



Da Coda AI Fine

Pushing Octave's Limits

A collaborative document authored by the Octave Forge community.

Table of Contents

1 CodaTutorial	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.....	3
1.1.4 Spicing up matpow.....	7
2 CodaAdvanced	13
2.1 Advanced Extension Programming.....	13
2.1.1 Defining Constants And Variables.....	13
2.1.2 Documenting Constants And Variables.....	16
2.1.3 Using static variables.....	16
2.1.4 Accessing global variables.....	17
3 CodaStandalone	19
3.1 Building Standalone Applications.....	19
4 CodaStruct	20
5 CodaTypes	21
5.1 Abstract.....	21
5.2 Introduction.....	21
5.2.1 Requirements.....	21
5.2.2 Need for a Type System.....	22
5.2.3 RTTI in GNU Octave.....	22
5.2.4 Making Custom Octave Objects.....	23
5.2.5 Dynamic Loaded Functions.....	27
5.2.6 Testing with DEFUN_DLD.....	27
5.2.7 See Also.....	28
5.3 Credits.....	28
5.4 GNU Free Documentation License.....	28

1 CodaTutorial

DaCodaAlFine – CodaTutorial

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 **CodaTutorial**. This will teach any reader who is familiar with C++ how to write her first dynamically linked extension for Octave. Having guided the reader in her first steps, **CodaAdvanced** covers more advanced topics of writing extensions.

1.1 Tutorial

The tutorial unfurls slowly; we do not want to scare anybody by the complexity of the task ahead. Also look at the sources in 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. 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 .

Note:

Octave already has a matrix power operator, the caret " \wedge ", which is more powerful than our example ever will be. This should not bother us.

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, than 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:

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

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

Easy does the job! `matpow` looks like it does what we want, but how can we be sure? The solution lies in unit tests -- extremely useful (and necessary) when implementing code in C++.

Add the following snippet to the bottom of `matpow`:

```
%!shared a
%!test
%! a = [ 2.0, -3.0;
%!      -1.0,  1.0];
%!
%!assert(matpow(a,0), diag([1,1]));
%!assert(matpow(a,1), a);
%!assert(matpow(a,2), a^2);
%!assert(matpow(a,3), a^3);
%!assert(matpow(a,4), a^4);
%!assert(matpow(a,22), a^22);
%!assert(matpow(a,23), a^23);
```

Running these tests on `matpow` gives us confidence!

```
octave:1> test matpow
PASSES 8 out of 8 tests
```

We now 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 the n -th power of A ? Starting with A , we can 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 advance with fewest 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 $n = (b[i] * 2^i, i = 0..w-1)$, 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 $n = 0$. See also [Golub:1996], section 11.2.5, page 569.

The new algorithm looks like this:

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

  ## handle special case: n = 0
  if (n == 0)
    b = eye(size(a));
    return;
  endif

  ## general case: n >= 1
  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 if n is even, false 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 our code works as expected.

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 like m–file functions can. This means that there must be a way of mapping a function from Octave code 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 often installed as /usr/include/octave–VERSION/octave/oct.h. Now, our skeletal source file of the dynamic extension has the following shape:

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

DEFUN_DLD(func, args, nargsout,
           "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. In this case the third argument to DEFUN_DLD can be omitted.

Tip – 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 nargsout 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 from class `octave_value_list` (defined in OCT/src/oct–obj.h) are useful:

```
{
  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 users 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 end in _type. Some of the more important of these methods (defined in OCT/src/ov-base.h) are:

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

1.1.3.3 Low-level implementation

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

```
#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();

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

  Matrix p(args(0).matrix_value());
  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;
    }
    p = p * p;
  }

  return octave_value(b);
}

bool
is_even(int n)
{
```

```

return n % 2 == 0;
}

```

- The matrix that we want to raise to the n -th power is the *first* argument and is therefore accessed through `args(0)`. The method `rows` returns the number of rows in the matrix.
- The exponent n is the *second* argument to `matpow` in `args(1)`. Its integer value is obtained by using the `int_value()` method.
- The method `columns` returns the number of columns in a matrix. (At this stage, we are tacitly assuming that all parameters passed to `matpow` are valid, which means especially that the matrix is square.)
- 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 (1 in our case).
- 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`.
- Multiplication has been conveniently overloaded to work on matrices. Want to 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: grepping through) the source code and the desired elementary operation is implemented in most cases.

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  matpow.m  matpow.cc
$ mkoctfile matpow.cc

```

Good! But while developing, we would like to compile several times, run our test suite and finally remove all files that can be re-generated from source. Enter: `Makefile`.

```

# 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
    echo "test matpow.cc" | octave --silent

%.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 matpow.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: read on to see how this is handled! [CodaAdvanced](#) describes how to document our newly written function properly.

1.1.4.1 Input parameter checking

`matpow` will fail when called with an invalid set of parameters. In order to make it more robust, we examine the different parameters before the function is executed, making sure that they conform to our requirements.

Parameter tests should be done in the following order:

1. 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)

Octave supplies the user with all necessary type– and size–tests as class member functions. The type–tests (defined in `OCT/src/ov–base.h`) share the common prefix `is_`. Here are a few:

```
{
  public const bool is_real_scalar();
  public const bool is_complex_scalar();
  public const bool is_real_matrix();
  public const bool is_complex_matrix();
}
```

Using these functions is seldomly necessary, however. Instead, we ask Octave to convert an input argument to the required type. If there is a problem (indicated by the `error_state` (`OCT/src/error.h`)), we abort, e.g.

```
const int n = args(1).int_value();
if (error_state) return octave_value_list();
```

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

```
{
  public const int rows();
  public const int columns();
  public const int length();
  public const bool is_empty();
  public const bool is_square();
}
```

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 modify `matpow` to include parameter checking:

```
...
const int n = args(1).int_value();
Matrix p(args(0).matrix_value());
if (error_state) return retval;
if (n < 0) {
  error("n must be positive");
  return retval;
}
if (!p.is_square()) {
  error("A must be square");
  return retval;
}
if (n == 0)
  return octave_value(
    DiagMatrix(args(0).rows(), args(0).columns(), 1.0)
  );

Matrix b(p);
int np = n - 1;
...
```

Our final task is to update the tests and run them. Take the tests from `matpow.m` and append them to the C++ source inside a comment:

```
/*
```

```

%!shared a
%!test
%! a = [ 2.0, -3.0;
%!      -1.0,  1.0];
...
%!assert(matpow(a,23), a^23);
*/

```

If you prefer to separate your tests from your code, you can put them in a file called 'matpow' instead (remember to update the Makefile).

Now, also add the following tests:

```

%!fail("matpow([1,1; 1,1])", "usage:");
%!fail("matpow()", "usage:");
%!fail("matpow([1,1; 1 1], 2, 1)", "usage:");
%!fail("matpow([1 2 3; 4 5 6], 1)");
%!fail("matpow(a, -1)");

```

```

$ octave -q
octave:1> test matpow.cc
PASSES 13 out of 13 tests

```

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! 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:

```

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

@cindex index term

Help text in Texinfo format...
@end deftypefn

```

- `-*- texinfo -*-` tells the parser that the doc–string is in Texinfo format.
- `@deftypefn{class} ... @end deftypefn` encloses the whole doc–string, like a LaTeX? environment or a DocBook? element does.
- `@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 text goes into both (or, more precisely: all) output processors. Usually, neither `@ifinfo`, nor `@iftex` appear alone, but always in pairs, as the same information must be conveyed in every output format.

Here is a sample docstring for `matpow.cc`:

```
-*- 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
@tex
$n = 0$
@end tex
@end iftex
@ifinfo
n = 0
@end ifinfo
is explicitly allowed, returning a unit-matrix of the same size as
@var{a}.

Note: @code{matpow} duplicates part of the functionality of the built-in
exponentiation operator
@iftex
@tex
``$\wedge$''.
@end tex
@end iftex
@ifinfo
@code{^}.
@end ifinfo
```

```
Example:
@example
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

```

- `@iftex ... @end iftex` selects text for conditional inclusion. Only if the text is processed with TeX will the included section be processed.
- `@tex ... @end tex` wraps parts of the text that will be fed through TeX?
- `@ifdoc ... @end ifdoc` selects text for conditional inclusion. Only if the text is processed with an info–tool will the included section be processed.
- `@code{code sequence}` marks up a code sequence.
- `@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) looks like this:

```

- Loadable Function: B = matpow(A, N)
  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.

```

Octave–Forge: CodaTutorial

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.

[DaCodaAlFine – CodaTutorial](#)

[ScaryOctave | RecentChanges](#)

2 CodaAdvanced

DaCodaAlFine – CodaAdvanced

2.1 Advanced Extension Programming

2.1.1 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 [CodaTutorial](#)). 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 copy `DEFCONST`'s definition from `defun.h` into the mentioned 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. **clearing** a protected constant does not give raise to a warning either. Clearing protected constants re–installs their original values.

```
#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.");
    }
}
```

Octave–Forge: CodaTutorial

```
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,
          "Gravitation 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`, `DEFVAR` is defined in `defun.h` and not in `defun-dld.h`, so the programmer must introduce the macro herself.

```
// 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 `syntab.h`

```
// syntab.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_functions* 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 represents 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 );

#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)
    {
```

Octave–Forge: CodaTutorial

```
    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;
}
```

2.1.2 Documenting Constants And Variables

Having studied [CodaTutorial](#), only one new Texinfo function remains: `defvr`.

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

2.1.3 Using static variables

In functions, you can declare variables as being 'persistent' or 'static'. In other words, their status is not lost during successive calls of the function.

As an example, examine:

```
function static_test()
  persistent m = 0;
  m++;
  disp(m);
endfunction
```

Which behaves as follows:

```
octave:1> static_test
1
octave:2> static_test
2
octave:3> static_test
3
```

Can we do the same in an `oct-file`? Sure!

```
#include <octave/oct.h>
static int m = 0;
DEFUN_DLD (static_test, args, , "")
{
    m++;
    octave_stdout << m << std::endl;
    return octave_value();
}
```

2.1.4 Accessing global variables

Examine the following snippet:

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

DEFUN_DLD (global_test, args, , "")
{
    octave_value MyGlobal = get_global_value ("MyGlobal");

    if (error_state)
    {
        error ("foo: global symbol MyGlobal not found");
        return octave_value();
    }

    int val = MyGlobal.int_value ();

    if (error_state)
    {
        error ("foo: global symbol MyGlobal is not an integer");
        return octave_value();
    }

    octave_stdout << "MyGlobal: " << val << std::endl;

    return octave_value ();
}
```

The function `get_global_value` is used to access the global. Now test it in the interpreter:

No global is defined and we try to access it:

```
octave:1> global_test
error: get_global_by_name: unknown symbol `MyGlobal'
error: foo: global symbol MyGlobal not found
```

We define the global and successfully access it:

```
octave:1> global MyGlobal = 1;
octave:2> global_test
MyGlobal: 1
```

Notice how `test_global` uses `error_state` to see whether `MyGlobal` is of the correct type. This is good practise!

DaCodaAlFine – CodaAdvanced

ScaryOctave | RecentChanges

3 CodaStandalone

DaCodaAIFine – CodaStandalone

3.1 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 uses class Matrix from `liboctave.a` or `liboctave.so`.

```
#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;
}
```

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

```
$ mkcoctfile --link-stand-alone hello.cc -o hello
$ ./hello
Hello Octave world!
 11 12
 21 22

$
```

All done!

DaCodaAIFine – CodaStandalone

[ScaryOctave | RecentChanges](#)

4 CodaStruct

This is an example of how to access octave structs from C++ as found on the octave mailing list. Please improve to the standards of the rest of this document.

You can use the Octave_map class to get access to the elements of a structure. Below is an example of how you could add an element "a.b.c" with a value of -1 to a structure "a" defined in the Octave workspace.

Here is the source file structure.cc:

```
#include <octave/oct.h>
#include <octave/oct-map.h>
DEFUN_DLD (structure, args, , "quick and dirty demo")
{
  octave_value_list retval;
  Octave_map a (args(0).map_value());
  Octave_map ab (a.contents (a.seek ("b"))(0).map_value());
  ab.assign ("c", -1.0);
  a.assign ("b", ab);
  retval(0) = a;
  return retval;
}
```

Compile it as usual:

```
$ mkoctfile structure.cc
```

And try it from Octave:

```
octave:1> a.a = 0; a.b.a = 1; a.b.b = 2;
octave:2> a = structure (a)
a =
{
  a = 0
  b =
  {
    a = 1
    b = 2
    c = -1
  }
}
```

DaCodaAIFine
ScaryOctave | RecentChanges

5 CodaTypes

DaCodaAlFine – CodaTypes

Octave Type System Muthiah Annamalai<gnumuthu@users.sf.net> Licensed under **GNU Free Documentation License**..

5.1 Abstract

This document explains the type system employed within GNU Octave, and how one can extend it to get specific functionality. It will be of most use to people who want to extend GNU Octave for language bindings to other API's like GSL, Gd-Octave, Octave-Vtk, Octave-Gtk. It however doesnot deter the inquisitive from hacking into the guts of GNU Octave, and aims to help such efforts with the important understanding of the Octave Type system.

5.2 Introduction

GNU Octave is written in C++ and provides an interpreted environment for scientific computing, and supports extensions on itself, by the use of dynamically loaded modules, and shared libraries. In this we explain the GNU Octave type system, and how one can extend it to get specific functionality. We shall round up with an example of custom extension.

GNU Octave has a type system that helps in reflection, meaning the interpreter of Octave language, can query the type at runtime to find its type information. Also this makes the Octave language dynamic typed like Python, Perl or JavaScript?. What we are going to do is essentially see what this type system is used for, and how we can customize/create our own types specific to each application.

5.2.1 Requirements

It is assumed that you have GNU Octave compiled with dynamic loading support, shared libraries, and have the GNU Octave development header files in your system. As is the case generally, you will have all these; if not, please get your copy from www.octave.org. Any version Octave-2.1.50 or greater will suffice. Later the better.

Programmers are expected to have some introduction to C++, and GNU Octave language, as it would ease your learning curve. Also experience with the beast [GNU Octave] itself is welcome, if not mandatory.

If you would like to learn more about GNU Octave and how to use it, an excellent place to start is the Octave manual, by John W Eaton. It's available at <http://www.octave.org/> There is also an online manual available at <http://www.gnu.org/manual/octave/index.html>. Alternatively, you can try the built-in octave tutorial. To start the tutorial, type `[user @ host]$ info octave`.

There are a number of things you must have before you install GNU Octave, including GNU/Linux, with several dependencies. For a comprehensive list of libraries needed to compile octave extensions, just see the file `[user @ host]$cat /usr/bin/mkcoctfile` or run [user @ host]$ldd /usr/lib/liboctave*` to see the list of dependencies.`

5.2.2 Need for a Type System

As discussed in the introduction, GNU Octave needs a type system for supporting the following features.

Implementing Octave the interpreted language, that allows us to perform scientific computation in a matrix based environment, for which we must use an interpreter. Within one interpreter, we can allow for the user to create variables, get/set their properties, and change their values. A serious user of GNU Octave will have noticed that Octave interpreter has a built-in command called **typeinfo()** which can be used to get the typeid of the object's current value. Executing the following piece of code in GNU Octave will give you the type of x, currently which is 'matrix'.

```
eg: >x=[1 2 3]; typeinfo(x);
     ans="matrix"
     >x=1; typeinfo(x);
     ans="scalar"
     >x="hello"; typeinfo(x);
     ans="string"
```

Object Hierarchy is supported in GNU Octave with the help of this type system. You can inherit a type from the **octave_value**, the canonical holder, and implement its virtual functions, so that we have a brand new type to work with. However, what you can do with your type is limited by what Octave interpreter allows you to. Its a bit restrictive, but all the same it works only to keep Octave what it is, and not morph into other avatars. eg: we can inherit from the matrix type to create an image object, and define our own affine transforms for it, as its member functions, overload operators like +,-,=,%,* which perform intuitive functions on the object from the interpreter itself.

5.2.3 RTTI in GNU Octave

As many experienced C++ programmers will tell you, RTTI [Run Time Type identification] is a very powerful feature that will allow you to make the object hierarchies interoperate. For this we must implement, RTTI using Octave, or use the one provided by C++. **FIXME** Personally I was successful with the RTTI present in GNU Octave only, and C++ RTTI failed me, for reasons I couldnt understand.

GNU Octave object hierarchy resembles this **octave_value** : This is the base type [abstract] for all the octave objects, and simply acts as a abstract data type, that allows, us to have derived classes override the base class virtual member functions to achieve our desired functionality. As such octave_value has the virtual functions delegating the actions to the derived types, and makes life hard [you have to implement every virtual functions, if you dont want a segmentation fault!] .so we use **octave_base_value**, which fills in default stuff for the uninteresting functions. We generally override the int_value(), string_value(), and ulong_value(),print() kind of member functions, for our functionality. Assume you want to use an Image class that is derived from the matrix class, you could override the print() method to display just the type of image, width, height, color and size information. Similarly you could override the member functions like matrix_value() to return the properly needed matrix itself.

1. class octave_value
2. class octave_base_value : public octave_value
3. template <class ST> class octave_base_scalar : public octave_base_value
4. template <class MT> class octave_base_matrix : public octave_base_value

5. class octave_bool : public octave_base_scalar<bool>
6. class octave_magic_colon : public octave_base_value // A type to represent ':' as used for indexing.
7. class octave_complex : public octave_base_scalar<Complex> // Complex scalar values.
8. class octave_cell : public octave_base_matrix<Cell>
 1. class Cell : public ArrayN? <octave_value>
 2. template <class T> class ArrayN? : public Array<T>
9. class octave_scalar : public octave_base_scalar<double>
10. class octave_struct : public octave_base_value
11. class octave_list : public octave_base_value

The above hierarchy would throw light on the inheritance tree in octave classes. We find a pattern in the inheritance tree. All scalar types are derived from octave_base_scalar which is a template class, so we have octave_scalar, octave_complex derive directly from octave_base_scalar. The matrix types naturally derive from the octave_base_matrix which is also a template class. We will have complex matrix types derive from the octave_base_matrix. Similar patterns can be found in octave_list, octave_cell, octave_map, octave_struct which are left as an exercise to the reader.

5.2.4 Making Custom Octave Objects

This section will show you how to create a custom octave object called Box which is designed to hold a value of a pointer, and its associated typename. Basically, it is designed to prevent the user from changing the value of the pointer by wrapping the pointer value within a octave_value object so that it appears opaque to the user.

First we will have to declare the class structure in Box.cpp. Here we will inherit the class from the octave_base_value type, and override the print(), and ulong_value(). We will add a new functions called check_ptr() and save_ptr().

```

/*
 * This program is free software.
 * This code is a part of gd-octave
 * Code may be used or distributed under GPL.
 * (C) 2004 Muthiah Annamalai
 *
 * Source File : Box.cpp
 */
#include<iostream>
#include<octave/oct.h>
#include<octave/parse.h>
#include<octave/dynamic-ld.h>
#include<octave/oct-map.h>
#include<octave/oct-stream.h>
#include<octave/ov-base-scalar.h>
#include<vector>
#include<string>

using namespace std;

class Box: public octave_base_value
{
public:

    /* Well the constructor!! */

```

Octave-Forge: CodaTutorial

```
Box(void):octave_base_value()
{
    objptr=0;
    objtype=new string("None");
}

Box(octave_value box_val):octave_base_value()
{
    const octave_value& rep = box_val.get_rep();
    objptr=((const Box&) rep).get_ptr();
    objtype=((const Box&) rep).get_str();
}

Box(long int ptr,string type):octave_base_value()
{
    objptr=ptr;
    objtype=new string(type);
}

~Box(void)
{
    if(objtype)
        delete objtype;
}

/* Your Home Work! */
static void save_ptr(int ptr,string type)
{
    // Make a static list, and store the data.
}

static bool check_ptr(Box b)
{
    // Iterate through the static list, and see if the pointers match,
    // else, just report not found->>false
    // or found->>true
}

/* Query Interface */
long int get_ptr(void) const
{
    return this->objptr;
}

string* get_str(void) const
{
    return this->objtype;
}

void print (std::ostream& os, bool pr_as_read_syntax = false) const
{
    os<<"Ptr ="<<get_ptr()<<" Str="<<*get_str()<<endl;
}

bool is_constant (void) const { return true; }
bool is_defined (void) const { return true; }

private:
```

Octave–Forge: CodaTutorial

```
/* Object name */
string *objtype;

/* store object pointer */
long int objptr;

DECLARE_OCTAVE_ALLOCATOR

DECLARE_OV_TYPEID_FUNCTIONS_AND_DATA
};

DEFINE_OCTAVE_ALLOCATOR (Box);
DEFINE_OV_TYPEID_FUNCTIONS_AND_DATA (Box, "Box");

DEFUN_DLD(BoxMake, args, , "BoxMake(ptr, string)\n\
makes new instances of Box objects\
to fill into the world of octave")
{
    if(args.length() < 2){
        cout<<"usage: BoxMake(ptr, string)";
        return octave_value();
    }
    Box *b=new Box(args(0).int_value(),args(1).string_value());
    return octave_value(b);
}

DEFUN_DLD(BoxTest, args, , "BoxTest(Boxobject)\
tests if the Box object is present in the\
saved pointer list present")
{
    if(args.length() < 1
        || args(0).type_id()!=Box::static_type_id()){
        cout<<"usage: BoxTest(Boxobject)"<<endl;
        return octave_value(-1);
    }
    const octave_value& rep = args(0).get_rep();
    const Box& b = ((const Box &)rep);
    b.print(cout,0);
    return octave_value();
}

/*
gcc --shared -o BoxMake.oct Box.cpp -I /usr/include/octave-2.1.50/octave/ -I /usr/include/octave-2.1.50/octave/
*/
```

Data Members of the Box class are:

1: long int objptr ;/* to hold the value of the pointer */ 2: string *objtype ;/* description where this pointer points to */

Members Functions of the Box class are: 1: long int get_ptr(void) const; 2: string* get_str(void) const; which are property accessors for the private data members. Actually the *save_ptr()* & *check_ptr()* functions are left as an exercise, as otherwise it would interfere with the comprehension of our *octave_value* mutation, and customization.

First you can see that the *constructor* for Box class has two arguments, one the pointer value, and the next the descriptive string that accompanies it. The default constructor, is elegant to include, and we initialize 0 and "None" to objptr and objtype fields.

The overridden functions from octave_base_value are 1: void print (std::ostream& os, bool pr_as_read_syntax = false) const; which is invoked if octave interpreter wants to print our object. 2: bool is_constant (void) const { return true; } 3: bool is_defined (void) const { return true; } -- **FIXME** -- These functions are necessary, if we want the object returned from our dynamic functions to be assignable within the octave interpreter. i.e they enable the object to be proper *R-values*.

The macros DECLARE_OCTAVE_ALLOCATOR [<liboctave/oct-alloc.h>] expands to

```
#define DECLARE_OCTAVE_ALLOCATOR \
    public: \
        void *operator new (size_t size, void *p) \ { return ::operator new (size, p); } \
        DECLARE_OCTAVE_ALLOCATOR_PLACEMENT_DELETE \
        void *operator new (size_t size) \ { return allocator.alloc (size); } \
        void operator delete (void *p, size_t size) \ { allocator.free (p, size); } \
    private: \
        static octave_allocator allocator; /* End of Macro */
```

These functions simply create our object, and allocate some memory.

Another macro DECLARE_OV_TYPEID_FUNCTIONS_AND_DATA defined in [src/ov.h] expands to

```
#define DECLARE_OV_TYPEID_FUNCTIONS_AND_DATA \
    public: \
        int type_id (void) const \ { return t_id; } \
        std::string type_name (void) const \ { return t_name; } \
        std::string class_name (void) const \ { return c_name; } \
        static int static_type_id (void) \ { return t_id; } \
        static void register_type (void); \
    private: \
        static int t_id; \
        static const std::string t_name; \
        static const std::string c_name;
```

These private variables **t_id**, **t_name**, **c_name** are useful in RTTI within Octave libraries and inherited types for accessing the type id with type_id(), type name with type_name(), class name with class_name(). Also the accessor helper/property functions static_type_id() and register_type() provide a clean interface to these variable.

We have in the code section, a macro DEFINE_OV_TYPEID_FUNCTIONS_AND_DATA(type, type-name, class-name) which expands to

```
#define DEFINE_OV_TYPEID_FUNCTIONS_AND_DATA(t, n, c) \
    int t::t_id (-1); \
    const std::string t::t_name (n); \
    const std::string t::c_name (c); \
    void t::register_type (void) \
    { \
        t_id = octave_value_typeinfo::register_type (t::t_name, \
                                                    t::c_name, \
                                                    octave_value (new t ()));}
```

This sets the `t_id`, `t_name`, `c_name` private variables to the given values of type, type–name and class–name. Also it registers the octave type which we have created with the octave interpreter using the `register_type()` function, and gets the unique type id. This makes the derived Box class a unique Octave Value derived type.

5.2.5 Dynamic Loaded Functions

Two functions `BoxMake?` and `BoxTest?` are defined which do functions of creating and testing a box expressing how we do conversions between the Box type to `octave_value` and vice versa. We will defer discussion of test script to the next section.

`BoxMake?` function creates a new Box object after checking if we have the 2 arguments. It converts the arguments 1,2 to integer and string respectively using the `octave_value` member functions `int_value()` and `string_value()`. Then we call the `Box()` constructor, and the new object we created is returned to the user as an `octave_object`. Finally the Box object is cast into a `octave_value` to be returned to the octave interpreter. If the user tries to print our object, then our own `print()` function gets called, and we can display relevant data for our objects string representation. Thus we have converted the Box object to the `octave_value` , and send it to the Octave Interpreter.

`BoxTest?` function is invoked by the user with a Box object created from `BoxMake?` function. We check the argument object, if its a type of `BoxTest?` using the `args(0).type_id()` function comparing it with `Box::static_type_id()`, and on success we proceed to the most important step of type casting `octave_value` object to a Box object which it actually is. The code magic is in this line

```
const octave_value& rep = args(0).get_rep();
const Box& b = ((const Box &)rep);
```

where we force the object to return its 'rep' which is actually the opaque pointer present as a union object within `octave_value`, to a `rep` object which is the actual Box object. This is necessary because octave language uses pass by value, not reference. So every copy of the object passed to our `BoxTest?` function has a pointer to the original Box object created within the `BoxMake?` function. So after our `type_id()` checks we are rightfully qualified to force the typecasting of the `octave_value` to a `const Box` reference. From now on we are free to call the requisite Box member functions, as and when necessary. Now we may call the `check_ptr()` function and return a true/false value. Thus we have converted the `octave_value` to the Box type , on the object received from the Octave Interpreter.

Compiling the program must be easy if you have "mkoctfile" script in your path. Just use it like

```
$mkoctfile Box.cpp -o BoxMake.oct && ln -s BoxMake.oct BoxTest.oct
```

5.2.6 Testing with DEFUN_DLD

Actually we compile the Box.cpp file into a shared object, so that the symbols are exported into the library without linking to the octave library. This will be recognised by the octave interpreter [after you set the extension from .so to .oct, and call the function 'myFunc' present in the file 'myFunc.oct']. Now the dl library with its functions like `dlopen` and `dlsym` found in [/usr/include/dlfcn.h] are used by Octave to search and load the function from you .oct file, and invoke the same with the arguments. Marshalling the arguments is easy as all the DEFUN_DLD functions take the same number of arguments, and we can access the functions arguments from the 'args' parameter. This 'args' parameter is of the type `octave_value_list`, which is the list of all arguments the user has

passed to us. We [have to ! ;–)] do checks on the type and number of the arguments before we get down to using them. Again, our example falls short of these expectations.

We now make symbolic links of the files `BoxMake?.oct` and `BoxTest?.oct` to the same shared file, and then execute this script. In `–s BoxMake?.oct BoxTest?.oct`

```
%Octave Testing script.
#! /usr/bin/octave -q
% Create a Few Boxes:
%Stress test.
for i=1:100
    b=BoxMake(i, "one" )
    BoxTest(b, "check" )
    typeinfo(b)
end
%`chmod +x Box.m` before executing the program.
```

Do ``chmod +x Box.m`` and then run the program. More interesting results are obtained if you try the octave interpreter, and execute the lines one by one. Also try the `typeinfo(BoxMake?(123, "BoxType?"))`, to see the custom type we have christened **Box** from the `octave_value`.

5.2.7 See Also

see also : `octave–x.y.z/examples/make_int.cc` in octave sources, CODA from octave–forge distribution. wiki.octave.org, www.octave.org, <http://octave.sourceforge.net>.

5.3 Credits

1. *OCtave Mailinglists*: Thanks to the cheerful people at octave mailinglists, like David Bateman, Paul Kienzile, Etienne and JWE.
2. *Octave Forge*: Thanks to Paul for maintaining octave forge.
3. *CODA* : Where I learnt more Octave. `octave–x.y.z/examples/make_int.cc` : Thanks to JWE
4. *wiki.octave.org* : Thanks to Etienne.

5.4 GNU Free Documentation License

1. This article is Â© copyright Muthiah Annamalai 2004.
2. You can freely read, modify, distribute and edit this document under the terms of the **GNU Free Documentation License**.
3. GNU FDL Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111–1307 USA