Numeric.LinearAlgebra.Data

Copyright	(c) Alberto Ruiz 2015
License	BSD3
Maintainer	Alberto Ruiz
Stability	provisional
Safe Haskell	None
Language	Haskell98

This module provides functions for creation and manipulation of vectors and matrices, IO, and other utilities.

Contents

Synopsis

Elements

```
type R = Double
```
Source

```
type C = Complex Double
```
Source

```
type I = CInt
```
Source

```
type Z = Int64
```
Source

```
type (./.) x n = Mod n x    infixr 5
```
Source

Vector

1D arrays are storable vectors directly reexported from the vector package.

```
fromList :: Storable a => [a] -> Vector a
```
#

O(n) Convert a list to a vector

```
toList :: Storable a => Vector a -> [a]
```
Source

```
(|>) :: Storable a => Int -> [a] -> Vector a    infixl 9
```
Source

Create a vector from a list of elements and explicit dimension. The input list is truncated if it is too long, so it may safely be used, for instance, with infinite lists.

```
>>> 5 |> [1..]
[1.0,2.0,3.0,4.0,5.0]
it :: (Enum a, Num a, Foreign.Storable.Storable a) => Vector a
```

```
vector :: [R] -> Vector R
```
Source

Create a real vector.

```
>>> vector [1..5]
[1.0,2.0,3.0,4.0,5.0]
it :: Vector R
```

```
range :: Int -> Vector I
```
Source

```
>>> range 5
[0,1,2,3,4]
it :: Vector I
```

```
idxs :: [Int] -> Vector I
```
Source

Create a vector of indexes, useful for matrix extraction using '(??)'

Matrix

The main data type of hmatrix is a 2D dense array defined on top of a storable vector. The internal representation is suitable for direct interface with standard numeric libraries.

```
(><) :: Storable a => Int -> Int -> [a] -> Matrix a
```
Source

Create a matrix from a list of elements

```
>>> (2><3) [2, 4, 7+2*iC,   -3, 11, 0]
(2><3)
 [       2.0 :+ 0.0,   4.0 :+ 0.0, 7.0 :+ 2.0
 , (-3.0) :+ (-0.0), 11.0 :+ 0.0, 0.0 :+ 0.0 ]
```

The input list is explicitly truncated, so that it can safely be used with lists that are too long (like infinite lists).

```
>>> (2><3)[1..]
(2><3)
 [ 1.0, 2.0, 3.0
 , 4.0, 5.0, 6.0 ]
```

This is the format produced by the instances of Show (Matrix a), which can also be used for input.

```
matrix
```
Source

`:: Int`	number of columns
`-> [R]`	elements in row order
`-> Matrix R`	

Create a real matrix.

```
>>> matrix 5 [1..15]
(3><5)
 [  1.0,  2.0,  3.0,  4.0,  5.0
 ,  6.0,  7.0,  8.0,  9.0, 10.0
 , 11.0, 12.0, 13.0, 14.0, 15.0 ]
```

```
tr :: Transposable m mt => m -> mt
```
Source

conjugate transpose

```
tr' :: Transposable m mt => m -> mt
```
Source

transpose

Dimensions

```
size :: Container c t => c t -> IndexOf c
```
Source

```
>>> size $ vector [1..10]
10
>>> size $ (2><5)[1..10::Double]
(2,5)
```

Synopsis

```
rows :: Matrix t -> Int
```
Source

```
cols :: Matrix t -> Int
```
Source

Conversion from/to lists

```
fromLists :: Element t => [[t]] -> Matrix t
```
Source

Creates a `Matrix` from a list of lists (considered as rows).

```
>>> fromLists [[1,2],[3,4],[5,6]]
(3><2)
 [ 1.0, 2.0
 , 3.0, 4.0
 , 5.0, 6.0 ]
```

```
toLists :: Element t => Matrix t -> [[t]]
```
Source

the inverse of `fromLists`

```
row :: [Double] -> Matrix Double
```
Source

create a single row real matrix from a list

```
>>> row [2,3,1,8]
(1><4)
 [ 2.0, 3.0, 1.0, 8.0 ]
```

```
col :: [Double] -> Matrix Double
```
Source

create a single column real matrix from a list

```
>>> col [7,-2,4]
(3><1)
 [  7.0
 , -2.0
 ,  4.0 ]
```

Conversions vector/matrix

```
flatten :: Element t => Matrix t -> Vector t
```
Source

Creates a vector by concatenation of rows. If the matrix is ColumnMajor, this operation requires a transpose.

```
>>> flatten (ident 3)
[1.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,1.0]
it :: (Num t, Element t) => Vector t
```

```
reshape :: Storable t => Int -> Vector t -> Matrix t
```
Source

Creates a matrix from a vector by grouping the elements in rows with the desired number of columns. (GNU-Octave groups by columns. To do it you can define `reshapeF r = tr' . reshape r` where r is the desired number of rows.)

Synopsis

```
>>> reshape 4 (fromList [1..12])
(3><4)
 [ 1.0,  2.0,  3.0,  4.0
 , 5.0,  6.0,  7.0,  8.0
 , 9.0, 10.0, 11.0, 12.0 ]
```

```
asRow :: Storable a => Vector a -> Matrix a
```
\# Source

creates a 1-row matrix from a vector

```
>>> asRow (fromList [1..5])
 (1><5)
  [ 1.0, 2.0, 3.0, 4.0, 5.0 ]
```

```
asColumn :: Storable a => Vector a -> Matrix a
```
\# Source

creates a 1-column matrix from a vector

```
>>> asColumn (fromList [1..5])
(5><1)
 [ 1.0
 , 2.0
 , 3.0
 , 4.0
 , 5.0 ]
```

```
fromRows :: Element t => [Vector t] -> Matrix t
```
\# Source

Create a matrix from a list of vectors. All vectors must have the same dimension, or dimension 1, which is are automatically expanded.

```
toRows :: Element t => Matrix t -> [Vector t]
```
\# Source

extracts the rows of a matrix as a list of vectors

```
fromColumns :: Element t => [Vector t] -> Matrix t
```
\# Source

Creates a matrix from a list of vectors, as columns

```
toColumns :: Element t => Matrix t -> [Vector t]
```
\# Source

Creates a list of vectors from the columns of a matrix

Indexing

```
atIndex :: Container c e => c e -> IndexOf c -> e
```
\# Source

generic indexing function

```
>>> vector [1,2,3] `atIndex` 1
2.0
```

```
>>> matrix 3 [0..8] `atIndex` (2,0)
6.0
```

```
class Indexable c t | c -> t, t -> c where
```
\# Source

Synopsis

Alternative indexing function.

```
>>> vector [1..10] ! 3
4.0
```

On a matrix it gets the k-th row as a vector:

```
>>> matrix 5 [1..15] ! 1
[6.0,7.0,8.0,9.0,10.0]
it :: Vector Double
```

```
>>> matrix 5 [1..15] ! 1 ! 3
9.0
```

Synopsis

Methods

`(!) :: c -> Int -> t` infixl 9 # Source

▼ Instances

⊞ `Indexable (Vector Double) Double` # Source

⊞ `Indexable (Vector Float) Float` # Source

⊞ `Indexable (Vector Z) Z` # Source

⊞ `Indexable (Vector I) I` # Source

⊞ `Indexable (Vector (Complex Double)) (Complex Double)` # Source

⊞ `Indexable (Vector (Complex Float)) (Complex Float)` # Source

⊞ `Element t => Indexable (Matrix t) (Vector t)` # Source

⊞ `(Storable t, Indexable (Vector t) t) => Indexable (Vector (Mod m t)) (Mod m t)` # Source

Construction

`scalar :: Container c e => e -> c e` # Source

create a structure with a single element

```
>>> let v = fromList [1..3::Double]
>>> v / scalar (norm2 v)
fromList [0.2672612419124244,0.5345224838248488,0.8017837257372732]
```

`class Konst e d c | d -> c, c -> d where` # Source

Methods

`konst :: e -> d -> c e` # Source

```
>>> konst 7 3 :: Vector Float
fromList [7.0,7.0,7.0]
```

```
>>> konst i (3::Int,4::Int)
(3><4)
 [ 0.0 :+ 1.0, 0.0 :+ 1.0, 0.0 :+ 1.0, 0.0 :+ 1.0
 , 0.0 :+ 1.0, 0.0 :+ 1.0, 0.0 :+ 1.0, 0.0 :+ 1.0
 , 0.0 :+ 1.0, 0.0 :+ 1.0, 0.0 :+ 1.0, 0.0 :+ 1.0 ]
```

▼ Instances

```
Container Vector e => Konst e Int Vector                         # Source
(Num e, Container Vector e) => Konst e (Int, Int) Matrix         # Source
```

```
class Build d f c e | d -> c, c -> d, f -> e, f -> d, f -> c, c e -> f, d e -> f
where                                                            # Source
```

Methods

```
build :: d -> f -> c e                                           # Source
```

```
>>> build 5 (**2) :: Vector Double
[0.0,1.0,4.0,9.0,16.0]
it :: Vector Double
```

Hilbert matrix of order N:

```
>>> let hilb n = build (n,n) (\i j -> 1/(i+j+1)) :: Matrix Double
>>> putStr . dispf 2 $ hilb 3
3x3
1.00  0.50  0.33
0.50  0.33  0.25
0.33  0.25  0.20
```

▼ Instances

```
Container Vector e => Build Int (e -> e) Vector e                # Source
Container Matrix e => Build (Int, Int) (e -> e -> e) Matrix e    # Source
```

```
assoc                                                            # Source
```

`:: Container c e`	
`=> IndexOf c`	size
`-> e`	default value
`-> [(IndexOf c, e)]`	association list
`-> c e`	result

Create a structure from an association list

```
>>> assoc 5 0 [(3,7),(1,4)] :: Vector Double
fromList [0.0,4.0,0.0,7.0,0.0]
```

```
>>> assoc (2,3) 0 [((0,2),7),((1,0),2*i-3)] :: Matrix (Complex Double)
(2><3)
 [      0.0 :+ 0.0, 0.0 :+ 0.0, 7.0 :+ 0.0
 , (-3.0) :+ 2.0, 0.0 :+ 0.0, 0.0 :+ 0.0 ]
```

```
accum                                                            # Source
```

`:: Container c e`	
`=> c e`	initial structure
`-> (e -> e -> e)`	update function
`-> [(IndexOf c, e)]`	association list
`-> c e`	result

Modify a structure using an update function

```
>>> accum (ident 5) (+) [((1,1),5),((0,3),3)] :: Matrix Double
(5><5)
 [ 1.0, 0.0, 0.0, 3.0, 0.0
 , 0.0, 6.0, 0.0, 0.0, 0.0
 , 0.0, 0.0, 1.0, 0.0, 0.0
 , 0.0, 0.0, 0.0, 1.0, 0.0
 , 0.0, 0.0, 0.0, 0.0, 1.0 ]
```

computation of histogram:

```
>>> accum (konst 0 7) (+) (map (flip (,) 1) [4,5,4,1,5,2,5]) :: Vector Double
fromList [0.0,1.0,1.0,0.0,2.0,3.0,0.0]
```

Synopsis

```
linspace :: (Fractional e, Container Vector e) => Int -> (e, e) -> Vector e
```
Source

Creates a real vector containing a range of values:

```
>>> linspace 5 (-3,7::Double)
[-3.0,-0.5,2.0,4.5,7.0]
it :: Vector Double
```

```
>>> linspace 5 (8,3:+2) :: Vector (Complex Double)
[8.0 :+ 0.0,6.75 :+ 0.5,5.5 :+ 1.0,4.25 :+ 1.5,3.0 :+ 2.0]
it :: Vector (Complex Double)
```

Logarithmic spacing can be defined as follows:

```
logspace n (a,b) = 10 ** linspace n (a,b)
```

Diagonal

```
ident :: (Num a, Element a) => Int -> Matrix a
```
Source

creates the identity matrix of given dimension

```
diag :: (Num a, Element a) => Vector a -> Matrix a
```
Source

Creates a square matrix with a given diagonal.

```
diagl :: [Double] -> Matrix Double
```
Source

create a real diagonal matrix from a list

```
>>> diagl [1,2,3]
(3><3)
 [ 1.0, 0.0, 0.0
 , 0.0, 2.0, 0.0
 , 0.0, 0.0, 3.0 ]
```

```
diagRect :: Storable t => t -> Vector t -> Int -> Int -> Matrix t
```
Source

creates a rectangular diagonal matrix:

```
>>> diagRect 7 (fromList [10,20,30]) 4 5 :: Matrix Double
(4><5)
 [ 10.0,  7.0,  7.0, 7.0, 7.0
 ,  7.0, 20.0,  7.0, 7.0, 7.0
 ,  7.0,  7.0, 30.0, 7.0, 7.0
 ,  7.0,  7.0,  7.0, 7.0, 7.0 ]
```

takeDiag :: Element t => Matrix t -> Vector t # Source

extracts the diagonal from a rectangular matrix

Vector extraction

subVector # Source

:: Storable t	
=> Int	index of the starting element
-> Int	number of elements to extract
-> Vector t	source
-> Vector t	result

takes a number of consecutive elements from a Vector

```
>>> subVector 2 3 (fromList [1..10])
[3.0,4.0,5.0]
it :: (Enum t, Num t, Foreign.Storable.Storable t) => Vector t
```

takesV :: Storable t => [Int] -> Vector t -> [Vector t] # Source

Extract consecutive subvectors of the given sizes.

```
>>> takesV [3,4] (linspace 10 (1,10::Double))
[[1.0,2.0,3.0],[4.0,5.0,6.0,7.0]]
it :: [Vector Double]
```

vjoin :: Storable t => [Vector t] -> Vector t # Source

concatenate a list of vectors

```
>>> vjoin [fromList [1..5::Double], konst 1 3]
[1.0,2.0,3.0,4.0,5.0,1.0,1.0,1.0]
it :: Vector Double
```

Matrix extraction

data **Extractor** # Source

Specification of indexes for the operator ??.

Constructors

All
Range Int Int Int
Pos (Vector I)
PosCyc (Vector I)
Take Int
TakeLast Int
Drop Int
DropLast Int

▼ **Instances**

⊞ `Show Extractor` # Source

```
(??) :: Element t => Matrix t -> (Extractor, Extractor) -> Matrix t
```
infixl 9 # Source

General matrix slicing.

```
>>> m
(4><5)
 [  0,  1,  2,  3,  4
 ,  5,  6,  7,  8,  9
 , 10, 11, 12, 13, 14
 , 15, 16, 17, 18, 19 ]
```

```
>>> m ?? (Take 3, DropLast 2)
(3><3)
 [  0,  1,  2
 ,  5,  6,  7
 , 10, 11, 12 ]
```

```
>>> m ?? (Pos (idxs[2,1]), All)
(2><5)
 [ 10, 11, 12, 13, 14
 ,  5,  6,  7,  8,  9 ]
```

```
>>> m ?? (PosCyc (idxs[-7,80]), Range 4 (-2) 0)
(2><3)
 [ 9, 7, 5
 , 4, 2, 0 ]
```

```
(?) :: Element t => Matrix t -> [Int] -> Matrix t
```
infixl 9 # Source

extract rows

```
>>> (20><4) [1..] ? [2,1,1]
(3><4)
 [ 9.0, 10.0, 11.0, 12.0
 , 5.0,  6.0,  7.0,  8.0
 , 5.0,  6.0,  7.0,  8.0 ]
```

```
(¿) :: Element t => Matrix t -> [Int] -> Matrix t
```
infixl 9 # Source

extract columns

(unicode 0x00bf, inverted question mark, Alt-Gr ?)

```
>>> (3><4) [1..] ¿ [3,0]
(3><2)
 [  4.0, 1.0
 ,  8.0, 5.0
 , 12.0, 9.0 ]
```

```
fliprl :: Element t => Matrix t -> Matrix t
```
Source

Reverse columns

```
flipud :: Element t => Matrix t -> Matrix t
```
Source

Reverse rows

subMatrix # Source

`:: Element a`	
`=> (Int, Int)`	(r0,c0) starting position
`-> (Int, Int)`	(rt,ct) dimensions of submatrix
`-> Matrix a`	input matrix
`-> Matrix a`	result

reference to a rectangular slice of a matrix (no data copy)

takeRows `:: Element t => Int -> Matrix t -> Matrix t` # Source

dropRows `:: Element t => Int -> Matrix t -> Matrix t` # Source

takeColumns `:: Element t => Int -> Matrix t -> Matrix t` # Source

dropColumns `:: Element t => Int -> Matrix t -> Matrix t` # Source

remap `:: Element t => Matrix I -> Matrix I -> Matrix t -> Matrix t` # Source

Extract elements from positions given in matrices of rows and columns.

```
>>> r
(3><3)
 [ 1, 1, 1
 , 1, 2, 2
 , 1, 2, 3 ]
>>> c
(3><3)
 [ 0, 1, 5
 , 2, 2, 1
 , 4, 4, 1 ]
>>> m
(4><6)
 [  0,  1,  2,  3,  4,  5
 ,  6,  7,  8,  9, 10, 11
 , 12, 13, 14, 15, 16, 17
 , 18, 19, 20, 21, 22, 23 ]
```

```
>>> remap r c m
(3><3)
 [  6,  7, 11
 ,  8, 14, 13
 , 10, 16, 19 ]
```

The indexes are autoconformable.

```
>>> c'
(3><1)
 [ 1
 , 2
 , 4 ]
>>> remap r c' m
(3><3)
 [  7,  7,  7
```

```
,  8, 14, 14
, 10, 16, 22 ]
```

Block matrix

```
fromBlocks :: Element t => [[Matrix t]] -> Matrix t
```
Source

Create a matrix from blocks given as a list of lists of matrices.

Single row-column components are automatically expanded to match the corresponding common row and column:

```
disp = putStr . dispf 2
```

```
>>> disp $ fromBlocks [[ident 5, 7, row[10,20]], [3, diagl[1,2,3], 0]]
8x10
1  0  0  0  0  7  7  7  10  20
0  1  0  0  0  7  7  7  10  20
0  0  1  0  0  7  7  7  10  20
0  0  0  1  0  7  7  7  10  20
0  0  0  0  1  7  7  7  10  20
3  3  3  3  3  1  0  0   0   0
3  3  3  3  3  0  2  0   0   0
3  3  3  3  3  0  0  3   0   0
```

```
(|||) :: Element t => Matrix t -> Matrix t -> Matrix t
```
infixl 3 # Source

horizontal concatenation

```
>>> ident 3 ||| konst 7 (3,4)
(3><7)
 [ 1.0, 0.0, 0.0, 7.0, 7.0, 7.0, 7.0
 , 0.0, 1.0, 0.0, 7.0, 7.0, 7.0, 7.0
 , 0.0, 0.0, 1.0, 7.0, 7.0, 7.0, 7.0 ]
```

```
(===) :: Element t => Matrix t -> Matrix t -> Matrix t
```
infixl 2 # Source

vertical concatenation

```
diagBlock :: (Element t, Num t) => [Matrix t] -> Matrix t
```
Source

create a block diagonal matrix

```
>>> disp 2 $ diagBlock [konst 1 (2,2), konst 2 (3,5), col [5,7]]
7x8
1  1  0  0  0  0  0  0
1  1  0  0  0  0  0  0
0  0  2  2  2  2  2  0
0  0  2  2  2  2  2  0
0  0  2  2  2  2  2  0
0  0  0  0  0  0  0  5
0  0  0  0  0  0  0  7
```

```
>>> diagBlock [(0><4)[], konst 2 (2,3)]  :: Matrix Double
(2><7)
 [ 0.0, 0.0, 0.0, 0.0, 2.0, 2.0, 2.0
 , 0.0, 0.0, 0.0, 0.0, 2.0, 2.0, 2.0 ]
```

repmat :: Element t => Matrix t -> Int -> Int -> Matrix t # Source

creates matrix by repetition of a matrix a given number of rows and columns

```
>>> repmat (ident 2) 2 3
(4><6)
 [ 1.0, 0.0, 1.0, 0.0, 1.0, 0.0
 , 0.0, 1.0, 0.0, 1.0, 0.0, 1.0
 , 1.0, 0.0, 1.0, 0.0, 1.0, 0.0
 , 0.0, 1.0, 0.0, 1.0, 0.0, 1.0 ]
```

Synopsis

toBlocks :: Element t => [Int] -> [Int] -> Matrix t -> [[Matrix t]] # Source

Partition a matrix into blocks with the given numbers of rows and columns. The remaining rows and columns are discarded.

toBlocksEvery :: Element t => Int -> Int -> Matrix t -> [[Matrix t]] # Source

Fully partition a matrix into blocks of the same size. If the dimensions are not a multiple of the given size the last blocks will be smaller.

Mapping functions

conj :: Container c e => c e -> c e # Source

complex conjugate

cmap :: (Element b, Container c e) => (e -> b) -> c e -> c b # Source

like `fmap` (cannot implement instance Functor because of Element class constraint)

cmod :: (Integral e, Container c e) => e -> c e -> c e # Source

`mod` for integer arrays

```
>>> cmod 3 (range 5)
fromList [0,1,2,0,1]
```

step :: (Ord e, Container c e) => c e -> c e # Source

A more efficient implementation of `cmap (\x -> if x>0 then 1 else 0)`

```
>>> step $ linspace 5 (-1,1::Double)
5 |> [0.0,0.0,0.0,1.0,1.0]
```

cond # Source

:: (Ord e, Container c e, Container c x)	
=> c e	a
-> c e	b
-> c x	l
-> c x	e
-> c x	g
-> c x	result

Element by element version of `case compare a b of {LT -> l; EQ -> e; GT -> g}`.

Arguments with any dimension = 1 are automatically expanded:

```
>>> cond ((1><4)[1..]) ((3><1)[1..]) 0 100 ((3><4)[1..]) :: Matrix Double
(3><4)
[ 100.0,   2.0,   3.0,  4.0
,   0.0, 100.0,   7.0,  8.0
,   0.0,   0.0, 100.0, 12.0 ]
```

```
>>> let chop x = cond (abs x) 1E-6 0 0 x
```

Synopsis

Find elements

`find :: Container c e => (e -> Bool) -> c e -> [IndexOf c]` # Source

Find index of elements which satisfy a predicate

```
>>> find (>0) (ident 3 :: Matrix Double)
[(0,0),(1,1),(2,2)]
```

`maxIndex :: Container c e => c e -> IndexOf c` # Source

index of maximum element

`minIndex :: Container c e => c e -> IndexOf c` # Source

index of minimum element

`maxElement :: Container c e => c e -> e` # Source

value of maximum element

`minElement :: Container c e => c e -> e` # Source

value of minimum element

`sortVector :: (Ord t, Element t) => Vector t -> Vector t` # Source

`sortIndex :: (Ord t, Element t) => Vector t -> Vector I` # Source

```
>>> m <- randn 4 10
>>> disp 2 m
4x10
-0.31   0.41   0.43  -0.19  -0.17  -0.23  -0.17  -1.04  -0.07  -1.24
 0.26   0.19   0.14   0.83  -1.54  -0.09   0.37  -0.63   0.71  -0.50
-0.11  -0.10  -1.29  -1.40  -1.04  -0.89  -0.68   0.35  -1.46   1.86
 1.04  -0.29   0.19  -0.75  -2.20  -0.01   1.06   0.11  -2.09  -1.58
```

```
>>> disp 2 $ m ?? (All, Pos $ sortIndex (m!1))
4x10
-0.17  -1.04  -1.24  -0.23   0.43   0.41  -0.31  -0.17  -0.07  -0.19
-1.54  -0.63  -0.50  -0.09   0.14   0.19   0.26   0.37   0.71   0.83
-1.04   0.35   1.86  -0.89  -1.29  -0.10  -0.11  -0.68  -1.46  -1.40
-2.20   0.11  -1.58  -0.01   0.19  -0.29   1.04   1.06  -2.09  -0.75
```

Sparse

```
type AssocMatrix = [((Int, Int), Double)]
```
Source

```
toDense :: AssocMatrix -> Matrix Double
```
Source

```
mkSparse :: AssocMatrix -> GMatrix
```
Source

```
mkDiagR :: Int -> Int -> Vector Double -> GMatrix
```
Source

```
mkDense :: Matrix Double -> GMatrix
```
Source

IO

```
disp :: Int -> Matrix Double -> IO ()
```
Source

print a real matrix with given number of digits after the decimal point

```
>>> disp 5 $ ident 2 / 3
2x2
0.33333  0.00000
0.00000  0.33333
```

```
loadMatrix :: FilePath -> IO (Matrix Double)
```
Source

load a matrix from an ASCII file formatted as a 2D table.

```
loadMatrix' :: FilePath -> IO (Maybe (Matrix Double))
```
Source

```
saveMatrix
```
Source

`:: FilePath`	
`-> String`	"printf" format (e.g. "%.2f", "%g", etc.)
`-> Matrix Double`	
`-> IO ()`	

save a matrix as a 2D ASCII table

```
latexFormat
```
Source

`:: String`	type of braces: "matrix", "bmatrix", "pmatrix", etc.
`-> String`	Formatted matrix, with elements separated by spaces and newlines
`-> String`	

Tool to display matrices with latex syntax.

```
>>> latexFormat "bmatrix" (dispf 2 $ ident 2)
"\\begin{bmatrix}\n1  &  0\n\\\\\n0  &  1\n\\end{bmatrix}"
```

```
dispf :: Int -> Matrix Double -> String
```
Source

Show a matrix with a given number of decimal places.

```
>>> dispf 2 (1/3 + ident 3)
"3x3\n1.33  0.33  0.33\n0.33  1.33  0.33\n0.33  0.33  1.33\n"
```

```
>>> putStr . dispf 2 $ (3><4)[1,1.5..]
3x4
1.00  1.50  2.00  2.50
3.00  3.50  4.00  4.50
5.00  5.50  6.00  6.50
```

```
>>> putStr . unlines . tail . lines . dispf 2 . asRow $ linspace 10 (0,1)
0.00  0.11  0.22  0.33  0.44  0.56  0.67  0.78  0.89  1.00
```

```
disps :: Int -> Matrix Double -> String
```
Source

Show a matrix with "autoscaling" and a given number of decimal places.

```
>>> putStr . disps 2 $ 120 * (3><4) [1..]
3x4  E3
 0.12  0.24  0.36  0.48
 0.60  0.72  0.84  0.96
 1.08  1.20  1.32  1.44
```

```
dispcf :: Int -> Matrix (Complex Double) -> String
```
Source

Pretty print a complex matrix with at most n decimal digits.

```
format :: Element t => String -> (t -> String) -> Matrix t -> String
```
Source

Creates a string from a matrix given a separator and a function to show each entry. Using this function the user can easily define any desired display function:

```
import Text.Printf(printf)
```

```
disp = putStr . format "  " (printf "%.2f")
```

```
dispDots :: Int -> Matrix Double -> IO ()
```
Source

```
dispBlanks :: Int -> Matrix Double -> IO ()
```
Source

```
dispShort :: Int -> Int -> Int -> Matrix Double -> IO ()
```
Source

Element conversion

```
class Convert t where
```
Source

Methods

```
real :: Complexable c => c (RealOf t) -> c t
```
Source

```
complex :: Complexable c => c t -> c (ComplexOf t)
```
Source

```
single :: Complexable c => c t -> c (SingleOf t)
```
Source

```
double :: Complexable c => c t -> c (DoubleOf t)
```
Source

Synopsis

```
toComplex :: (Complexable c, RealElement t) => (c t, c t) -> c (Complex t)
```
Source #

```
fromComplex :: (Complexable c, RealElement t) => c (Complex t) -> (c t, c t)
```
Source

Instances

- Convert Double # Source
- Convert Float # Source
- Convert (Complex Double) # Source
- Convert (Complex Float) # Source

```
roundVector :: Vector Double -> Vector Double
```
Source

```
fromInt :: Container c e => c I -> c e
```
Source

```
>>> fromInt ((2><2) [0..3]) :: Matrix (Complex Double)
(2><2)
[ 0.0 :+ 0.0, 1.0 :+ 0.0
, 2.0 :+ 0.0, 3.0 :+ 0.0 ]
```

```
toInt :: Container c e => c e -> c I
```
Source

```
fromZ :: Container c e => c Z -> c e
```
Source

```
toZ :: Container c e => c e -> c Z
```
Source

Misc

```
arctan2 :: (Fractional e, Container c e) => c e -> c e -> c e
```
Source

```
separable :: Element t => (Vector t -> Vector t) -> Matrix t -> Matrix t
```
Source

matrix computation implemented as separated vector operations by rows and columns.

```
fromArray2D :: Storable e => Array (Int, Int) e -> Matrix e
```
Source

```
module Data.Complex
```

```
data Mod (n :: Nat) t
```
Source

Wrapper with a phantom integer for statically checked modular arithmetic.

Instances

- KnownNat m => Container Vector (Mod m Z) # Source
- KnownNat m => Container Vector (Mod m I) # Source
- KnownNat m => Num (Vector (Mod m Z)) # Source
- KnownNat m => Num (Vector (Mod m I)) # Source

Synopsis

- KnownNat m => Testable (Matrix (Mod m I)) # Source
- KnownNat m => Normed (Vector (Mod m Z)) # Source
- KnownNat m => Normed (Vector (Mod m I)) # Source
- (Storable t, Indexable (Vector t) t) => Indexable (Vector (Mod m t)) (Mod m t) # Source
- (Integral t, Enum t, KnownNat m) => Enum (Mod m t) # Source
- (Eq t, KnownNat m) => Eq (Mod m t) # Source
- (Show (Mod m t), Num (Mod m t), Eq t, KnownNat m) => Fractional (Mod m t) # Source

  this instance is only valid for prime m
- (Integral t, KnownNat m, Num (Mod m t)) => Integral (Mod m t) # Source
- (Integral t, KnownNat n) => Num (Mod n t) # Source
- (Ord t, KnownNat m) => Ord (Mod m t) # Source
- (Integral t, Real t, KnownNat m, Integral (Mod m t)) => Real (Mod m t) # Source
- Show t => Show (Mod n t) # Source
- Storable t => Storable (Mod n t) # Source
- NFData t => NFData (Mod n t) # Source
- KnownNat m => Element (Mod m Z) # Source
- KnownNat m => Element (Mod m I) # Source
- KnownNat m => Product (Mod m Z) # Source
- KnownNat m => Product (Mod m I) # Source
- KnownNat m => Numeric (Mod m Z) # Source
- KnownNat m => Numeric (Mod m I) # Source
- type RealOf (Mod n Z) # Source
- type RealOf (Mod n I) # Source

Synopsis

```
data Vector a
```
#

Storable-based vectors

▼ Instances

- Complexable Vector # Source
- LSDiv Vector # Source
- Storable a => Vector Vector a
- Container Vector t => Linear t Vector # Source
- Container Vector Double # Source
- Container Vector Float # Source
- Container Vector Z # Source

Container Vector I # Source

Container Vector e => Konst e Int Vector # Source

Container Vector (Complex Double) # Source

Container Vector (Complex Float) # Source

KnownNat n => Sized ℂ (C n) Vector # Source

KnownNat n => Sized ℝ (R n) Vector # Source

KnownNat m => Container Vector (Mod m Z) # Source

KnownNat m => Container Vector (Mod m I) # Source

Container Vector e => Build Int (e -> e) Vector e # Source

Storable a => IsList (Vector a)

(Storable a, Eq a) => Eq (Vector a)

Floating (Vector Double) # Source

Floating (Vector Float) # Source

Floating (Vector (Complex Double)) # Source

Floating (Vector (Complex Float)) # Source

(Container Vector a, Num (Vector a), Fractional a) => Fractional (Vector a) # Source

(Data a, Storable a) => Data (Vector a)

Num (Vector Double) # Source

Num (Vector Float) # Source

Num (Vector (Complex Double)) # Source

Num (Vector (Complex Float)) # Source

Num (Vector Z) # Source

Num (Vector I) # Source

KnownNat m => Num (Vector (Mod m Z)) # Source

KnownNat m => Num (Vector (Mod m I)) # Source

(Storable a, Ord a) => Ord (Vector a)

(Read a, Storable a) => Read (Vector a)

(Show a, Storable a) => Show (Vector a)

Storable a => Semigroup (Vector a)

Storable a => Monoid (Vector a)

(Binary a, Storable a) => Binary (Vector a) # Source

NFData (Vector a)

Storable t => TransArray (Vector t) # Source

Container Vector t => Additive (Vector t) # Source

Normed (Vector Float) # Source

Normed (Vector (Complex Float)) # Source

Normed (Vector C) # Source

Normed (Vector R) # Source

Normed (Vector Z) # Source

Normed (Vector I) # Source

KnownNat m => Normed (Vector (Mod m Z)) # Source

KnownNat m => Normed (Vector (Mod m I)) # Source

Indexable (Vector Double) Double # Source

Indexable (Vector Float) Float # Source

Indexable (Vector Z) Z # Source

Indexable (Vector I) I # Source

Indexable (Vector (Complex Double)) (Complex Double) # Source

Indexable (Vector (Complex Float)) (Complex Float) # Source

Element t => Indexable (Matrix t) (Vector t) # Source

(Storable t, Indexable (Vector t) t) => Indexable (Vector (Mod m t)) (Mod m t) # Source

type Mutable Vector

type IndexOf Vector # Source

type Item (Vector a)

type Trans (Vector t) b # Source

type TransRaw (Vector t) b # Source

data **Matrix** t # Source

Matrix representation suitable for BLAS/LAPACK computations.

Instances

Complexable Matrix # Source

LSDiv Matrix # Source

Container Matrix t => Linear t Matrix # Source

(Num a, Element a, Container Vector a) => Container Matrix a # Source

(Num e, Container Vector e) => Konst e (Int, Int) Matrix # Source

(KnownNat m, KnownNat n) => Sized ℂ (M m n) Matrix # Source

(KnownNat m, KnownNat n) => Sized ℝ (L m n) Matrix # Source

Container Matrix a => Eq (Matrix a) # Source

(Floating a, Container Vector a, Floating (Vector a), Fractional (Matrix a)) => Floating (Matrix a) # Source

(Container Vector a, Fractional a, Fractional (Vector a), Num (Matrix a)) => Fractional (Matrix a) # Source

(Container Matrix a, Num a, Num (Vector a)) => Num (Matrix a) # Source

(Element a, Read a) => Read (Matrix a) # Source

(Show a, Element a) => Show (Matrix a) # Source

(Container Vector t, Eq t, Num (Vector t), Product t) => Semigroup (Matrix t) # Source

```
⊞ (Container Vector t, Eq t, Num (Vector t), Product t) => Monoid (Matrix t)        # Source
⊞ (Binary (Vector a), Element a) => Binary (Matrix a)        # Source
⊞ (Storable t, NFData t) => NFData (Matrix t)        # Source
⊞ Storable t => TransArray (Matrix t)        # Source
⊞ Testable (Matrix I)        # Source
⊞ KnownNat m => Testable (Matrix (Mod m I))        # Source
⊞ Container Matrix t => Additive (Matrix t)        # Source
⊞ Normed (Matrix C)        # Source
⊞ Normed (Matrix R)        # Source
⊞ (CTrans t, Container Vector t) => Transposable (Matrix t) (Matrix t)        # Source
⊞ Element t => Indexable (Matrix t) (Vector t)        # Source
⊞ Container Matrix e => Build (Int, Int) (e -> e -> e) Matrix e        # Source
⊞ type IndexOf Matrix        # Source
⊞ type Trans (Matrix t) b        # Source
⊞ type TransRaw (Matrix t) b        # Source
```

```
data GMatrix        # Source
```

General matrix with specialized internal representations for dense, sparse, diagonal, banded, and constant elements.

```
>>> let m = mkSparse [((0,999),1.0),((1,1999),2.0)]
>>> m
SparseR {gmCSR = CSR {csrVals = fromList [1.0,2.0],
                      csrCols = fromList [1000,2000],
                      csrRows = fromList [1,2,3],
                      csrNRows = 2,
                      csrNCols = 2000},
                      nRows = 2,
                      nCols = 2000}
```

```
>>> let m = mkDense (mat 2 [1..4])
>>> m
Dense {gmDense = (2><2)
 [ 1.0, 2.0
 , 3.0, 4.0 ], nRows = 2, nCols = 2}
```

▼ **Instances**

```
⊞ Show GMatrix        # Source
⊞ Testable GMatrix        # Source
⊞ Transposable GMatrix GMatrix        # Source
```

```
nRows :: GMatrix -> Int        # Source
```

```
nCols :: GMatrix -> Int        # Source
```

Synopsis