introduction to *hmatrix*

Alberto Ruiz `<aruiz@um.es>`

version 0.14, June 25, 2012

Contents

1 Introduction

The package *hmatrix* provides a high level, purely functional Haskell [1] interface to matrix computations [11] and other numerical algorithms, implemented using LAPACK [2], BLAS [3], and GSL [4]. Using `ghci` [5] we can interactively experiment with simple operations, much like working with GNU-Octave [6] or similar systems, and large applications can be more easily developed and maintained using Haskell's expressive power. Detailed documentation is available on-line [19].

Vectors and matrices are different types but dimension checking is performed at run time. This is a good balance between safety and simplicity, and this package can be used as the underlying engine for more advanced interfaces. For instance, Reiner Pope's *hmatrix-static* [18] supports compile-time checking of matrix and vector sizes. For a more complete collection of Haskell bindings to the above numeric libraries see the related packages [16, 17, 10, 14, 15].

This document has been generated from literate Haskell sources using *lsh2TeX* [12]. We use `\eval` and `\perform` to automatically insert the results of the computations.

1.1 Installation

The library requires the Glasgow Haskell Compiler [5] `ghc >= 6.10` and the development packages for GSL, BLAS and LAPACK. An optimized implementation like ATLAS [7] or Intel's MKL [8] is recommended. The packages `gnuplot` and `imagemagick` are also required for graphics[1], and `octave` is useful to check the results. For example, in *Ubuntu* we need the following packages:

```
$ sudo apt-get install libgsl0-dev liblapack-dev gnuplot imagemagick
$ sudo apt-get install haskell-platform
$ cabal update
```

The latest stable version of *hmatrix* can be automatically installed from *Hackage* [9]:

```
$ cabal install hmatrix
$ cabal install hmatrix-special
$ cabal install hmatrix-tests
```

The optional package `hmatrix-special` provides selected GSL special functions and is not required for linear algebra applications.

See the website [20] for detailed installation instructions for MacOS X and Windows.

1.2 Tests

We should verify that the library works as expected by running the tests:

```
ghci> Numeric.LinearAlgebra.Tests.runTests 20
```

No errors should be reported (the argument for *runTests* is the maximum matrix size; some tests may fail with bigger sizes due to precision loss). There is also a simple benchmark:

```
ghci> Numeric.LinearAlgebra.Tests.runBenchmarks
```

[1]See also the packages `plot` and `plot-gtk` by Vivian McPhail and `Gnuplot` by Henning Thielemann.

2 Linear Algebra

The linear algebra functions work with dense inmutable 1D arrays (*Vector*) and 2D arrays (*Matrix*) of real (*Double*) or complex (*Complex Double*) elements. *Float* and *Complex Float* elements are allowed for simple numeric operations and matrix products, but full linear algebra is only supported in double precision. Most functions admit both real and complex arguments but they are different types and conversions between them must be explicit. There is no difference between row and column vectors.

The *Vector* type is a *Storable* vector from Roman Leshchinskiy's `vector` package [13], so all array processing tools provided by this library are directly available. However, for compatibility with LAPACK the matrix type is not a *Vector* (*Vector t*).

2.1 Quick reference

Most Matlab/Octave array operations can be replicated using the tools provided by *hmatrix* and standard Haskell functions. The following table shows the chosen names for frequently used functions and operators.

Matlab/Octave	***hmatrix***	**description**
`[a,b;c,d]`	`(><)`, *fromList*, *fromBlocks*, etc.	vector/matrix creation
m'	*ctrans m*	conjugate transpose
m.'	*trans m*	transpose
*	*multiply* (<>)	matrix product
* or *dot*	*dot* (`<.>`)	dot product
*	*scale*, *scalar x* *	scale all entries
/ or \	`<\>` or *linearSolve*	multiplication by (pseudo)inverse
`+ .* ./ .^` etc.	`+ * / ^ **` etc.	element by element operations
diag	*diag*, *takeDiag*, etc.	diagonal matrices
reshape	*reshape*, *flatten*, etc.	changing structure
repmat		"replicate" matrix
eye, *zeros*, *ones*	*ident*, *constant*, etc.	useful constants
norm	*pnorm*	matrix and vector norms
eig	*eigenvalues*, *eig*, *eigSH*, etc.	eigensystem
svd	*singularValues*, *svd*, *thinSVD*, etc.	singular value decomposition
qr		QR factorization
chol		Cholesky factorization
hess		Hessenberg factorization
schur		Schur factorization
inv		inverse matrix
pinv		pseudoinverse
det		determinant
lu		LU factorization
expm		matrix exponential

The numeric instances for *Vector* and *Matrix* are defined element-by-element. Therefore, $(*)$ is like `.*` in Matlab/Octave. Numeric literals are interpreted as singleton structures which are automatically replicated to match operands of any other shape. If this behavior is not desired we can alternatively import just the required modules and functions. Data organization, normalization constants, and other conventions are usually the same as those of GNU-Octave.

2.2 Simple examples

To get a first impression of the library we show a simple `ghci` session. A 'user friendly' interface is provided by the module *Numeric.LinearAlgebra*:

```
ghci> import Numeric.LinearAlgebra
```

A matrix can be created by giving its dimensions with the operator $(><)$ (resembling $\times$) and a list with the elements in row order:

```
ghci> let m = (3><4) [1..]  :: Matrix Double
```

Most functions are overloaded for both real and complex matrices, so a type hint is often needed.

```
ghci> m

(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 ]
```

The operator `(><)` is also used by `show`, so printed matrices can be directly read back or included in source code. The operator `|>` (resembling an arrow tip) can be used to create vectors:

```
ghci> let w = 4 |> [2,0,-3,0::Double]
```

The operator $(<>)$ is the matrix product, and also the matrix-vector and vector-matrix products:

```
ghci> m <> w

fromList [-7.0,-11.0,-15.0]
```

The euclidean inner product of vectors is represented by $(< . >)$:

```
ghci> w <.> w

13.0
```

We can try the standard matrix computations. For instance, we can check that the singular values of m are related to the eigenvalues of mm^T:

```
ghci> singularValues m

fromList [25.436835633480243,1.7226122475210646,2.797126318607423e-16]

ghci> sqrt . eigenvalues $ m <> trans m

fromList [25.43683563348025 :+ 0.0,1.7226122475210628 :+ 0.0,2.046526718156048e-7 :+ 0.0]
```

The eigenvalues of an arbitrary matrix are in general complex, so in this case is better to use a specialized version for symmetric or hermitian matrices, which obtains a *Vector Double* with the eigenvalues in decreasing order:

```
ghci> sqrt . eigenvaluesSH $ m <> trans m

fromList [25.43683563348025,1.7226122475210661,1.3376362286897434e-7]
```

The full singular value decomposition of m is:

```
ghci> let (u,s,v) = fullSVD m
```

The obtained factors reconstruct the original matrix:

```
ghci> u <> s <> trans v

(3><4)
 [ 0.9999999999999997, 1.9999999999999998,                3.0, 3.9999999999999996
 ,   4.999999999999998,                6.0, 6.999999999999999,  7.999999999999997
 ,   8.999999999999998,               10.0,              11.0, 11.999999999999998 ]
```

The *Show* instances for *Matrix* use the standard *show* defined for the base types, which produces an unpredictable number of decimal places. A nicer result can be obtained by the available formatting utilities. For instance, *dispf* (display with fixed format) shows a real matrix as a table of numbers with a given number of decimal places.

```
ghci> let disp = putStr . dispf 2

ghci> disp $ u <> s <> trans v

3x4
1.00   2.00   3.00   4.00
5.00   6.00   7.00   8.00
9.00  10.00  11.00  12.00
```

From the above factorization we can easily obtain the null-space of m:

```
ghci> let x = last (toColumns v)

ghci> x

fromList [-0.22899601185305568,0.6761835031987941,-0.6653789708384209,
0.21819147949268256]


ghci> m <> x

fromList [-5.551115123125783e-17,5.551115123125783e-17,1.6653345369377348e-16]
```

This can be computed more directly by a predefined function:

```
ghci> nullVector m

fromList [-0.22899601185305568,0.6761835031987941,-0.6653789708384209,
0.21819147949268256]
```

Vectors can also be ‘pretty printed’. In this case we choose ‘autoscaled’ format:

```
ghci> let dispv = putStr . vecdisp (disps 2)

ghci> dispv x

4 |>  E-1 -2.29  6.76 -6.65  2.18
```

The function *singularValues* is more efficient than a full decomposition if the singular vectors are not required. There are also functions to efficiently compute ‘economy’ decompositions of rectangular matrices. For example, consider the following matrix:

```
ghci> let b = diagRect 0 (fromList [3,2]) 5 10 :: Matrix Double
ghci> disp b

5x10
3  0  0  0  0  0  0  0  0  0
0  2  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
```

(For legibility, if the matrix appears to be composed of 'approximate integers' fixed format display does not print the decimal places.)

The thin SVD only removes the information corresponding to the 'extra' dimensions:

```
ghci> let (u,s,v) = thinSVD b

ghci> disp (trans u)

5x5
1  0  -0  -0  -0
0  1  -0  -0  -0
0  0  -0  -0   1
0  0  -0   1  -0
0  0   1  -0  -0

ghci> dispv s

5 |>  E0  3.00  2.00  0.00  0.00  0.00

ghci> disp (trans v)

5x10
1  0  0  0  0  0  0  0  0  0
0  1  0  0  0  0  0  0  0  0
0  0  0  0  1  0  0  0  0  0
0  0  0  1  0  0  0  0  0  0
0  0  1  0  0  0  0  0  0  0
```

(The singular vectors corresponding to the null singular values may differ from those obtained by other computing programs[2].) The *compactSVD* only includes the numerically nonzero singular values:

```
ghci> let (u,s,v) = compactSVD b

ghci> disp (trans u)

2x5
1  0  -0  -0  -0
0  1  -0  -0  -0

ghci> dispv s

2 |>  E0  3.00  2.00
```

[2]In recent *hmatrix* versions the 'default' SVD is internally implemented using LAPACK's `[d|z]gesdd`, which is more efficient than the previously used `[d|z]gesvd`. For example, the singular vectors obtained for the above matrix `b` by Octave 3.0.1 (using `[d|z]gesvd`) correspond to identity matrices without any permutation. See `Numeric.LinearAlgebra.LAPACK` for all available SVD variants.

```
ghci> disp (trans v)

2x10
1  0  0  0  0  0  0  0  0  0
0  1  0  0  0  0  0  0  0  0
```

We can try other factorizations:

```
ghci> let (q,r) = qr m

ghci> disp q

3x3
-0.10   0.91   0.41
-0.48   0.32  -0.82
-0.87  -0.28   0.41

ghci> disp r

3x4
-10.34  -11.79  -13.24  -14.69
  0.00    0.95    1.89    2.84
  0.00    0.00    0.00    0.00

ghci> disp $ q <> r

3x4
1.00   2.00   3.00   4.00
5.00   6.00   7.00   8.00
9.00  10.00  11.00  12.00
```

The Cholesky factorization works on symmetric matrices:

```
ghci> let t = (diag.fromList) [4,7,3] + 1 :: Matrix Double

ghci> disp t

3x3
5  1  1
1  8  1
1  1  4

ghci> let c = chol t

ghci> disp c

3x3
2.24  0.45  0.45
0.00  2.79  0.29
0.00  0.00  1.93

ghci> disp $ trans c <> c

3x3
5.00  1.00  1.00
1.00  8.00  1.00
1.00  1.00  4.00
```

2.3 Array manipulation

Many linear algebra applications require auxiliary data manipulation tasks: conversion of matrices into lists of vectors, extraction of submatrices, construction of matrices from blocks, etc. The library provides a few utilities for this kind of tasks.

$$\textbf{import}\ \mathit{Numeric.LinearAlgebra}$$
$$\mathit{disp} = \mathit{putStr} \circ \mathit{dispf}\ 2$$

Vectors are created from ordinary Haskell lists:

$$u = \mathit{fromList}\ [1, 7, -5, 2.8 :: \mathit{Double}]$$
$$v = \mathit{fromList}\ [1 \mathinner{..} 4] :: \mathit{Vector}\ \mathit{Double}$$
$$w = 7 \mid > [3 - 2 * i, 0, 5, 8, 7 * i, \mathit{pi}, \mathit{exp}\ (i * \mathit{pi})]$$

```
ghci> u

fromList [1.0,7.0,-5.0,2.8]

ghci> w

fromList [3.0 :+ (-2.0),0.0 :+ 0.0,5.0 :+ 0.0,8.0 :+ 0.0,0.0 :+ 7.0,3.
141592653589793 :+ 0.0,(-1.0) :+ 1.2246063538223773e-16]
```

There are functions for vector concatenation, extraction of subvectors and individual elements:

```
ghci> join [u,v,u]

fromList [1.0,7.0,-5.0,2.8,1.0,2.0,3.0,4.0,1.0,7.0,-5.0,2.8]

ghci> subVector 2 3 w

fromList [5.0 :+ 0.0,8.0 :+ 0.0,0.0 :+ 7.0]

ghci> v @> 2

3.0
```

In addition to the already mentioned operator ($><$) there are several ways to create matrices. For instance, they can be built from a list of vectors using *fromRows* or *fromColumns*:

$$a = \mathit{fromRows}\ [u, v, \mathit{constant}\ 2\ 4, \mathit{linspace}\ 4\ (0, 0.75), 3 * v]$$
$$b = \mathit{fromColumns}\ (\mathit{take}\ 12\ (\mathit{cycle}\ [u, v]))$$

```
ghci> a

(5><4)
 [ 1.0,  7.0, -5.0,  2.8
 , 1.0,  2.0,  3.0,  4.0
 , 2.0,  2.0,  2.0,  2.0
 , 0.0, 0.25,  0.5, 0.75
 , 3.0,  6.0,  9.0, 12.0 ]
```

```
ghci> disp b

4x12
 1.00  1.00   1.00  1.00   1.00  1.00   1.00  1.00   1.00  1.00   1.00  1.00
 7.00  2.00   7.00  2.00   7.00  2.00   7.00  2.00   7.00  2.00   7.00  2.00
-5.00  3.00  -5.00  3.00  -5.00  3.00  -5.00  3.00  -5.00  3.00  -5.00  3.00
 2.80  4.00   2.80  4.00   2.80  4.00   2.80  4.00   2.80  4.00   2.80  4.00
```

Matrices can be created from a list of lists of elements or other matrices:

$$\begin{array}{l} c = \mathit{fromLists} \\ \quad [[1,2,3,4] \\ \quad ,[4,3,2,1] \\ \quad ,[0,1,0,1]] :: \mathit{Matrix\ Double} \end{array}$$

```
ghci> disp c

3x4
1  2  3  4
4  3  2  1
0  1  0  1

ghci> disp $ fromBlocks [[c, ident 3],[8+3*ident 4, trans c]]

7x7
 1   2   3   4  1  0  0
 4   3   2   1  0  1  0
 0   1   0   1  0  0  1
11   8   8   8  1  4  0
 8  11   8   8  2  3  1
 8   8  11   8  3  2  0
 8   8   8  11  4  1  1
```

If a matrix block has a single row or column it will be automatically replicated to match the corresponding common dimension. We define some utility functions [3] to illustrate this behavior:

$$\begin{array}{l} \mathit{vector}\ \mathit{xs} = \mathit{fromList}\ \mathit{xs} :: \mathit{Vector\ Double} \\ \mathit{eye}\ n = \mathit{ident}\ n :: \mathit{Matrix\ Double} \\ \mathit{diagl} = \mathit{diag} \circ \mathit{vector} \\ \mathit{row} = \mathit{asRow} \circ \mathit{vector} \\ \mathit{col} = \mathit{asColumn} \circ \mathit{vector} \\ \mathit{ones}\ r\ c = \mathit{konst}\ (1 :: \mathit{Double})\ (r, c) \\ \mathit{blocks}\ \mathit{mms} = \mathit{fromBlocks}\ \mathit{mms} :: \mathit{Matrix\ Double} \end{array}$$

Numeric literals are interpreted as 1×1 matrices, so the following expression is also valid, although not very efficient:

```
ghci> disp $ blocks [[1,2,3],[5,7,8]]

2x3
1  2  3
5  7  8
```

Constants are automatically expanded:

[3] Some of them are available in the module `Numeric.LinearAlgebra.Util`.

```
ghci> disp $ blocks [[7*ones 3 5,0],[0,1]]

4x6
7  7  7  7  7  0
7  7  7  7  7  0
7  7  7  7  7  0
0  0  0  0  0  1

ghci> disp $ blocks [[3+8*eye 5, 25,3],[700, eye 3,4]]

8x9
 11    3    3    3    3  25  25  25  3
  3   11    3    3    3  25  25  25  3
  3    3   11    3    3  25  25  25  3
  3    3    3   11    3  25  25  25  3
  3    3    3    3   11  25  25  25  3
700  700  700  700  700   1   0   0  4
700  700  700  700  700   0   1   0  4
700  700  700  700  700   0   0   1  4
```

and single row or column blocks are appropriately replicated:

```
ghci> disp $ blocks [[5*ones 4 6, row[1..4]],[col[1..3], 0]]

7x10
5  5  5  5  5  5  1  2  3  4
5  5  5  5  5  5  1  2  3  4
5  5  5  5  5  5  1  2  3  4
5  5  5  5  5  5  1  2  3  4
1  1  1  1  1  1  0  0  0  0
2  2  2  2  2  2  0  0  0  0
3  3  3  3  3  3  0  0  0  0
```

We can define simpler horizontal and vertical concatenation operators which do not require a rectangular arrangement:

infixl $3 \, !$
$a \, ! \, b = \mathit{blocks} \; [[a, b]]$
infixl $2 \, \#$
$a \, \# \, b = \mathit{blocks} \; [[a], [b]]$

They are similar to Matlab/Octave's matrix construction operators "," and ";", extended with automatic replication of single rows or columns:

$$\begin{aligned} g = \; & \mathit{ones} \; 4 \; 6 \, ! \, \mathit{row} \; [5, 6, 7] \\ & \# \, 9 \, ! \, \mathit{row} \; [2 \mathinner{..} 7] \, ! \, \mathit{eye} \; 2 \\ & \# \, 8 \end{aligned}$$

```
ghci> disp g

7x9
1  1  1  1  1  1  5  6  7
1  1  1  1  1  1  5  6  7
1  1  1  1  1  1  5  6  7
1  1  1  1  1  1  5  6  7
9  2  3  4  5  6  7  1  0
9  2  3  4  5  6  7  0  1
8  8  8  8  8  8  8  8  8
```

Most of the above functions have their corresponding inverses: *toRows*, *toLists*, *subMatrix*, *takeRows*, *dropColumns*, etc. See the documentation for full information.

Finally, vectors and matrices can be created from association lists:

```
ghci> assoc (3,4) 7 [((0,1),3),((2,1),5)] :: Matrix Double

(3><4)
 [ 7.0, 3.0, 7.0, 7.0
 , 7.0, 7.0, 7.0, 7.0
 , 7.0, 5.0, 7.0, 7.0 ]
```

```
ghci> accum (fromList[0..9]) (+) [(2,1),(7,5),(2,2)] :: Vector Double

fromList [0.0,1.0,5.0,3.0,4.0,5.0,6.0,12.0,8.0,9.0]
```

and also from plain functions. For instance:

$$h = build\ (5,5)\ (\lambda r\ c \to r - 2 + i * (c - 2)) :: Matrix\ (Complex\ Double)$$

```
ghci> dispcf 2 h

5x5
-2-2i  -2-i  -2  -2+i  -2+2i
-1-2i  -1-i  -1  -1+i  -1+2i
  -2i    -i   0     i     2i
 1-2i   1-i   1   1+i   1+2i
 2-2i   2-i   2   2+i   2+2i
```

A matrix can be displayed with LaTeX format. Given the auxiliary definition

$$latex\ m = putStrLn\ \$\ latexFormat\ \texttt{"bmatrix"}\ (dispcf\ 2\ m)$$

we insert the matrix in the document with the *lhs2TeX* command `$$\perform{latex h}$$`:

$$\begin{bmatrix} -2-2i & -2-i & -2 & -2+i & -2+2i \\ -1-2i & -1-i & -1 & -1+i & -1+2i \\ -2i & -i & 0 & i & 2i \\ 1-2i & 1-i & 1 & 1+i & 1+2i \\ 2-2i & 2-i & 2 & 2+i & 2+2i \end{bmatrix}$$

2.4 Type conversions

Since vectors and matrices are different types we need conversion functions. The elements of a vector can be arranged into a matrix with a given number of columns using *reshape*:

```
ghci> reshape 4 (fromList [1..12 :: Double])

(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 ]
```

A vector with all matrix elements in row order[4] is given by *flatten*:

```
ghci> flatten (ident 3 :: Matrix Double)

fromList [1.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,1.0]

ghci> disp $ reshape 8 . flatten $ ident 4

2x8
1  0  0  0  0  1  0  0
0  0  1  0  0  0  0  1
```

The function *real* converts a real object into a generic real or complex object with zero imaginary components:

```
ghci> let x = fromList[1,2,3] :: Vector Double
ghci> x

fromList [1.0,2.0,3.0]

ghci> real x :: Vector Double

fromList [1.0,2.0,3.0]

ghci> real x :: Vector (Complex Double)

fromList [1.0 :+ 0.0,2.0 :+ 0.0,3.0 :+ 0.0]
```

The function *complex* takes a real or complex object and converts it into a complex object.

```
ghci> complex x

fromList [1.0 :+ 0.0,2.0 :+ 0.0,3.0 :+ 0.0]


ghci> let z = fromList[1,2+i,3-i] :: Vector (Complex Double)
ghci> complex z

fromList [1.0 :+ 0.0,2.0 :+ 1.0,3.0 :+ (-1.0)]
```

These functions are useful to write generic code. For instance, if $(s, v) = eig\ m$ then the property $complex\ m <> v \simeq v <> diag\ s$ is valid for both real and complex matrices. We need *complex* because the eigenvalues and eigenvectors of a general real matrix can be complex, so the reconstruction of both real and complex matrices must be done in the complex domain. If $(u, s, v) = fullSVD\ m$ then the property $m \simeq u <> real\ d <> trans\ v$ is valid for real and complex matrices. In this case we need *real* because the singular values are always real, so the reconstruction can be done in the real domain for real matrices.

[4]Octave/Matlab uses column order in `reshape`.

2.5 Numeric instances

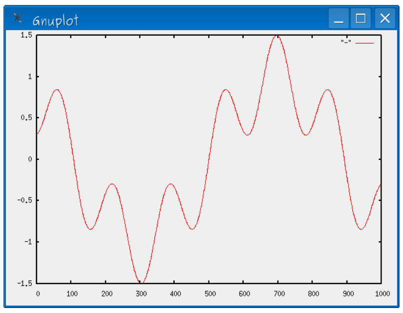
The numeric instances (*Num*, *Fractional*, and *Floating*) of vectors and matrices have been defined in such a way that we can easily write 'vectorized' expressions:

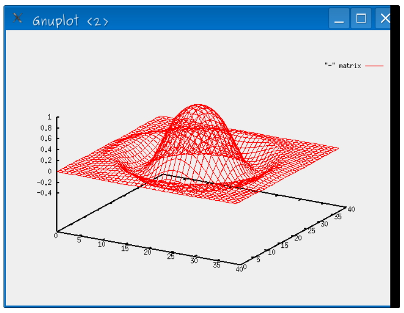
```
import Numeric.LinearAlgebra
import Graphics.Plot

disp = putStr ∘ dispf 2

vector xs = fromList xs :: Vector Double
```

```
ghci> 1 + 2 * vector[1..5]

fromList [3.0,5.0,7.0,9.0,11.0]
```

This behaviour is useful to work with generic functions defined elsewhere. For instance, the following program plots standard *Floating* functions:

```
plot2D f n = mesh (f x y) where
  (x, y) = meshdom range range
  range = linspace n (−2, 2)

sombrero :: (Floating a) ⇒ a → a → a
sombrero x y = exp (−r2) ∗ cos (2 ∗ r2)
  where r2 = x ∗ x + y ∗ y

f :: (Floating a) ⇒ a → a
f x = sin x + 0.5 ∗ sin (5 ∗ x)

main = do
  let x = linspace 1000 (−4, 4)
  mplot [f x]
  plot2D sombrero 40
```





However, explicit conversions or specific operators are required in some cases, since only numeric literals are interpreted as containers:

```
x = vector [1 . . 3]
norm x = pnorm PNorm1 x
```

```
ghci> x / norm x
```

```
 <interactive>:1:4:
 Couldn't match expected type Vector Double
            against inferred type Double
     In the second argument of (/), namely norm x
     In the expression: x / norm x
     In the definition of it: it = x / norm x
```

```
ghci> scale (recip (norm x)) x

fromList [0.16666666666666666,0.3333333333333333,0.5]

ghci> x / scalar (norm x)

fromList [0.16666666666666666,0.3333333333333333,0.5]
```

Since numeric literals stand both for vectors and elements we can define an extremely simple (although not very efficient) way to create small vectors:

$$\textbf{infixl}\ 3\ \&$$
$$a\ \&\ b = \mathit{join}\ [a, b] :: \mathit{Vector\ Double}$$

```
ghci> 3 & 5 & 7

fromList [3.0,5.0,7.0]

ghci> 19 & constant 3 5 & (-2) & 11

fromList [19.0,3.0,3.0,3.0,3.0,3.0,-2.0,11.0]
```

A matrix with a single row or column is automatically expanded in arithmetic operations to match the corresponding dimension of the other argument. This is useful to define vector-like operations on all rows or columns of a matrix:

$$m = (4 >< 5)\ [1\,..] :: \mathit{Matrix\ Double}$$
$$\mathit{row} = \mathit{asRow} \circ \mathit{vector}$$
$$\mathit{col} = \mathit{asColumn} \circ \mathit{vector}$$

```
ghci> disp m

4x5
 1   2   3   4   5
 6   7   8   9  10
11  12  13  14  15
16  17  18  19  20

ghci> disp $ m + row [10,20 .. 50]

4x5
11  22  33  44  55
16  27  38  49  60
21  32  43  54  65
26  37  48  59  70
```

```
ghci> disp $ m * col [100,200 .. 400]

4x5
 100   200   300   400   500
1200  1400  1600  1800  2000
3300  3600  3900  4200  4500
6400  6800  7200  7600  8000
```

Using this feature we can directly 'center' the rows of a matrix by subtracting the mean:

$$mean\ a = constant\ (recip \circ fromIntegral \circ rows\ \$\ a)\ (rows\ a) <> a$$

```
ghci> m - asRow (mean m)

(4><5)
 [ -7.5, -7.5, -7.5, -7.5, -7.5
 , -2.5, -2.5, -2.5, -2.5, -2.5
 ,  2.5,  2.5,  2.5,  2.5,  2.5
 ,  7.5,  7.5,  7.5,  7.5,  7.5 ]
```

As a final example of this kind of implicit replication of single rows or columns we define conversion functions for homogeneous vectors stored as rows of a matrix.

$$homog\ a = fromBlocks\ [[a, 1]]$$
$$inhomog\ a = takeColumns\ c\ a\ /\ dropColumns\ c\ a$$
$$\quad \mathbf{where}\ c = cols\ a - 1$$

The transformation h performs some translation and scaling of vectors in $\mathbb{R}^2$ using homogeneous coordinates:

$$h = (3 >< 3)\ [1, 0, 1, 0, 2, 3, 0, 0, 1] :: Matrix\ Double$$

```
ghci> disp h

3x3
1  0  1
0  2  3
0  0  1
```

We prepare a few test points stored as the rows of p:

$$p = (5 >< 2)\ [1\,..] :: Matrix\ Double$$

```
ghci> disp p

5x2
1   2
3   4
5   6
7   8
9  10
```

The transformation is applied to the homogeneous coordinates and the result is converted back to the ordinary ones:

```
ghci> disp $ inhomog . (<> trans h). homog $ p
```

```
5x2
 2   7
 4  11
 6  15
 8  19
10  23
```

It possible to import only the required linear algebra functions without this kind of ‘adaptable’ numeric instances.

2.6 In-place updates

Efficient in-place modification of vectors and matrices can be safely done using the ST monad. For instance, the following function creates a diagonal matrix:

```
import Numeric.LinearAlgebra
import Data.Packed.ST
import Control.Monad.ST
import System.Random (randomIO)
disp = putStr ∘ dispf 2


dg :: (Num t, Element t) ⇒ Int → Int → [t] → Matrix t
dg r c l = runSTMatrix $ do
  m ← newMatrix 0 r c
  let set v k = writeMatrix m k k v
  sequence_ $ zipWith set l [0 .. min r c − 1]
  return m
```

```
ghci> dg 3 4 [5,7::Double]
```

```
(3><4)
 [ 5.0, 0.0, 0.0, 0.0
 , 0.0, 7.0, 0.0, 0.0
 , 0.0, 0.0, 0.0, 0.0 ]
```

In general, this kind of manipulation requires ‘thawing’ and ‘freezing’ the appropriate arrays:

```
clear k v = runST $ do
  w ← thawVector v
  writeVector w k 0
  v′ ← freezeVector w
  return v′
```

```
ghci> let x = fromList [1,2,5+i,-7*i,8]
```

```
ghci> clear 3 x
```

```
fromList [1.0 :+ 0.0,2.0 :+ 0.0,5.0 :+ 1.0,0.0 :+ 0.0,8.0 :+ 0.0]
```

The original vector is of course not affected:

```
ghci> x

fromList [1.0 :+ 0.0,2.0 :+ 0.0,5.0 :+ 1.0,(-0.0) :+ (-7.0),8.0 :+ 0.0]
```

More examples can be found in the file `examples/inplace.hs` included in the distribution tarball. The functions *assoc* and *accum* are implemented using the ST monad.

2.7 Random vectors and matrices

Simulations often require pseudorandom numbers. There are functions to easily create vectors and matrices with random entries internally using the GSL random number utilities. The seed must be explicitly supplied:

```
ghci> randomVector 7777 Uniform 10

fromList [0.8115051253698766,0.21379995718598366,0.692324141273275,0.5
189151456579566,0.45248611830174923,0.8809234916698188,0.9966296646744
013,0.8711196009535342,0.9875703777652234,0.2944871049840003]
```

We can use *randomIO* to set a different seed in each function call:

```
rand r c = do
  seed ← randomIO
  return (reshape c $ randomVector seed Uniform (r * c))
```

```
ghci> m <- rand 4 3

ghci> disp m

4x3
0.64  0.69  0.87
0.43  0.09  0.40
0.64  0.31  0.34
0.75  0.75  0.12

ghci> m <- rand 4 3

ghci> disp m

4x3
0.47  0.16  0.40
0.30  0.89  0.71
0.04  0.70  0.66
0.36  0.74  0.85
```

There are functions to obtain multivariate samples from uniform distributions on desired intervals, and from gaussian distributions with desired mean vector and covariance matrix:

```
ghci> disp $ gaussianSample 44444 10 (3|>[1,0,-2]) (diag.fromList$[3,2,1])
```

```
10x3
 2.65   1.14  -2.24
 0.10  -0.61  -2.66
-0.91  -0.60  -1.38
 1.97  -1.35  -0.94
 5.56   0.39   0.04
 2.29  -0.03  -2.76
 0.72   0.54  -1.35
-2.09   2.25  -2.56
 1.45   2.62  -1.89
-1.19  -0.95  -1.78
```

See `Numeric.Container` for details.

2.8 Vectorized boolean operations

There is not explicit support for boolean elements, but many constructions based on vectorized comparisons can be easily defined using `step`, `cond`, and `find`.

```
ghci> m <- rand 3 4
```

```
ghci> disp m
```

```
3x4
0.90  0.62  0.75  0.63
0.93  0.44  0.15  0.87
0.71  0.34  0.05  0.55
```

The unit `step` function maps $(\lambda x \to \textbf{if } x > 0 \textbf{ then } 1 \textbf{ else } 0)$ on a vector or matrix:

```
ghci> step (m - 0.5)
```

```
(3><4)
 [ 1.0, 1.0, 1.0, 1.0
 , 1.0, 0.0, 0.0, 1.0
 , 1.0, 0.0, 0.0, 1.0 ]
```

The function $cond\ a\ b\ lt\ eq\ gt$ is a vectorized form of $\textbf{case}\ compare\ a\ b\ \textbf{of}\ \{LT \to lt; EQ \to eq; GT \to gt\}$. The next expression creates a copy of `m` in which the elements equal or greater than 0.5 are replaced by the constant 3:

```
ghci> disp $ cond m 0.5 m 3 3
```

```
3x4
3.00  3.00  3.00  3.00
3.00  0.44  0.15  3.00
3.00  0.34  0.05  3.00
```

Similar to the behavior of arithmetic operators described above, arguments with any dimension = 1 are automatically expanded. For instance, using the following auxiliary definitions:

$$\begin{aligned} row &= asRow \circ fromList :: [Double] \to Matrix\ Double \\ col &= asColumn \circ fromList :: [Double] \to Matrix\ Double \end{aligned}$$

we can simulate a mask for the upper triangular part of a matrix:

```
ghci> disp $ cond (row[1..7]) (col[1..4]) 0 0 ((4><7)[1..])
```

```
4x7
0  2   3   4   5   6   7
0  0  10  11  12  13  14
0  0   0  18  19  20  21
0  0   0   0  26  27  28
```

Note that all five arguments to *cond* will be fully evaluated, not only the first two used in the comparisons. Vectors and matrices in this package are strict in all elements.

Finally, the function `find` returns the indexes of elements satisfying a predicate:

```
ghci> find (>0) (ident 3 :: Matrix Double)
```

```
[(0,0),(1,1),(2,2)]
```

This is useful in combination with *assoc* or *accum*. More examples of these functions can be found in `examples/bool.hs`.

3 Numeric functions

All functions not directly related to linear algebra are reexported by *Numeric.GSL*. Alternatively, we can import only selected modules or functions.

import *Numeric.GSL*
import *Numeric.GSL.Special*

3.1 Integration

The following function computes a numerical integral and the estimated error in the result:

$$quad = integrateQNG\ 1\ E - 6$$
$$f\ x = 4\ /\ (1 + x \uparrow 2)$$

```
ghci> quad f 0 1
```

```
(3.141592653589793,3.487868498008632e-14)
```

A multiple integral can be easily defined using Haskell's higher order functions:

$$quad1\ f\ a\ b = fst\ \$\ integrateQAGS\ 1\ E - 9\ 100\ f\ a\ b$$

$$quad2\ f\ a\ b\ g1\ g2 = quad1\ h\ a\ b$$
$$\quad \textbf{where}\ h\ x = quad1\ (f\ x)\ (g1\ x)\ (g2\ x)$$

$$volSphere\ r = 8 * quad2\ (\lambda x\ y \to sqrt\ (r * r - x * x - y * y))$$
$$\quad 0\ r$$
$$\quad (const\ 0)\ (\lambda x \to sqrt\ (r * r - x * x))$$

```
ghci> volSphere 2.5
```

```
65.44984694978737
```

```
ghci> 4/3*pi*2.5**3
65.44984694978736
```

3.2 Differentiation

Here is an example of numerical differentiation:

$$\mathit{deriv}\ f = \mathit{fst} \circ \mathit{derivCentral}\ 0.01\ f$$
$$\mathit{gaussian} = \mathit{deriv}\ (\lambda x \to 0.5 * \mathit{erf}\ (x\ /\ \mathit{sqrt}\ 2))$$

```
ghci> gaussian 0.5
0.35206532676400276

ghci> erf_Z 0.5
0.35206532676429947
```

(*erf_Z* is the gaussian probability density function provided by GSL).

3.3 Ordinary differential equations

The module *Numeric.GSL.ODE* provides the function *odeSolve*, which obtains a numeric solution for a system of differential equations. We must only supply the derivatives of the state vector, the initial value, and the desired times for the solution. It uses reasonable default parameters for the underlying numeric integration algorithm. (There is an alternative version in which the integration method and the parameters can be freely chosen. See the documentation for details.)

import *Numeric.GSL.ODE*
import *Numeric.LinearAlgebra*
import *Graphics.Plot* (*mplot*)

As a first example, we solve the differential equation for the harmonic oscillator with frequency ω and damping ratio δ:

$$\frac{d^2x}{dt^2} + 2\delta\omega\frac{dx}{dt} + \omega^2 x = 0$$

The equation must be converted to a system of first order equations. The derivative of the state vector is:

```
ghci> let harmonic w d t [x,v] = [v, -w^2*x -2*d*w*v]
```

We ask for the solution at 100 regularly spaced points between $t_0 = 0$ and $t_1 = 20$:

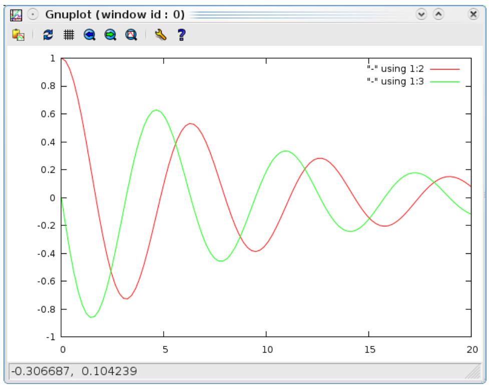
```
ghci> let ts = linspace 100 (0,20::Double)
```

Then we compute the solution for $\omega = 1$ and $\delta = 0.1$ starting from $x = 1$ and $dx/dt = 1$:

```
ghci> let sol = odeSolve (harmonic 1 0.1) [1,0] ts
```

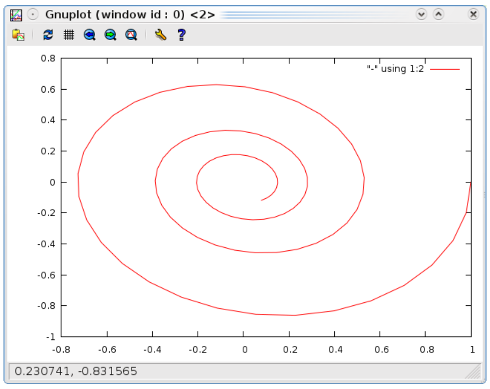
The rows of matrix `sol` are the estimated state vectors at the requested times. We can plot each variable against time using `mplot`:

```
ghci> mplot (ts: toColumns sol)
```



The path in the phase space is:

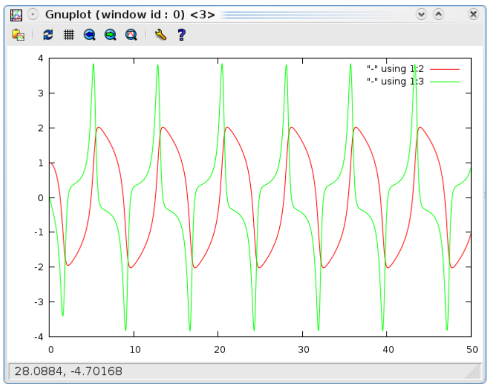
```
ghci> mplot (toColumns sol)
```

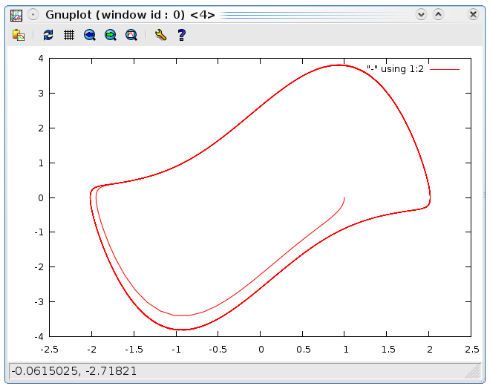


The following code shows the behaviour of the van der Pol oscillator:

```
vanderpol mu = do
  let xdot mu t [x, v] = [v, −x + mu ∗ v ∗ (1 − x ↑ 2)]
      ts = linspace 1000 (0, 50)
      sol = toColumns $ odeSolve (xdot mu) [1, 0] ts
  mplot (ts : sol)
  mplot sol
```

```
ghci> vanderpol 2
```

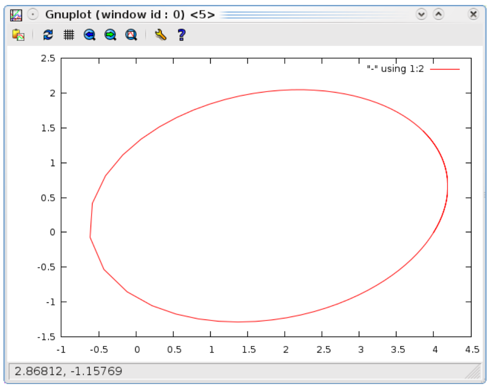




Finally, a simple newtonian orbit can be obtained by the following code:

```
kepler v a = mplot (take 2 $ toColumns sol) where
  xdot t [x, y, vx, vy] = [vx, vy, x ∗ k, y ∗ k] where
    g = 1
    k = (−g) ∗ (x ∗ x + y ∗ y) ∗∗ (−1.5)
  ts = linspace 100 (0, 30)
  sol = odeSolve xdot [4, 0, v ∗ cos (a ∗ degree), v ∗ sin (a ∗ degree)] ts
  degree = pi / 180
```

```
ghci> kepler 0.3 60
```



3.4 Iterative minimization

The optimization functions work with functions of several variables represented by lists of *Double*. The next example uses the Nelder-Mead method, which does not require the gradient:

```
import Numeric.GSL
import Numeric.GSL.Special
import Graphics.Plot (mplot)
import Text.Printf (printf)
import Numeric.Container (toColumns, format)


disp d x = putStrLn ∘ format "   " (printf ("%." ⧺ show d ⧺ "f")) $ x


f [x, y] = 10 ∗ (x − 1) ↑ 2 + 20 ∗ (y − 2) ↑ 2 + 30


minimizeS f xi = minimize NMSimplex2 1 E − 2 100 (replicate (length xi) 1) f xi


(s, p) = minimizeS f [5, 7]
```

The approximate solution is found at:

```
ghci> s
[0.9920430849306288,1.9969168063253182]
```

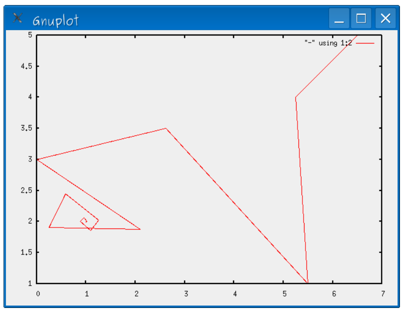
The optimization path, including objective function values and size of the search region, is:

```
ghci> disp 3 p
```

```
 1.000  512.500  1.130  6.500  5.000
 2.000  290.625  1.409  5.250  4.000
 3.000  290.625  1.409  5.250  4.000
 4.000  252.500  1.409  5.500  1.000
 5.000  101.406  1.847  2.625  3.500
 6.000  101.406  1.847  2.625  3.500
 7.000   60.000  1.847  0.000  3.000
 8.000   42.275  1.321  2.094  1.875
 9.000   35.684  1.069  0.258  1.906
10.000   35.664  0.841  0.588  2.445
11.000   30.680  0.476  1.258  2.025
12.000   30.680  0.367  1.258  2.025
13.000   30.539  0.300  1.093  1.849
14.000   30.137  0.172  0.883  2.004
15.000   30.137  0.126  0.883  2.004
16.000   30.090  0.106  0.958  2.060
17.000   30.005  0.063  1.022  2.004
18.000   30.005  0.043  1.022  2.004
19.000   30.005  0.043  1.022  2.004
20.000   30.005  0.027  1.022  2.004
21.000   30.005  0.022  1.022  2.004
22.000   30.001  0.016  0.992  1.997
23.000   30.001  0.013  0.992  1.997
24.000   30.001  0.008  0.992  1.997
```

Since this is a 2D problem we can show a graphical representation of the path using *mplot*:

```
ghci> mplot $ drop 3 (toColumns p)
```



The conjugate gradient method needs the gradient of the function. We can compare the solution to the above minimization problem using the true gradient and a numerical approximation.

$$minimizeC = minimizeD\ ConjugateFR\ 1\ E - 3\ 30\ 1\ E - 2\ 1\ E - 4$$

$$df\ [x, y] = [20 * (x - 1), 40 * (y - 2)]$$

$$\begin{array}{l} gradient\ f\ v = [partialDerivative\ k\ f\ v \mid k \leftarrow [0\mathinner{\ldotp\ldotp} length\ v - 1]]\ \textbf{where} \\ \quad partialDerivative\ n\ f\ v = fst\ (derivCentral\ 0.01\ g\ (v\ !!\ n))\ \textbf{where} \\ \quad\quad g\ x = f\ (a +\!\!+ x : b) \\ \quad\quad (a, \_ : b) = splitAt\ n\ v \end{array}$$

The solution with the true gradient:

```
ghci> fst $ minimizeC f df [5,7]
```

```
[0.9999999999999999,1.9999999999999998]
```

and with a numeric estimation of the gradient:

```
ghci> fst $ minimizeC f (gradient f) [5,7]
```

```
[1.000000000000031,2.000000000000102]
```

Different optimization methods are compared in the program `examples/minimize.hs` distributed in the package.

3.5 General root finding

Nonlinear systems of equations are numerically solved in a similar way:

$$rosenbrock\ a\ b\ [x, y] = [a * (1 - x), b * (y - x \uparrow 2)]$$

```
ghci> fst $ root Hybrids 1E-7 30 (rosenbrock 1 10) [-10,-5]
```

```
[1.0,1.0]
```

3.6 Roots of polynomials

The function $polySolve :: [Double] \to [Complex\ Double]$ admits the coefficients of a real polynomial in ascending order and returns all its complex roots. For instance, the five fifth roots of unity (solutions of $x^5 - 1 = 0$) are:

$$pol = [-1, 0, 0, 0, 0, 1.0]$$

```
ghci> polySolve pol
```

```
[(-0.8090169943749472) :+ 0.5877852522924731,(-0.8090169943749472) :+
(-0.5877852522924731),0.30901699437494756 :+ 0.9510565162951535,0.3090
1699437494756 :+ (-0.9510565162951535),1.0000000000000002 :+ 0.0]
```

We can check the result:

$$polyEval\ cs\ x = foldr\ (\lambda c\ ac \to ac * x + (c :+ 0))\ 0\ cs$$

```
ghci> map (polyEval pol) (polySolve pol)
```

```
[(-6.661338147750939e-16) :+ (-4.996003610813204e-16),(-6.661338147750
939e-16) :+ 4.996003610813204e-16,(-1.1102230246251565e-16) :+ (-6.106
226635438361e-16),(-1.1102230246251565e-16) :+ 6.106226635438361e-16,1
.1102230246251565e-15 :+ 0.0]
```

3.7 Special functions

The package `hmatrix-special` includes automatic bindings to more than 200 GSL special functions (usually in two variants, with and without error estimate).

```
ghci> gamma 5
```

```
24.0
```

```
ghci> synchrotron_1_e 5
```

```
(2.1248129774981993e-2,5.4485809058681077e-17)
```

3.8 Linear programming

The package `hmatrix-glpk` provides an interface to the simplex algorithm for optimization of a linear function subject to linear constraints. It is available from Hackage using `cabal install`, and requires `libglpk-dev`. Usage examples are provided in the online documentation.

4 Examples

4.1 Least squares data fitting

We would like to estimate a polynomial model for a set of 2D observations. This can be easily done by the least squares solution of an overconstrained linear system. Suppose that the observations are contained in a plain text data file `data.txt` like this:

```
 0.9    1.1
 2.1    3.9
 3.1    9.2
 4.0   51.8
 4.9   25.3
 6.1   35.7
 7.0   49.4
 7.9    3.6
 9.1   81.5
10.2   99.5
```

(it is a noisy quadratic dependency with outliers). First we read the file:

```
ghci> import Numeric.LinearAlgebra

ghci> dat <- fmap readMatrix $ readFile "data.txt"
ghci> dat

(10><2)
 [  0.9,  1.1
 ,  2.1,  3.9
 ,  3.1,  9.2
 ,  4.0, 51.8
 ,  4.9, 25.3
 ,  6.1, 35.7
 ,  7.0, 49.4
 ,  7.9,  3.6
 ,  9.1, 81.5
 , 10.2, 99.5 ]

ghci> let [x,b] = toColumns dat
```

Now we build the coefficient matrix with powers of `x`:

```
ghci> let a = fromColumns $ map (x^) [0..3]
ghci> let disp = putStr . dispf 3
ghci> disp a

10x4
1.000   0.900    0.810     0.729
1.000   2.100    4.410     9.261
1.000   3.100    9.610    29.791
1.000   4.000   16.000    64.000
1.000   4.900   24.010   117.649
1.000   6.100   37.210   226.981
1.000   7.000   49.000   343.000
1.000   7.900   62.410   493.039
1.000   9.100   82.810   753.571
1.000  10.200  104.040  1061.208
```

Note that `x^0` produces a singleton `1::Vector Double` which is automatically expanded to match the dimension of the other columns. The coefficients of the polynomial are given by the solution to the linear system:

```
ghci> a <\> b

fromList [-35.80526354723337,38.50215935012768,-7.665077852195382,0.5103332989916711]
```

The operator $< \backslash >$ represents multiplication by the (pseudo)inverse, implemented efficiently. We can define the polynomial as a regular Haskell function:

```
ghci> let polyEval cs x = foldr (\c ac->ac*x+c) 0 cs

ghci> let f = polyEval $ toList (a <\> b)

ghci> f 2.5

20.51735604860955
```

Normal Haskell functions can be mapped into vectors:

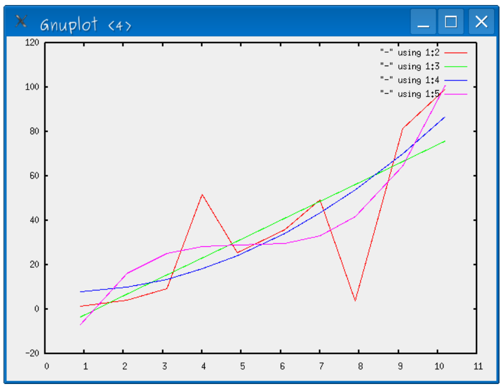
```
ghci> mapVector f (linspace 5 (0,1))

fromList [-35.805263547233366,-26.65081711766691,-18.406661672844415,-
11.024953465985398,-4.457848750309399]
```

Alternatively, if the function is to be applied to many data points, we can create a function which directly admits vectors and efficiently evaluates the polynomial on all elements. This is illustrated in the next program, which also graphically compares models of several degrees.

```
import Numeric.LinearAlgebra
import Graphics.Plot
import Text.Printf (printf)


expand :: Int → Vector Double → Matrix Double
expand n x = fromColumns $ map (x↑) [0 .. n]


polynomialModel :: Vector Double → Vector Double → Int → (Vector Double → Vector Double)
polynomialModel x y n = f where
  f z = expand n z <> ws
  ws = expand n x <\> y


main = do
  [x, y] ← (toColumns ∘ readMatrix) `fmap` readFile "data.txt"
  let pol = polynomialModel x y
  let view = [x, y, pol 1 x, pol 2 x, pol 3 x]
  putStrLn $ "   x        y        p 1      p 2      p 3"
  putStrLn $ format "   " (printf "%.2f") $ fromColumns view
  mplot view
```

```
ghci> main
```

```
  x       y       p 1     p 2     p 3
 0.90    1.10   -3.41    7.70   -6.99
 2.10    3.90    6.83    9.80   15.97
 3.10    9.20   15.36   13.39   25.09
 4.00   51.80   23.04   18.05   28.22
 4.90   25.30   30.72   24.05   28.86
 6.10   35.70   40.96   34.16   29.68
 7.00   49.40   48.64   43.31   33.17
 7.90    3.60   56.32   53.82   41.60
 9.10   81.50   66.57   69.92   64.39
10.20   99.50   75.95   86.80  101.01
```



4.2 Principal component analysis

The following program shows a graphical representation of the eigenvectors of the distribution of images of ‘4’ in the MNIST database of handwritten digits. For simplicity we will use only the first 5000 samples of the database in ASCII format:

```
$ wget -nv http://dis.um.es/~alberto/material/sp/mnist.txt.gz
$ gunzip mnist.txt.gz
```

```
import Numeric.LinearAlgebra
import Graphics.Plot

mean a = constant (recip ∘ fromIntegral ∘ rows $ a) (rows a) <> a
cov x = (trans xc <> xc) / fromIntegral (rows x − 1)
  where xc = x − asRow (mean x)

splitEvery k [] = []
splitEvery k l = take k l : splitEvery k (drop k l)

main = do
  m ← loadMatrix "mnist.txt"
```

Extract image vectors and labels:

```
  let xs = toRows $ takeColumns (cols m − 1) m
  let cs = toList $ flatten $ dropColumns (cols m − 1) m
```

Select the images of digit '4':

$$\textbf{let}\ g = \mathit{fromRows}\ [x \mid (x, c) \leftarrow \mathit{zip}\ \mathit{xs}\ \mathit{cs}, c \equiv 4]$$

Show a few images:

$$\textbf{let}\ \mathit{images} = \mathit{map}\ (\mathit{reshape}\ 28 \circ \mathit{negate})\ (\mathit{toRows}\ g)$$
$$\mathit{imshow}\ \$\ \mathit{fromBlocks} \circ \mathit{splitEvery}\ 4 \circ \mathit{take}\ 16\ \$\ \mathit{images}$$

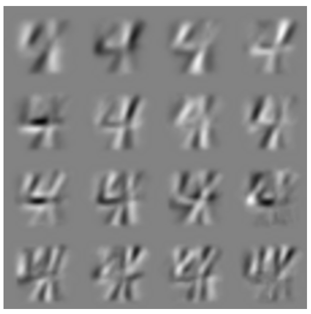
Compute the eigenvectors of the distribution:

$$\textbf{let}\ v = \mathit{toColumns}\ \$\ \mathit{snd}\ \$\ \mathit{eigSH}\ \$\ \mathit{cov}\ \$\ g$$

And show the 16 most important 'eigendigits':

$$\mathit{imshow}\ \$\ \mathit{fromBlocks}\ \$\ \mathit{splitEvery}\ 4\ \$\ \mathit{map}\ (\mathit{reshape}\ 28)\ \$\ \mathit{take}\ 16\ v$$

The program creates the following images:



4.3 Golden ratio

Linear recursive sequences can be expressed as an iterated linear transformation. For example, the Fibonacci sequence $x_k = x_{k-1} + x_{k-2}$ corresponds to the following transformation:

$$\begin{bmatrix} x_k \\ x_{k-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_{k-1} \\ x_{k-2} \end{bmatrix}$$

Therefore the n-th term is just a matrix power:

$$\begin{bmatrix} x_n \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} x_1 \\ x_0 \end{bmatrix}$$

This can be computed in closed form through diagonalization. We first write a function to create the coefficient matrix of a linear transformation.

$$\textbf{import}\ \mathit{Numeric.LinearAlgebra}$$
$$\textbf{import}\ \mathit{Control.Arrow}\ ((***))$$

$$\mathit{fromMap}\ n\ f = \mathit{fromRows} \circ f \circ \mathit{toRows}\ \$\ \mathit{ident}\ n :: \mathit{Matrix}\ \mathit{Double}$$

The canonical basis is extracted from the identity matrix, and we can take advantage of the Num instance of *Vector* to define the linear transformation using normal arithmetic expressions. For the Fibonacci recurrence we can write:

$$\mathit{fibo}\ [\mathit{f1},\mathit{f0}] = [\mathit{f1} + \mathit{f0},\mathit{f1}]$$
$$m = \mathit{fromMap}\ 2\ \mathit{fibo}$$

```
ghci> m

(2><2)
 [ 1.0, 1.0
 , 1.0, 0.0 ]
```

This is not safe, since the user may supply a non linear function, which would produce an absurd matrix. In any case, a matrix power can be expressed in closed form by diagonalization.

```
ghci> eig m

(fromList [1.618033988749895 :+ 0.0,(-0.6180339887498949) :+ 0.0],(2><2)
 [ 0.8506508083520399 :+ 0.0, (-0.5257311121191335) :+ 0.0
 , 0.5257311121191335 :+ 0.0,    0.8506508083520399 :+ 0.0 ])
```

We have used the general eig, producing a complex result, because the transformation associated to an arbitrary linear recurrence need not be symmetric. In this case the result is real, so for simplicity we take the real part of the eigensystem:

$$(l, v) = (\mathit{fst} \circ \mathit{fromComplex} *** \mathit{fst} \circ \mathit{fromComplex})\ (\mathit{eig}\ m)$$

```
ghci> v <> diag l <> trans v

(2><2)
 [                1.0,      0.9999999999999998
 , 0.9999999999999998, -1.5281829621183185e-16 ]
```

So we can write

$$\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^n = \begin{bmatrix}0.851 & -0.526\\ 0.526 & 0.851\end{bmatrix}\begin{bmatrix}1.618 & 0.000\\ 0.000 & -0.618\end{bmatrix}^n\begin{bmatrix}0.851 & 0.526\\ -0.526 & 0.851\end{bmatrix}$$

and equivalently:

$$x_n = \begin{bmatrix}0.851 & -0.526\end{bmatrix}\begin{bmatrix}1.618 & 0.000\\ 0.000 & -0.618\end{bmatrix}^n\begin{bmatrix}0.851\\ -0.526\end{bmatrix}$$

which can be implemented as

$$\mathit{v1} = \mathit{head}\ (\mathit{toRows}\ v)$$
$$\mathit{fib}\ k = \mathit{v1} <> \mathit{diag}\ (l \uparrow k) < . > \mathit{v1}$$

```
ghci> map fib [1..10]

[1.0,1.9999999999999996,3.0,4.999999999999999,7.999999999999999,12.999
999999999998,20.999999999999996,33.99999999999999,54.99999999999986,8
8.99999999999999]
```

```
ghci> map (round.fib) [1..20]

[1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946]
```

The remarkable result that the Fibonacci sequence can be computed with a closed form expression involving the golden ratio (the first eigenvalue of the transformation) can be easily understood in this framework.

Finally note that Haskell automatically reduces a^0 to the singleton `(1><1) [1]` for any matrix size, so we have a problem with x_0:

```
ghci> fib 0
*** Exception: inconsistent dimensions in matrix product (1,2) x (1,1)
```

Perhaps we should extend the automatic conformability to matrix products, with the meaning that a singleton is interpreted as a scalar or a diagonal matrix with constant elements.

4.4 Multiplication operators

This package provides different multiplication operators depending on the dimensions of the arguments. We currently have four contractions: mXm, vXm, mXv, and $dot = (<.>)$. We also have $scale$ and the element-by-element product $(*)$, as well as $outer$ and $kronecker$. This is good because we can easily deduce the type of the arguments in the source code. Compiler messages for type errors are also more user friendly.

Alternatively, the operation of contraction, related to composition of linear transformations, is a general concept that could also be represented by a single overloaded symbol. For this reason we already provide $(<>)$, which works with $Vector$ and $Matrix$. Furthermore, contraction is associative, so it can be somewhat strange that we must use different symbols depending on the grouping of the arguments. For instance, if s is a scalar, v is a vector, and m is a matrix, the combined product $s\ v\ m\ v$ can be expressed equivalently as

$$scale\ s\ v <> m <.> v$$
$$scale\ s\ v <.> (m <> v)$$
$$s * (v <> m <.> v)$$

In this section we explore an alternative interface. There are four types of multiplication: scaling, contraction, outer product and element-by-element multiplication. The following table shows the combinations of scalar, vector and matrix with uncontroversial results:

	s	v	m
s	s	v	m
v	v		
m	m		

scale

	s	v	m
s			
v		s	v
m		v	m

contraction

	s	v	m
s	s	v	m
v	v	m	
m	m		

outer

	s	v	m
s	s		
v		v	
m			m

element-by-element

The element-by-element product is already available through the standard Num method $(*)$, which provides automatic conformability. A more general elementwise multiplication operator could be introduced to combine different types, but this functionality is probably better achieved by $(*)$ and explicit $asRow$ or $asColumn$.

The outer product is more interesting. The matrix-vector and matrix-matrix outer (tensor) products actually require 3D and 4D arrays, but we could provide a flattened 2D representation similar to the already available kronecker product, so we could define a general outer product operator $\otimes$ with the following results:

$\otimes$	s	v	m
s	s	v	m
v	v	m	m
m	m	m	m

We have $outer\ u\ v = kronecker\ (asColumn\ u)\ (asRow\ v)$, so for consistency we define $v \otimes m = kronecker\ (asColumn\ v)\ m$ and $m \otimes v = kronecker\ m\ (asRow\ v)$. If we want the opposite orientation we must insert explicit $asRow$ or $asColumn$. The outer product with a scalar works as a scaling. Other possibility is that all results of $\otimes$ are matrices, with $s \otimes v$ = row, $v \otimes s$ = column, and $s \otimes s$ = `(1><1)[s]`. The outer product is not associative. It has no problems of dimension conformability, and there is also no problem with the interpretation of singleton structures.

The scaling and contraction operations are defined on complementary arguments, so we could merge them into a single operator. It is common practice in mathematical notation to use concatenation both for contraction and scaling, and this is the convention used in some scientific packages like *Matlab/Octave*, although in other systems like *Mathematica* the `Dot` operator only evaluates the expression when the arguments are lists. We can define a generalized dot/scaling operator like this:

$\cdot$	s	v	m
s	s	v	m
v	v	s	v
m	m	v	m

In this case singletons should be interpreted as scalars so that the type of a numeric literal does not change the result: $(3 :: Double) \cdot v$ = $(3 :: Vector\ Double) \cdot v$.

An alternative is that the contraction is extended to scalars in the sense of an adaptable vector of constant elements. For instance, the sum of the rows of a matrix m could be directly written as $1 \cdot m$ instead of $constant\ 1\ (rows\ m) <> m$.

$\cdot$	s	v	m
s	s	s	v
v	s	s	v
m	v	v	m

In this case singleton vectors should be interpreted as adaptable constants, and singleton matrices must be scalars (equivalent to diagonal matrices). The design space is large.

One small problem of these operators using multiparameter type classes is that we often need explicit signatures for the arguments (something like $(1 :: Double) \cdot m$). We want clear code, which is a matter of personal taste, and static error checking. There is probably not a single best interface for all users and applications.

Some experimental code can be found in the file `examples/multiply.hs` included in the distribution.

5 Related Projects

easyVision

hmatrix is a core package of a Haskell system for fast prototyping of real-time computer vision and image processing applications. You can find more information and screenshots in the web page of the project [21].

hTensor

This is an experimental package for multidimensional arrays and simple tensor computations. In contrast with other libraries with ‘anonymous’ array dimensions indexed by integers, dimensions in this library have an ‘identity’ which is preserved in data manipulation. Dimensions are explicitly selected by name in expressions, and Einstein’s summation convention for repeated indices is automatically applied. See [22] for more information.

References

[1] `http://www.haskell.org`.

[2] `http://www.netlib.org/lapack`.

[3] `http://www.netlib.org/blas`.

[4] `http://www.gnu.org/software/gsl`.

[5] `http://www.haskell.org/ghc`.

[6] `http://www.octave.org`.

[7] `http://math-atlas.sourceforge.net`.

[8] `http://software.intel.com/en-us/intel-mkl`.

[9] `http://hackage.haskell.org/packages/hackage.html`.

[10] Mauricio C. Antunes. `http://hackage.haskell.org/package/bindings-gsl`.

[11] G.H. Golub and C.F. Van Loan. *Matrix Computations, 3rd ed.* John Hopkins Univ. Press, 1996.

[12] R. Hinze, A. Loeh, S. Wehr, and B. Smith. *lhs2TeX*. `http://people.cs.uu.nl/andres/lhs2tex`.

[13] Roman Leshchinskiy. `http://hackage.haskell.org/package/vector`.

[14] Vivian McPhail. `http://hackage.haskell.org/package/hstatistics`.

[15] Vivian McPhail. `http://hackage.haskell.org/package/hmatrix-gsl-stats`.

[16] Patrick Perry. `http://hackage.haskell.org/package/blas`.

[17] Patrick Perry. `http://github.com/patperry/lapack`.

[18] Reiner Pope. `http://hackage.haskell.org/package/hmatrix-static`.

[19] A. Ruiz. `http://hackage.haskell.org/package/hmatrix`.

[20] A. Ruiz. `https://github.com/albertoruiz/hmatrix`.

[21] A. Ruiz. `https://github.com/albertoruiz/easyVision`.

[22] A. Ruiz. Introduction to *hTensor*, 2010. `https://github.com/albertoruiz/hTensor`.