Introduction to **RcppSimpleTensor**

Thibaut Lamadon and Florian Oswald
https://github.com/tlamadon/RcppSimpleTensor

April 29, 2013

1 Introduction

In this vignette we will introduce the **RcppSimpleTensor** package by some examples. For information regarding installation, please refer to the website (above). We will first demonstrate usage with the help of a very simple example that permits a graphical representation. Then we will present an application that uses **RcppSimpleTensor**, where we want to find an approximatin to the integral of a multivariate function.

Contents

2 Usage

Tensor algebra is a convenient way to deal with high-dimensional objects. In a way, it is like matrix multiplication for arrays of general dimension N-D. Note, however, that the operations one can perform with the package are **not restricted to be matrix multiplication only**, but **any kind of mathematical operation** *along a given index* of an array. In this section we demonstrate some ways to use the library by operating on three arrays `A`, `b` and `B`, defined as follows:

```
A <- array(1:8, dim = c(2, 2, 2))
b <- array(1, dim = c(2, 1))
B <- array(1, dim = c(2, 2))
print(A)

## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8

print(b)

##      [,1]
## [1,]    1
## [2,]    1

print(B)

##      [,1] [,2]
## [1,]    1    1
## [2,]    1    1
```

2.1 Multidimensional Multiplication

We will use the two main functions in **RcppSimpleTensor**, `tensorFunction` and `TI`, to show different forms of matrix multiplication. The two functions perform the same tasks, the difference is that `TI` is for inline use, and `tensorFunction` needs to be defined before usage. First of all, notice that array `A` can be visualized as a cube (see figure 1).

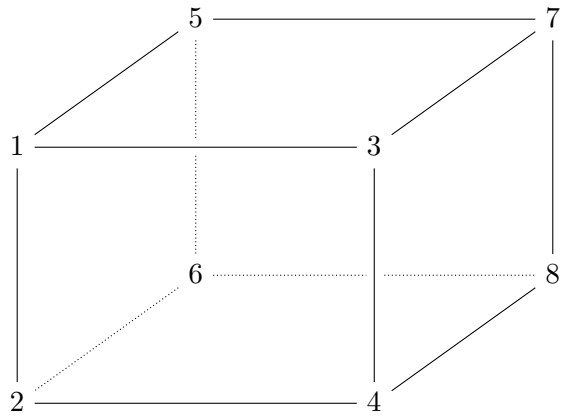


Figure 1: Array `A`

We will now perform operations along the 3 different indices of `A`, and we will denote those indices `i` (vertical direction), `j` (horizontal direction) and `k` (in-depth direction). All of our examples are a form of tensor reduction, i.e. we operate along a certain dimension of an array to reduce the array in this dimension. In terms of figure 1, we would flatten the cube along a certain index, applying a certain operation to the elements. Needless to say, the gains from using **RcppSimpleTensor** get larger as the dimension of a grows, but we will go with this easy example for illustration.

2.1.1 Matrix multiply `A` with `b` along index `i`

As a first task, we want to multiply `A` with vector `b` along their common index `i`, that is, along the vertical direction of `A`. This means our result will still have indices `j` and `k`, as we only "flattened-out" `A` along dimension `i`:

```
library(RcppSimpleTensor)
# 1. define tensor function
tensFunc <- tensorFunction( R[k,j] ~ A[i,j,k] * b[i] )
# 2. and call it
y <- tensFunc( A, b)
y
```

```
##      [,1] [,2]
## [1,]    3    7
## [2,]   11   15

# or create the inline tensor
TI <- createInlineTensor()
y <- TI( A[i,j,k] * b[i], k + j)
y

##      [,1] [,2]
## [1,]    3    7
## [2,]   11   15
```

Notice that this is like looking at the cube from above and performing the operation `A[ ,j,k] * b` at each pair of coordinates (j,k). If we let $y_{i,j}$ be the element in row i, column j, we obtained

$$\begin{aligned} y_{1,1} &= A_{1,1,1}b_1 + A_{2,1,1}b_2 = 1 \cdot 1 + 2 \cdot 1 = 3 \\ y_{2,1} &= A_{1,2,1}b_1 + A_{2,2,1}b_2 = 3 \cdot 1 + 4 \cdot 1 = 7 \\ y_{1,2} &= A_{1,1,2}b_1 + A_{2,1,2}b_2 = 5 \cdot 1 + 6 \cdot 1 = 11 \\ y_{2,2} &= A_{1,2,2}b_1 + A_{2,2,2}b_2 = 7 \cdot 1 + 8 \cdot 1 = 15 \end{aligned}$$

In (Einstein) tensor notation[1], our operation is defined as

$$\begin{aligned} y_{j,k} &= A_{i,j,k}b_i \\ &= \sum_i A_{i,j,k}b_i \end{aligned}$$

2.1.2 Matrix multiply along `j`

In exact analogy to above, we can assign `b` to a different (matching!) index of `A` – in this case `j`. We want to do

$$y_{i,k} = \sum_j A_{i,j,k}b_j$$

Thus, we obtain an `ni` by `nk` array, where `nx` is the number of elements in dimension `x`

```
TI(A[i, j, k] * b[j], k + i)

## '.find.package' is deprecated.
## Use 'find.package' instead.
## See help("Deprecated")
```

[1]The tensor notation is named after Einstein if one dispenses with the summation sign. The convention is that one is summing over the index that disappears from the resulting left hand side. See http://en.wikipedia.org/wiki/Einstein_notation.

```
##      [,1] [,2]
## [1,]    4    6
## [2,]   12   14
```

2.1.3 Matrix multiply with array

Now we do the same with 2 dimensional array `B`. Much in the same way, we specify common indices, and give the index we wish to keep in the result to `TI()`. In terms of tensor notation, we want to do

$$y_k = \sum_i \sum_j A_{i,j,k} B_{i,j}$$

```
TI(A[i, j, k] * B[i, j], k)

## '.find.package' is deprecated.
## Use 'find.package' instead.
## See help("Deprecated")

## [1] 10 26
```

2.2 Other Multidimensional Operations

As mentioned above, you are not restricted to do multiplication. In general, the functions will accept any binary operator. Let's see two examples. First, let's take `A` to the power of `b` along index `k`:

$$y_{i,j} = \sum_k (A_{i,j,k})^{b_k}$$

```
TI(A[i, j, k]^b[k], i + j)

## '.find.package' is deprecated.
## Use 'find.package' instead.
## See help("Deprecated")

##      [,1] [,2]
## [1,]    6   10
## [2,]    8   12
```

Secondly, let's sum up the cosines of elements of `A` and `B`, keeping only dimension `k`:

$$y_k = \sum_i \sum_k \cos\left(A_{i,j,k}, B_{i,j}\right)$$

```
TI(cos(A[i, j, k], B[i, j]), k)

## '.find.package' is deprecated.
## Use 'find.package' instead.
## See help("Deprecated")

## [1] 4 4
```

3 Under the hood

In this section we'll present more about what is happening under the hood of RcppSimpleTensor. This is for users that want to understand better what exactly is going on in this package.

3.1 Expression parsing and compilation

TensorFunction and TI take the expression, extract the relevant indices and then generates a C++ file. The C++ file is then compiled. The source file and the compiled library are located in the .tensor folder in the working directory. You can go to that folder and see their content. Additionally, if you call the tensorFunction with the verbose argument, this will display all of the compilation steps.

After the compilation is done, the library is linked to the R environment and wrapped into an R function that correctly deals with the different dimensions.

3.2 Caching

Obviously the compilation phase takes quite a long time and is unnecessary if the tensor expression has not changed. For that reason RcppSimpleTensor has a caching mechanism. When tensorFunction is called, a hash is created from the tensor expression. This hash is used to create the C++ file and the library. Before compiling the tensor, the function checks if such a library already exists locally and if it does, it links to it.

You can see how much faster the second call for a similar tensor is:

```
# 1. define tensor function, it will need to create the source and compile
system.time(tensFunc <- tensorFunction( R[k,j] ~ A[i,j,k] * b[i,n] ,cache=FALSE))

##    user  system elapsed
##   1.648   0.197   1.899

# 2. call it again, it uses the cached library
system.time(tensFunc <- tensorFunction( R[k,j] ~ A[i,j,k] * b[i,n] ))

##    user  system elapsed
##   0.013   0.000   0.014
```

This caching mechanism will also work for the TI() function. However linking is itself slow compared to just a function call. For that reason, to make TI() as fast as possible we use another caching mechanism. The TI context stores a list of all TI expressions. When TI gets called, it looks in this list if the tensor called has already been linked to the R context. If that is the case, we don't want to load the library again, it's already there.

As a demo, we are going to make 3 calls to TI, the first one will require compilation, the second one will require a linking, the third one will use level-2 caching. Let's compare the speeds:

```
# first call
system.time(TI(A[i,j,k] * b[i,n],j+k))

##    user  system elapsed
##   1.643   0.197   1.871

# second call
system.time(TI(A[i,j,k] * b[i,n],j+k))

##    user  system elapsed
##   0.001   0.000   0.000
```

3.3 Setting compiler flags for better performance

By default your compiler flags won't have the -O3 flag which will give you the best performance. To activate that create a MakeVars file in your ~/.R/ directory and add the following to it:

CXXFLAGS = −O3

That will significantly increase the performances of your tensor expressions!

4 Application: Evaluation of multidimensional B-splines

Disclaimer: This section is likely to be useful only if you have some prior knowledge about B-spline approximation.

We will use observational data to approximate a multivariate function (or a *data generating process*) on a space of tensor products of univariate B-splines. The function we want to approximte is defined as

$$\begin{aligned} f : X \times Y \times Z &\mapsto \mathbb{R} \\ f(x, y, z) &= (x + y - 5)^2 + (z - 5)^2 \end{aligned}$$

and $X, Y, Z \subset \mathbb{R}$. We will get data on f at a grid of points $\{x_i, y_j, z_k\}$, and use this to compute an approximant to f. The coefficients obtained from the approximation procedure can in conjunction

with the basis be used to approximate the function at an arbitrary set of points (in the approximation domain). We will use this to perform integration over one, and then 2 dimensions of this function with quadrature methods, i.e. we want to approximate

$$\begin{aligned} E_Y\left[f(x,y,z)\right] &= \int f(x,y,z)g(y)dy \\ E_Z\left[f(x,y,z)\right] &= \int f(x,y,z)s(z)dz \\ E_{YZ}\left[f(x,y,z)\right] &= \int f(x,y,z)s(z)g(y)dzdy \end{aligned}$$

where g, s are the pdf of y, z respectively. The advantages of **RcppSimpleTensor** are that we can write our code in a way which is very close to the mathematical expression, and we have a way to easily increase the number of dimensions with little additional cost.

4.1 Data Setup

The general idea is to estimate the f at relatively few grid points (`num.x`, `num.y` and `num.z`), to then be able to obtain a function value at an arbitrary point. Imagine a setup where calculating f is costly, such that we want `num.x + num.y + num.z` to be small.

```
library(splines)
if (!require(statmod)) install.packages('statmod')
library(statmod)
# number of evaluation points in each dimension
num.x <- 10
num.y <- 8
num.z <- 4
# choose number of integration nodes and method
num.int.z <- 50
num.int.y <- 40
int.z <- gauss.quad(n=num.int.z,kind="hermite")
int.y <- gauss.quad(n=num.int.y,kind="hermite")
# select degree of splines
degree <- 3
# get spline knot vector defined on nodes
xknots <- c(rep(0,times=degree),seq(0,10,le=num.x),rep(10,times=degree))
zknots<-c(rep(min(int.z$nodes),times=degree),
seq(int.z$nodes[1],int.z$nodes[num.int.z],le=num.z),
rep(max(int.z$nodes),times=degree))
yknots<-c(rep(min(int.y$nodes),times=degree),
seq(int.y$nodes[1],int.y$nodes[num.int.y],le=num.y),
rep(max(int.y$nodes),times=degree))
# get grid points where to evaluate function
```

```
xdata <- as.array(seq(0,10,length=length(xknots)-degree-1))
ydata <- as.array(seq(int.y$nodes[1],int.y$nodes[num.int.y],length=length(yknots)-degree-1))
zdata <- as.array(seq(int.z$nodes[1],int.z$nodes[num.int.z],length=length(zknots)-degree-1))
# design matrices
X <- splineDesign(knots=xknots,x=xdata)
Y <- splineDesign(knots=yknots,x=ydata)
Z <- splineDesign(knots=zknots,x=zdata)
DGP <- function( x,y,z ) { return((x + y - 5)^2 + (z-5)^2) }
```

4.2 Generate Data: Evaluate the function

We compare evaluation time of the 3-dimensional function with a mapply operation and with **RcppSimpleTensor**, as follows:

```
data.grid <- expand.grid(x=xdata,y=ydata,z=zdata)
trad.time<-system.time(
traditional<-array(with(data.grid,mapply(DGP,x,y,z)),
dim=c(length(xdata),length(ydata),length(zdata))))
# evaluate function with RcppSimpleTensor
# define a tensor function to calculate function values:
RcppVals <- tensorFunction( R[i,j,k] ~ (X[i] + Y[j] - 5)^2 + (Z[k] - 5)^2 )

## '.find.package' is deprecated.
## Use 'find.package' instead.
## See help("Deprecated")

# read: return array indexed by [i,j,k], defined as (x[i] + y[j] - 5)^2 + (z[k]-5)^2
Rcpp.time <- system.time( RcppArray <- RcppVals(xdata,ydata,zdata) )
# check result
sum(abs(traditional - RcppArray))

## [1] 0

# see timing
print(rbind(trad.time,Rcpp.time))

##           user.self sys.self elapsed user.child sys.child
## trad.time     0.004        0   0.004          0         0
## Rcpp.time     0.000        0   0.000          0         0
```

4.3 Visualize the function

Just for orientation, let's plot the function at the highest and lowest value of variable z, respectively (see figure 2).

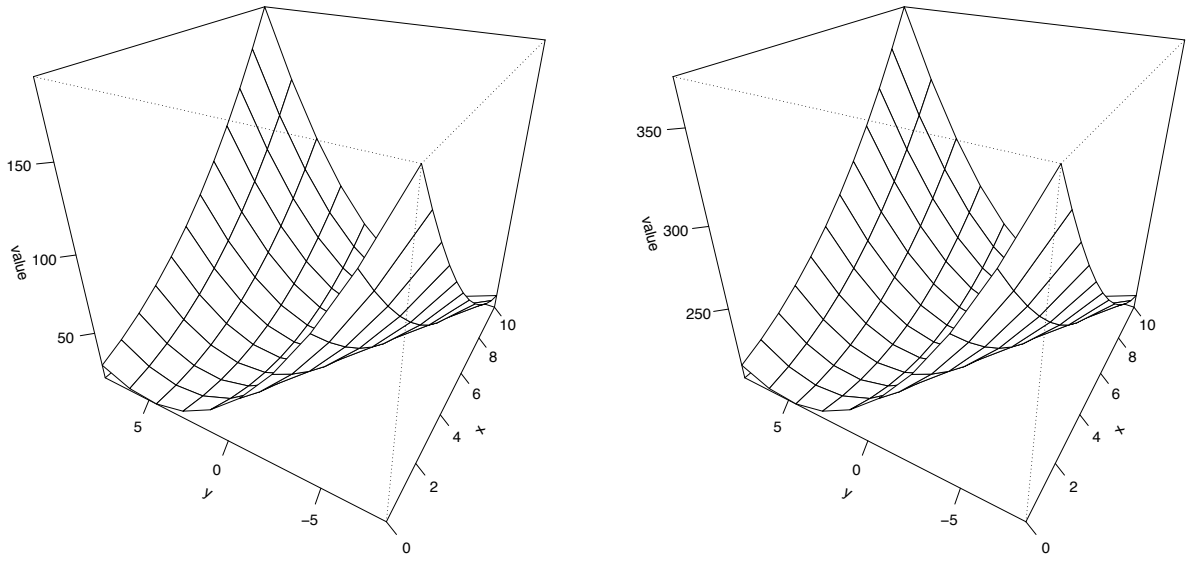


Figure 2: function `DGP` on the evaluation grid at highest and lowest point of z

4.4 Compute Approximant on Grid

Now we obtain the approximating coefficients b by solving the system

$$y = bA$$

where y are the function values at each grid point, b is our vector of coefficients and A is the tensor product of spaces of grid points X, Y and Z.

```
b <- solve(kronecker(Z, kronecker(Y, X)), as.vector(RcppArray))
# put coefficients into a 3-dimensional array
bb <- array(b, dim = c(length(xdata), length(ydata), length(zdata)))
```

4.5 Use RcppSimpleTensor to Evaluate Spline

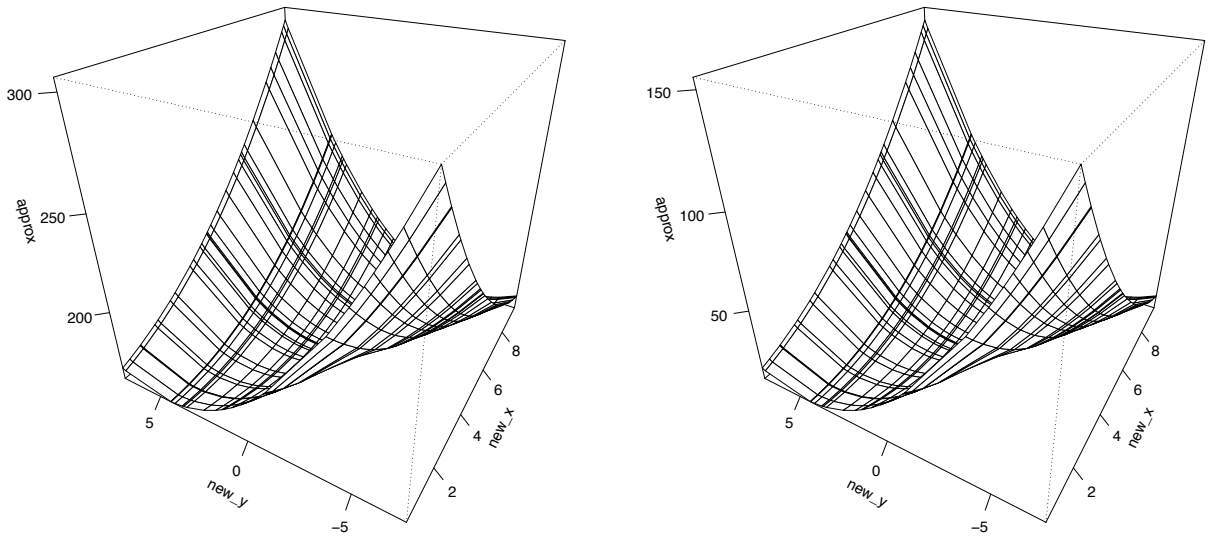
Next, we generate random data in the domain of our approximation, and evaluate the following expression, which is our approximant $\hat{f}$ at arbitrary data (x, y, z):

$$\hat{f}(x, y, z) = \sum_i \sum_j \sum_k b_{i,j,k} X_i(x) Y_j(y) Z_k(z)$$

where X_i, Y_j, Z_k are the values of the i-th, j-th and k-th basis functions at the values x, y, z, respectively.

```
new_xdata <- sort( runif(n=30,min=min(xdata),max=max(xdata)) )
new_ydata <- sort( runif(n=25,min=min(int.y$nodes),max=max(int.y$nodes) ) )
new_zdata <- sort( runif(n=21,min=min(int.z$nodes),max=max(int.z$nodes) ) )
# basis for new values
new_X <- splineDesign(knots=xknots,x=new_xdata)
new_Y <- splineDesign(knots=yknots,x=new_ydata)
new_Z <- splineDesign(knots=zknots,x=new_zdata)
# define RcppSimpleTensor function
spline.eval <- tensorFunction(R[nx,ny,nz] ~ coeffs[mx,my,mz] * Xbase[nx,mx] * Ybase[ny,my] * Zba

## '.find.package' is deprecated.
## Use 'find.package' instead.
## See help("Deprecated")

pred.vals <- spline.eval( bb, new_X, new_Y, new_Z )
# or simply with inline:
TIpred.vals <- TI( bb[m1,m2,m3] * new_X[n1,m1] * new_Y[n2,m2] * new_Z[n3,m3], n1+n2+n3)

## '.find.package' is deprecated.
## Use 'find.package' instead.
## See help("Deprecated")
```

Let's see how we are doing by plotting our approximant, again for the two most extreme values of z:



4.6 Perform numerical Integration

Here we calculate the basis on the entire set of integration nodes, and then we multiply by quadrature weights and sum over the correct indices. We want to approximate

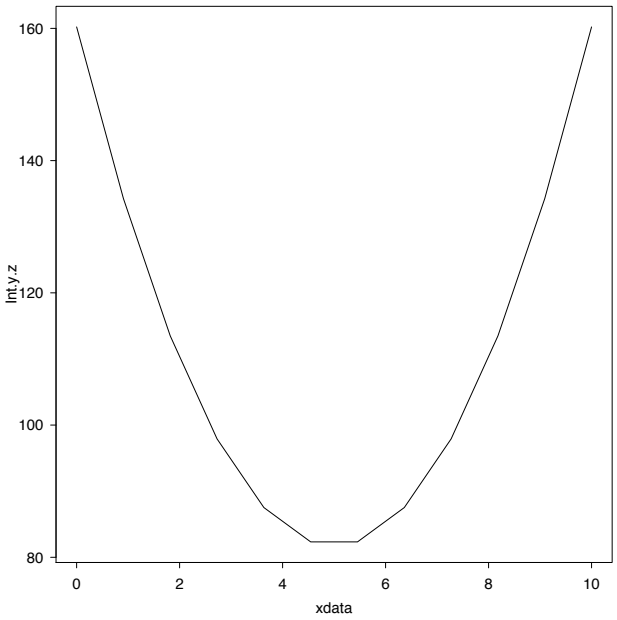
$$E_{YZ}\left[f(x,y,z)\right] = \int f(x,y,z)s(z)g(y)dzdy$$

by

$$\hat{E}_{YZ}\left[f(x,y,z)\right] = \sum_i \sum_j \omega_i^y \omega_j^z \hat{f}(x,\tilde{y}_i,\tilde{z}_j)$$

where $(\omega^y, \omega^z, \tilde{y}, \tilde{z})$ are quadrature weights and nodes for y, z respectively.

```
Int.base.z <- splineDesign(knots = zknots, x = int.z$nodes)
Int.base.y <- splineDesign(knots = yknots, x = int.y$nodes)
# evaluate spline with RcppSimple
Intdata <- spline.eval(bb, X, Int.base.y, Int.base.z)
yweights <- int.y$weights
zweights <- int.z$weights
#### integrate i.e. weighted sum over corresponding dimensions
RcppIntFun <- tensorFunction(R[nx] ~ Data[nx, ny, nz] * yweight[ny] * zweight[nz])

## '.find.package' is deprecated.
## Use 'find.package' instead.
## See help("Deprecated")

Int.y.z <- RcppIntFun(Intdata, yweights, zweights)
# or inline
TI.Int.y.z <- TI(Intdata[nx, ny, nz] * yweights[ny] * zweights[nz], nx)

## '.find.package' is deprecated.
## Use 'find.package' instead.
## See help("Deprecated")

# compare
sum(Int.y.z - TI.Int.y.z)

## [1] 0

dim(Int.y.z)

## [1] 12

# plot integrated function
plot(xdata, Int.y.z, type = "l")
```



If we want to integrate only one out of many dimensions, like in

$$\hat{E}_Y\left[f(x,y,z)\right] = \sum_i \omega_i^y \hat{f}(x,\tilde{y}_i,z)$$

or

$$\hat{E}_Z\left[f(x,y,z)\right] = \sum_j \omega_j^z \hat{f}(x,y,\tilde{z}_j)$$

this is very easily accomplished with RcppSimpleTensor – just select the right dimensions and recompute:

```
# integrate w.r.t. z only
Int.z <- TI( Intdata[nx,ny,nz] * zweights[nz], nx + ny)

## '.find.package' is deprecated.
## Use 'find.package' instead.
## See help("Deprecated")

dim(Int.z)

## [1] 12 40

# integrate w.r.t. y only
Int.y <- TI( Intdata[nx,ny,nz] * yweights[ny], nx + nz )

## '.find.package' is deprecated.
## Use 'find.package' instead.
## See help("Deprecated")
```

```
dim(Int.y)

## [1] 12 50
```

We can again inspect the result graphically: