Introduction to \textit{hTensor}

Alberto Ruiz
University of Murcia, Spain
version 0.7.0, September 28, 2010

Contents

1 Introduction 1

2 Simple Arrays 3
  2.1 Array creation 3
  2.2 Automatic conformability 6
  2.3 Structure Manipulation 7
  2.4 Display options 7
  2.5 Transposition 9
  2.6 Index renaming 9

3 Array operations 10
  3.1 Arithmetic operations 10
  3.2 Contraction 12
  3.3 Tensor product 12

4 Tensors 17
  4.1 Change of basis 18

5 Exterior Algebra 21
  5.1 Dual 23
  5.2 Geometric constructions 23

6 Geometric Algebra 24
  6.1 Geometric constructions 24
  6.2 Complex numbers, quaternions, and sandwich product 25
  6.3 Inverse 26
  6.4 Outermorphism 26

7 Multidimensional linear systems 26
  7.1 Nonhomogeneous systems 27
  7.2 Homogeneous systems 29
  7.3 ‘Partially homogeneous’ systems 30
  7.4 Multilinear systems 33

1 Introduction

This document describes an experimental Haskell \footnote{1} package for multidimensional arrays, oriented to support simple tensor computations and multilinear algebra. In contrast with the standard approach based on ‘anonymous’ dimensions indexed by integers, array dimensions (indices) 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 \footnote{2} is
automatically applied. The library has a purely functional interface: arrays are immutable, and operations work on whole structures which can be assembled and decomposed using simple primitives. Two alternative array types are provided: simple arrays, in which contractions only require equal dimension, and tensors, whose indices are labeled as covariant or contravariant (subindex or superindex), and contractions can only be done on pairs of complementary indices. Another interesting feature is that arguments are automatically made conformant by replicating them along extra dimensions appearing in an operation.

Multidimensional arrays are built on top of hmatrix [16] so big dense arrays can in principle be efficiently processed (coordinates are internally stored in C arrays of Double or Complex Double elements, and contractions and tensor products are implemented using BLAS [3]). However, the library is still very experimental and not yet fully optimized, so it may be useful only for toy problems. Consistency of dimension sizes is currently checked at run time. Future work includes static checking of conformability and a GUI for diagrammatic tensor expressions [4, 17]. Bug reports, suggestions and contributions are welcome.

This document has been generated from literate Haskell sources by the excellent Ish2TeX preprocessor [13]. We use \eval and \perform to automatically insert the results of the computations.

The library requires the Glasgow Haskell Compiler ghc >= 6.10 [5], and can be installed from Hackage [6] using cabal-install [7]. Since it depends on hmatrix the development packages for GSL [8] and LAPACK [9] must be available. For example, in Ubuntu/Debian we need the following packages:

$$\text{sudo apt-get install libgsl0-dev liblapack-dev}$$
$$\text{cabal install hTensor}$$

All required dependencies will be automatically satisfied. See the website of the library for detailed installation instructions.

Motivation

The ‘named indices’ approach is motivated by the following considerations. A scalar-valued multilinear function of, for example, 3 arguments, denoted by \(M(\cdot,\cdot,\cdot)\), is represented by a 3-index array of coefficients \(m_{ijk}\). Each array dimension corresponds to an input ‘slot’ of the function. Given vectors \(a\), \(b\), and \(c\), the value \(M(a,b,c) = m_{ijk}a^ib^jc^k\) is the contraction of the input vectors along the appropriate dimensions of \(m\). Interestingly, we can apply only some of the inputs, obtaining a multilinear function with fewer arguments, and represented by an array of fewer dimensions. For example, \(M(\cdot,b,\cdot) = R(\cdot,\cdot) \sim m_{ijk}b^j = r_{ik}\). This is similar to partial application in functional programming. Also, composition of multilinear functions corresponds to contraction of the appropriate ‘modes’ of the coefficient arrays: if \(T(\cdot,\cdot)\) is a vector-valued function represented by the array \(t^p_{qr}\), the composition \(M(a,\cdot,T(\cdot,b)) = S(\cdot,\cdot)\) is represented by the 2-index array \(m_{ijk}a^ir^k_{qr}b^r = s_{jq}\). Partial application and composition of multilinear functions can be nicely precomputed into compact representations.

In this process of application and composition the array dimension associated to each function argument is not uniquely determined. Any ordering of elements can be chosen as long as we keep track of the correspondence between dimensions and arguments. Note that contraction of the first or last index of multidimensional arrays can be efficiently done using standard matrix products, while internal indices require auxiliary transpositions. If we impose some fixed convention for dimension ordering then many useless transpositions may be wasted in complex computations. This library explores the idea of giving explicit index names to array dimensions so standard algebraic tensor expressions can be easily translated to Haskell without worrying about the internal
layout of coordinates. This approach is in some sense similar to keyword arguments available in some programming languages. We try to keep syntax simple, but this kind of expressions with lots of indices rapidly becomes unreadable so a GUI for diagrammatic notation would be very useful.

This package was initially developed to support tensor computations arising in Multiple View Geometry \cite{12,11}. In this field there are many interesting multilinear relations whose covariant or contravariant nature must be respected and tracked down in complex expressions. This tutorial presents a general overview of $hTensor$. Applications to multiview geometry can be found in \cite{15}.

2 Simple Arrays

Simple multidimensional arrays are provided by the following modules:

```
import Numeric.LinearAlgebra.Array
import Numeric.LinearAlgebra.Array.Util
```

2.1 Array creation

The simplest function for array creation is `listArray`, which defines an array from a list of dimensions and a list of elements (ordered such that the last index runs ‘faster’). We will use an auxiliary operator to avoid ambiguity in the type of the elements:

```
infixl 9 #
ds # cs = listArray ds cs :: Array Double
```

For instance, the following expression defines a two-dimensional array with $2 \times 3$ elements:

```
a = [2,3] # [1,-3,2,0,4,7.8]
```

The `Show` instance produces an explicit representation that can be parsed back as an array.

```
ghci> a
index "1" [index "2" [1.0,-3.0,2.0], index "2" [0.0,4.0,7.8]]
```

but in most cases this flat representation is not very useful. Arrays can be more nicely displayed as nested tables using special formatting functions. We typically define convenient shortcuts:

```
sh x = putStrLn $ formatFixed 2 $ x
```

```
t = [2,3,4] # [1,1.5..]
```

```
ghci> sh t
1:2 x 2:3 x 3:3 x 4:4
2
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1.00 1.50 2.00 2.50</td>
<td>7.00 7.50 8.00 8.50</td>
<td>13.00 13.50 14.00 14.50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 3.00 3.50 4.00 4.50</td>
<td>9.00 9.50 10.00 10.50</td>
<td>15.00 15.50 16.00 16.50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.00 5.50 6.00 6.50</td>
<td>11.00 11.50 12.00 12.50</td>
<td>17.00 17.50 18.00 18.50</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1-------------------------------------------------------------------------------
2
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>19.00 19.50 20.00 20.50</td>
</tr>
<tr>
<td>3 21.00 21.50 22.00 22.50</td>
</tr>
<tr>
<td>23.00 23.50 24.00 24.50</td>
</tr>
</tbody>
</table>
```

The output includes information about array structure, and index names are shown at the top and left sides of all nested tables.
The default index names produced by listArray are successive integers, but they can be easily changed using simple operators. The next definition uses the explicit renaming operator (⊿), written as (!>).

\[ t2 = [3, 4, 5] \# [1..] \triangleright "2r 1k" \]

The name "2" is changed to "r" and "1" to "k". The remaining indices are not changed.

\[ ghci> \text{sh } t2 \]

\[
\begin{array}{ccc}
3 & 3 & 3 \\
1 & 22 & 23 & 24 & 25 \\
31 & 32 & 33 & 34 & 35 \\
36 & 37 & 38 & 39 & 40 \\
41 & 42 & 43 & 44 & 45 \\
46 & 47 & 48 & 49 & 50 \\
51 & 52 & 53 & 54 & 55 \\
56 & 57 & 58 & 59 & 60 \\
\end{array}
\]

The decimal points are not shown by formatFixed if all elements are (approximate) integers. Other formatting utilities will be described later.

The elements of an array can be expressed as a function of index values. The first index is zero.

\[ q = \text{mkFun} [2, 4, 3] \$ \lambda[i, j, k] \to \text{fromIntegral} (i + j) \uparrow k \]

\[ ghci> \text{sh } \$ q!"ijk" \]

\[
\begin{array}{ccc}
i & i & i \\
1 & 2 & 3 \\
1 & 2 & 4 \\
1 & 3 & 9 \\
1 & 4 & 16 \\
\end{array}
\]

The default index names are again successive integers, but we have printed \( q \) using the direct renaming operator (!), which changes all index names, in lexicographical order, by new single-letter names given in a String. Therefore, in this case, \( q!"ijk" \) is equivalent to \( q \triangleright "1i 2j 3k" \).

Arrays can also be created from association lists. The default element is zero.

\[ r = \text{mkAssoc} [2, 3, 4] [[(0, 1, 1), 2], [(1, 0, 3), -7]] !"abc" \]

\[ ghci> \text{sh } r \]

\[
\begin{array}{ccc}
\text{a} & \text{c} & \text{c} \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\end{array}
\]

An alternative method for array creation is based on the function index, which assembles a new array from a list of (sub)arrays. Again we define an auxiliary operator to remove ambiguity in the element type:

\[ \text{Index names may contain more than one character, but some practical operators like (!) and (⊿) assume single-letter names. The reason is that a list of single-letter names can be given with low syntactic overhead as a simple string. In any case there are functions to deal with general names. See the documentation for details.} \]
\textbf{infixl 8 $-|$

\((-|) :: \text{Name} \rightarrow [\text{Array Double}] \rightarrow \text{Array Double}

\((-|) = \text{index}

\) Numeric literals are interpreted as 0th-order arrays (scalar values) in the \textit{Num} instance, so this operator can be directly used to build one dimensional arrays:

\texttt{ghci> sh \$ "p" $-| [2,5,0,8]}

\texttt{p:4}

\texttt{P}

\texttt{2 5 0 8}

\) It may be useful to define shortcuts for frequently used index names:

\texttt{[i,j,k] = map (-|) ["i", "j", "k"]}

so we can write this kind of expressions:

\texttt{b = i [}
\texttt{j [2,3,5],}
\texttt{j [10,0,87]]}

\texttt{ghci> sh b}

\texttt{i:2 x j:3}

\texttt{j}

\texttt{i 2 3 5}

\texttt{10 0 87}

\texttt{u = i \circ map j \circ map (map k) \$ [[[1,3],[5,7]],[[0,2],[1,1]]]}

\texttt{ghci> sh u}

\texttt{i:2 x j:2 x k:2}

\texttt{i}

\texttt{k | k}

\texttt{j 1 3 | j 0 2}

\texttt{5 7 | 1 1}

\texttt{m = k [b,2*b,0,1-b]}

\texttt{ghci> sh m}

\texttt{k:4 x i:2 x j:3}

\texttt{k}

\texttt{i 2 3 5 | j 4 6 10 | i 0 0 0 | i -1 -2 -4}

\texttt{10 0 87 | 20 0 174 | 0 0 0 | -9 1 -86}

\) The library also includes methods to directly create arrays from \textit{hmatrix}'s vectors and matrices.
2.2 Automatic conformability

The scalar values in the above definition for $m$ are interpreted as arrays of appropriate dimensions so that the expressions have sense. An important feature of this library is that array arguments are usually automatically made conformant. Consider the following expressions:

```
ghci> sh $ k [ i [1,2,3], i [10,20,30], 7 ]
k:3 x i:3
  i
    1 2 3
k 10 20 30
   7 7 7

ghci> sh $ k [ i [1,2,3], j [10,20,30], 7 ]
k:3 x i:3 x j:3
  j  | j  | j  
    1 1 1 | 10 20 30 | 7 7 7
i 2 2 2 | i 10 20 30 | i 7 7 7
   3 3 3 | 10 20 30 | 7 7 7
```

The ‘mental model’ for this behavior is that an array has the same value for all other indices not explicitly mentioned in its definition. Therefore,

```
ghci> sh $ i [1,2,3] + i [10,20,30]
i:3
  i
    11 22 33

and
ghci> sh $ i [1,2,3] + j [10,20,30,40]
i:3 x j:4
  j
    11 21 31 41
i 12 22 32 42
   13 23 33 43
```

Common indices do require equal dimensions:

```
ghci> sh $ i [1,2,3] + i [10,20,30,40]
*** Exception: makeConformantT with inconsistent dimensions ([i:3],[i:4])
k:4 x i:2 x j:3
  j  | j  | j  | j  
    i -1 -2 -4 | i 0 0 0 | i 4 6 10 | i 2 3 5
   -9 1 -86 | 0 0 0 | 20 0 174 | 10 0 87
```

Automatic conformability is very useful, but we must be careful because wrong index names may create big arrays instead of producing an exception.


2.3 Structure Manipulation

The library includes several functions for manipulation of array structure. We can extract the parts of an array at a given index, apply a function at a certain depth, etc.

A useful tool is \texttt{onIndex} (intended to be used in infix form), which processes the list of parts of an array in a selected dimension with a given function. For example:

\begin{verbatim}
ghci> sh $ (reverse `onIndex` "k") m

k:4 x i:2 x j:3

<table>
<thead>
<tr>
<th>j</th>
<th>j</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>i</td>
<td>-1</td>
<td>-2</td>
</tr>
<tr>
<td>j</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

ghci> sh $ onIndex (take 2) "k" m

k:2 x i:2 x j:3

<table>
<thead>
<tr>
<th>j</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>2</td>
</tr>
<tr>
<td>j</td>
<td>10</td>
</tr>
</tbody>
</table>

ghci> sh $ onIndex (i[2,-7]:) "j" m

k:4 x i:2 x j:4

<table>
<thead>
<tr>
<th>j</th>
<th>j</th>
<th>j</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>j</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>i</td>
<td>2</td>
<td>-1</td>
<td>-2</td>
</tr>
<tr>
<td>j</td>
<td>-7</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>i</td>
<td>-7</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>

Taking advantage of automatic conformability we can write:

\begin{verbatim}
homog name = (+[1])`onIndex` name
\end{verbatim}

\begin{verbatim}
ghci> sh $ homog "k" m

k:5 x i:2 x j:3

<table>
<thead>
<tr>
<th>j</th>
<th>j</th>
<th>j</th>
<th>j</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>j</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>10</td>
<td>0</td>
<td>87</td>
<td>20</td>
</tr>
</tbody>
</table>

ghci> sh $ homog "i" m

k:4 x i:3 x j:3

<table>
<thead>
<tr>
<th>j</th>
<th>j</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>j</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

ghci> sh $ homog "i" m

k:4 x i:3 x j:3

<table>
<thead>
<tr>
<th>j</th>
<th>j</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>j</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

2.4 Display options

The format of the display tables can be customized. For instance, index labels can be omitted:
ghci> `sh . noIdx $ t`

2 x 3 x 3 x 4
1.00 1.50 2.00 | 2.50 | 7.00 7.50 8.00 8.50 | 13.00 13.50 14.00 14.50
3.00 3.50 4.00 | 4.50 | 9.00 9.50 10.00 10.50 | 15.00 15.50 16.00 16.50
5.00 5.50 6.00 | 6.50 | 11.00 11.50 12.00 12.50 | 17.00 17.50 18.00 18.50
--------------------------------------------------------------------------
19.00 19.50 20.00 20.50 | 25.00 25.50 26.00 26.50 | 31.00 31.50 32.00 32.50
21.00 21.50 22.00 22.50 | 27.00 27.50 28.00 28.50 | 33.00 33.50 34.00 34.50
23.00 23.50 24.00 24.50 | 29.00 29.50 30.00 30.50 | 35.00 35.50 36.00 36.50

There is an autoscaled format:
ghci> `putStrLn . formatScaled 1 . noIdx $ t`

(2 x 3 x 3 x 4) E1
0.1 0.2 0.3 | 0.7 0.8 0.9 | 1.3 1.4 1.5
0.3 0.4 0.5 | 0.9 1.0 1.1 | 1.5 1.6 1.7
0.5 0.6 0.7 | 1.1 1.2 1.3 | 1.7 1.8 1.9
--------------------------------------------------------------------------
1.9 2.0 2.1 | 2.5 2.6 2.7 | 3.1 3.2 3.3
2.1 2.2 2.3 | 2.7 2.8 2.9 | 3.3 3.4 3.5
2.3 2.4 2.5 | 2.9 3.0 3.1 | 3.5 3.6 3.7

Arrays with an odd number of dimensions are formatted by default as a row of 2D structures:
ghci> `sh m`

k:4 x i:2 x j:3

<table>
<thead>
<tr>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
</tr>
<tr>
<td>i</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

This behavior can be modified by inserting a dummy index at the desired level. For instance, we can display m as a 'column of matrices':
ghci> `sh . dummyAt 1 $ m`

k:4 x i:2 x j:3

<table>
<thead>
<tr>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
</tr>
<tr>
<td>20</td>
</tr>
</tbody>
</table>

| k:4 x i:2 x j:3
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
</tr>
<tr>
<td>i</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
</tr>
<tr>
<td>-9</td>
</tr>
</tbody>
</table>

or as a 2D table of row vectors:
The arrays produced by formatting functions like \texttt{noIdx} or \texttt{dummyAt} are not intended to be used in other computations (this will be enforced in future versions).

### 2.5 Transposition

Index order can be changed using the transposition operator $\leftarrow$. 

transposing $\text{m}$:

```ghci
ghci> sh m
k:4 x i:2 x j:3

<table>
<thead>
<tr>
<th>j</th>
<th>j</th>
<th>j</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>-4</td>
<td>-9</td>
</tr>
</tbody>
</table>
```

Transposition is only useful to change the layout of coordinates for display. Dimensions are always identified by names, so the transposed structure is equivalent to the original one for all computational purposes.

### 2.6 Index renaming

Generally speaking, index renaming should always be based on explicit substitutions using $\leftarrow$. However, if we create an array from scratch we can also use direct renaming ($!$) since dimension roles and names are usually evident from the definition itself. Direct renaming can be safely used on symmetric arrays, or homogeneous antisymmetric arrays, since all dimensions are equivalent.

Mistakes in renaming may produce run-time errors (because dimensions are not conformable) or legal but unexpected results. It is difficult to prevent this kind of semantic errors.

---

2 Implementation of low level functions may require access to the ‘internal’ order of coordinates, but, by design, all high level array computations are independent of the physical layout of the array, which in general depends on previous operations and should not be exposed to the user.
Furthermore, in many cases we must work with arrays coming from other computations, or from an unspecified source, so index roles for our application cannot be easily inferred if the names are arbitrary. The recommendation is to set up a naming convention that must be satisfied by the caller of the function, or, perhaps better, to include the names of each particular dimension as additional arguments to the function.

As a final remark, remember that the direct renaming operator (!) must be used with care. Several successive renaming operations with ‘non increasing’ names may produce subtle errors:

$$\text{ghci} > \text{sh} \; q!"pijk"$$

\[
p:2 \times i:4 \times j:3
\]

\[
\begin{array}{ccc}
  & j & 1 \\
 i & 1  & 1  & 1  & 1 \\
 1  & 2  & 4  & 1  & 3  \\
 1  & 3  & 9  & 1  & 4  \\
\end{array}
\]

$$\text{ghci} > \text{sh} \; q!"pijk"!"pijk"$$

\[
j:2 \times p:4 \times i:3
\]

\[
\begin{array}{ccc}
  & i & 1 \\
p & 1  & 1  & 1 \\
1  & 2  & 4  & 1  & 3  \\
1  & 3  & 9  & 1  & 4  \\
\end{array}
\]

$$\text{ghci} > q!"pijk" == q!"pijk"!"pijk"$$

False

3 Array operations

3.1 Arithmetic operations

Arrays are instances of the standard Haskell numeric classes. In most operations arguments are automatically made conformant. They are replicated along all required extra dimensions to get a common structure.

$$\text{ghci} > \text{sh} \; m$$

\[
k:4 \times i:2 \times j:3
\]

\[
\begin{array}{ccc}
  & j & 1 \\
 i & 2  & 3  & 5  & 4  & 6  & 10  \\
 10  & 0  & 87  & 20  & 0  & 174  & 0  \\
\end{array}
\]

$$\text{ghci} > \text{sh} \; m + i \; [0,100]$$

\[
k:4 \times i:2 \times j:3
\]

\[
\begin{array}{ccc}
  & j & 1 \\
 i & 2  & 3  & 5  & 4  & 6  & 10  \\
 110  & 100  & 187  & 120  & 100  & 274  & 100  \\
\end{array}
\]

The symbol (\(\ast\)) is used for the tensor product (explained below), while elementwise multiplication
is denoted by (\times):

```
ghci> sh $ m .* j [1,0,-1]
k:4 x i:2 x j:3
```
```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>i</td>
<td>j</td>
</tr>
<tr>
<td>i 2</td>
<td>0</td>
<td>-5</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>-87</td>
</tr>
</tbody>
</table>

Renaming indices is the standard method to specify the desired dimensions in the operations. For example, given

\[
\begin{align*}
  w &= [2,2,2] \# [1..] \\
  c &= [2,2] \# [10,100,20,200]
\end{align*}
\]

```
ghci> sh w
```
```
1:2 x 2:2 x 3:2
1
3 | 3
2 1 2 | 2 5 6
3 4 | 7 8
```
```
ghci> sh c
```
```
1:2 x 2:2
2
1 10 100
20 200
```

we can add \(c\) to different ‘slices’ of \(w\):

```
ghci> sh $ w!"kij" + c"ij"
```
```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>j</td>
<td>j</td>
</tr>
<tr>
<td>i 11 102</td>
<td>i 15 106</td>
<td></td>
</tr>
<tr>
<td>23 204</td>
<td>27 208</td>
<td></td>
</tr>
</tbody>
</table>
```
```
ghci> sh $ w!"kij" + c"jk"
```
```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>j</td>
<td>j</td>
</tr>
<tr>
<td>i 11 22</td>
<td>i 105 206</td>
<td></td>
</tr>
<tr>
<td>13 24</td>
<td>107 208</td>
<td></td>
</tr>
</tbody>
</table>
```

General element-by-element operations on arrays are defined using `mapArray` and `zipArray`\(^3\).

\(^3\)They apply a function on `Vectors` to the flattened coordinates of the arrays. This is to take advantage of the `hmatrix` operations, internally implemented using optimized numeric libraries, but we can apply any general function on individual elements using `mapArray` (`mapVector f`).

11
3.2 Contraction

If an array is given the same name in two dimensions it is automatically contracted out. The indices with the common name ‘dissapear’ by summation of the diagonal subarrays at these levels. For example, given

\[
s = [3, 3, 3, 3] \# [1\ldots]!"ijkl"
\]

ghci> sh s

\[
i:3 \times j:3 \times k:3 \times l:3
\]

\[
\begin{array}{|c|c|c|}
\hline
1 & 2 & 3 \\
\hline
10 & 11 & 12 \\
\hline
19 & 20 & 21 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
k & 4 & 5 & 6 \\
\hline
k & 13 & 14 & 15 \\
\hline
k & 22 & 23 & 24 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
7 & 8 & 9 \\
\hline
16 & 17 & 18 \\
\hline
25 & 26 & 27 \\
\hline
\end{array}
\]

the contraction of the \(kl\) indices produces the following 2D array, with the trace of each block:

ghci> sh $ s!"ijkl"

\[
i:3 \times j:3
\]

\[
\begin{array}{|c|c|c|}
\hline
15 & 42 & 69 \\
\hline
96 & 123 & 150 \\
\hline
177 & 204 & 231 \\
\hline
\end{array}
\]

and the contraction of \(ij\) produces a ‘block matrix’ trace:

ghci> sh $ s!"ji"

\[
k:3 \times l:3
\]

\[
\begin{array}{|c|c|c|}
\hline
k & 58 & 59 & 60 \\
\hline
k & 61 & 62 & 63 \\
\hline
73 & 74 & 75 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
1 & 46 & 47 & 48 \\
\hline
k & 120 & 123 & 126 \\
\hline
k & 70 & 71 & 72 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
k & 55 & 56 & 57 \\
\hline
k & 61 & 62 & 63 \\
\hline
79 & 80 & 81 \\
\hline
\end{array}
\]

Two indices can be contracted out if they are compatible. In the case of the simple arrays considered in this section all indices are compatible with each other, and contractions only require equal dimension. (We get a runtime error if we perform the contraction of two indices with different number of elements).

3.3 Tensor product

Contractions can be performed on single arrays by renaming, but they are probably more useful in tensor products. The tensor product is just the normal product, but if the magnitudes depend on indices (i.e., they are arrays), then we get a result for all possible combinations of index values. For instance:
\[ x = \begin{bmatrix} 2 & 2 \end{bmatrix} \# \begin{bmatrix} 1 & 0 & 2 & -3 \end{bmatrix} !^{ij} \]
\[ y = \begin{bmatrix} 2 & 3 \end{bmatrix} \# \begin{bmatrix} 1 \end{bmatrix} !^{pq} \]

ghci> `sh x`
\[ i:2 \times j:2 \]
\[ j \]
\[ i \quad 1 \quad 0 \]
\[ 2 \quad -3 \]

ghci> `sh y`
\[ p:2 \times q:3 \]
\[ q \]
\[ p \quad 1 \quad 2 \quad 3 \]
\[ 4 \quad 5 \quad 6 \]

ghci> `sh $ x \times y`
\[ i:2 \times j:2 \times p:2 \times q:3 \]
\[ j \]
\[ q \]
\[ p \quad 1 \quad 2 \quad 3 \]
\[ 0 \quad 0 \quad 0 \]
\[ 4 \quad 5 \quad 6 \]
\[ 0 \quad 0 \quad 0 \]

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p 2 4 6</td>
<td>p -3 -6 -9</td>
</tr>
<tr>
<td>8 10 12</td>
<td>-12 -15 -18</td>
</tr>
</tbody>
</table>

We can see the result as the first array with all coordinates replaced by the second one, appropriately scaled.

Common indices in a tensor product are automatically contracted out. For example, if \( x \) and \( y \) are interpreted as matrices, the matrix product \( xy \) can be expressed as

ghci> `sh $ x !^{ik} \times y !^{kq}`
\[ i:2 \times q:3 \]
\[ q \]
\[ i \quad 1 \quad 2 \quad 3 \]
\[ -10 \quad -11 \quad -12 \]

and \( x^T y \) is

ghci> `sh $ x !^{kj} \times y !^{kq}`
\[ j:2 \times q:3 \]
\[ q \]
\[ j \quad 9 \quad 12 \quad 15 \]
\[ -12 \quad -15 \quad -18 \]

In standard matrix notation computations are implicitly defined by argument order and transposition. The index renaming scheme in this package has some advantages when working with multidimensional arrays, since the desired ‘modes’ can be selected in a more natural way.

Index names specify the level, ‘depth’, or ‘mode’ at which a transformation is applied. Given
the above $x$, interpreted as a transformation, and

$$d = [2, 2] \# [1..4]! "pq" \ast [2, 2] \# [1, 0, 0, 1]! "rs"$$

ghci> \texttt{sh d}

```
  p:2 x q:2 x r:2 x s:2
    q
    s | s
    r 1 0 | r 2 0
    0 1 0 | 0 2
-+----------
    s | s
    r 3 0 | r 4 0
    0 3 | 0 4
```

we can transform the inner matrices:

ghci> \texttt{sh $ (x!"ir" \ast d!"pqrs") \rightarrow "pqis"}$

```
  p:2 x q:2 x i:2 x s:2
    q
    s | s
    i 1 0 | i 2 0
    2 -3 | 4 -6
-+----------
    s | s
    i 3 0 | i 4 0
    6 -9 | 8 -12
```

or the bigger $2 \times 2$ block:

ghci> \texttt{sh $ x!"ip" \ast d!"pqrs"} \\

```
i:2 x q:2 x r:2 x s:2
    q
    s | s
    r 1 0 | r 2 0
    0 1 | 0 2
-+----------
    s | s
    r -7 0 | r -8 0
    0 -7 | 0 -8
```

Compare the tensor product ($\ast$) with element-by-element multiplication ($\ast$):

ghci> \texttt{sh $ x!"ik" \ast y!"kj"$} \\

```
i:2 x j:3
  j
  i 1 2 3
-10 -11 -12
```
The product function using \((\cdot \cdot)\) is called *outers*. It is useful to build outer products along the common indices, which are not contracted.

It looks like the result of a tensor product depends on the order of the arguments. Compare \(x \cdot y\) and \(y \cdot x\):

ghci> `sh $ (x !"ik" .* y !"kj") \"-> \"kij"

\[
\begin{array}{c|ccc}
 & i & 0 & 0 \\
\hline
j & 2 & 4 & 6 \\
\end{array}
\]

ghci> `sh $ x*y

\[
\begin{array}{c|ccc}
 & p & 0 & 0 & 0 \\
\hline
q & 1 & 2 & 3 \\
4 & 5 & 6 & 0 & 0 & 0 \\
\end{array}
\]

ghci> `sh $ y*x

\[
\begin{array}{c|ccc}
 & i & 0 & 0 \\
\hline
j & 2 & -3 & -6 \\
\end{array}
\]

However, the tensor product is (trivially) commutative:

ghci> `sh $ y*x-y*x

\[
\begin{array}{c|ccc}
 & p & 0 & 0 & 0 \\
\hline
q & 0 & 0 & 0 \\
\end{array}
\]

The display order of coordinates is the internal storage order, which depends on how the array has been created. It can be changed for human interpretation using the transposition operator \((\cdots)\). But the result of the algebraic operations defined in this library does not depend on the particular coordinate storage order of the arguments.
(The standard matrix product is not commutative because $AB$ means something like $A^j_k B^k_l = B^k_l A^j_k$, while $BA$ means $A^k_l B^l_j = B^l_j A^k_l$. The order of the matrices in the product implicitly selects the dimensions to be contracted out.)

Since dimensions are explicitly identified by name, argument order in a tensor product is irrelevant. However, even though the final result is numerically the same, the order chosen to perform a sequence of tensor products may have an important effect in the size of intermediate structures.

The function `smartProduct` is like `foldl1'` ($\ast$) with the argument list rearranged so that the intermediate arrays are as small as possible.

Many operations on arrays can be expressed in terms of tensor products and contractions. As a final example, we write a function to ‘center’ an ‘array of matrices’ by subtracting the mean, which can be computed by contraction with a constant vector:

$$\text{mean name } t = t \ast \text{name } - \left( \text{replicate } n \ v \right) \text{where}$$

$$n = \text{size name } t$$

$$v = \text{scalar } \circ \text{recip } \circ \text{fromIntegral } \$ \ n$$

We check this function with the following array `ms`, which can be interpreted as a collection, indexed by $k$, of $3 \times 3$ matrices:


ghci> sh ms

k:4 x i:3 x j:3

  k
  j | j | j | j
  1 2 3 | 2 4 6 | 3 6 9 | 4 8 12
i 4 5 6 | i 8 10 12 | i 12 15 18 | i 16 20 24
i 7 8 9 | i 14 16 18 | i 21 24 27 | i 28 32 36

ghci> sh (mean "k" ms)

i:3 x k:4

  j
  2.50 5.00 7.50
i 10.00 12.50 15.00
i 17.50 20.00 22.50

ghci> sh $ (ms - mean "k" ms)

k:4 x i:3 x j:3

  k
  j | j | j | j
  -1.50 -3.00 -4.50 | -0.50 -1.00 -1.50 | 0.50 1.00 1.50 | 1.50 3.00 4.50
i -6.00 -7.50 -9.00 | i -2.00 -2.50 -3.00 | i 2.00 2.50 3.00 | i 6.00 7.50 9.00
i -10.50 -12.00 -13.50 | i -3.50 -4.00 -4.50 | i 3.50 4.00 4.50 | i 10.50 12.00 13.50

We can center any other index:

ghci> sh (mean "j" ms $>"ik")

i:3 x k:4

  k
  2.00 4.00 6.00 8.00
i 5.00 10.00 15.00 20.00
i 8.00 16.00 24.00 32.00
The above operations automatically create and rearrange the arrays into consistent structures.

4 Tensors

So far the indices used to specify dimensions in array operations had all the same ‘neutral’ status, and contractions could always be performed along dimensions with the same size. This library also provides an alternative type of multidimensional array in which each dimension can be either a superindex (intended to represent a contravariant coordinate, one that transforms as a vector in a change of basis), or a subindex (intended to represent a covariant dimension, which transforms as a linear function). Contractions can only be performed on complementary indices.

There is actually nothing special about this new index type. We can define absurd computations that respect the consistency rule for contractions. The goal is that an exception is raised (or, in future versions, a compile-time error is produced) if by mistake we define tensor expressions which are inconsistent with the assumed mathematical role of the arguments.

\[
\text{import} \ \text{Numeric.LinearAlgebra.Tensor} \\
\text{import} \ \text{Numeric.LinearAlgebra (inv)} \\
\]

\[
sh \ t = \text{printA} "\%7.2f" \ t \\
\]

**Tensors** can be created from a list of parts using superindex and subindex, in a similar fashion as the function *index* used for simple arrays. The library also provides a few additional utilities. For instance, the function *vector* creates an one-dimensional array of *Double* with contravariant coordinates:

\[
x = \text{vector} \ [1, 2, 3] \\
\]

\[
\text{ghci>} \ x!"k" \ \\
\text{superindex} \ "k" \ [1.0, 2.0, 3.0] \\
\]

When we display *Tensor* objects we get information about index types:

\[
\text{ghci>} \ sh \ (x!"k") \\
k^3 \\
1.00 \quad 2.00 \quad 3.00 \\
\]

Analogously, *covector* defines a 1-D covariant array (subindex), which can be interpreted as a scalar-valued linear function:

\[
a = \text{covector} \ [2, -1, 3] \\
\]
ghci> \texttt{sh} $ \texttt{a!"k"}$

\[
\begin{pmatrix}
2.00 & -1.00 & 3.00 \\
\end{pmatrix}
\]

Since \(x^k\) and \(a_k\) have complementary indices we can contract them out. The expression \(x^k a_k\) is written as

ghci> \texttt{x!"k" * a!"k"}

scalar 9.0

However, we cannot directly contract \(x\) with itself:

\[
x!"k" * x!"k" = \star \star \star \text{Exception : wrong contraction2 : } [k \uparrow 3 \text{ and } k \uparrow 3]
\]

This requires an inner product or manual index rising (see below).

The function \texttt{transf} creates a 2D tensor with a superindex and a subindex:

\[
m = \texttt{transf} |
\begin{pmatrix}
[1,0,2], \\
[7,1,-3], \\
[4,4,0]
\end{pmatrix}
\]

ghci> \texttt{sh} $ m!"ij"$

\[
\begin{pmatrix}
1.00 & 0.00 & 2.00 \\
7.00 & 1.00 & -3.00 \\
4.00 & 4.00 & 0.00
\end{pmatrix}
\]

This tensor can be interpreted as a linear transformation of vectors (using as input the covariant ‘slot’). For example, \(m^i_j x_j\) can be obtained by

ghci> \texttt{sh} $ m!"ij"*x!"j"$

\[
\begin{pmatrix}
7.00 & 0.00 & 12.00
\end{pmatrix}
\]

It is also a transformation of covectors (using the contravariant slot). The value of \(a_i m^i_j\) is

ghci> \texttt{sh} $ m!"ij"*a!"i"$

\[
\begin{pmatrix}
7.00 & 11.00 & 7.00
\end{pmatrix}
\]

Again, ‘inappropriate’ tensor products are not allowed:

\[
m!"ij" * a!"j" = \star \star \star \text{Exception : wrong contraction2 : } [i \uparrow 3, j_3 \text{ and } j_3].
\]

4.1 Change of basis

The following figure illustrates the expansion of a vector in two different basis.
Tensors are abstract objects (multilinear functions) which require coordinates to work with, but which do not depend on the particular coordinate system chosen. Hence they must be transformed according to the proper rules of a change of basis. If a tensor expression reduces to a scalar value, this value will be independent on the particular basis chosen for the vector spaces. For example, we can check that 
\[ a_{ij}m^j_i \] (the value of function \( a \) on the image of \( x \) under \( m \)) is an invariant:

\[ m^! \ "ij" * a^! \ "i" * x^! \ "j" = scalar \ 50.0 \]

We will compute this value in a different basis. The canonical basis is:

\[ \begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix} = basisOf( x^! \ "j" ) \]

**ghci>** \texttt{sh es}

\[
\begin{bmatrix}
1.00 & 0.00 & 0.00 \\
0.00 & 1.00 & 0.00 \\
0.00 & 0.00 & 1.00
\end{bmatrix}
\]

Trivially, the coordinates of \( x \) in the canonical basis are:

**ghci>** \texttt{sh $ x!"e" * es}

\[
\begin{bmatrix}
1.00 & 2.00 & 3.00
\end{bmatrix}
\]

We define the new basis as a certain linear combination of the old one:

\[ [b1, b2, b3] = [e1 + e2, 2 * e3 - e1, (e2 - 3 * e3) / 2] \]

**ghci>** \texttt{sh bs}

\[
\begin{bmatrix}
1.00 & 1.00 & 0.00 \\
-1.00 & 0.00 & 2.00 \\
0.00 & 0.50 & -1.50
\end{bmatrix}
\]

The coordinates of a vector in the new basis are obtained with the inverse (hence the contravariant term) of the matrix \( bs \) containing the new basis in terms of the old one.

\[ ibs = applyAsMatrix inv bs \]
We check that the inverse is correct:

```
ghci> sh $bs!"bj" * ibs!"jk"
```


Then we apply the change of basis to vector $x$:

```ghci
ghci> sh x'
```

```
j3

2.20 1.20 -0.40
```

In contrast, the coordinates of $a$ in the new basis are obtained directly with $bs$ (linear functions are covariant):

```ghci
ghci> sh a'
```

```
i3

1.00 4.00 -5.00
```

Finally, the coordinates of the transformation $m$ in the new basis require $bs$ on the covariant index and the inverse $ibs$ in the contravariant one:

```ghci
ghci> sh m'
```

```
i3 x j3

6.80 -7.40 2.20
```

When all objects are expressed in the new basis we get the same result (this is obvious, since all changes of basis cancel out in the contractions):

```ghci
m' ! "ij" * a' ! "i" * x' ! "j" = scalar 49.99999999999999
```
As another example, consider a conic. It can be defined as the set of (homogeneous) vectors which are mapped to the scalar value 0 by a certain quadratic function (expressed as a bilinear function with both inputs connected to the same vector). The conic can be represented as a symmetric matrix, but it is actually a doubly covariant tensor, and therefore the equations for a change of basis are different from that of a transformation. Ordinary matrix notation uses the same kind of 2D table of numbers for representation of objects of very different nature. In some applications it is important keep this difference explicit. Furthermore, matrix notation cannot easily deal with multilinear functions of order greater than 2.

5 Exterior Algebra

Exterior (Grassmann) Algebra is a natural and powerful method for representation and manipulation of subspaces. The subspace \( S \) spanned by two vectors \( x \) and \( y \) can be expressed as a linear combination \( \alpha x + \beta y \) for any \( \alpha \) and \( \beta \). A disadvantage of this representation is that it is not unique, any other pair of linearly independent vectors in \( S \) can be used to represent the same subspace.

In contrast, the exterior (wedge) product obtains a representation of the subspace as an an-tysymmetric tensor (2-vector). For example,

```
import Numeric.LinearAlgebra.Exterior

sh t = putStrLn o formatFixed 2 $ t
fact n = product [1..n]

x = vector [3,4,7]
y = vector [0,0,5]

s = x ∧ y
```

```
ghci> sh s
1^3 x 2^3
 2
 0 0 15
1 0 0 20
-15 -20 0
```

All slots/indices of an (anti)symmetric tensor are equivalent except for a possible sign change. The \( r \)-vectors in \( \mathbb{R}^n \), also known as blades, are more economically expressed in a basis with only \( \binom{n}{r} \) elements:

```
ghci> asMultivector s
15{13} + 20{23}
```

This output is to be interpreted as \( s = 15 \, e_{13} + 20 \, e_{23} \). The exterior product is just the antisymmetrization of the ordinary tensor product. This process removes all linear dependencies, leaving just the ‘essence’ of the subspace. We obtain the same 2-vector from any other two independent vectors in the subspace: \( \text{vector } [3,4,-2] \wedge \text{vector } [6,8,7] = 33 \, e_{13} + 44 \, e_{23} \). The result only differs in the overall scale. This scaling factor is irrelevant for subspace representation, but it is of course very useful in many other applications. In fact, there is a norm for \( r \)-vectors, which
measures the area (in general the hypervolume) of the parallelogram defined by the component 1-vectors. For instance, \( \text{vector} \ [2, 700] \land \text{vector} \ [0, 3] = 6 \, e_{12} \). This norm can be defined in terms of an inner product between \( r \)-vectors / subspaces, computed as the contraction along all slots of both antisymmetric tensors (for simplicity we assume Euclidean metric).

\[
\text{norm } t = \sqrt{\text{inner } t \, t}
\]

ghci> \text{norm } s
scalar 25.0

This inner product is consistent with the standard inner product of (1-)vectors:

ghci> \text{inner } x \, y
scalar 35.0

And the norm can also be directly computed from the ‘vectorized’ coordinates:

ghci> \text{pnorm } \text{PNorm2} \ (\text{coords } s) \ / \ \sqrt{\text{fromIntegral} (\text{fact} (\text{order } s))})
25.0

The inner product for antisymmetric tensors/subspaces of different dimension is the orthogonal complement of the projection. For instance

ghci> \text{sh } \$ \text{inner} \ (\text{vector}[1,1,1]) \ (\text{vector}[1,0,0] \lor \text{vector}[0,1,0])
2^3
2
-1 1 0

The full antisymmetric tensor \( e_1 \land e_2 \land \ldots \land e_n \) represents the whole space. We can take a look at the explicit expansion of \( \epsilon^{ijk} \) and \( \epsilon^{pqrs} \):

ghci> \text{sh} \ (\text{leviCivita} \ 3!"ijk")
i^3 \times j^3 \times k^3

\[
\begin{array}{ccc|ccc|ccc}
\hline
k & k & k \\
0 & 0 & 0 & 0 & 0 & -1 & 1 & 1 \\
j & j & j \\
0 & 0 & 1 & 0 & 0 & 0 & j & -1 & 0 \\n0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{array}
\]

22
But it is better to show them in compact form: \( \text{asMultivector} \left( \text{leviCivita} \ 4 \right) = e_{1234} \). We can only work with explicit antisymmetric tensors of low dimension, since array size and computation time grow exponentially. See Geometric Algebra (Section 6).

### 5.1 Dual

Some objects are more economically represented in dual form. The dual is the inner product with the full space. If \( s = 15 \ e_{13} + 20 \ e_{23} \), then \( \text{dual} \ s = 20 \ e_{1} - 15 \ e_{2} \). Both objects have the same norm: \( \text{norm} \ s = \text{scalar} \ 25.0 \), and \( \text{norm} \ (\text{dual} \ s) = \text{scalar} \ 25.0 \). Dualization is an involution: \( \text{dual} \ (\text{dual} \ s) = 15 \ e_{13} + 20 \ e_{23} \). (The dimension of the space is deduced from the explicit antisymmetric tensor representation of \( s \).)

### 5.2 Geometric constructions

Exterior algebra is useful for geometric constructions in projective space with homogeneous representation. As an example, we will compute the intersection of a plane defined by three points \( a \), \( b \), and \( c \), and the line through points \( u \) and \( v \). We need a few auxiliary functions:

```haskell
-- create a homogeneous vector from ordinary coordinates
hvector xs = vector (xs ++ [1])

-- obtain inhomogeneous coordinates at a given dimension
inhomog n t = (onIndex init n) t / last (parts t n)

-- Extract to a list the inhomogeneous representation from a homogeneous vector
inh t
    | order t ≡ 1 = toList $ asVector $ inhomog (head (names t)) t
    | otherwise = error "inhomog requires a 1st order tensor"
```

Let us give some values to the points:

\[
\begin{align*}
a &= \text{hvector} \ [1,0,0] \\
b &= \text{hvector} \ [0,1,0] \\
c &= \text{hvector} \ [0,0,1]
\end{align*}
\]
\[ u = \text{hvector} \begin{bmatrix} 0, 0, 0 \end{bmatrix} \]
\[ v = \text{hvector} \begin{bmatrix} 1, 1, 1 \end{bmatrix} \]

The required result can be easily obtained using the ‘join’ (\( \land \)) and ‘meet’ (\( \lor \)) operations:
\[ p = a \land b \land c \]
\[ l = u \land v \]
\[ r = p \lor l \]

The plane is \( p = e_{123} + e_{124} - e_{134} + e_{234} \), or dual \( p = -e_1 - e_2 - e_3 + e_4 \), and the line is \( l = -e_{14} - e_{24} - e_{34} \). The intersection is \( r = e_1 + e_2 + e_3 + 3 e_4 \), whose ordinary coordinates are \( \text{inh} \ r = [0.3333333333333333, 0.3333333333333333] \).

We can take a look at line \( l \) as an antisymmetric tensor:
\[
\begin{bmatrix}
0 & 0 & 0 & -1 \\
0 & 0 & 0 & -1 \\
0 & 0 & 0 & -1 \\
1 & 1 & 1 & 0
\end{bmatrix}
\]

Its elements are closely related to Plucker coordinates of a 3D line:
\[
\text{plucker} :: \text{Tensor Double} \rightarrow [\text{Double}] \\
\text{plucker} \ r = [l_{12}, l_{13}, l_{14}, l_{23}, l_{42}, l_{34}] \text{ where}
\begin{align*}
[l_{12}, l_{13}, l_{14}] & = \text{toLists} (\text{asMatrix} \ r) \\
[l_{23}, l_{42}, l_{34}] & = \text{toLists} (\text{asMatrix} \ r)
\end{align*}
\]

See [17, 15] for applications of exterior algebra to visual geometry.

### 6 Geometric Algebra

The library includes a very basic and incomplete experimental implementation of *multivectors* [10]. A multivector is a sum of *blades* of different degree.

\[
\text{import} \ \text{Numeric.LinearAlgebra.Multivector}
\]

The geometric product \( ab = a \land b + a \cdot b \) is represented by \( \ast \). For instance, \( \text{vector} \ [1, 2, 3] \ast e_2 = 2 + e_{12} - 3 e_{23} \). Multivectors are shown and internally represented in the antisymmetric basis.

#### 6.1 Geometric constructions

We reproduce the geometric construction in the previous section. Homogeneous vectors in 3D space can be created simply adding \( e_4 \) to a normal vector, and the inhomogeneous representation can be retrieved with the help of the contractive inner product \( (\cdot\mid) \) :
\[
in h \ v = v / (v - e \ 4) - e \ 4
\]

The points are:

\[
a = vector \ [1, 0, 0, 1] \\
b = e \ 2 + e \ 4 \\
c = vector \ [0, 0, 1] + e \ 4
\]

\[
u = e \ 4 \\
v = vector \ [1, 1, 1] + e \ 4
\]

and the plane, line, and intersection point are obtained with the same meet and join operators, now working on multivectors:

\[
p = a \land b \land c \\
l = u \land v \\
r = p \lor l
\]

Both implementations obtain consistent results:

\[
\bullet \ p = e_{123} + e_{124} - e_{134} + e_{234} \\
\bullet \ l = -e_{14} - e_{24} - e_{34} \\
\bullet \ r = -e_1 - e_2 - e_3 - 3 \ e_4 \\
\bullet \ inh \ r = 0.333 \ e_1 + 0.333 \ e_2 + 0.333 \ e_3
\]

The inhomogeneous \( r \) is correct but we have a sign change in the meet operation for \( l \) with respect to the explicit antisymmetric tensor computation of the previous section. The problem here is that we have \( e_1 \land \text{full} \ 4 = e_{234} \) and \( (e_1 \land e_2) \lor (e_1 \land e_3) = e_1 \), but \( e_1 \land e_2 \lor \text{full} \ 4 = -e_{34} \) and \( e_1 \land e_2 \lor e_3 \lor \text{full} \ 4 = -e_4 \), etc. This coincides with the definition of \( \text{lcem} \) in the \texttt{gaviewer} program accompanying [10].

Note also that \( \text{full} \ 3 \land \text{full} \ 3 = -1 \), but \( \text{full} \ 4 \land \text{full} \ 4 = 1 \), which depends on the number of inversions required by the dimension of space. So the dual requires a reversion: \( \text{full} \ 3 \land \text{rever} (\text{full} \ 3) = 1 \), while undualization does not: \( (e_2 \lor \text{rever} (\text{full} \ 3)) \land \text{full} \ 3 = e_2 \). In the exterior algebra module described in the previous section dualization works as an involution in all dimensions.

### 6.2 Complex numbers, quaternions, and sandwich product

Complex numbers and quaternions are just a consequence of geometric algebra principles. For example, if we define

\[
i = e \ 1 \ast e \ 2
\]

we get \((3 + i) \ast (2 - i) = 7 - e_{12}\).

Many interesting geometric operations can be efficiently expressed using the ‘sandwich’ product. For example, we can rotate a whole line using a \textit{rotor}:

\[
rot = \text{rotor} \ 3 (pi / 4) (e \ 3) \quad \text{-- \ rotates an angle of pi/4 around the z axis, in R3.}
\]
rot = 0.924 − 0.383 e_{12}

\text{line} = \text{vector} [0, 0, 1] \land \text{vector} [1, 1, 1]
\text{rotatedline} = \text{rot} \ast \text{line} \ast \text{rever rot}

To check the rotation we define a function to compute two (inhomogeneous) points in a line:

twopts \ l = (inh \ l \lor (e_4 \land e_1 \land e_2),
inh \ (\text{vector} [0, 0, 1] \land \text{vector} [1, 0, 1] \land \text{vector} [0, 1, 1]) \lor l)

Then, \text{line} = -e_{14} - e_{24} - e_{34}, \text{twopts line} = (0, e_1 + e_2 + e_3), and \text{rotatedline} = -1.414 e_{24} - e_{34}, \text{twopts rotatedline} = (0, 1.414 e_2 + e_3).

6.3 Inverse

An important feature of geometric algebra is that some multivectors have inverses. For example,
\frac{1}{(2 + l + p \ast v)} = 0.105 - 0.082 e_{12} - 0.105 e_{1234} + 0.082 e_{13} + 0.029 e_{14} - 0.082 e_{23} + 0.029 e_{24} + 0.029 e_{34}, and therefore \frac{1}{(2 + l + p \ast v)} \ast (2 + l + p \ast v) = 1.

6.4 Outermorphism

Finally, linear transformation of multivectors are applied using apply. For example, if we define
\begin{align*}
f 1 &= 2 \ast e_1 + 3 \ast e_2 \\
f 2 &= e_3 - 2 \ast e_1 \\
f 3 &= e_1 + e_2 + e_3
\end{align*}

then

- apply f (vector [1, 2, 3]) = e_1 + 6 e_2 + 5 e_3.
- apply f (e_1 + e_3) = 3 e_1 + 4 e_2 + e_3.
- apply f (e_1 \land e_2) = 6 e_{12} + 2 e_{13} + 3 e_{23}.
- apply f (full 3) = 7 e_{123} (the determinant of f).
- apply (\lambda k \rightarrow 2 \ast e_k) (7 + (e_1 \land e_2)) = 7 + 4 e_{12} (isotropic scaling).

Interestingly, the linearCombinationOf type constructor is a Monad, so apply is closely related to the (\gg=) operator.

7 Multidimensional linear systems

The equation \(ax = b\) can be easily solved if \(a\) and \(b\) are scalar magnitudes, and basic linear algebra extends this problem to vectors and matrices. This library includes a number of functions to solve this equation for general multidimensional arrays. We allow both nonhomogeneous (\(b \neq 0\)) and homogeneous equations (\(b = 0\)), and also ‘partially’ homogeneous equations, in which equality is relaxed to proportionality in a selected dimension. Many interesting problems can be reduced to any of these simple forms. Let us see a few examples.
import Numeric.LinearAlgebra.Array as A
import Numeric.LinearAlgebra.Array.Util
import Numeric.LinearAlgebra.Array.Solve
import Numeric.LinearAlgebra as LA
import Numeric.LinearAlgebra.Tensor
import Numeric.LinearAlgebra.Array.Decomposition
import System.Random
import Text.Printf

infixl 9 #
(#) :: [Int] → [Double] → Array Double
(#) = listArray

sh x = putStrLn o formatFixed 2 $ x

norm x = sqrt (x * x)

For demonstration purposes we use random arrays to create full rank systems (later we will describe more realistic applications). For simplicity we use the standard random number generator

randomArray seed dims = dims # cs where
g = mkStdGen seed
cs = randomRs (-1, 1) g

7.1 Nonhomogeneous systems

By construction the following arrays verify the nonhomogeneous equation $ax = b$.

$a = randomArray 777 [2, 4, 3, 2] ! "ijkl"
x = randomArray 888 [4, 3, 2, 3] ! "pqlk"
b = a * x$

We can take a look at the array of coefficients $a$ and the right hand side of the equation $b$:

ghci> sh a

```text
i:2 x j:4 x k:3 x l:2

| 1 | 1 | 1 | 1 |
| 0.03 0.24 | -0.42 -0.65 | -0.54 0.65 | 0.49 -0.42 |
| 0.81 -0.15 | 0.16 -0.23 | 0.84 -0.22 | 0.38 0.32 |
| 0.27 -0.85 | 0.38 -0.60 | -0.68 -0.30 | 0.10 0.28 |

| 1 | 1 | 1 | 1 |
| -0.39 0.09 | 0.74 0.01 | 0.27 -1.00 | -0.76 0.21 |
| 0.21 -0.95 | 0.30 0.29 | 0.55 0.85 | 0.40 0.42 |
| 0.57 -0.67 | -0.16 0.24 | 0.61 -0.30 | 0.78 -0.34 |
```

4See also Data.Packed.Random in hmatrix and other random number generators available in Hackage.
### Solution of a Multidimensional Linear System

The solution of this multidimensional linear system is obtained by the function `solve`:

\[
xI = \text{solve } a \cdot b
\]

The solution satisfies the equation:

\[
x = \text{solve } a \cdot b
\]

And, since the system is full rank, the obtained solution \(xI\) is the same as the original \(x\) used to generate the problem: \(\text{norm } (xI - x) = \text{scalar} 3.080696979004693e-15\).

Depending on the dimensions of the arrays and the rank of the linear system that must be internally solved we may have underconstrained or overconstrained systems. The `solve` function obtains the minimum squared error solution of overconstrained systems, and the minimum norm solution of underconstrained systems.

The number of unknowns in the internal linear system arising from \(x = \text{solve } a \cdot b\) is the product of the sizes of the common dimensions in \(a\) and \(x\). The extra dimensions in \(a\) play the role of additional constraints for the solution, and the extra dimensions in \(b\) play the role of additional right-hand sides.

In the above example `dims a = [i : 2, j : 4, k : 3, l : 2]`, `dims x = [p : 4, q : 3, l : 2, k : 3]` and `dims b = [i : 2, j : 4, p : 4, q : 3]`. The solution \(x\) has 72 coordinates, but the common dimensions are \(k\) and \(l\), so we must solve a system with only 6 unknowns. The extra dimensions of \(a\) are \(i\)
and \( j \), so we have 8 equations. The extra dimensions of \( b \) are \( p \) and \( q \), so we have 12 right-hand sides. The \texttt{solve} function internally uses \texttt{linearSolveSVD} from \texttt{hmatrix}.

### 7.2 Homogeneous systems

To solve a multidimensional homogeneous system we must only provide the index names for the desired solution. For instance,

\[
x_2 = \text{solveH} \ a \ "k\ j"
\]

ghci> \texttt{sh x2}

\begin{verbatim}
k:3 x j:4
  j
-0.13 -0.06 -0.34  0.32
  k -0.05 -0.06 -0.31  0.16
-0.03  0.06 -0.01  0.79
\end{verbatim}

We get zeros in all remaining dimensions:

ghci> \texttt{sh \$ a*x2}

\begin{verbatim}
l:2 x i:2
  i
l -0.00 -0.00
  0.00 -0.00
\end{verbatim}

If the system is overconstrained the solution minimizes the squared error:

\[
x_3 = \text{solveH} \ a \ "j"
\]

ghci> \texttt{sh x3}

\begin{verbatim}
j:4
  j
-0.29 -0.83 -0.08 -0.47
\end{verbatim}

ghci> \texttt{sh \$ a*x3}

\begin{verbatim}
k:3 x l:2 x i:2
  k
  i |  i |  i
l 0.16 -0.16 | l 0.29 -0.16 | l -0.22 -0.45
0.61 -0.05 | 0.10 -0.22 | 0.64  0.19
\end{verbatim}

If the system is underconstrained \texttt{solveH} gives one of the solutions\textsuperscript{6}:

\[
x_4 = \text{solveH} \ a \ "i k l"
\]

\textsuperscript{6}The one associated to the last singular vector in the SVD decomposition of the coefficient matrix.

\textsuperscript{5}There is an alternative way (\texttt{solve' infoRank}) for calling \texttt{solve}, which prints in the console debugging information (sizes and rank) about the internal linear system. This may change in future versions.
ghci> **sh x4**

\[
i:2 \times k:3 \times l:2
\]

\[
\begin{array}{ccc|ccc}
1 & 1 \\
0.31 & -0.04 & -0.11 & -0.06 \\
k & 0.22 & 0.26 & k & 0.06 & 0.01 \\
0.01 & -0.21 & 0.30 & 0.79
\end{array}
\]

ghci> **sh $ a*x4**

\[
j:4
\]

\[
\begin{array}{cccc}
-0.00 & -0.00 & -0.00 & -0.00
\end{array}
\]

But we may ask for all of them using a more general function:

\[
x4s = \text{solveHomog} \ a \ ["i", "k", "l"] \ (\text{Left} \ \text{eps})
\]

There are length \( x4s = 8 \) solutions (an orthonormal basis) spanning the null-space of the array \( a \) along dimensions \( ikl \). Any of them (or any linear combination) solves the problem:

ghci> **sh $ (2*(x4s!!3) -5*(x4s!!6)) * a**

\[
j:4
\]

\[
\begin{array}{cccc}
-0.00 & 0.00 & -0.00 & 0.00
\end{array}
\]

See the documentation for details about the available functions for solving homogeneous systems.

### 7.3 ‘Partially homogeneous’ systems

This library includes functions to solve the multidimensional system \( ax = b \) in the case that one of the dimensions of \( b \) has homogeneous coordinates and equality is relaxed to proportionality in this dimension. This kind of systems are frequently found, for example, in computer vision applications, where we must estimate projective transformations (homographies, camera matrices, etc.) from input/output correspondences.

As a simple example, consider the following four points in the plane:

\[
tl = \text{subindex} \ "n" \circ \text{map} (\text{superindex} \ "x")
\]

\[
v = tl \ [[1, 2], [2, 4], [4, 5], [5, 1]] \ ! "nv"
\]

ghci> **sh v**

\[
n_4 \times v^2
\]

\[
\begin{array}{cccc}
v & 1 & 2 \\
n & 2 & 4 \\
4 & 5 \\
5 & 1
\end{array}
\]

Imagine that these points are the vertices of the image of a unit square. We are interested in a rectifying transformation which takes this points back to their ‘true’ locations:
\[ u = tl \left( \frac{[0, 0], [0, 1], [1, 1], [1, 0]}{\nu} \right) \]

ghci> \textbf{sh} u

\begin{verbatim}
 n_4 \times u^2
  u
  0 0
  n 0 1
  1 1
  1 0
\end{verbatim}

This cannot be done using a linear transformation, since there are more constraints than degrees of freedom (only two linearly independent arbitrary points can be transformed). However, using homogeneous coordinates we can get more flexible transformations.

\[
\text{homog}T \ n = (+[1])^{onIndex} \ n
\]
\[
\text{inhomog}T \ n \ t = (\text{init}^{onIndex} \ n) \ t \ / \ last (\text{parts} \ t \ n)
\]

\[ uh = \text{homog}T \ "u" \ u \]
\[ vh = \text{homog}T \ "v" \ v \]

ghci> \textbf{sh} vh

\begin{verbatim}
 n_4 \times v^3
  v
  1 2 1
  n 2 4 1
  4 5 1
  5 1 1
\end{verbatim}

Up to three arbitrary point correspondences can be exactly captured by a linear transformation (more precisely, an affine one, since coordinates are augmented with a constant). This transformation is the solution of a simple linear system:

\[ t3 = \text{take} \ 3^{onIndex} \ "n" \]
\[ m1 = \text{solve} (t3 \ vh) (t3 \ uh) \]

ghci> \textbf{sh} m1

\begin{verbatim}
 v_3 \times u^3
  u
  0.67 -0.33 -0.00
  v -0.33 0.67 0.00
  0.00 -1.00 1.00
\end{verbatim}

ghci> \textbf{sh} $ m1 \times t3 \ vh$

\begin{verbatim}
 u^3 \times n_3
  n
  0.00 0.00 1.00
  u -0.00 1.00 1.00
  1.00 1.00 1.00
\end{verbatim}

To do this we don’t actually need homogeneous coordinates in the righ-hand side:
\[ m_2 = solve (t3 \, vh) \, (t3 \, u) \]

ghci> sh m2

\[
\begin{array}{ccc}
0.67 & -0.33 \\
-0.33 & 0.67 \\
0.00 & -1.00 \\
\end{array}
\]

ghci> sh $ m2 * t3 vh

\[
\begin{array}{ccc}
0.00 & -0.00 & 1.00 \\
-0.00 & 1.00 & 1.00 \\
\end{array}
\]

In any case, four points cannot be perfectly transformed:

\[ m_3 = solve \, vh \, u \]

ghci> sh $ m3 * vh

\[
\begin{array}{cccc}
0.14 & 0.23 & 0.86 & 1.05 \\
-0.00 & 1.14 & -0.05 \\
\end{array}
\]

We have obtained the minimum squared error solution. An exact transformation of four arbitrary points requires a projective transformation (a linear transformation of homogeneous coordinates). It can be computed using \texttt{solveP} (which stands for ‘solve partially homogeneous system’, or ‘solve proportional’ or ‘solve projective’, etc.). We must indicate the index in the right-hand side for which equality is relaxed to proportionality.

\[ m = solveP \, vh \, uh \, "u" \]

ghci> sh m

\[
\begin{array}{ccc}
0.37 & 0.04 & 0.22 \\
-0.18 & 0.15 & -0.22 \\
0.00 & -0.33 & 0.77 \\
\end{array}
\]

The homogeneous coordinates are transformed as follows:

ghci> sh $ m * vh

\[
\begin{array}{cccc}
0.00 & 0.00 & 0.55 & 1.66 \\
-0.00 & 0.33 & 0.55 & 0.00 \\
0.55 & 0.33 & 0.55 & 1.66 \\
\end{array}
\]

which correspond to the expected points:
The function \( \text{solveP} \) internally solves a homogeneous system arising from the desired proportionality condition, implemented as an appropriate contraction with the Levi-Civita tensor. For simplicity and improved numeric stability we currently include all the equations, although most of them are redundant. Hence the homogeneous dimension for \( \text{solveP} \) must be small (not much larger than 5 or 6), because the number of equations grows exponentially.

7.4 Multilinear systems

This package includes some experimental functions to solve multilinear tensor equations of the type \( axyz \ldots = b \) (with two or more tensor unknowns \( x, y, z \ldots \)) using alternating least squares. For each solve function (\( \text{solve}, \text{solveH}, \text{solveP} \)) there is a corresponding multilinear version (\( \text{mlSolve}, \text{mlSolveH}, \text{mlSolveP} \)) which refines an initial solution of the system. These are also a few utilities (\( \text{solveFactors}, \text{cpAuto}, \text{etc.} \)) which automatically initialize and solve the system.

As an example, consider the following pseudorandom array:

\[
ra = \text{product} \left( \text{randomArrayRank} \ 7777 \ 3 \ [4, 4, 6, 5] \right) ! \ "ijlk"
\]

By construction it is actually a simple diagonal ‘core’ whose ‘modes’ have been randomly transformed. We would like to find a change of basis for each dimension which diagonalizes the array. This can be done with the CP decomposition [14]. The \( \text{cpAuto} \) function computes decompositions of increasing rank until one of them gives acceptable reconstruction error.

\[
sol@((\text{core} : \text{xs})) = \text{cp} \ ra
\]
We have been lucky and the optimization converged at the correct rank:

The transformations for each mode are:

```
ghci> sh (xs!!0)
1:3 x i:4
 i
-0.51 0.65 -0.56 0.08
 0.16 0.62 0.64 0.42
 0.59 0.36 -0.65 -0.32

ghci> sh (xs!!1)
2:3 x j:4
 j
0.75 0.11 0.33 0.56
-0.90 0.07 0.30 0.32
-0.38 0.78 0.48 0.11

ghci> sh (xs!!2)
3:3 x l:6
 l
-0.52 0.05 0.12 0.35 -0.61 0.47
 0.39 -0.51 -0.60 -0.06 0.23 0.41
-0.15 0.67 0.10 -0.32 0.52 0.37

ghci> sh (xs!!3)
4:3 x k:5
 k
0.73 0.00 -0.55 -0.03 0.40
0.32 0.47 -0.62 -0.10 0.54
-0.31 0.06 0.50 0.74 -0.32
```
We have used the \texttt{cpAuto} function with initialization based on the high-order SVD.

\[
\texttt{cp t = cpAuto (finit t) defaultParameters t where finit = cpInitSvd \circ \texttt{fst} \circ \texttt{hosvd'}}
\]

It is possible to compute the decomposition for a fixed rank. In this case we get the history of errors in the optimization. For rank 2 we get

\[
\texttt{cpRank r t = cpRun (cpInitSvd (fst \$ \texttt{hosvd'} t) r) defaultParameters t}
\]

\[\texttt{(_,- errs2) = cpRank 2 ra}\]

\begin{verbatim}
ghci> \texttt{putStr$ concatMap (printf "%.2g " ) (reverse errs2)}
80.00 29.77 26.64
\end{verbatim}

For rank 3 the optimization converges in length \texttt{errs} = 7 steps.

\[\texttt{(y, errs) = cpRank 3 ra}\]

\begin{verbatim}
ghci> \texttt{putStr$ concatMap (printf "%.1g " ) (reverse errs)}
80.8 13.2 1.5 0.2 3.5e-2 5.6e-3 9.8e-4
\end{verbatim}

The error tolerance in the default parameters is \texttt{epsilon defaultParameters = 1.0e \textasciitilde -3 \%}.

An alternative to the CP decomposition is the ‘High-Order SVD’, which decomposes the array as a reduced, non-sparse core and orthogonal transformations on each mode. It is often applied to multidimensional principal component analysis.

\[
\texttt{h:rs = hosvd ra}
\]

\begin{verbatim}
ghci> \texttt{map dims (h:rs)}

\[
[[1:3,2:3,3:3,4:3],[1:3,i:4],[2:3,j:4],[3:3,1:6],[4:3,k:5]]
\]

\[\begin{array}{cccc}
\text{ghci> sh h}
1:3 & 2:3 & 3:3 & 4:3 \\
\hline
4 & 4 & 4 & 4 \\
2.86 & -0.52 & 0.10 & -0.07 & 0.02 & -0.02 \\
3 & 0.41 & -0.14 & 0.02 & 3 & 0.27 & 0.05 & -0.03 & 3 & -0.00 & -0.00 & -0.04 \\
0.09 & -0.01 & -0.02 & -0.12 & -0.02 & 0.14 & -0.13 & -0.02 & 0.13 \\
\hline
\end{array}
\]

\[\begin{array}{cccc}
\hline
4 & 4 & 4 & 4 \\
1 & 0.19 & 0.27 & 0.02 & -0.70 & -0.31 & -0.02 & 0.09 & 0.05 & 0.02 \\
3 & -0.95 & -0.46 & -0.06 & 3 & 1.24 & 0.65 & 0.12 & 3 & -0.24 & -0.11 & 0.02 \\
-0.05 & -0.01 & 0.02 & 0.17 & 0.03 & -0.14 & 0.13 & 0.01 & -0.13 \\
\hline
\end{array}
\]

\[\begin{array}{cccc}
\hline
4 & 4 & 4 & 4 \\
-0.09 & 0.03 & 0.01 & 0.01 & -0.00 & -0.06 & 0.06 & 0.01 & -0.05 \\
3 & -0.07 & -0.02 & 0.02 & 3 & 0.17 & 0.04 & -0.13 & 3 & 0.11 & 0.01 & -0.12 \\
0.06 & 0.01 & -0.07 & -0.45 & -0.06 & 0.46 & -0.41 & -0.05 & 0.41 \\
\end{array}
\]

35
ghci> sh (rs!!3)

4:3 x k:5
k
0.63 0.13 -0.60 -0.09 0.46
4 -0.56 0.74 -0.19 -0.16 0.27
0.13 0.31 0.09 0.94 0.04

ghci> sh $ (rs!!3) * (rs!!3)!>"4p"

4:3 x p:3
p
1.00 0.00 0.00
4 0.00 1.00 -0.00
0.00 -0.00 1.00

ghci> norm (ra - product (h:rs))

scalar 3.182158964790961e-15

References

