

Lisplab manual

A mathematics library for Common Lisp

Joern Inge Vestgaarden

This manual is for Lisplab version 0.2, updated 10. May 2010.
Copyright © 2009 Joern Inge Vestgaarden

Table of Contents

1	Introduction	1
2	Getting started	2
2.1	Dependencies	2
2.2	Portability	2
2.3	Installing	2
2.4	Naming conventions	3
2.5	Status - past and future	3
2.6	Bugs and limitations	3
3	Tutorial	5
3.1	Starting	5
3.2	Matrix classes	5
3.3	Matrix construction	5
3.4	Matrix element reference	6
3.5	Elementwise operators and functions	6
3.6	Linear algebra	7
3.7	Matrix IO	8
3.8	Matrices without store	8
3.9	Matrix map	8
3.10	Ordinary functions	9
3.11	Special functions	9
3.12	Infix notation	9
4	Structure	10
4.1	Design principles	10
4.2	Package structure	10
4.3	The four levels, 0 – 3.	10
4.4	Matrix class hierarchy	11
4.4.1	The structure	11
4.4.2	The element type	11
4.4.3	The implementation	11
5	Discussion	13
5.1	The foreign function interfaces	13
5.2	Parallel execution	13
5.3	Symbolic calculations	13
5.4	Missing features	13

1 Introduction

Lisplab is a mathematics/matrix library in Common Lisp. It is placed under the GNU General Public License (GPL) and offers an easy-to-use and rich programming framework for mathematics, including Matlab-like matrix handling, linear algebra, Fast Fourier Transform, special functions, Runge-Kutta solver, infix notation, and interfaces to BLAS, LAPACK, FFTW, SLATEC and QUADPACK.

The name Lisplab is inspired by Matlab and Lisplab offers much of the same kind of programming-style as Matlab, with high level manipulation of matrices, but contrary to Matlab, Lisplab benefits from being a part of Common Lisp. Hence, you have lexical scope, dynamic scope, macros, first class functions, iterations, CLOS, fast execution, and working in an free general purpose programming language. And best of all: you can enjoy you favorite data-types in addition to the matrices: functions, hash tables, structures, classes, arbitrary precision integers, rationals, and lists.

Lisplab is not unique in building Matlab-like syntax on top of Common Lisp. Other similar frameworks are Matlisp, Femlisp, NLISP and GSSL. Lisplab itself was started as a branch of Matlisp, but there is now only little of the original code left.

2 Getting started

2.1 Dependencies

In order to enjoy the full power of Lisplab you must install some foreign libraries. These are

- BLAS – Basic Linear Algebra Subprograms. Preferably the Atlas build.
- LAPACK – Matrix library. Preferably the Atlas build.
- FFTW – The fastest Fast Fourier Transform available.

2.2 Portability

Lisplab has been developed with SBCL, SLIME and ASDF on Linux, and there are yet unnecessary bindings to these platforms.

- Some of the optimized lisp code uses the SBCL macro `truly-the`.
- The FFTW FFI works only for SBCL.
- The Matlisp FFI should in theory be portable to other lisps and Windows, but it has not yet been tested.
- The `*READ-DEFAULT-FLOAT-FORMAT*` must be `double-float` when compiling Slatec. This is a minor problem.

Except from this, Lisplab should be self-contained and not depend on any other projects.

2.3 Installing

Lisplab is ASDF installable, but before you come so far you need to specify the location of the foreign libraries. You specify these in three special variables,

- `*lisplab-libblas-path*`
- `*lisplab-liblapack-path*`
- `*lisplab-libfftw-path*`

that live their lives in the Common-Lisp-User package. You can either assign them on the top-level, in you Common Lisp installation file, in `start.lisp`, or probably the easiest: directly in `lisplab.asd`.

ASDF sub-systems:

- *Lisplab* – the full Lisplab installation.
- *Lisplab-base* – the part of Lisplab without external dependencies.
- *Lisplab-matlisp* – FFIs to BLAS and LAPACK. These are modified version from Matlisp.
- *Lisplab-fftw* – FFI to FFTW for Fast Fourier Transform.
- *Slatec* – special functions, generated from Fortran by f2cl. Originally made for Maxima.
- *Quadpack* – integration routines, generated from Fortran by f2cl.

The simplest way to test Lisplab is to load `start.lisp`. If you have problems loading, first look at `start.lisp` and see if you can hack it. Then look at `lisplab.asd`.

To install BLAS, LAPACK, and FFTW, if you are too lazy to do a custom build, and is lucky enough to administer a Debian or Ubuntu machine, you typically write

```
# aptitude install libatlas3gf-base
# aptitude install libfftw3-3
```

2.4 Naming conventions

- The matrix classes and constructors follow the naming convention from BLAS where you give names based on element type and matrix structure. The most used types are *f* - float, *d* - double, *c* - complex float, *z* - complex double float, while for matrix structure *ge* - general, *di* - diagonal, and many more. So *matrix-dge* is a general matrix with double float elements, while *matrix-zge* is a general matrix with complex double float elements.
- The generic functions of the basic algebra start with a dot: `.+`, `.-`, `.*`, `./`, `.^`, `.sin`, `.cos`, `.tan`, `.besj`, `.re`, etc. On numbers these functions work as the non-dotted Common Lisp functions and on matrices they work elementwise.
- Linear algebra functions tend to start with *m*: `m*`, `minv`, `mmax`, `mtp`, etc., but this conventions is not strictly followed.
- The naming convention of files follow the layered structure of Lisplab, with level0 to level3.
- See [Chapter 4 \[Structure\]](#), page 10, for more about the naming conventions of matrix classes.

2.5 Status - past and future

Lisplab has been developed for physics simulations and data handling. Lisplab started as a refactoring of Matlisp, but most of the code has been replaced and a lot new code has been written. The only Matlisp code that is kept is the interfaces to BLAS and LAPACK.

Some large extensions that could be fun to do:

- Parallel computation, e.g. using MPI.
- More native linear algebra routines, e.g. eigenvalue computation.
- New non-matrix algebra items, e.g. quaternions, polynomes or arbitrary precision floats.
- Symbolic manipulation. Could make something like *ginac* in `c++`.
- New matrix optimization for new usage, e.g. integer matrices for image processing or cryptography.
- Interface to new foreign libraries, e.g. GSL.
- More special functions.

An of course a lot more linear algebra and matrix stuff.

2.6 Bugs and limitations

The purpose of Lisplab is to be a platform for mathematical computations. From this perspective it is clear that it will never be complete. Also, since there is no spec it is not obvious what is a bug and what is not!

Hence, the list in this section must be read as non-systematic gathering of problem features.

- Lisplab runs only on SBCL. Lisplab is mainly ANSI Common Lisp, so just minor changes in the build-system should make it run on other lisps, but the problem is that it will most probably be slow. It should be fast on CMUCL, though.
- Lacks a formal spec.
- Poorly tested.
- Lacks error checks (but these should not be made before a spec!)

3 Tutorial

3.1 Starting

On SBCL, make sure that `asdf:*central-registry*` contains `lisplab.asd` and type

```
CL-USER> (let ((*read-default-float-format* 'double-float))
           (require :lisplab))
CL-USER> (in-package :ll)
```

The `*read-default-float-format*` is only needed when requiring for first time. The main package of Lisplab is `lisplab`, with nickname `ll`, and the package `lisplab-user`, with nickname `ll-user`.

3.2 Matrix classes

The matrix classes hierarchy has three lines of inheritance: on structure, on element type and on implementation. This hierarchy is similar to the stream example in *Object-Oriented Programming in Common Lisp*, by Sonya E. Keene.

Objects of the most important classes can be created by read macros. For double-float matrices, `matrix-dge`,

```
LL-USER> #md((1 2) (3 4))
#md(( 1.000      2.000   )
     ( 3.000      4.000   ))
```

For complex double-float matrices, `matrix-zge`,

```
LL-USER> #mz((1 2) (#c(1 2) #c(3 4)))
#mz((#c( 1.0      0.0   ) #c( 2.0      0.0   ))
     (#c( 1.0      2.0   ) #c( 3.0      4.0   )))
```

Untyped matrices, `matrix-ge`,

```
LL-USER> #mm(('a 'b) ('c 'd))
#mm((A B)
     (C D))
```

There are more matrix classes, but these are not exported. Even more matrix classes can be created dynamically, but these capabilities are not yet fully in use. See [Chapter 4 \[Structure\]](#), page 10.

3.3 Matrix construction

There are many ways to create a matrix, such as

```
LL-USER> (dnew 0 2 2)
#md(( 0.000      0.000   )
     ( 0.000      0.000   ))
```

```
LL-USER> (drow 1 2)
#md(( 1.000      2.000   ))
```

```
LL-USER> (dcol 1 2)
```

```
#md(( 1.000  )
     ( 2.000  ))
```

```
LL-USER> (dmat '((1 2) (3 4)))
#md(( 1.000  2.000  )
     ( 3.000  4.000  ))
```

Similarly, there are `znew`, `zcol`, `zrow`, and `zmat` for complex double float matrices and `mnew`, `mcol`, `mrow`, and `mmat` for any matrices. The latter take matrix class as first argument.

Often you want to create a matrix of the same type as a input matrix. Then you can use `mcreate` and `mcreate*` for extended syntax. They are useful when creating methods that should operate on many matrix types.

To create matrices from something else, use `convert`

```
LL-USER> (convert '((1 2) (3 4)) '(:d :ge :any))
#md(( 1.000  2.000  )
     ( 3.000  4.000  ))
```

`Convert` also converts between matrix types. If the matrix contents cannot be converted directly (e.g., conversion from complex to real), use `copy-contents` instead.

Other matrix constructors are `mcreate`, `mcreate*`, `drange`, `dgrid`, `fmat`, `funmat`. The most fundamental constructor is `make-matrix-instance`.

3.4 Matrix element reference

Lisplab matrices are zero-based and in column major order. Matrix reference is with the settable generic function `mref`

```
LL-USER> (mref #md((1 2) (3 4)) 0 1)
2.0
```

Matrices can also list their elements as vectors (similar to `row-major-aref` for arrays). Vector access is with the generic function `vref`

```
LL-USER> (vref #md((1 2) (3 4)) 1)
3.0
```

which is also settable.

3.5 Elementwise operators and functions

Lisplab introduces general mathematical operators `.+`, `.-`, `.*`, `./`, and `.^`. These are generalization of `+`, `-`, `*`, `/`, and `^`. For numbers they work the same,

```
LL-USER> (.+ 1 2)
3
```

But the lisplab functions have a much wider functionality since they are based on the binary generic functions `.add`, `.sub`, `.mul`, `.div`, and `.expt`.

On matrices the all functions starting with a dot works elementwise. Hence

```
LL-USER> (.sin (drow 0 1))
#md(( 0.000  0.8415  ))
```

The dotted operators ignores the internal structure of the matrices

```
LL-USER> (let ((a (drow 0 1))
              (b (dcol 0.1 0.2)))
          (.+ a (. * b 2)))
#md((0.2000      1.400      ))
```

The output need not have same type as the input

```
LL-USER> (.asin #md((1 2)))
#mz((#c( 1.6      0.0      ) #c( 1.6      1.3      )))
```

The dotted functions and operators are optimized and very fast for the double-float matrices, but slow for the matrices with arbitrary element-types.

```
LL-USER> (.+ #mm((1/2 3/2)) 1)
#mm((3/2 5/2))
```

3.6 Linear algebra

The linear algebra functions feel and maybe also change the matrix structure. The linear algebra functions often start with m , although this conventions is not strictly followed. The number of linear algebra functions is small compared with LAPACK, but you will find matrix multiplication, matrix inversion, transpose, conjugate transpose, LU-decomposition and eigenvalues.

Matrix multiplication

```
LL-USER> (let ((a #md((1 0) (0 -1)))
              (b #md((0.1) (0.2))))
          (m* a b))
#md((0.1000      )
     (-.2000      ))
```

Matrix inversion

```
LL-USER> (minv #md((1 2)
                  (-2 1)))
#md((0.2000      -.4000      )
     (0.4000      0.2000      ))
```

Transpose

```
LL-USER> (mtp #md((1 2)
                  (-2 1)))
#md(( 1.000      -2.000      )
     ( 2.000      1.000      ))
```

Conjugate transpose

```
LL-USER> (mct #mz((1 (* 2 %i)) (-2 1)))
#mz((#c( 1.0      -0.0      ) #c(-2.0      -0.0      ))
     (#c( 0.0      -2.0      ) #c( 1.0      -0.0      )))
```

Eigenvalues

```
LL-USER> (eigenvalues #md((1 2) (0 0.5)))
#md(( 1.000      )
     (0.5000      ))
```

Eigenvectors

```
LL-USER> (eigenvectors #md((1 2) (0 0.5)))
(
#md(( 1.000   )
     (0.5000  ))

#md(( 1.000   -.9701   )
     ( 0.000   0.2425  )))
```

Some of the linear algebra functions also work for general element matrices

```
LL-USER> (let ((a #mm((1 0) (0 -1)))
              (b #mm((1/2) (2/3))))
          (m* a b))
#mm((1/2)
     (-2/3))

LL-USER> (minv #mm((1 2)(-2 1)))
#mm((1/5 -2/5)
     (2/5 1/5))
```

3.7 Matrix IO

Delimited files are read with `dload`. Delimited files are written with `dloadwrite`. You can also write matrices as images with `pgmwrite` and `pswrite`, writing portable graymap and postscript respectively.

3.8 Matrices without store

The class `function-matrix` implements matrices with functions and has no store, but where the elements are given by a function

```
LL-USER> (funmat '(2 2) (lambda (i j)
                        (if (= i j)
                            1
                            0)))

#<FUNCTION-MATRIX 2x2
1 0
0 1
{ACOF489}>
```

The similar macro `fmat` creates functions of any type by a function.

Function matrices are also used to view a part or restructured other matrix with `view-matrix`, `view-col`, or `view-row`.

3.9 Matrix map

There are two methods for mapping of matrices: `mmap` and `mmap-into`. For example

```
LL-USER> (mmap 'matrix-dge #'re #mz((1 %i) (-%i 2)))
#md(( 1.000   0.000   )
     ( 0.000   2.000   ))
```

The output matrix takes the structure from the first arguments, but ignores in general matrix structure. If first argument `t` output type is same as type of first matrix argument.

3.10 Ordinary functions

`.sin`, `.cos`, `.sin`, `.tan`, `.sinh`, `.cosh`, `.sinh`, `.tanh`, `.asin`, `.acos`, `.asin`, `.atan`, `.asinh`, `.acosh`, `.asinh`, `.atanh`, `.conj`, `.re`, `.im`, `.abs`, `.sqrt`, `.exp`.

3.11 Special functions

`.besj`, `.besy`, `.besi`, `.besk`, `.besh1`, `.besh2`, `.ai`, `.gamma`.

3.12 Infix notation

Infix input is with the macro `w/infix`

```
LL-USER> (w/infix
           (let ((x 3))
             (1 .+ 2 .* x)))
```

7

The `w/infix` is compatible with the Lisp semantics. The infix math also works with the functions `+`, `-`, `*`, `/` and `^`.

4 Structure

4.1 Design principles

High level principles

- Lisplab will be a *homogeneous platform* for mathematics.
- Lisplab is free software.
- User applications should need to stay only in Common Lisp. (There should be no need for optimized math in FFIs or special languages like Maxima)
- Lisplab will steal as much code as possible from as many as possible.

General design principles

- Every common mathematical operator and function is represented by a *CLOS generic function*.
- Modular structure (Inspired by GSL).
- Trust the Lisp virtual machine (Avoid use of FFIs and destructive functions).
- Avoid mathematical algorithms in macros.
- Error checks is primarily caller's responsibility.

Design principles for the matrix code

- Layered structure where dependencies are primarily to the layer below – not vertical within the layer.

4.2 Package structure

So far, there is only one main package, called, you might guess it: *lisplab*. Except from that there are only a few special packages for generated code and FFIs: *Slatec*, *Blas*, and *FFTW*. For test code an applications you have the package *lisplab-user*.

4.3 The four levels, 0 – 3.

The Lisplab matrix and linear algebra code has a layered structure with four levels, 0 – 3, where

- **Level 0** is matrix independent and contains generic functions for mathematics. In this level only specialization for numbers and non-matrix objects.
- **Level 1** defines matrices and implements `mref`, `vref`, `cols`, `rows`, `size` and `make-matrix-instance`.
- **Level 2** Implements level 0 for matrices and defines core functionality related to matrices such as matrix constructors `dnew`, `dcol`, etc. and other matrix helper functions, such as `mmax`, `mmin`, `circ-shift`, etc. Optimizations are mainly in level 2.
- **Level 3** is everything else that uses matrices, including linear algebra, FFTs, solvers, etc.

The levels are unequal in size: level 0 is potentially large, level 1 is small, level 2 and 3 are large.

The intention with the structure are the following

- To minimize the work needed to add new matrices implementations. Actually, you need only to implement level 1, provided you inherit from matrix base. Otherwise you need also to implement level 2.
- To encourage code reuse. For example, if you add a new type to the dotted algebra, such as polynomials, symbolic expressions, or arbitrary precision floats, you can immediately perform matrix operations on them without write one single extra line of code.
- To manage optimizations and special implementations by overloading for specialized matrix types, mainly at level 2, but also at level 3.
- To make foreign libraries available in Common Lisp.
- To create a closed, constant and extensible system of operations.

The filenames often also denote the levels. If a filename does not denote the level, it is most probably level 3, or outside the level system (non matrix code).

4.4 Matrix class hierarchy

All matrices are subclasses of `matrix-base`, and as far as possible the generic functions specialize on one or many of its subclasses.

The matrix class hierarchy has three independent lines of inheritance

- On structure
- On element type
- On implementation

The structure is inspired by the stream example in Object-Oriented Programming in Common Lisp, by Sonya E. Keene.

4.4.1 The structure

The structure tells what kind of symmetries or other special properties the matrix has. Two matrix classes

- `matrix-structure-general`, where the matrix has no known symmetries.
- `matrix-structure-diagonal`, where only diagonal elements are non-zero.
- Other types to come.

4.4.2 The element type

The element type classes has no other purpose than to be represents a Common Lisp types.

- `matrix-element-base`, represents `t`
- `matrix-element-double-float`, represents `double-float`
- `matrix-element-complex-double-float`, represents `complex double-float`

4.4.3 The implementation

Since Lisplab has many competing implementation of the same generic functions, the implementation class structure tells which one too choose. There are currently four classes in a straight line of inheritance

- `matrix-implementation-base`

- `matrix-implementation-lisp`, use native Common Lisp if possible.
- `matrix-implementation-blas`, use foreign libraries if possible.

This standard method dispatch ensures foreign library methods are chosen before native lisp. *It is the responsibility of the methods to call next method if the library is not loaded.*

5 Discussion

5.1 The foreign function interfaces

The foreign function interface comes from Matlisp and is mainly on level 3. The elements of Lisplab typed matrices (double-float and complex double-float) are stored as 1D simple arrays, and most Lisps will then have a SAP inside the array pointer which is binary compatible with Fortran. This adds some overhead to matrix element references, but simplifies the memory management considerably compared to a bare pointer type. Also, it is important to avoid pointers from being moved by the garbage collector. In Matlisp, this was handled by stopping the garbage collector, but a more smoother and better way would be to just pin the pointers in action.

Lisplab has an extra layer with Fortran compatibility above the ordinary C FFI. This layer is mainly unchanged compared to Matlisp, but only the SBCL FFI is distributed with Lisplab.

The FFI for FFTW is a mock-up for SBCL, and it only is for standard complex transforms and inverse transforms. Of course, a lot more can be done, but this simple version works quite OK. In long term the SBCL FFI should be replaced with a general one.

5.2 Parallel execution

To request usage of more threads, call `init-threads`. Currently only the calls to FFTW will react to this.

5.3 Symbolic calculations

There is draft code for symbolic calculations, but it is not yet included in the build. The symbolic code need not know anything about the rest of Lisplab, except for the generic methods of the dotted algebra (The level 0).

5.4 Missing features

- There is no way to iterate through the elements of a general matrix in a fast way. (The map functions are currently the only thing, but these are structure agnostic and also not fast.) There should maybe be an macro `w/matrix`.
- There should be linear algebra primitives, like row exchange, in level 2, so that level 3 can be made entirely without knowledge about internal structure of matrices. (Structure similar to blas - lapack)
- Integer matrices.
- Vectorized execution of operations.
- Numerical integration.
- Symbolic math. Should be separate module, only with knowledge of the dotted algebra generic functions.
- The dotted algebra should also work on functions so the one could write `(.+ (lambda (x) (+ x 1)) 3)` and get a new functions as result. It might even be possible to make beautiful optimizations this way.