Da Coda Al Fine

Pushing Octave's Limits

Christoph L. Spiel
Hammersmith Consulting

Da Coda Al FinePushing Octave's Limits

by Christoph L. Spiel

Octave is the Matlab®-clone which comes closest to the original. Though it tries to be fully compatible, it features its own extensions and improvements over the functionality of the commercial Matlab® product.

Octave is shipped with a 340+ page reference manual, which also tries to be a bit of a user guide. Despite its size the documentation is incomplete, and where it stops, Coda starts: Da Coda Al Fine approaches Octave purely from the user's point of view, explaining e.g. `oct`-file construction in a tutorial like form.

Revision History

Revision 0.1 2001-2-21 Revised by: cls
Initial revision
Revision 0.2 2004-02-06 Revised by: sjw
Corrections, unit testing improved

Table of Contents

1. OCT Files **1**

1.1. Tutorial 1

1.1.1. Problem Definition 1

1.1.2. High-Level Implementation 1

1.1.3. Our First Dynamic Extension 4

1.1.4. Spicing up `matpow` 9

1.2. Advanced Extension Programming 16

1.2.1. Defining Constants And Variables 16

1.2.2. Documenting Constants And Variables 20

1.3. Abridged Quick-Reference 20

1.3.1. Types 21

1.3.2. List Operations 22

1.3.3. Access to Value 23

1.3.4. Type Classification 24

2. Building Standalone Applications **25**

A. Complete examples **26**

A.1. Unit-test 26

A.1.1. `unit_init.m` 26

A.1.2. `unit_test.m` 27

A.1.3. `unit_test_equals.m` 28

A.1.4. `unit_test_err.m` 28

A.1.5. `errcheck.m` 28

A.1.6. `erreval.m` 29

A.1.7. `unit_results.m` 29

B. GNU Free Documentation License **30**

0. PREAMBLE 30

1. APPLICABILITY AND DEFINITIONS 30

2. VERBATIM COPYING 31

3. COPYING IN QUANTITY 32

4. MODIFICATIONS 32

5. COMBINING DOCUMENTS 34

6. COLLECTIONS OF DOCUMENTS 34

7. AGGREGATION WITH INDEPENDENT WORKS 34

8. TRANSLATION 35

9. TERMINATION 35

10. FUTURE REVISIONS OF THIS LICENSE 35

C. GNU General Public License **37**

0. APPLICABILITY[1] 38

1. VERBATIM COPYING 38

2. MODIFICATIONS 38

3. DISTRIBUTION 39

4. TERMINATION 40

5. ACCEPTANCE 40

6. REDISTRIBUTION 40

7. CONSEQUENCES 41

8. LIMITATIONS 41
9. FUTURE REVISIONS OF THIS LICENSE 41
10. AGGREGATION WITH INDEPENDENT WORKS 42
11. NO WARRANTY 42
12. LIABILITY 42
Bibliography 44
Index 45

List of Tables

1-1. Types.....21

Chapter 1. OCT Files

Octahedron, octopus, oculist, `oct`-file? – What?

An `oct`-file is a dynamical extension of the Octave interpreter, in other words a *shared library* or *shared object*. The source files, that make up an `oct`-file, are written in C++, and therefore, conventionally, carry the extension `cc`.

Why would you ever want to write in C++ again, having the wonderful Octave environment at your service? Well, sometimes the performance of native Octave-code is not enough to attack a problem. Octave-code is interpreted, thus it is inherently slow when executed (especially if the problem cannot be vectorised). In such a case moving from `m`-code to compiled C++-code speeds up the execution by a factor of ten or more. The second group of problems, where reverting to low-level code is necessary, is when interfacing with foreign libraries (think of LAPACK) or routines written in languages other than C++, most notably Fortran-77.

Having convinced ourselves that we have to bite the bullet, we start with an easy tutorial (Section 1.1). This will teach any reader who is sufficiently used to C++ how to write her first dynamically linked extension for Octave. Having guided the reader with her first steps, Section 1.2 covers more advanced topics of writing extensions. For later reference, we provide a quick reference of the most helpful methods of the Octave interface classes in Section 1.3.

1.1. Tutorial

The tutorial unfurls slowly; we do not want to scare anybody by the complexity of the task ahead. At any time, if you feel like it, look into the directories OCT`/src` or OCT`/src/DLD-FUNCTIONS`, where OCT is the root directory of your Octave source-tree and consult the header-files and already existent dynamic extensions.

1.1.1. Problem Definition

Instead of giving an abstract treatise, we want to explain the whole business of dynamical extensions with the help of a simple, yet non-trivial problem. This example shall be analyzed in detail from beginning to end, and we shall elucidate some modern software construction principles on the way.

We shall implement a matrix power function.[1] Given a square matrix A of integers, reals, or complex values, and a non-negative integral exponent n, the function `matpow`(A, n) is to calculate A^n.

We have to watch out for boundary cases: an empty matrix A or a zero exponent n.

1.1.2. High-Level Implementation

We postpone writing the C++-implementation until we are completely satisfied with our implementation in Octave. Having Octave, a rapid prototyping environment at hand, it would be stupid to throw away its possibilities.

Hobby Astronomer's Mirror Making Rule

It is faster to grind a 3 inch mirror and then a 5 inch mirror, that to grind a 5 inch mirror.

Now that the problem is exactly defined, we can start thinking about an implementation. Here is our naive first shot:

Example 1-1. Naive implementation of `matpow`

```
function b = matpow0(a, n)
    ## Return b = a^n for square matrix a, and non-negative, integral n.
    ## Note: matpow0() is an extremely naive implementation.

    b = diag(ones(size(a, 1), 1));
    for i = 1:n
        b = b * a;
    endfor
endfunction
```

Easy does the job! `matpow0` looks like it does what we want, but how can we be sure? We write a test-suite[2]! A test-suite is needed when we switch to C++. We have a piece of running code, so let us write some tests. The unit testing functions are defined in Appendix A, Section A.1.

```
### name:        test_matpow.m  --  test matrix power functions
### original:    Ch. L. Spiel

###
### The following tests should succeed.
###

unit_init(0, {"a"}); ## Quiet testing, use global variable a

a = [ 2.0, -3.0;
     -1.0,  1.0];

unit_test_equals("a^0",  matpow0(a, 0), diag([1, 1]));
unit_test_equals("a^1",  matpow0(a, 1), a);
unit_test_equals("a^2",  matpow0(a, 2), a^2);
unit_test_equals("a^3",  matpow0(a, 3), a^3);
unit_test_equals("a^4",  matpow0(a, 4), a^4);
unit_test_equals("a^22", matpow0(a, 22), a^22);
unit_test_equals("a^23", matpow0(a, 23), a^23);
```

```
###
### print results
###

unit_results();
```

Running the tests on `matpow0` gives us confidence

```
octave:2> test_matpow
................
# of testcases attempted   7
# of expected passes       7
# of expected failures     0
# of unexpected passes     0
# of unexpected failures   0
# of unresolved testcases  0
```

but we also get more ambitious!

Our algorithm is really naive, and matrix multiplications are computationally expensive. Let us cut down on the number of multiplications. What is the minimum number of multiplications needed to compute A^n? Starting with A, we can only square it, getting A^2. Again squaring is the fastest way to get to the fourth power. In general squaring our highest power lets us avance with least multiplications. This is the heart of our new algorithm.

Improved matrix power algorithm. If the exponent n is a w-bit number, we can apply the binary expansion $n = \text{sum}(b_i * 2^i, i = 0..w\text{-}1)$, where the b_i are either 0 or 1. In other words, we square A for every bit in the binary expansion of n, multiplying this value with the final result if $b_i = 1$. Special care has to be taken for the cases where $n = 0$ or 1. See also [Golub:1996], section 11.2.5, page 569.

Example 1-2. Implementation of improved matrix power algorithm

```
function b = matpow1(a, n)
    ## Return b = a^n for square matrix a, and non-negative, integral n.

    -- handle special cases: n = 0, and n = 1
    -- first, to allow for an early return
    if (n == 0)
        b = diag(ones(size(a, 1), 1));
        return;
    endif

    if (n == 1)
        b = a;
        return;
    endif

    -- general case: n >= 2
    p = a;                                  -- p holds the current square
    b = a;
```

```
    np = n - 1;
    while (np >= 1)
        if (is_even(np))                    -- is_even is defined below
            -- zero in the binary expansion
            np = np / 2;
        else
            -- one in the binary expansion
            np = (np - 1) / 2;
            b *= p;
        endif
        p *= p;
    endwhile
endfunction


function e = is_even(n)
    ## Return true (1), if n is even, false (0) otherwise.

    e = (rem(n, 2) == 0)
endfunction
```

The new algorithm reduces the number of matrix multiplications from O(n), to O($\log_2(n)$), which is a remarkable improvement for large matrices as matrix multiplication itself is an O(m^3) process (m is the number of rows and columns of the matrix).

Running the test-suite again ensures that we did not code any nonsense.

1.1.3. Our First Dynamic Extension

Ready for the jump to light speed? Wait – we have to feed the navigation computer first!

1.1.3.1. Feature Compatibility

In principle, the functions defined in `oct`-files must have the same capabilities as functions in `m`-files. In particular, the input and output arguments lists must be able to carry different numbers of arguments, which are moreover of different type, just as `m`-file functions can. This means that there must be a way of mapping a function from Octave like

```
function [array, real-scalar, integer] =
    func(complex-scalar, array, list, integer)
    ## func frobs the snafu, returning all gromniz coefficients.

    -- actual function code goes here

endfunction
```

to C++. To this end, Octave's core interface supplies the macro (in OCT/src/`defun-int.h`)

```
octave_value_list
DEFUN_DLD(function_name,
          const octave_value_list& input_argument_list,
          [ int number_of_output_arguments ],            -- optional argument
          const char* documentation_string)
```

We have decorated the macro arguments with their types. Note that the first argument is the name of the function to be defined.

Of course the macro has to be defined somewhere. The easiest way to pull in all necessary definitions is to include OCT/src/`oct.h`. Now, our skeletal source file of the dynamic extension has the following shape:

```
#include <oct.h>

DEFUN_DLD(func, args, nargout,
          "func frobs the snafu, returning the gromniz coefficients.")
{
    -- actual function code goes here

    return possibly_empty_list_of_values;
}
```

Often functions do not depend on the actual number of output parameters. Then, the third argument to `DEFUN_DLD` can be omitted.

> **Naming convention:** `DEFUN_DLD` gives the user the freedom to choose any name for *`input_argument_list`* and *`number_of_output_arguments`*, but conventionally `args` and `nargout` are used (thus reminding of the parameter names in `main`, which are `argc` and `argv`).

1.1.3.2. Essential types and methods

Before we start to write a low-level implementation of `matpow`, we look at the most essential types and methods used to handle data that flows through the Octave-interpreter interface.

As has been said above, the arguments passed to dynamically loaded functions are bundled in an octave_value_list. Results returned from a function are also passed in an octave_value_list. The default constructor, `octave_value_list`, creates an empty list which, when used as return value, yields an Octave function returning "void". To access the elements of a list the following methods form class `octave_value_list` (defined in OCT/src/`oct-obj.h`) are useful:

```
class octave_value_list {
 public octave_value_list();
```

```
  public octave_value& operator()(int index);
  public const octave_value operator()(int index);
  public const int length();
}
```

`length` returns the number of elements in the list. The overloaded parenthesis operator selects a single element from the list. The index base for *`index`* is 0 and *not* 1 as Fortran adepts might infer. The value returned by `operator()` is an octave_value, i.e., any type known to Octave, e.g. integers, real matrices, complex matrices, and so on.

Knowing how to access the arguments of unspecified type, the next thing we would like to do is get their values. Octave follows a uniform naming scheme: all functions that return an object of a certain type ends in `_type`. Some of the more important of these methods (defined in OCT/src/`ov-base.h`) are:

```
 class octave_base_value {
  public const int int_value();
  public const double double_value();
  public const Complex complex_value();
  public const Matrix matrix_value();
  public const ComplexMatrix complex_matrix_value();
}
```

1.1.3.3. Low-level implementation

Now, we are ready to implement `matpow` as a dynamically loaded extension to Octave.

Example 1-3. C++ implementation of `matpow`

```
#include <octave/oct.h>


static bool is_even(int n);


DEFUN_DLD(matpow, args, ,
          "Return b = a^n for square matrix a, and non-negative, integral n.")
{
    const int n = args(1).int_value();  ❶

    if (n == 0)
        return octave_value(
               DiagMatrix(args(0).rows()❷, args(0).columns()❸, 1.0) ❹
               );
    if (n == 1)
        return args(0);

    Matrix p(args(0).matrix_value());   ❺
    Matrix b(p);
    int np = n - 1;
    while (np >= 1)
```

```
    {
        if (is_even(np))
        {
            np = np / 2;
        }
        else
        {
            np = (np - 1) / 2;
            b = b * p;                    ❻
        }
        p = p * p;
    }

    return octave_value(b);               ❼
}


bool
is_even(int n)
{
    return n % 2 == 0;
}
```

❶ Get the exponent `n`, which is the *second* argument to `matpow` through `args`(1) and retrieve its integral value with `int_value`.

❷ The matrix that we want to raise to the n-th power is the *first* argument, therefore it is accessed through `args`(0). The method `rows` returns the number of rows in the matrix.

❸ The method `columns` returns the number of columns in the matrix. (The actual value is assumed to be equal to the number of rows. At this stage, we are tacitly assuming that all parameters passed to `matpow` are valid, which means especially that the matrix is square.)

❹ We call a constructor for diagonal matrices, `DiagMatrix` (defined in OCT/liboctave/`dDiagMatrix.h`), that accepts the size of the matrix and the value to put on the diagonal, which is 1 in our case.

❺ Initialise the matrix `p` that will store the powers of the base. The `Matrix` constructor cannot take an octave_value and we have to supply the matrix itself by invoking `matrix_value`.

❻ Multiplication has been conveniently overloaded to work on matrices. Wanna give John a treat for this one?

> **Tip:** The Octave library overloads all elementary operations of scalars, (row-/column-) vectors and matrices. If in doubt as to whether a particular operation has been overloaded, simply try it. It takes less time than browsing (read: grep through) all the sources – the desired elementary operation is implemented in most cases.

❼ We can return the result matrix `b` directly, as an appropriate constructor is invoked to convert it to an octave_value_list.

We learn from the example that the C++ closely resembles the Octave function. This is due to the clever class structure of the Octave library.

1.1.3.4. Compiling

Now that we are done with the coding, we are ready to compile and link. The Octave distribution contains the script **mkoctfile**, which does exactly this for us. In the simplest case it is called with the C++-source as its only argument.

```
$ ls
Makefile  matpow0.m  matpow1.m  matpow.cc  test_matpow.m
$ mkoctfile matpow.cc
```

Good, but not good enough! Presumably, we shall compile several times, so we would like to run our test suite and finally remove all files that can be re-generated from source. Enter: `Makefile`.

Example 1-4. Simple `Makefile` for `oct`-files

```
# Makefile for Octave extension matpow

.phony: all
all: matpow.oct

.phony: clean
clean:
        rm -f matpow.oct *.o

.phony: distclean
distclean: clean
        rm -f *~ octave-core core

.phony: test
test: matpow.oct
        octave --silent test_matpow.m

%.oct: %.cc
        mkoctfile $<
```

1.1.3.5. Running

If the `oct`-file is in the LOADPATH, it will be loaded automatically – either when requesting **help** on the function or when invoking it directly.

```
$ ls
Makefile  matpow0.m  matpow1.m  matpow.cc  matpow.o  matpow.oct  test_matpow.m
$ octave -q
```

```
octave:1> help matpow
matpow is the dynamically-linked function from the file
/home/cspiel/hsc/octave/src/matpow/matpow.oct

Return b = a^n for square matrix a, and non-negative, integral n.

octave:2> matpow([2, 0; 0, -1], 4)
ans =

  16   0
   0   1

octave:3> Ctrl-D
```

1.1.4. Spicing up `matpow`

The keen reader will have noticed that `matpow` is highly vulnerable to unexpected input. We take care of this deficiency in Section 1.1.4.1. Also, the documentation is not ready for prime time, something we attend to in Section 1.1.4.2.

1.1.4.1. Input parameter checking

Why do we need parameter checking at all? A rule of thumb is to perform a *complete* parameter checking on

- every function that is exposed to the user (e.g. at Octave's interpreter level), i.e. for every function created with DEFUN_DLD, or
- any function that might be used in a different context than the writer intended Functions that are used only internally do not require full parameter checking.

One problem which arises frequently with parameter checking of (interfacing) functions is that the checks easily take up more space than the routine itself, thereby distracting the reader from the function's original purpose. Often, including all the checks into the main function bloats it beyond the desired maintainability. Therefore, we have to take precautions against these problems by consistently factoring out all parameter checks.

The rule of thumb here is to group logically related tests together in separate functions. The testing functions get the original functions' arguments *by constant reference* and return a boolean value. *Constant referencing* avoids any constructor calls and – in addition to that – protects the arguments against modification. In our example a single test function is enough:

```
static bool any_bad_argument(const octave_value_list& args);
```

`any_bad_argument` prints a detailed message, raises an Octave error, and returns `true` if the arguments fail any test; otherwise it returns `false` and we can continue processing. The only thing we have to change in `matpow` is to add a call to `any_bad_argument` and on failure to return an Octave-void value, i.e. an empty octave_value_list. The first few lines of `matpow` then take the following form:

```
DEFUN_DLD(matpow, args, ,
          "Return b = a^n for square matrix a, and non-negative, integral n.")
{
    if (any_bad_argument(args))
        return octave_value_list();

    const int n = args(1).int_value();

    -- rest of matpow is unchanged
}
```

As we are convinced that we have to check the input arguments, the question is how to do the checks.

> **Ordering of Tests:** The correct ordering of tests is essential!
>
> 1. Actual number of arguments
> 2. Type of argument
> 3. Size of argument (where applicable)
> 4. Range of argument (where applicable and necessary)
> 5. Inter-argument relations (if necessary)
>
> Item 1 always goes first. Items 2-4 usually repeat for each of the arguments. The final tests check the relations between the arguments, i.e., belong to Item 5.

Octave supplies the user with all necessary type- and size-testing functions. The type-tests (defined in OCT/src/`ov-base.h`) share the common prefix `is_`. Here are the most commonly used:

```
 class octave_base_value , public class foo_value {
  public const bool is_real_scalar();
  public const bool is_complex_scalar();
  public const bool is_real_matrix();
  public const bool is_complex_matrix();
}
```

To examine the sizes of different Octave objects, the following methods prove useful:

```
 class octave_list , public class octave_base_value {
  public const int length();
}
 class octave_base_matrix , public class octave_base_value {
  public const int rows();
  public const int columns();
  public const int length();
}
```

Remember that the methods available depend on the underlying type. For example, a ColumnVector only has a length (OCT/src/`ov-list.h`), whereas a Matrix has a number of rows and columns (OCT/src/`ov-base-mat.h`).

We have all the knowledge we need to write the argument testing function to augment `matpow`.

```
static bool
any_bad_argument(const octave_value_list& args)
{
    if (!args(0).is_real_matrix())
    {
        error("matpow: expecting base A (arg 1) to be a real matrix");
        return true;
    }

    if (args(0).rows() != args(0).columns())
    {
        error("matpow: expecting base A (arg 1) to be a square matrix");
        return true;
    }

    if (!args(1).is_real_scalar())
    {
        error("matpow: exponent N (arg 2) must be a real scalar.");
        return true;
    }

    if (args(1).scalar_value() < 0.0)
    {
        error("matpow: exponent N (arg 2) must be non-negative.");
        return true;
    }

    if (floor(args(1).scalar_value()) != args(1).scalar_value())
    {
        error("matpow: exponent N (arg 2) must be an integer.");
        return true;
    }

    return false;
}
```

Out final duty is to update the test-frame and run it. For brevity, we only list the new tests in `testmp.m`

```
###
### The following tests should trigger the error exits.
###

## number of arguments
```

```
unit_test_err("error exit, too few arguments",
              "matpow:", "matpow([1,1; 1,1])");

unit_test_err("error exit, too few arguments",
              "matpow:", "matpow()");

unit_test_err("error exit, too many arguments",
              "matpow:", "matpow([1,1; 1 1], 2, 1)");

## argument type and size

unit_test_err("error exit, A not a matrix",
              "matpow:", "matpow(1, 1)");

unit_test_err("error exit, A not a square matrix",
              "matpow:", "matpow([1 2 3; 4 5 6], 1)");

unit_test_err("error exit, N not a real scalar(here: complex)",
              "matpow:", "matpow(a, 1+i)");

unit_test_err("error exit, N not a real scalar(here: non-scalar)",
              "matpow:", "matpow(a, ones(2,2))");

unit_test_err("error exit, N negative",
              "matpow:", "matpow(a, -1)");

unit_test_err("error exit, N non-integral",
              "matpow:", "matpow(2.5)");
```

```
$ octave -q test_matpow.m
................
# of testcases attempted   16
# of expected passes       16
# of expected failures     0
# of unexpected passes     0
# of unexpected failures   0
# of unresolved testcases  0

Unit testing completed successfully!
```

1.1.4.2. Texinfo documentation strings

Our much improved `matpow` still carries around the poor documentation string:

"Return b = a^n for square matrix a, and non-negative, integral n."

Let us improve on this one! Octave supports documentation strings – docstrings for short – in Texinfo format. The effect on the online documentation will be small, but the appearance of printed documentation will be greatly improved.

The fundamental building block of Texinfo documentation strings is the Texinfo-macro `@deftypefn`, which takes two arguments: The class the function is in, and the function's signature. For functions defined with `DEFUN_DLD`, the class is `Loadable Function`. A skeletal Texinfo docstring therefore looks like this:

Example 1-5. Skeletal Texinfo Docstring

```
-*- texinfo -*- ❶
@deftypefn{Loadable Function}❷ {return_values =} function_name(arguments)

@cindex index term ❸

Help text in Texinfo format...
@end deftypefn
```

❶ Tell the parser that the doc-string is in Texinfo format.

❷ `@deftypefn{class} ... @end deftypefn` encloses the whole doc-string, like a LaTeX environment or a DocBook element does.

❸ `@cindex index entry` generates an index entry. It can be used multiple times.

Texinfo has several macros which control the markup. In the group of format changing commands, we note `@var{variable_name}`, and `@code{code_piece}`. The Texinfo format has been designed to generate output for online viewing with text-terminals as well as generating high-quality printed output. To these ends, Texinfo has commands which control the diversion of parts of the document into a particular output processor. Two formats are of importance: info and TeX. The first is selected with

```
@ifinfo
text for info only
@end ifinfo
```

the latter with

```
@iftex
@tex
text for TeX only
@end tex
@end iftex
```

If no specific output processor is chosen, by default, the test goes into both (or, more precisely: every) output processors. Usually, neither `@ifinfo`, nor `@iftex` appear alone, but always in pairs, as the same information must be conveyed in every output format.

Example 1-6. Documentation string in Texinfo format

```
-*- texinfo -*-
@deftypefn{Loadable Function} {@var{b} =} matpow(@var{a}, @var{n})

@cindex matrix power

Return matrix @var{a} raised to the @var{n}-th power.  Matrix @var{a} is
a square matrix and @var{n} a non-negative integer.
@iftex ❶
@tex ❷
$n = 0$
@end tex
@end iftex
@ifinfo ❸
n = 0
@end ifinfo
is explicitly allowed, returning a unit-matrix of the same size as
@var{a}.

Note: @code{matpow} ❹ duplicates part of the functionality of the built-in
exponentiation operator
@iftex
@tex
“$\wedge$”.
@end tex
@end iftex
@ifinfo
‘@code{^}’.
@end ifinfo


Example:
@example ❺
matpow([1, 2; 3, 4], 4)
@end example
@noindent
returns
@example
ans =

  199  290
  435  634
@end example

The algorithm takes
@iftex
@tex
$\lfloor \log_{2}(n) \rfloor$
@end tex
@end iftex
@ifinfo
floor(log2(n))
```

```
@end ifinfo
matrix multiplications to compute the result.
@end deftypefn
```

❶ `@iftex ... @end iftex` selects text for conditional inclusion. Only if the text is processed with an Tex the included section will be processed.

❷ `@tex ... @end tex` wraps parts of the text that will be fed through TeX.

❸ `@ifdoc ... @end ifdoc` selects text for conditional inclusion. Only if the text is processed with an info-tool the included section will be processed.

❹ `@code{code sequence}` marks up a a code sequence.

❺ `@example ... @end example` wraps examples.

For further information about Texinfo consult the Texinfo documentation. For TeX-beginners we recommend "The Not So Short Introduction to LaTeX" by Tobias Oetiker et. al.

One thing we held back is the true appearance of a Texinfo docstring – mainly because it looks so ugly. The C++-language imposes the constraint that the docstring must be a string-constant. Moreover, because `DEFUN_DLD` is a macro, every line-end has to be escaped with a backslash. The backslash does not insert any whitespace and TeX separates paragraphs with empty lines, so that we have to put in new-lines as line-separators. Thus, the Texinfo docstring in source form has each line end decorated with "`\n\`".

```
DEFUN_DLD(matpow, args, ,
          "-*- texinfo -*-\n\
@deftypefn{Loadable Function} {@var{b}} = matpow(@var{a}, @var{n})\n\
\n\
@cindex matrix power\n\
\n\
Return matrix @var{a} raised to the @var{n}-th power.  Matrix @var{a} is\n\
a square matrix, and @var{n} a non-negative integer.\n\
...")
```

At least the formatted versions look much better. The info-version, which will be used in Octave's online help has approximately the following appearance:

```
b = matpow(a, n)                                        Loadable Function
Return matrix a raised to the n-th power. Matrix a is a square matrix, and
n a non-negative integer. n = 0 is explicitely allowed, returning a
unit-matrix of the same size as a.

Note: matpow duplicates part of the functionality of the built-in
exponentiation operator `^'.

Example:
matpow([1, 2; 3, 4], 4)

returns
```

```
ans =

  199  290
  435  634

The algorithm takes floor(log2(n)) matrix multiplications to compute the
result.
```

whereas the TeX-version will look like this:

b = **matpow**(a, n) [Loadable Function]

Return matrix a raised to the n-th power. Matrix a is a square matrix, and n a non-negative integer.

Return matrix a raised to the n-th power. Matrix a is a square matrix and n a non-negative integer. $n = 0$ is explicitly allowed, returning a unit-matrix of the same size as a.

Note: matpow duplicates part of the functionality of the built-in exponentiating operator "^".

Example:

```
matpow([1, 2; 3, 4], 4)
```

returns

```
ans =

  199  290
  435  634
```

The algorithm takes $\lfloor \log_2(n) \rfloor$ matrix multiplications to compute the result.

Docstring of `matpow` after rendering with TeX. Note the differences between the info-version and the TeX-version, that have been introduced with `@ifinfo` and `@iftex`.

1.2. Advanced Extension Programming

...

1.2.1. Defining Constants And Variables

The definition of constants and variables in a dynamically linked GNU/Octave extension resembles the header of a dynamically linked function (see, for example, Section 1.1.3). However, the appropriate macro `DEFCONST` is not available when creating a dynamically loadable extension for it is defined in `defun.h` and not in `defun-dld.h`. The latter is necessary to set up an dynamically loadable extension.

The easiest, moderately clean way is to duplicate DEFCONST's definition from defun.h into the respective extension.

```
// same definition as found in defun.h

#ifndef DEFCONST
#define DEFCONST(name, defn, doc) DEFCONST_INTERNAL(name, defn, doc)
#endif
```

DEFCONST introduces a constant named constant_name at the interpreter level, giving it the value of defining_expression and endowing it with the documentation documentation_string. The newly created constant will be protected against deletion, but *not* against change.

```
void
DEFCONST(constant_name,
         defining_expression,
         const std::string& documentation_string)
```

The name of the constant constant_name must be a valid C++ identifier, because it is *not* quoted. Octave automatically casts the variable's definition, defining_expression, to type octave_value.

A constant can be assigned to and then takes on the new value! Assigning to a constant does not even produce a warning. **clear**ing a protected constant does not give raise to a warning either. Clearing a protected constants re-installs its original value.

Example 1-7. Constant Definition

```
#include <oct.h>

// 'DEFCONST' from "defun.h"
#ifndef DEFCONST
#define DEFCONST(name, defn, doc) \
    DEFCONST_INTERNAL(name, defn, doc)
#endif

static const double h = 6.626176e-34; // Planck's constant in J*s

DEFUN_DLD(defconst, args, ,
          "Install some fundamental physical constants.")
{
    if (args.length() == 0)
    {
        DEFCONST(c, 2.99792458e8,
                 "Speed of light in m/s.");
        DEFCONST(hbar, h / (2.0 * M_PI),
                 "Reduced Planck's constant hbar, that is, h/(2*pi) in J*s.");
        DEFCONST(G, 6.672e-11,
                 "Graviation constant in N*m^2/kg^2.");
        DEFCONST(e, 1.6021892e-19,
                 "(Absolute value of the) Charge of an electron in C.");
```

```
    }
    else
    {
        error("defconst: expecting no arguments.");
    }

    return octave_value_list();
}
```

Tip: Long documentation strings in a long series of definitions tend to obscure the code. Assigning the documentation string to a macro allows for a separation of the help text and the definition.

```
#define DOCSTRING_HBAR \
"Reduced Planck's constant hbar, this is, h/(2*pi) in J*s."

...

DEFCONST(hbar, h / (2.0 * M_PI), DOCSTRING_HBAR);
```

This is also useful for describing the function's documentation string.

Like `DEFCONST` is defined in `defun.h` and not in `defun-dld.h`. So, the programmer must introduce the macro himself.

```
// same definition as found in defun.h

#ifndef DEFVAR
#define DEFVAR(name, defn, chg_fcn, doc) \
    DEFVAR_INTERNAL(#name, SBV_ ## name, defn, false, chg_fcn, doc)
#endif

void
DEFVAR(variable_name,
       defining_expression,
       symbol_record::change_function changing_function,
       const std::string& documentation_string)
```

The parameters `variable_name`, *`defining_expression`*, and *`documentation_string`* are analogous to those of `DEFCONST`. Only *`changing_function`* calls for further explanation.

`changing_function` is a pointer to a function that gets called whenever variable variable_name is given a new value. *`changing_function`* can be `NULL` if there is no function to call. change_function is defined in `symtab.h`

```
// symtab.h

typedef int (*change_function) (void);
```

A *changing_function* never takes on any parameters! Therefore, it must have a built-in knowledge of which interpreter variable to take care of. Usually, *changing_function*s correspond one-to-one with variable_names. Note the *changing_function* is called to initialize variable_name with the value of defining_expression. This means that *changing_function* is called at least once even if variable_name never gets changed withing the interpreter. The return value 0 from *changing_function* signals success to the caller, any other value stands for failure.

DEFVAR installs and initializes a variable in the interpreter's workspace. To access a variable or constant, variables.h declares three functions:

```
#include <variables.h>

std::string builtin_string_variable ( const std::string& symbol_name );
int builtin_real_scalar_variable ( const std::string& symbol_name ,  double&
value );
octave_value builtin_any_variable ( const std::string& symbol_name );
```

Example 1-8. Variable Definition

```
#include <oct.h>
#include <variables.h>            // for 'builtin_*_variable'

// 'DEFVAR' from "defun.h"
#ifndef DEFVAR
#define DEFVAR(name, defn, chg_fcn, doc) \
    DEFVAR_INTERNAL(#name, SBV_ ## name, defn, false, chg_fcn, doc)
#endif

static double counter_var;
static unsigned count = 0;

static int counter_set();

//
// documentation strings
//

#define DOCSTRING_DEFVAR \
"Define two variables in the workspace: simple and counter.\n\
See the respective documentations, that is,\n\
'help simple' and 'help counter'."

#define DOCSTRING_SIMPLE \
"Variable 'simple' is initialized to 0.5.\n\
It is not linked to any low-level variable."

#define DOCSTRING_COUNTER \
"Variable 'counter' is initialized to 1.0.\n\
It is linked to the C++ variable 'counter_var' in file 'defvar.cc'.\n\
Whenever 'counter' is assigned to the number of assigments\n\
```

```
is printed."

//
// body
//

DEFUN_DLD(defvar, args, , DOCSTRING_DEFVAR)
{
    if (args.length() == 0)
    {
        DEFVAR(simple, 0.5, 0, DOCSTRING_SIMPLE);
        DEFVAR(counter, 1.0, counter_set, DOCSTRING_COUNTER);
    }
    else
    {
        error("defvar: expecting no arguments.");
    }

    return octave_value_list();
}

static int
counter_set()
{
    if (builtin_real_scalar_variable("counter", counter_var) == 0)
    {
        error("counter_set: internal error, non-existent variable");
        return 1;
    }

    count++;
    cout << "==> `counter' has been assigned to " << count << " times;\n"
         << "==> its new value is " << counter_var << ".\n";

    return 0;
}
```

1.2.2. Documenting Constants And Variables

Having studied Section 1.1.4.2, only one new Texinfo function is waiting: `defvr`.

```
-*- texinfo -*-
@defvr {Built-in Variable} my_own_variable
...
@end defvr
```

1.3. Abridged Quick-Reference

Question to Radio Erivan: "Is Octave lacking a Quick Reference?" Answer: "No, Octave ships with a very nice 3 page Quick Reference card, but it only describes the interpreter, not the library-interface."

Coda has to fill in the gap. The library quick reference is called "abridged", which is a fancy word for "incomplete". According to the establisheed 80-20 rule, the reader will find 80% of the functions she needs for usual programming jobs. The missing 20% will take 80% of the total coding time to hunt down.

All methods listed in this section have public access specification. Therefore, we have dropped the `public` modifiers in the class' synopis.

To generate an up-to-date class reference using doxygen, issue the command

```
doxygen -g octave-dox
```

Edit `octave-dox`, set `EXTRACT_ALL` and `RECURSIVE` to YES and `INPUT` to the location of the Octave include files, e.g. `/usr/include/octave-2.1.50`. Now execute

```
doxygen octave-dox
```

to generate the documentation in `./html`.

1.3.1. Types

We follow the old tradition from the days of Pascal and introduce first the types the Octave library works on.

Table 1-1. Types

Type	Files	Description
octave_value_list	`oct-obj.h`	Heterogenous container that is used in passing arguments to and from a dynamicaly loadable extension. I.e. the parameters and return values of a loadable function are mapped onto octave_value_lists. See Section 1.3.2.
octave_value	`ov.h`	Single element of an octave_value_list, which can hold *any* of Octave's types.

Type	Files	Description
octave_matrix	ov-re-mat.h	Any of Octave's real-valued matrix (Matrix, DiagMatrix) or vector types (RowVector, ColumnVector).
octave_complex_matrix	ov-cx-mat.h	Any of Octave's complex-valued matrix (ComplexMatrix, ComplexDiagMatrix) or vector types (ComplexRowVector, ComplexColumnVector).
octave_bool_matrix	ov-bool-mat.h	?
octave_char_matrix	ov-char-mat.h	?
octave_scalar	ov-scalar.h	Real (double) valued, complex (Complex) valued, or boolean (bool) scalar.
RowVector	oct-obj.h, ov-re-mat.h, ov.h	Real (double) valued row vector; works like a matrix with a single row.
ComplexRowVector	oct-obj.h, ov-cx-mat.h, ov.h	Complex (double) valued row vector; works like a matrix with a single row.
ColumnVector	oct-obj.h, ov-re-mat.h, ov.h	Real (double) valued column vector; works like a matrix with a single column.
ComplexColumnVector	oct-obj.h, ov-cx-mat.h, ov.h	Complex (Complex) valued column vector; works like a matrix with a single column.
Matrix	ov-re-mat.h,	Real (double) valued matrix.
ComplexMatrix	ov-cx-mat.h,	Complex (Complex) valued matrix.
boolMatrix	?	Matrix of booleans (bool).
charMatrix	?	Matrix of characters (char).
DiagMatrix	ov-re-mat.h, ov.h	Real (double) valued diagonal matrix.
ComplexDiagMatrix	ov-cx-mat.h, ov.h	Complex (double) valued diagonal matrix.
Cell array	ov-cell.h, Cell.h,	Cell array.

1.3.2. List Operations

.

```
class octave_value_list {
```

```
  octave_value_list();
  octave_value_list(const T& t);
  octave_value& operator()(int index);
  const octave_value operator()(int index);
  const int length();
  const bool empty();
  octave_value_list& prepend(const octave_value& val);
  octave_value_list& append(const octave_value& val);
  octave_value_list& append(const octave_value_list& lst);
  octave_value_list& reverse();
  const octave_value_list& splice(int offset, int length, const octave_value_list& lst);
  const octave_value_list& index(idx_vector& i);
}
```

Most of the methods in `octave_value_list` are self explaining. The default constructor, `splice`, and `index` require some explanation.

Default Constructor. The default constructor makes an empty list. When an empty list is returned from a dynamically loaded extension, the function at the interpreter level behaves lieka procedure, i.e. returns "void" in other words returns nothing.

Converting Constructors. `octave_value_list` defines constructors for the following types *T*: double, Matrix, DiagMatrix, RowVector, ColumnVector, Complex, ComplexMatrix, ComplexDiagMatrix, ComplexRowVector, ComplexColumnVector, char*, std::string, string_vector, Range, and octave_value_list.

`splice`. Replaces *length* elements of the list, starting at *offset*, and then inserts *lst*. See also Octave documentation of `splice`.

`index`. Extract elements at the index values given by idx_vector *i*.

1.3.3. Access to Value

.

```
 class octave_base_value {
  const bool bool_value();
  const int int_value();
  const int nint_value();
  const double double_value();
  const Complex complex_value();
  const double scalar_value();
  const std::string string_value();
  const Range range_value();
  const Octave_map map_value();
  const octave_stream stream_value();
  octave_function* function_value(bool silent);
  const Cell cell_value();
  const boolMatrix bool_matrix_value();
  const Matrix matrix_value();
  const ComplexMatrix complex_matrix_value();
```

```
  const charMatrix char_matrix_value();
  const octave_value_list list_value();
}
```

1.3.4. Type Classification

.

```
 class octave_base_value , public class foo_value {
  const bool is_scalar_type();
  const bool is_bool_type();
  const bool is_complex_type();
  const bool is_constant();
  const bool is_numeric_type();
  const bool is_range();
  const bool is_real_scalar();
  const bool is_real_type();
  const bool is_string();
  const bool is_matrix_type();
  const bool is_char_matrix();
  const bool is_complex_matrix();
  const bool is_complex_scalar();
  const bool is_real_matrix();
  const bool is_all_va_args();
  const bool is_builtin_function();
  const bool is_cell();
  const bool is_defined();
  const bool is_dld_function();
  const bool is_empty();
  const bool is_function();
  const bool is_list();
  const bool is_magic_colon();
  const bool is_map();
  const bool is_stream();
  const bool is_true();
  const bool is_zero_by_zero();
}
```

Notes

1. Octave alread has a matrix power operator, the caret “^”, which is more powerful than our example ever will be. This should not bother us.
2. Octave-forge contains a good unit testing framework. The framework provided here is very simple and only intended as a teaching aid.

Chapter 2. Building Standalone Applications

The libraries Octave itself uses, can be utilized in standalone applications. These applications then have access, for example, to the vector and matrix classes as well as to all the Octave algorithms.

The following C++ program, Example 2-1, uses class `Matrix` from `liboctave.a` or `liboctave.so`.

Example 2-1. "Hello World!" program using Octave's libraries.

```
#include <iostream>
#include "oct.h"

int
main(void)
{
    std::cout << "Hello Octave world!\n";

    const int size = 2;
    Matrix a_matrix = Matrix(size, size);
    for (int row = 0; row < size; ++row)
    {
        for (int column = 0; column < size; ++column)
        {
            a_matrix(row, column) = (row + 1)*10 + (column + 1);
        }
    }
    std::cout << a_matrix;

    return 0;
}
```

`mkoctfile` can once again be used to compile our application:

```
$ mkoctfile --link-stand-alone hello.cc -o hello
$ ./hello

Hello Octave world!
 11 12
 21 22

$
```

Appendix A. Complete examples

This appendix gathers all examples that are either too long for the main text or are referenced (but not defined in) the main text.

A.1. Unit-test

A.1.1. `unit_init.m`

```
function unit_init(verbosity, global_vars)
    ## Initialize the global structure unittest_results, which is needed
    ## in all functions of the *unittest module.  Debugging information
    ## is printed if verbosity==1.  global_vars is a cell array of the
    ## names of the global variables used in the tests.
    ##
    ## e.g. unit_init(1, {"g", "a", "x"})

    global unittest_results;

    unittest_results.verbose = 0;
    unittest_results.eval_globals = {};
    if (nargin > 0)
        if (!isscalar(verbosity) || verbosity < 0 || verbosity > 1)
            warning("unit_init: verbose must be 0 or 1");
        else
            unittest_results.verbose = verbosity;
        endif

        if (nargin == 2 && iscell(global_vars))
            for i = 1:length(global_vars)
                unittest_results.eval_globals{i} = strcat("global ", global_vars{i}, ";");
            endfor
        else
            error("global_vars must be a cell array");
        endif

        if (nargin > 2)
            usage("expecting 2 arguments only");
        end
    endif

    unittest_results.total = 0; # number of testcases attempted
    unittest_results.pass = 0;  # number of expected passed
    unittest_results.fail = 0;  # number of unexpected failures
    unittest_results.upass = 0; # number of unexpected passes
    unittest_results.xfail = 0; # number of expected failures
    unittest_results.unresolved = 0; # number of unresolved testcases
```

```
    default_eval_print_flag = 0;
endfunction
```

A.1.2. unit_test.m

```
function result = unit_test(test_title, expect_pass, actual_result)
     ## Function unittest compares the ACTUAL_RESULT of running
     ## a test (either  0 for failure, or 1 for success) with the
     ## expected outcome of the test EXPECT_PASS (either 0 for expecting
     ## a failure, or 1 for expecting pass).  TEST_TITLE is the name of
     ## the test.  All test results will be accompanied by the test's
     ## title.
     ##
     ## The result of unittest is on of the following: UNRESOLVED: The
     ## test did neither return 0 nor 1.  PASS: expected pass, got pass.
     ## FAIL: expected pass, got fail.  UPASS: expected fail, got pass.
     ## XFAIL: expected fail, got fail.
     ##
     ## A call to unittest typically looks like this:
     ##
     ## unittest("scalar integer addition", 1, eval("1 + 1 == 2;"));

     global unittest_results;

     ## Sanity check input parameters
     if ( nargin < 3 | nargin > 4 )
         error("Function run_rest expects 3 or 4 parameters.");
     endif

     if (!isstr(test_title))
         error("Expecting TEST_TITLE (arg 1) to be a string.");
     endif

     if (expect_pass != 0 & expect_pass != 1)
         error("Expecting EXPECT_PASS (arg 2) to be 0 or 1.");
     endif

     unittest_results.total++;

     ## Take actions depending on what test result we expect
     ## (expect_pass), and what we actually got (actual_result).
     if (actual_result != 0 & actual_result != 1)
         result = "UNRESOLVED";
         unittest_results.unresolved++;
         if (actual_result == 2)
             printf("SYNTAX ERROR: %s\n", test_title);
         endif
         return;
     endif

     if (expect_pass == 1 & actual_result == 1)
```

```
        result = "PASS";
        if (unittest_results.verbose != 0)
            printf("PASS: %s\n", test_title);
        else
            printf('.');
        endif
        unittest_results.pass++;
    elseif (expect_pass == 1 & actual_result == 0)
        result = "FAIL";
        printf("FAIL: %s\n\n", test_title);
        unittest_results.fail++;
    elseif (expect_pass == 0 & actual_result == 0)
        result = "XFAIL";
        printf("XFAIL: %s\n", test_title);
        unittest_results.xfail++;
    elseif (expect_pass == 0 & actual_result == 1)
        result = "UPASS";
        printf("UPASS: %s\n", test_title);
        unittest_results.upass++;
    endif
endfunction
```

A.1.3. unit_test_equals.m

```
function unit_test_equals(test_title, result_A, result_B)

    unit_test(test_title, 1, all(all(result_A == result_B)));

endfunction
```

A.1.4. unit_test_err.m

```
function unit_test_err(test_title, expected_error_prefix, evaluated_string)

    unit_test(test_title, 1, erreval(expected_error_prefix, evaluated_string));

endfunction
```

A.1.5. errcheck.m

```
function e = errcheck(error_prefix)

    e = 1;
    if (index(__error_text__, error_prefix) != 1)
        e = 0;
        printf("\nUNEXPECTED ERROR: %s", __error_text__);
    endif
```

```
end
```

A.1.6. erreval.m

```
function rv = erreval(error_prefix, try_str, catch_str)
    ## erreval() extends the built-in function eval().  Return 0 if
    ## try_str does not raise the error of type error_prefix, return 1
    ## otherwise.

    global unittest_results;
    for k = 1:length(unittest_results.eval_globals)
        eval(unittest_results.eval_globals{k});
    end

    rv = 0;
    try
        eval(try_str);
    catch
        rv = errcheck(error_prefix);
    end

endfunction
```

A.1.7. unit_results.m

```
 function unit_results()
     ## Print the results from previous unittest calls in pretty format.

     global unittest_results;

     printf("\n");
     printf("# of testcases attempted   %d\n", unittest_results.total);
     printf("# of expected passes       %d\n", unittest_results.pass);
     printf("# of expected failures     %d\n", unittest_results.xfail);
     printf("# of unexpected passes     %d\n", unittest_results.upass);
     printf("# of unexpected failures   %d\n", unittest_results.fail);
     printf("# of unresolved testcases  %d\n", unittest_results.unresolved);
     printf("\n");

     if (unittest_results.total == unittest_results.pass + unittest_results.xfail)
         printf("Unit testing completed successfully!\n");
     else
         printf("One or more tests failed!\n");
     endif
 endfunction
```

Appendix B. GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of

Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled “History” in the various original documents, forming one section entitled “History”; likewise combine any sections entitled “Acknowledgements”, and any sections entitled “Dedications”. You must delete all sections entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an “aggregate”, and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document’s Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See *GNU Copyleft* (http://www.gnu.org/copyleft/).

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of

following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Appendix C. GNU General Public License

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU General Public License

Terms And Conditions For Copying, Distribution And Modification

0. APPLICABILITY[1]

This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. VERBATIM COPYING

You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. MODIFICATIONS

You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. DISTRIBUTION

You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. TERMINATION

You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. ACCEPTANCE

You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. REDISTRIBUTION

Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these

terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. CONSEQUENCES

If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. LIMITATIONS

If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in

detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. AGGREGATION WITH INDEPENDENT WORKS

If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. NO WARRANTY

BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. LIABILITY

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO

OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Notes

1. The titles of the sections have been added by the authors. They do not occur in the original GNU General Public License. Everything else has been copied in verbatim.

Bibliography

[Golub:1996] Gene H. Golub and Charles F. van Loan, *Matrix Computations*, third edition, The Johns Hopkins University Press, 1996, 0-8018-5413-X.

Index

A

application
  standalone, 25
args (dynamic function parameter)
  (see macro, DEFUN_DLD)

C

constant
  access from dynamical extension, 19
  behavior of a, 17
  definition
    (see definition of a constant (oct))

D

DEFCONST
  (see Octave library, macro, DEFCONST)
definition
  of a constant (oct), 16
  of a variable (oct), 16
DEFUN_DLD
  (see macro, DEFUN_DLD)
DEFVAR
  (see Octave library, macro, DEFVAR)
dynamic extension
  (see file, oct)

E

examples, 26
  unit test, 26

F

file
  oct, 1

G

GNU
  Free Documentation License (FDL), 30
  General Public License (GPL), 37

H

header file
  defun-dld.h, 16
  defun.h, 16

M

macro
  DEFUN_DLD, 5
mkoctfile, 8

N

nargout (dynamic function parameter)
  (see macro, DEFUN_DLD)

O

oct-file
  (see file, oct)
Octave function
  clear, 17
Octave library
  function
    builtin_any_variable, 19
    builtin_real_scalar_variable, 19
    builtin_string_variable, 19
    symbol_record::change_function, 18
  macro
    DEFCONST, 17
    DEFVAR, 18
  typedef
    change_function, 18

S

standalone applications
(see applications, standalone)

T

tests
ordering of, 10

V

variable
access from dynamical extension, 19
definition
(see definition of a variable (oct))