Introducing the float package: 32-Bit Floats for R

April 13, 2020

Drew Schmidt
wrathematics@gmail.com

Version 0.2-4

Acknowledgements and Disclaimer

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. The findings and conclusions in this article have not been formally disseminated by the U.S. Department of Health & Human Services nor by the U.S. Department of Energy, and should not be construed to represent any determination or policy of University, Agency, Administration and National Laboratory.

This publication was typeset using LaTeX.

Contents

1 Introduction

R has a "numeric" type for vectors and matrices. This type must be either integer or double precision. As such, R has no real ability to work with 32-bit floats. However, sometimes single precision (or less!) is more than enough for a particular task. The **float** package [9] extends R's linear algebra facilities to include single precision (float) data. Float vectors/matrices have half the precision of their "numeric"-type counterparts, for a performance vs accuracy trade-off.

The internal representation is an S4 class, which allows us to keep the syntax identical to that of base R's. Interaction between base types for binary operators is generally possible. In these cases, type promotion always defaults to the higher precision (more on this in Section 2.2). The package ships with copies of the single precision 'BLAS' and 'LAPACK', which are automatically built in the event they are not available on the system.

1.1 Installation

You can install the stable version from CRAN using the usual `install.packages()`:

```
install.packages("float")
```

The development version is maintained on GitHub. You can install this version using any of the well-known installer packages available to R:

```
remotes::install_github("wrathematics/float")
```

Note that for best performance, you will need to build the package from source, either from the GitHub repository or from the CRAN source distribution. See Section 1.2 for more details as to why a source installation is recommended.

1.2 BLAS and LAPACK Libraries

The linear algebra operations in the **float** package are handled by the BLAS and LAPACK [7, 2]. By default, R will not ship with the single precision versions of these functions, so we include a source code distribution within the package. This is the "reference" or NetLib implementation, which is not particularly efficient. Additionally, compiling these can take a very long time.

To take advantage of the enhanced run-time performance and reduced compilation times of using tuned BLAS/LAPACK with **float**, you will need to choose an implementation and install it. Typical implementations include **OpenBLAS** [12], Intel **MKL** [6], AMD **ACML** [1], **Atlas** [11], and Apple's **Accelerate** [3]. You can read more about using different BLAS implementations with R in the R Installation and Administration manual [10].

Once you switch BLAS implementations with R, you will need to rebuild the **float** package from source.

2 For Users

2.1 Basics

R does not have a 32-bit float type (hence the package). You can cast your data from integer/numeric to float using `fl()` (you can also cast a float to a numeric via `dbl()`):

```
library(float)

x = matrix(1:9, 3)
```

```
x
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9

s = fl(x)
s
## # A float32 matrix: 3x3
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

This will of course require 1.5x the memory of the input matrix (storing it as both a double and as a float). For workflows requiring many operations, the memory savings will still be substantial. At this time, we do not have a reader, so casting is the best way to go. However, once you cast the matrix to a float, you can serialize it as usual with `save()` and/or `saveRDS()`.

For testing or other cases where random matrices are needed (e.g., PCA via random normal projections [5]), we include several random generators. The functions `flrunif()` and `flrnorm()` are somewhat like R's `runif()` and `rnorm()` in that they produce vectors (but also matrices) of floating point random uniform/normal values:

```
set.seed(1234)

flrunif(5)
## # A float32 vector: 5
## [1] 0.1137034 0.6222994 0.6092747 0.6233795 0.8609154
flrunif(2, 3)
## # A float32 matrix: 2x3
##             [,1]      [,2]      [,3]
## [1,] 0.640310585 0.2325505 0.5142511
## [2,] 0.009495757 0.6660838 0.6935913
flrunif(5, min=10, max=20)
## # A float32 vector: 5
## [1] 15.44975 12.82734 19.23433 12.92316 18.37296
```

Arbitrary generators can be used with the `flrand()` interface. It behaves more like R's `runif()`, `rnorm()`, etc., except that it accepts a generator function for its first argument. For example:

```
flrand(generator=rexp, n=5, rate=.1)
## # A float32 vector: 5
## [1]  8.624105  6.745913  8.380404  7.604303 18.800766
flrand(function(n) sample(5, size=n, replace=TRUE), 5)
## # A float32 vector: 5
## [1] 2 2 2 1 1
```

This is conceptually similar to first generating `n` random values and then casting them over to floats, but more memory efficient. The function processes the generator data in 4KiB chunks (for double precision generators).

2.2 Arithmetic and Type Promotion

Perhaps a mistake in hindsight, but floats and numeric vectors/matrices will interoperate with each other in binary arithmetic operations. So you can multiply `2L`, `2.0`, and `fl(2)` in any binary combination you like. But the output will be determined by the highest precision; in fact, the arithmetic itself will be carried out in the highest possible precision. So adding a float matrix with a double matrix is really just adding 2 double matrices together after casting the float up (which uses quite a bit of additional memory) and returning a double matrix.

This even works for more complicated functions like `%*%` (matrix multiplication). For example:

```
set.seed(1234)
x = matrix(1:4, 2)
y = flrunif(2, 2)

x
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
y
## # A float32 matrix: 2x2
##           [,1]      [,2]
## [1,] 0.1137034 0.6092747
## [2,] 0.6222994 0.6233795

x %*% y
## # A float32 matrix: 2x2
##          [,1]     [,2]
## [1,] 1.980602 2.479413
## [2,] 2.716604 3.712067

storage.mode(x) = "double"
x %*% y
##          [,1]     [,2]
## [1,] 1.980602 2.479413
## [2,] 2.716605 3.712067
```

Long story short, be careful when mixing types.

3 For Developers

3.1 Basics

A `float32` matrix/vector is really a very simple S4 class. It has one slot, `@Data`, which should be an ordinary R integer vector or matrix. The values of that integer matrix will be interpreted as floats in the provided methods. If you wish to create your own method, say using C kernels or Rcpp [4], then you will have to play the same game. More on that later.

To create a `float32` object, use `float::float32()`:

```
Data = 1:3
x = float32(Data)
x
## # A float32 vector: 3
```

```
## [1] 1.401298e-45 2.802597e-45 4.203895e-45
```

To access the integer data of a `float32`, just grab the `@Data` slot:

```
> x@Data
## [1] 1 2 3
```

In general there's no relationship between the integer vs float interpretations of the values residing in the same block of memory, with the exception of 0:

```
x = fl(0:3)
x@Data
## [1]          0 1065353216 1073741824 1077936128
dbl(x+x)
## [1] 0 2 4 6
(x+x)@Data
## [1]          0 1073741824 1082130432 1086324736
x@Data + x@Data
## [1]          0 2130706432         NA         NA
## Warning message:
## In x@Data + x@Data : NAs produced by integer overflow
```

So when creating new functionality not provided by existing **float** package methods, you will probably have to move to compiled code.

3.2 Compiled Code

Using 32-bit floats from **float** in compiled code is not terribly difficult, but maybe a bit annoying. The general way to proceed for a 32-bit float `x` is:

- Pass `x@Data` (an integer) to `.Call()`
- Inside the C/C++ function, use a `float` pointer to the integer data.
- Return from `.Call()` an integer vector/matrix.
- Put the return from `.Call()` (say `ret`) in the `float` S4 class: `float32(ret)`.

One can access the data with the `FLOAT()` macro. If writing an R package, add **float** to the `LinkingTo` list in the package DESCRIPTION file. Then add `#include <float/float32.h>` and the macro will be available. If you are working outside the construct of a package (not recommended), then you can define the macro as follows:

```
#define FLOAT(x) ((float*) INTEGER(x))
```

This is the "`DATAPTR`" way of doing things, similar to `REAL()` for double precision and `INTEGER` for ints. There is no Rcpp-like idiom for floats similar to `NumericVector` and `NumericMatrix` at this time.

Here's a basic example of how one would create a new function `add1()` (ignoring that we could just do `x+1`) using C. We will do this outside of a package framework for simplicity of demonstration, but again, it is recommended that you use the `LinkingTo` way mentioned above.

add1.c

```
#include <Rinternals.h>
```

```
2  #include <R.h>
3
4  #define FLOAT(x) ((float*) INTEGER(x))
5
6  SEXP R_add1(SEXP x_)
7  {
8    SEXP ret;
9    PROTECT(ret = allocVector(INTSXP, 1));
10
11   float *x = FLOAT(x_);
12   FLOAT(ret)[0] = x[0] + 1.0f;
13
14   UNPROTECT(1);
15   return ret;
16 }
```

Note that using `INTEGER(ret)[0]` instead of `FLOAT(ret)[0]` on line 12 above is not correct. That would first cast the value to an integer before storing the data. Then back at the R level, once put in the `float32` class, that integer value would be treated as though it were a `float`. If that explanation doesn't make sense, try modifying the above to the wrong thing and see what happens.

We can build that function with `R CMD SHLIB add1.c`, and then call it via:

add1.r

```
1  dyn.load("add1.so")
2  library(float)
3
4  add1 = function(x)
5  {
6    ret = .Call("R_add1", x@Data)
7    float32(ret)
8  }
9
10 add1(fl(1))
11 ## # A float32 vector: 1
12 ## [1] 2
13 add1(fl(pi))
14 ## # A float32 vector: 1
15 ## [1] 4.141593
```

Like I said, not really difficult, but annoying.

3.3 Linking and Additional Functions

If you are writing C/C++ code on single precision vectors and matrices, there is a good chance that you will need to link with the **float** package. For sure if you want to efficiently do linear algebra (say via BLAS/LAPACK or you are using the float interface from Armadillo via **RcppArmadillo** [?]), you will need to do this for CRAN safety[1]. To do this, you will need to set the `LDFLAGS` line of your `src/Makevars` file to include something like this:

```
FLOAT_LIBS = `${R_HOME}/bin${R_ARCH_BIN}/Rscript -e "float:::ldflags()"`
```

[1] If you are just working on your own machine and linking with high-performance BLAS/LAPACK, then no linking is necessary. For portability, you need to link.

```
PKG_LIBS = $(LAPACK_LIBS) $(BLAS_LIBS) $(FLIBS) $(FLOAT_LIBS)
```

By default, `float:::ldflags()` will try to dynamically link on Linux and Mac, but you can force static linking via `float:::ldflags(static=TRUE)`. In my opinion, dynamic linking is preferential, but you are free to make up your own mind about that. However, dynamic linking to an R package shared library is (I think?) impossible on Windows. So Windows will always statically link.

In addition to BLAS and LAPACK symbols, there are a few helpers available. First, we include float values `NA_FLOAT` and `R_NaNf`, which are 32-bit analogues to `NA_REAL` and `R_NaN`.

```
int ISNAf(const float x);
int ISNANf(const float x);
```

which you can find in the `float/float32.h` header.

Finally, we also provide `float:::cppflags()` for the `PKG_CPPFLAGS` include flags. But using the `LinkingTo` field should usually be sufficient (i.e., you don't need it most of the time). One notable exception is if you are doing something goofy with a different compiler, like `nvcc` where you may need to explicitly pass the include flags.

4 Some Benchmarks

We will be examining two common applications from statistics which are dominated by linear algebra computations: covariance and principal components analysis. The setup for each of these benchmarks is:

```
library(float)
library(rbenchmark)
set.seed(1234)

reps = 5
cols = c("test", "replications", "elapsed", "relative")

m = 7500
n = 500
x = matrix(rnorm(m*n), m, n)
s = fl(x)
```

All benchmarks were performed using 2 cores of on an Intel Core i5-5200U (2.20GHz CPU) laptop running Linux and:

- gcc 7.2.0
- R version 3.4.2
- libopenblas 0.2.20

Note that the benchmarks are highly dependent on the choice of BLAS library and hardware used. The cache sizes for this machine are:

```
memuse::Sys.cachesize()
## L1I:   32.000 KiB
## L1D:   32.000 KiB
## L2:   256.000 KiB
## L3:     3.000 MiB
```

4.1 Covariance

Since covariance is just the crossproducts matrix $x^T x$ on mean-centered data, we can very easily create a custom covariance function:

```
custcov = function(x)
{
  s = scale(x, TRUE, FALSE)
  crossprod(s) / max(1L, nrow(x)-1)
}
```

This function will work for numeric inputs as well as 32-bit floats. We can compare these two cases against R's internal covariance function:

```
benchmark(custcov(x), custcov(s), cov(x), replications=reps, columns=
    cols)
##          test replications elapsed relative
## 3     cov(x)            5   8.113   43.385
## 2 custcov(s)            5   0.187    1.000
## 1 custcov(x)            5   0.719    3.845
```

R's `cov()` is clearly not designed with performance in mind. The performance difference between the `custcov(s)` (float) and `custcov(x)` (double) calls should only be about 2x. The higher performance we see is likely due to the fact that our implementation of `scale()` is better than R's. We can compare this to a highly optimized implementation of covariance, namely `covar()` from the **coop** package [8]:

```
benchmark(custcov(s), coop::covar(x), replications=reps, columns=cols)
##             test replications elapsed relative
## 2 coop::covar(x)            5   0.358    1.817
## 1     custcov(s)            5   0.197    1.000
```

This looks more in line with what we would expect moving from double to single precision.

4.2 Principal Components Analysis

PCA is just SVD with some statistical window dressing:

```
pca = function(x)
{
  p = svd(scale(x, TRUE, FALSE), nu=0)
  p$d = p$d / max(1, sqrt(nrow(x) - 1))
  names(p) = c("sdev", "rotation")

  p
}
```

Once again, our function will work for both numeric inputs as well as 32-bit floats. We again compare the performance of these two cases against R's internal function (in this case, `prcomp()`):

```
benchmark(pca(x), pca(s), prcomp(x), replications=reps, columns=cols)
##        test replications elapsed relative
## 2    pca(s)            5   1.592    1.000
## 3    pca(x)            5   3.663    2.301
## 1 prcomp(x)            5   4.293    2.697
```

Again, our improved `scale()` implementation is giving an edge (and possibly because `prcomp()` is doing more *useful* work, as opposed to `cov()...`), although it is much less pronounced here since the SVD calculation is dominating. Indeed, the overall run time is roughly 10x higher here for the single precision PCA case compared to the single precision covariance calculation.

References

[1] AMD. Core math library (acml). *URL http://developer. amd. com/acml. jsp*, 2012.

[2] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. *LAPACK Users' guide*. SIAM, 1999.

[3] Apple. Accelerate. *URL https://developer.apple.com/documentation/accelerate*, 2017.

[4] Dirk Eddelbuettel and Romain François. Rcpp: Seamless R and C++ integration. *Journal of Statistical Software*, 40(8):1–18, 2011.

[5] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.

[6] Intel Corporation. Intel Math Kernel Library (Intel MKL). http://software.intel.com/en-us/intel-mkl.

[7] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[8] Drew Schmidt. *Co-Operation: Fast Correlation, Covariance, and Cosine Similarity*, 2016. R package version 0.6-0.

[9] Drew Schmidt. *float: Single Precision Floats*, 2017. R package version 0.1-0.

[10] R Core Team. R installation and administration. https://cran.r-project.org/doc/manuals/r-release/R-admin.html#BLAS.

[11] R Clint Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–27. IEEE Computer Society, 1998.

[12] Zhang Xianyi, Wang Qian, and Zaheer Chothia. Openblas. *URL: http://xianyi. github. io/OpenBLAS*, 2012.