with subsidiary functions. Care should be taken to return the state of these messages to the state they were in when the function was invoked. Similarly, the attributes of built-in symbols should not be affected by loading a package.

Documentation

All symbols visible outside your package must be documented with usage messages. A usage message should be written in complete sentences, and should begin with the name of the function the message is documenting. The arguments of a function should not be capitalized, and they should not be very long. For example, the usage message for the function `Fit` includes the fragment `Fit[data, funs, vars]`, not `Fit[Data, Functions, Variables]`.

Error and warning messages should be provided in anticipation of mistakes, and should be as explicit as possible. These messages must be generated using the `Message` function, so that they can be caught by the function `Check` and can make use of the global symbol `$NewMessage`. The WRI Technical Report *Mathematica Warning Messages* gives a listing of the message tags and message texts associated with built-in symbols. New error messages should be defined only if existing messages are inappropriate. The function `Print` should be used only for lengthy diagnostic information, and the default behavior of a function should be that diagnostic information is not printed out.

The use of comments for internal documentation is encouraged (see also the section on Notebooks). For the purpose of internal documentation, your package must have certain special, standardized comments in the format described in the example package below. These comments can be read in a *Mathematica* session using the functions defined in the V2.2 package ``'Utilities`Package` ``.

Testing

Your package should be accompanied by one or more files containing tests. Ideally, the tests will exercise at least once every definition and branch in the package's functions, including those that handle error conditions; they will also cover a range of complexity, from the straightforward to the sophisticated. If it is likely that users will use your package in conjunction with others in the standard library, the tests should verify that this is possible without the shadowing of important symbols.

The package `Testing.m` (available on *MathSource*) provides functions for comparing *Mathematica* input and output, as well as functions for creating tests and automating test runs. A set of test files documenting correct package behavior simplifies the task of verifying whether a package created under one version of *Mathematica* works consistently under another, or whether the package works consistently from platform to platform.

Guidelines for *Mathematica* Notebooks

Notebooks can be used to document and illustrate *Mathematica* packages. They contain all the information that should be in a package, together with explanatory text and examples. All the concerns discussed in the previous section apply to the code in Notebooks. The guidelines described here primarily pertain to Notebooks that, like packages, introduce new functionality into *Mathematica*. Notebooks intended as courseware focus on explicating a topic, so sections may cover particular subtopics, rather than serve as a means of separating *Mathematica* examples from *Mathematica* function definitions.

Formatting

In order to promote consistency, Notebooks should contain certain information in a standardized format, illustrated in the example at the end of this section. Appropriate cell styles should be used to distinguish between cells; for example, section header cells should be in the "Section" cell style. Sections and subsections should be placed in cell groups as shown in the example.

After a header giving the title, authors and summary, a standard Notebook has three sections: Reference, Discussion and Implementation. The Reference section follows closely the format for packages, up to but not including the `:Discussion:` field.

The Discussion section can be substantially longer for Notebooks than it is for packages. Typically, it should include, grouped under a subsection cell, a selection of examples designed to illustrate the different ways in which the functions defined in the Notebook can be used. Examples can be input lines only, or evaluated input/output pairs. To save space, examples with a lot of output—including all graphics output—should be left unevaluated. The input cells in the examples should be marked as active cells, but not as initialization cells. (This is the default for newly created cells.)

The Implementation section contains the definitions and usage messages for the Notebook. It should also include documentation about the implementation of the functions; documentation about their use belongs in the Discussion section. The input cells in the Implementation section should be marked as active cells and as initialization cells. When a Notebook is loaded into *Mathematica* using `Needs` or `<<`, only such cells are evaluated; other cells are treated as comments. Care should be taken to put context-manipulation commands on separate lines and in separate cells.

Testing

Each Notebook should have one or more companion Notebooks containing tests. To create such a Notebook, copy the title cell and essential cells in the Reference section from the main Notebook to an empty Notebook. Add the words "Tests for" to the title cell, and add a section named "Tests" to hold the tests. The coverage of the tests should be summarized below the title. A cell describing test history should be added to the Reference section.

Each test should be in the form of one of the functions defined within the package `Testing.m` and include an evaluated input/output cell pair, in the form produced automatically when an input cell is evaluated. It is important to have the "Generating Unformatted Text" option in the "Action Preferences" panel set correctly. It should be set to either "all results except those containing Graphics or Short" (the default), or to "all results" if you intend to include graphics output in your tests.

Portability issues

Because some versions of *Mathematica* don't support changing the font, face, size or color of text within a cell, the intelligibility of your Notebook should not depend on such changes. On the other hand, all versions of *Mathematica* support changing the font, face, size and color of cells as a unit.

Avoid using fonts with non-standard symbols, which may not be available on other systems. The PostScript Symbol font can be used freely, as it is available on all systems.

The Macintosh PICT graphics format works only on Macintosh computers. Because of this, Notebooks containing pictures pasted in from other Macintosh applications are not portable to other systems. You should convert such pictures into a portable form using the "Convert to PostScript" menu command.

Example Package

Title; standard capitalization rules for titles apply.	(* :Title: Continued Fractions *)
Author(s); if more than one, separate by commas.	(* :Authors: Stephen Wolfram, Ilan Vardi *)
A short description of what the package does. This field is made use of by the function browser in the *Mathematica* V2.2 front end, so it should be descriptive but succinct.	(* :Summary: This package generates continued fraction approximations to real numbers, represented as ContinuedFractionForm objects. It extends the Normal function to convert such objects to rational numbers. *)
The externally visible context defined in this package. Also specifies (in a system-independent way) the name of the file into which this package should be put.	(* :Context: NumberTheory‘ContinuedFractions‘ *)
Version number of the package (cf. :History:, below).	(* :Package Version: 1.4 *)
The minimum version of *Mathematica* required to run the package.	(* :Mathematica Version: 2.0 *)
Copyright notice.	(* :Copyright: Copyright 1987-1993, Wolfram Research, Inc. *)
Revision history.	(* :History: Modified by John Novak (Wolfram Research), May 1992, to allow nonrational nonfloating-point numbers. Modified by John Novak (Wolfram Research), March 1992, to allow rational numbers. Revised by Ilan Vardi (Wolfram Research), September 1989. Version 1.1 by Stephen Wolfram (Wolfram Research), May 1987. *)
Keyword(s) for indexing; if more than one, separate by commas. Capitalize proper names only.	(* :Keywords: number theory, continued fraction, rational approximation, partial quotient *)
References for the algorithms or data in the package. `:Source:` fields may be repeated or absent.	(* :Source: Hardy and Wright: "An Introduction to the Theory of Numbers", Cambridge University Press. *)
Any aspects of this package that might interfere with ordinary *Mathematica* operations. `:Warning:` fields may be repeated or absent.	(* :Warning: Loading this package extends the definition of Normal. *)
One or more `:Requirement:` fields may be present here, stating packages on which this one depends (unless they are part of the standard *Mathematica* distribution), and machine or operating system dependencies.	
Known limitations in the package. This field may be repeated or absent.	(* :Limitation: This package does not define mathematical operations on ContinuedFractionForm objects. *)

The optional `:Discussion:` field may elaborate on the summary, contain detailed usage instructions, or explanations about the algorithms. This may be a briefer form of hard-copy documentation.

```
(* :Discussion:
The continued fraction of a real number  x  is
ContinuedFractionForm[{a0,a1,a2,...}] if  x  equals a0+1/(a1+1/(a2+...)).
All numbers are forced to bignums, minimizing roundoff error problems.
If a nonrational nonfloating-point number is given, ContinuedFraction
estimates the precision needed to get a continued fraction of the correct
order.
*)
```

The *Mathematica* code itself.

```
BeginPackage["NumberTheory`ContinuedFractions`"]
```

Include usage messages for each externally visible function.

```
ContinuedFraction::usage =
"ContinuedFraction[x, n] generates the continued fraction representation for
the number x to order n.  Note that the order returned may be less than n if
the continued fraction terminates before n steps."
ContinuedFractionForm::usage =
"ContinuedFractionForm[{a0,a1,...}] is a continued fraction;  it is
converted to an ordinary rational using Normal."
```

Use a private context for the actual definitions. The convention "`‘Private‘`" allows users to readily extract the hidden definitions.

```
Begin["`Private`"]
ContinuedFraction::incomp =
"Warning: ContinuedFraction either terminated or the argument was
unable to be evaluated to sufficient precision in `` attempts."
ContinuedFraction[x_?MachineNumberQ, n_Integer?Positive] :=
   ContinuedFraction[SetPrecision[x, $MachinePrecision], n]
ContinuedFraction[x:(_Real | _Rational), n_Integer?Positive] :=
   ContinuedFractionForm[Floor[Prepend[cf[x, n - 1], x]]]
```

One or more `:Note:` fields may be present here, explaining any non-obvious implementation details.

```
(* :Note:
A comfortable overestimate for the required precision is to start with one
decimal digit for every two continued fraction coefficients required.
*)
ContinuedFraction[x_?(Not[NumberQ[#]] && NumberQ[N[#]] &),
   n_Integer?Positive] :=
      Module[{nx, prec = n/2 + 10, out = {}, i = 0},
         While[Length[out] < n && i++ < n,
            nx = N[x, prec];
            out = cf[nx, n-1];
            prec += (n - Length[out])/2 + 10
         ];
         If[Length[out] < n-1,
            Message[ContinuedFraction::incomp, n]
         ];
         ContinuedFractionForm[Floor[Prepend[out, nx]]]
      ]
cf[num_, 0] := {}
cf[num_, ord_] := {} /; ((num - Floor[num]) == 0)
cf[num_, ord_] :=
      Module[{tmp = 1/(num - Floor[num])},
        Join[{tmp}, cf[tmp, ord - 1]]
      ]
ContinuedFractionForm/: Normal[ContinuedFractionForm[a_List]] :=
   Fold[ 1 / #1 + #2 &, Last[a], Rest[Reverse[a]]]
End[]
EndPackage[]
```

The `:Examples:` field contains sample input showing instructive examples of the use of the package. Pathological examples should be restricted to test files.	`(* :Examples:`
	`ContinuedFraction[Pi, 8]`
	`ContinuedFraction[1234/4321, 10]`
	`Table[Normal[ContinuedFraction[Pi, n]], {n, 1, 7}]`
	`*)`

Example Test File

Test files should be numbered.	`(* :Test Title: Continued Fractions Test 1 *)`

A history of modifications made to the tests in response to package code changes.	`(* :Test History:`
	`Added tests for non-numerical numbers, John Novak (Wolfram Research), May 1992.`
	`Initial version by John Novak (Wolfram Research), March 1992.`
	`*)`

Use `:Test Coverage:` fields to explain what aspects of the package the tests cover.	`(* :Test Coverage: Continued fractions of rational and non-numerical numbers. *)`

Repeat essential package information here.	`(* :Title: Continued Fractions *)`
	`(* :Authors: Stephen Wolfram, Ilan Vardi *)`
	``(* :Context: NumberTheory`ContinuedFractions` *)``
	`(* :Package Version: 2.0 *)`

A set of tests using the functions defined in `Testing.m` (available on *MathSource*). You should start by loading your package.	`Test[`
	``<<NumberTheory`ContinuedFractions` ``
	`,`
	`Null`
	`]`

Examples from the `:Examples:` field of the package should be tested.	`Test[`
	`ContinuedFraction[Pi, 8]`
	`,`
	`ContinuedFractionForm[{3, 7, 15, 1, 292, 1, 1, 1}]`
	`]`
	`Test[`
	`ContinuedFraction[1234/4321, 10]`
	`,`
	`ContinuedFractionForm[{0, 3, 1, 1, 153, 1, 3}]`
	`]`
	`Test[`
	`Table[Normal[ContinuedFraction[Pi, n]], {n, 1, 7}]`
	`,`
	`{3, 22/7, 333/106, 355/113, 103993/33102, 104348/33215, 208341/66317}`
	`]`
	`Test[`
	`ContinuedFraction[(10^(-20) + 3^(1/2))/3^(1/2), 1]`
	`,`
	`ContinuedFractionForm[{1}]`
	`]`

Here an example is tested for both the correct answer and the correct message. Other functions in the package `Testing.m` permit numerical tests, the interactive generation of tests from a *Mathematica* session, and the launch and completion of test runs.	`Test[`
	`ContinuedFraction[(10^(-20) + 3^(1/2))/3^(1/2), 3]`
	`,`
	`ContinuedFractionForm[{1, 17320508075688772608}]`
	`,`
	`{MessageName[ContinuedFraction, "incomp"]}`
	`]`

Example Notebook

A cell in the Title style, containing the title of the Notebook. A cell in the Subsubtitle style, containing the authors of the Notebook. A short summary of the contents of the Notebook, in the Text style. *Mathematica* symbols should be in the Input style font. A cell in the Section style, beginning the Reference section. A sequence of cells in the Subsubsection and Text styles, one pair for each of the special comments listed in the package description, using the same names as subsection headers. A Notebook parser will be able to extract reference information from Notebooks following this convention. A cell in the Section style, beginning the Discussion section. A general discussion of the contents of the Notebook. A cell in the Subsection style, beginning the Examples subsection. A mixture of Text, Input, and Output cells containing examples. More complicated Notebooks may contain subsubsections here. It is usually a good idea to include all the examples from this section in the companion Notebooks containing tests.	

A cell in the Section style, beginning the Implementation section.

A cell in the Subsection style, beginning the Preparation subsection.

The context name for the Notebook.

Usage messages for all the externally visible symbols.

A cell in the Subsection style, beginning the Definitions subsection.

A private context for the definitions.

A mixture of Text and Input cells containing the actual definitions and documentation. In this example, the code is broken up into subsubsections.

The end of the Implementation section.	