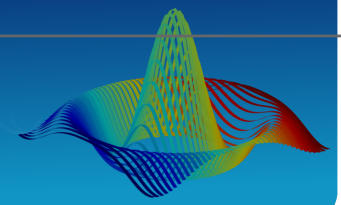
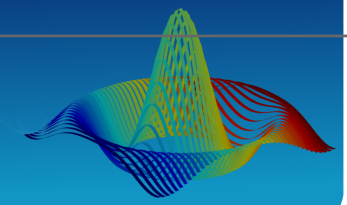


Octave's architecture

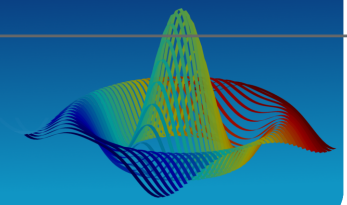
Jordi Gutiérrez Hermoso

July 22, 2012

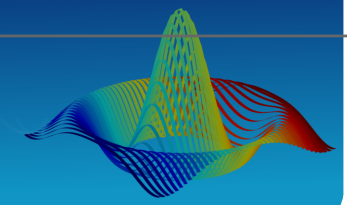
Outline



Outline

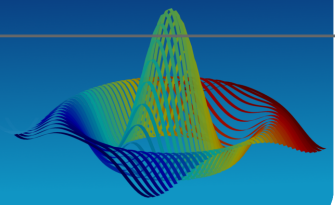


How is Octave?



A lot of this can be found in `etc/HACKING`

How is Octave?

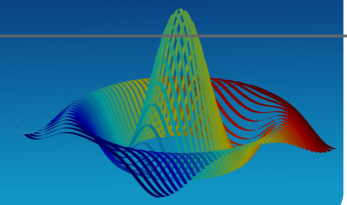


A lot of this can be found in `etc/HACKING`

ls -d */

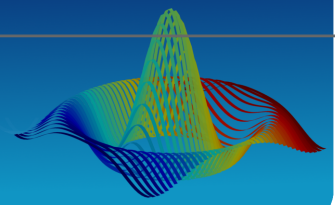
doc/	Documentation
examples/	oct-file examples (these are referenced in the manual)
gnulib/	Compatibility GNU utlity functions
m4/	Build system helpers
build-aux/	Build system helpers
test/	Test suite
etc/	Crufty old docs

How is Octave?



But those were not the most interesting bits

How is Octave?



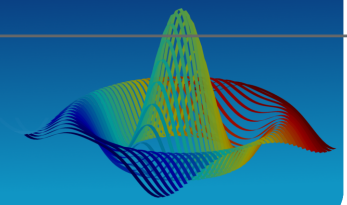
But those were not the most interesting bits

ls -d */

These are:

libcruft/	Old Fortran code
scripts/	m-script code (further categorised by subdirectories
liboctave/	C++ array and numerical classes
src/	Interpreter implementation, octave_value class hierarchy

Under src/



More things in src/:

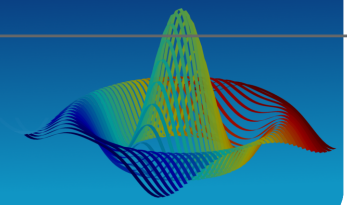
ls -d */

TEMPLATE-INST Some C++ template instantiations

OPERATORS Operators macro'ed for various types

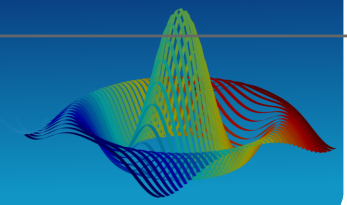
DLD-FUNCTIONS Oct files

libcruft/



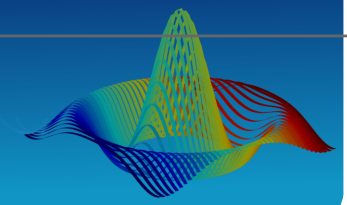
- Fortran code is in here

libcruft/



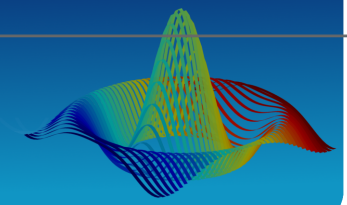
- Fortran code is in here
- Most of the time, not interesting

libcruft/

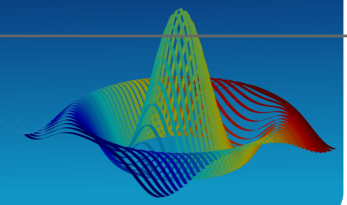


- Fortran code is in here
- Most of the time, not interesting
- Fortran code gets called from C++ with F77 CPP macro.

Outline

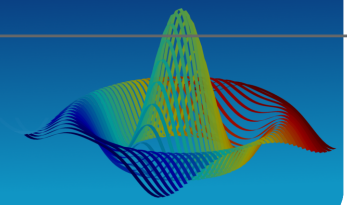


liboctave/



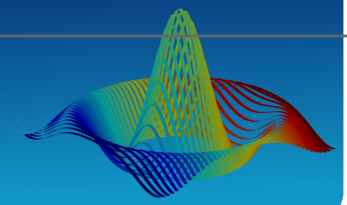
- Numerical classes

liboctave/



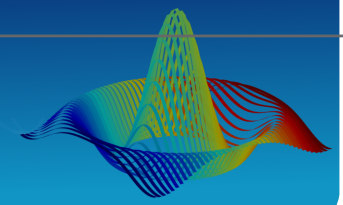
- Numerical classes
- Important base classes: Array, Sparse, idx_vector, octave_fftw

liboctave/



- Numerical classes
- Important base classes: Array, Sparse, idx_vector, octave_fftw
- Roughly following Fortran naming conventions for types: f - float, d - double, c - complex – these names are not wholly consistent.

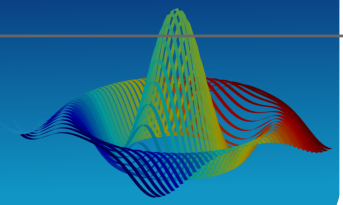
liboctave/



Array class

- Very general array class: used to build matrices, N-d arrays, cell arrays

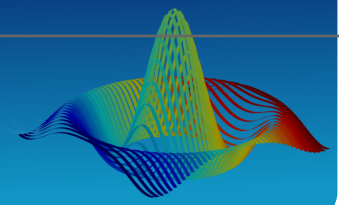
liboctave/



Array class

- Very general array class: used to build matrices, N-d arrays, cell arrays
- Templated class. With different template parameter can build all sorts of arrays.

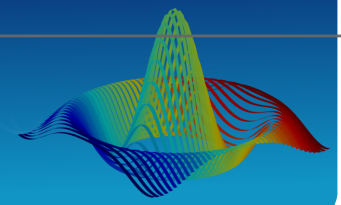
liboctave/



Array class

- Very general array class: used to build matrices, N-d arrays, cell arrays
- Templated class. With different template parameter can build all sorts of arrays.
- Data is stored in contiguous Fortran column-major order (this is later passed on to Fortran routines for some computations)

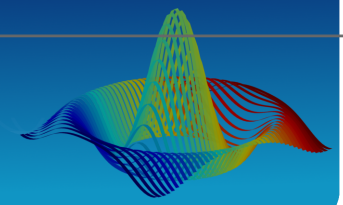
liboctave/



Array class

- Very general array class: used to build matrices, N-d arrays, cell arrays
- Templated class. With different template parameter can build all sorts of arrays.
- Data is stored in contiguous Fortran column-major order (this is later passed on to Fortran routines for some computations)
- *Not* used to build Sparse.

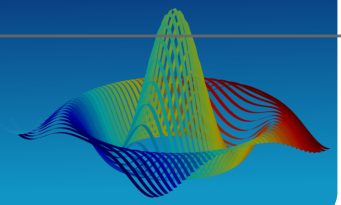
liboctave/



Sparse class

- Used to build double, float, and complex sparse matrices

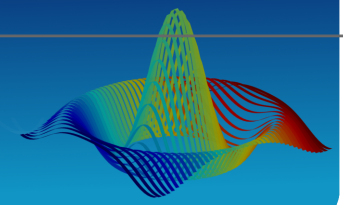
liboctave/



Sparse class

- Used to build double, float, and complex sparse matrices
- Data structure is *compressed column sparse* (Bateman & Adler article on ArXiv)

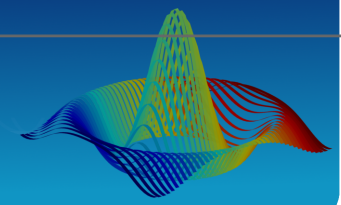
liboctave/



Sparse class

- Used to build double, float, and complex sparse matrices
- Data structure is *compressed column sparse* (Bateman & Adler article on ArXiv)
- Indexing here is full of dirty tricks...

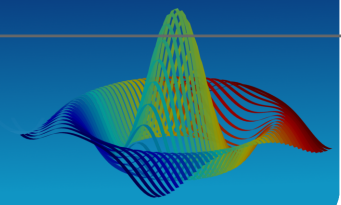
liboctave/



idx_vector

- Highly polymorphic class. Represents all sorts of indexing operations (slices, colons).

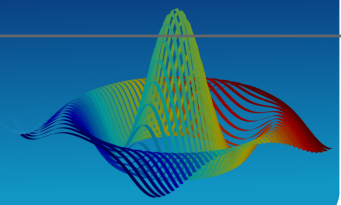
liboctave/



idx_vector

- Highly polymorphic class. Represents all sorts of indexing operations (slices, colons).
- Polymorphism is achieved through the rep pointer (this is a general design pattern in Octave)

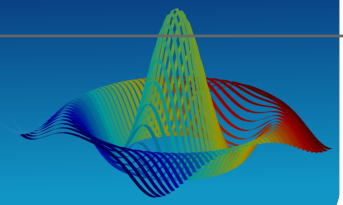
liboctave/



idx_vector

- Highly polymorphic class. Represents all sorts of indexing operations (slices, colons).
- Polymorphism is achieved through the rep pointer (this is a general design pattern in Octave)
- Highly optimised for many kinds of indexing

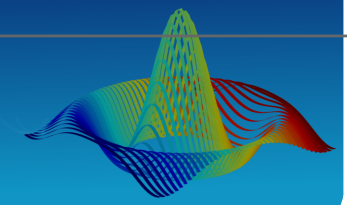
liboctave/



dim_vector

- Stores dimensions (always at least two, implicit trailing ones)

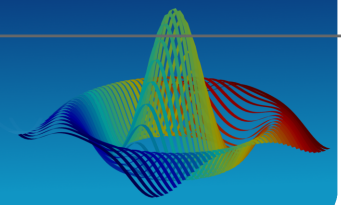
liboctave/



dim_vector

- Stores dimensions (always at least two, implicit trailing ones)
- Curious implementation detail: stores ndims and count in rep[-1] and rep[-2]

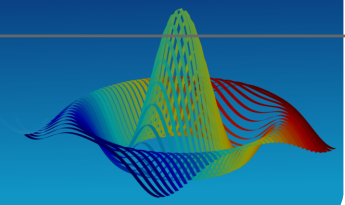
liboctave/



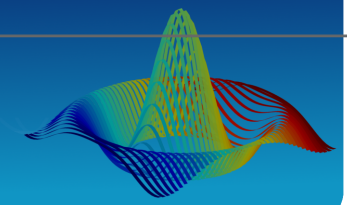
dim_vector

- Stores dimensions (always at least two, implicit trailing ones)
- Curious implementation detail: stores ndims and count in rep[-1] and rep[-2]
- Passing dimensions around without dim_vector is *deprecated*

Outline



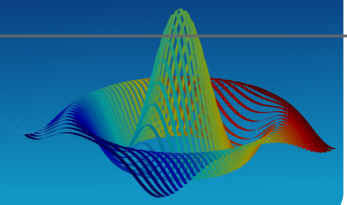
src/



interpreter

- lexer in lex.ll (flex)
- parser in oct-parse.yy (bison)
- Relatively simple code. Bison manual is sufficient for understanding how to hack these sources.

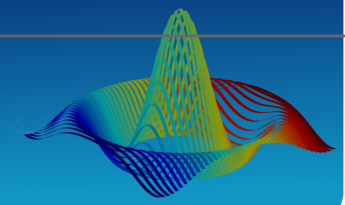
src/



octave_value class

- Polymorphic class (rep pointer again) that represents all user-visible Octave types

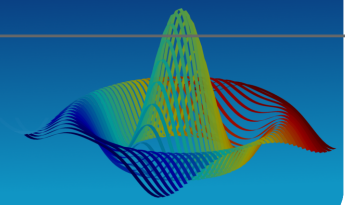
src/



octave_value class

- Polymorphic class (rep pointer again) that represents all user-visible Octave types
- From the interpreter, can see internal octave_value type with typeinfo() command.

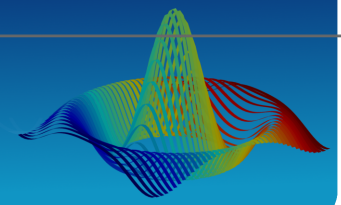
src/



octave_value class

- Polymorphic class (rep pointer again) that represents all user-visible Octave types
- From the interpreter, can see internal octave_value type with typeinfo() command.
- These classes are mostly for the interpreter. Built-in functions (DEFUN) are usually in here too.

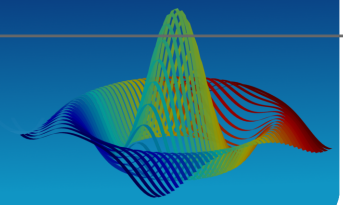
src/DLD-FUNCTIONS



oct files

- Mostly self-contained bits of code here

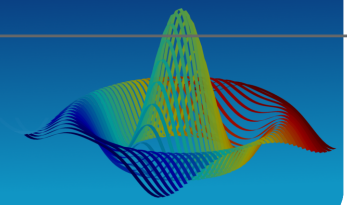
src/DLD-FUNCTIONS



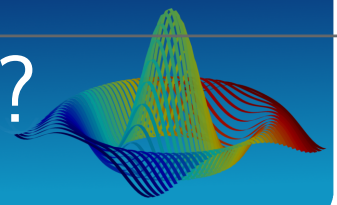
oct files

- Mostly self-contained bits of code here
- Filename tends to match functions contained in it

Outline

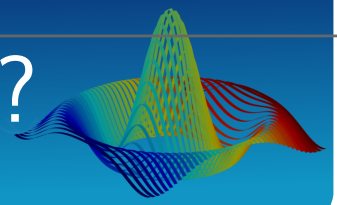


Where is that damn function(ality)...?



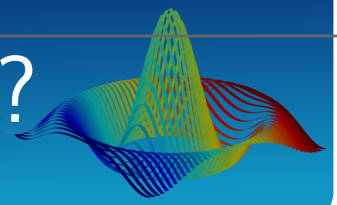
- Do you want to use a function? Use *which* command to see what kind of function it is. Can be...

Where is that damn function(ality)...?



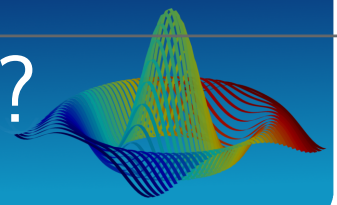
- Do you want to use a function? Use *which* command to see what kind of function it is. Can be...
  - m-file (easiest case)

Where is that damn function(ality)...?



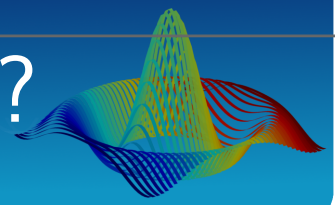
- Do you want to use a function? Use *which* command to see what kind of function it is. Can be...
  - m-file (easiest case)
  - Dynamically loaded function (second easiest, look under DLD-FUNCTIONS)

Where is that damn function(ality)...?



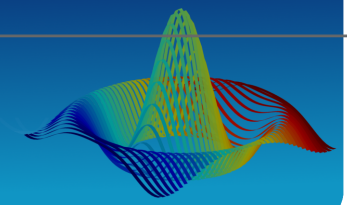
- Do you want to use a function? Use *which* command to see what kind of function it is. Can be...
  - m-file (easiest case)
  - Dynamically loaded function (second easiest, look under DLD-FUNCTIONS)
  - Built-in function (gotta go digging under src/ for it)

Where is that damn function(ality)...?



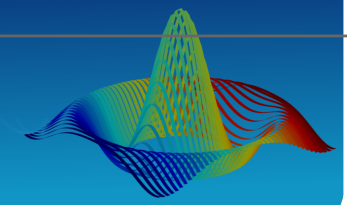
- Do you want to use a function? Use *which* command to see what kind of function it is. Can be...
  - m-file (easiest case)
  - Dynamically loaded function (second easiest, look under DLD-FUNCTIONS)
  - Built-in function (gotta go digging under src/ for it)
- More internal functionality? (indexing, parsing, error message...)

Finding internal code



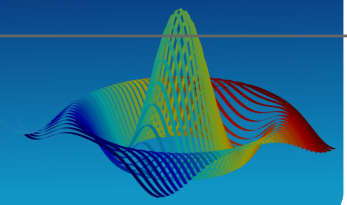
- Easiest way: step through the code with a debugger

Finding internal code



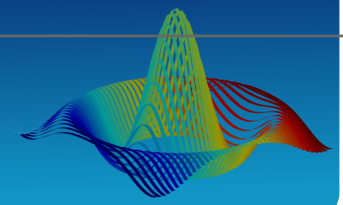
- Easiest way: step through the code with a debugger
- Common technique: produce output related to the functionality you want to change (perhaps an error message?)

Finding internal code



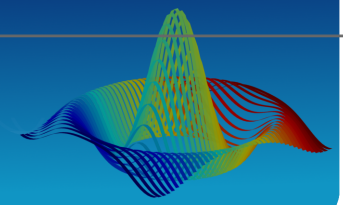
- Easiest way: step through the code with a debugger
- Common technique: produce output related to the functionality you want to change (perhaps an error message?)
  1. Grep the source code for that error message (think of the *static* part of the message)

Finding internal code



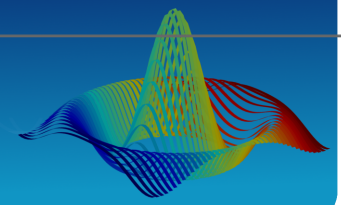
- Easiest way: step through the code with a debugger
- Common technique: produce output related to the functionality you want to change (perhaps an error message?)
  1. Grep the source code for that error message (think of the *static* part of the message)
  2. Put a breakpoint there

Finding internal code



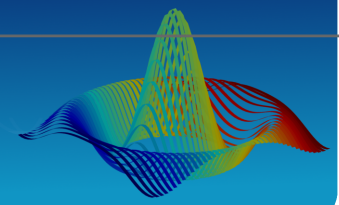
- Easiest way: step through the code with a debugger
- Common technique: produce output related to the functionality you want to change (perhaps an error message?)
  1. Grep the source code for that error message (think of the *static* part of the message)
  2. Put a breakpoint there
  3. Run Octave

Finding internal code



- Easiest way: step through the code with a debugger
- Common technique: produce output related to the functionality you want to change (perhaps an error message?)
  1. Grep the source code for that error message (think of the *static* part of the message)
  2. Put a breakpoint there
  3. Run Octave
  4. Look up the call stack, see if you can find the code

Finding internal code



- Easiest way: step through the code with a debugger
- Common technique: produce output related to the functionality you want to change (perhaps an error message?)
  1. Grep the source code for that error message (think of the *static* part of the message)
  2. Put a breakpoint there
  3. Run Octave
  4. Look up the call stack, see if you can find the code
- Less common technique: think deeply about Octave's design and where the code might be (this gets easier later on).