Benchmarks for Discrete Fourier Transforms

Andrew J. Barbour

January 29, 2013

Abstract

The base DFT calculator in R, `stats::fft`, uses the Mixed-Radix algorithm of Singleton (1969). In this vignette we show how this calculator compares to `FFT` in the `fftw` package (Krey et al., 2011), which uses the FFTW algorithm of Frigo and Johnson (2005). For univariate DFT computations, the methods are nearly equivalent with two exceptions which are not mutually exclusive: (A) the series to be transformed is very long, and especially (B) when the series length is not highly composite. In both exceptions the algorithm `FFT` outperforms `fft`.

Contents

1 Benchmarking function

We use both functions in their default state, and ask them to transform the same univariate random series. Benchmark information comes from the `rbenchmark` program, and the versatile `plyr` and `reshape2` packages are used to manipulate the information for this presentation; `ggplot2` is used for plotting. First we load the libraries needed:

```
> rm(list=ls())
> library(fftw)
> library(rbenchmark)
> library(plyr)
> library(reshape2)
> library(ggplot2)
```

and create a benchmark function:

```
> reps <- 10
> dftbm <- function(nd, repls=reps){
+         set.seed(1234)
+         x <- rnorm(nd, mean=0, sd=1)
+         bmd <- benchmark(replications=repls, fftw::FFT(x), stats::fft(x))
+         bmd$num_dat <- nd
+         bmd$relative[is.na(bmd$relative)] <- 1   # NA happens.
+         return(bmd)
+ }
```

2 Highly composite (HC) series

It's well known that DFT algorithms are most efficient for "Highly Composite Numbers"[1], specifically multiples of (2,3,5).

So, we create a vector of series lengths we wish to benchmark

```
> (nterms.even <- round(2**seq.int(from=4,to=20,by=1)))
```

```
 [1]      16      32      64     128     256     512    1024    2048    4096
[10]    8192   16384   32768   65536  131072  262144  524288 1048576
```

and use it with `lapply` and the benchmark function previously defined. These data are further distilled into a usable format with `ldply`:

```
> bench.even <- function(){
+   benchdat.e <- plyr::ldply(lapply(X=nterms.even, FUN=dftbm))
+   }
> bench.even()
```

In order to plot the results, we need to perform some map/reduce operations on the data (Wickham, 2011):

```
> pltbench <- function(lentyp=c("even","odd")){
+   benchdat <- switch(match.arg(lentyp), even=benchdat.e, odd=benchdat.o)
+   stopifnot(exists("benchdat"))
+   tests <- unique(benchdat$test)
+   ## subset only information we care about
+   allbench.df.drp <- subset(benchdat,
+         select=c(test, num_dat, user.self, sys.self, elapsed, relative))
+   ## reduce data.frame with melt
+   allbench.df.mlt <- reshape2::melt(allbench.df.drp,
```

[1] This is the reason for the `stats::nextn` function.

```
+                                    id.vars=c("test","num_dat"))
+   ## calculate the summary information to be plotted:
+   tmpd <- plyr::ddply(allbench.df.mlt,
+                       .(variable,  num_dat),
+                       summarise,
+                       summary="medians", # just a name
+                       value=ggplot2::mean_cl_normal(value)[1,1])
+   ## create copies for each test and map to data.frame
+   allmeds <<- plyr::ldply(lapply(X=tests,
+                                  FUN=function(x,df=tmpd){
+                                        df$test <- x; return(df)
+                                      }))
+   ## plot the benchmark data
+   g <- ggplot(data=allbench.df.mlt,
+                aes(x=log10(num_dat),
+                    y=log2(value),
+                    # 1/sqrt(n) standard errors [assumes N(0,1)]
+                    ymin=log2(value*(1-1/sqrt(reps))),
+                    ymax=log2(value*(1+1/sqrt(reps))),
+                    colour=test,
+                    group=test)) +
+        scale_colour_discrete(guide="none") +
+        theme_bw()+
+        ylim(c(-11,11))+
+        xlim(c(0.5,6.5))+
+        ggtitle(sprintf("DFT benchmarks of %s length series",toupper(lentyp)))
+   ## add previous summary curves if exist
+   if (exists("allmeds.prev")){
+      g <- g + geom_path(size=1.5, colour="dark grey", data=allmeds.prev,
+                          aes(group=test))
+                          }
+   ## create a facetted version
+   g2 <- g + facet_grid(variable~test) #, scales="free_y")
+   ## add the summary data as a line
+   g3 <- g2 + geom_path(colour="black", data=allmeds, aes(group=test))
+   ## and finally the data
+   print(g4 <<- g3 + geom_pointrange())
+ }
```

For each row of the figure we plot summary curves[2] so we can easily inter-compare the

[2] Based on this post:
http://geokook.wordpress.com/2012/12/29/row-wise-summary-curves-in-faceted-ggplot2-figures/

```
> pltbench("even")
```

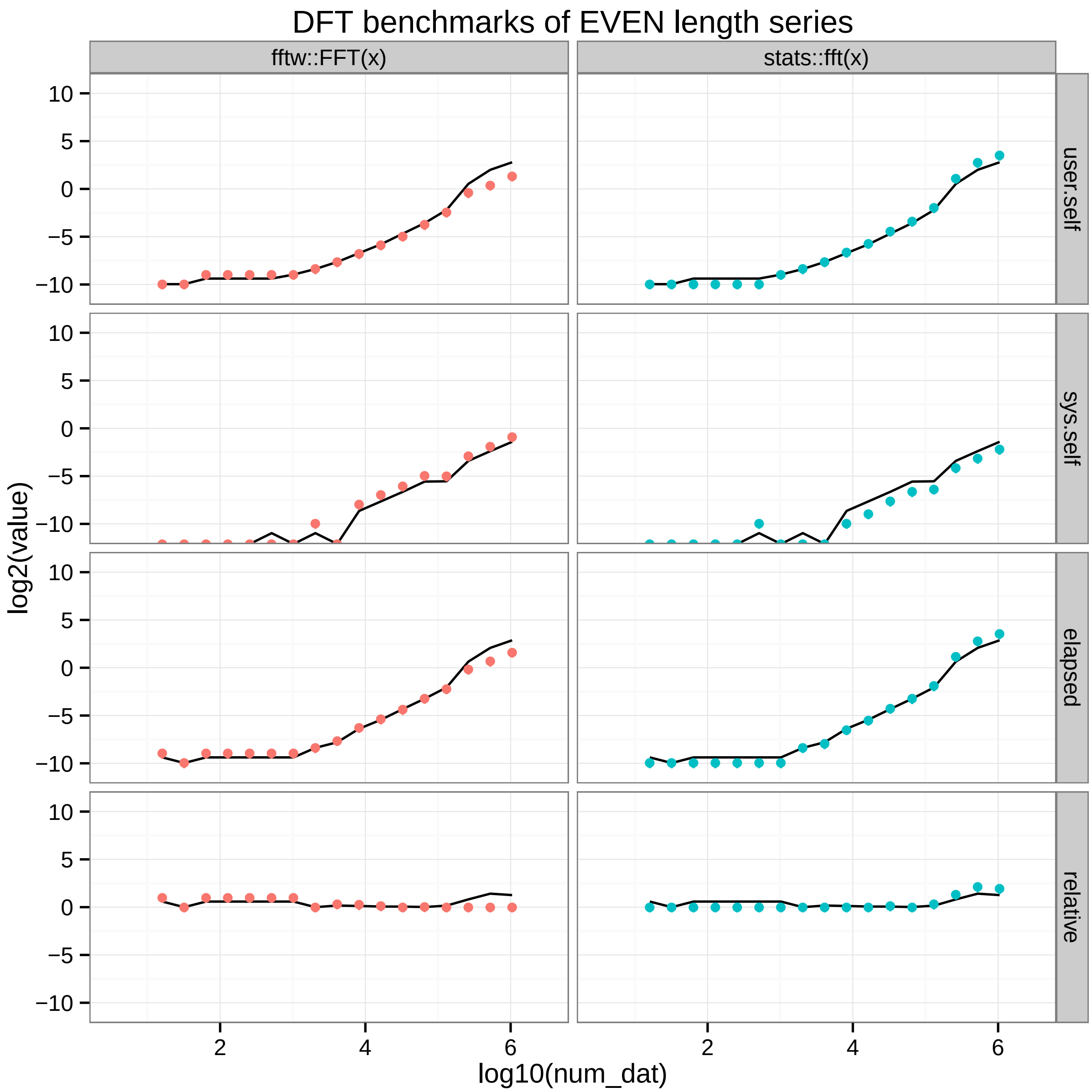

Figure 1: DFT benchmark results for HC series lengths.

benchmark data.

3 Non highly composite (NHC) series

DFT algorithms can have drastically reduced performance if the series length is not highly composite (NHC). We now test NHC series by adding one to the HC series-length vector (also restricting the total length for sanity's sake):

```
> nterms.odd <- nterms.even + 1
> nterms.odd <- nterms.odd[nterms.odd < 50e3] # painfully long otherwise!
```

and performing the full set of benchmarks again:

```
> bench.odd <- function(){
+   benchdat.o <- plyr::ldply(lapply(X=nterms.odd, FUN=dftbm))
+   }
> bench.odd() # FAIR WARNING: this can take a while!!
```

We can now visualize the results, with the addition of the HC summary curves:

4 Conclusion

Figures 2 and 3 compare the DFT calculations for HC and NHC length series. For univariate DFT computations, the methods are nearly equivalent with two exceptions which are not mutually exclusive: (A) the series to be transformed is very long, and especially (B) when the series length is not highly composite. In both exceptions the algorithm `FFT` outperforms `fft`. In the case of exception (B), both methods have drastically increased computation times; hence, zero padding should be done to ensure the length does not adversely affect the efficiency of the DFT calculator.

References

Frigo, M. and Johnson, S. G. (2005). The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

Krey, S., Ligges, U., and Mersmann, O. (2011). *fftw: Fast FFT and DCT based on FFTW*. R package version 1.0-3.

Singleton, R. C. (1969). An Algorithm for Computing the Mixed Radix Fast Fourier Transform. *IEEE Transactions on Audio and Electroacoustics*, AU-17(2):93–103.

Wickham, H. (2011). The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, 40(1):1–29.

```
> allmeds.prev <- allmeds
> pltbench("odd")
```

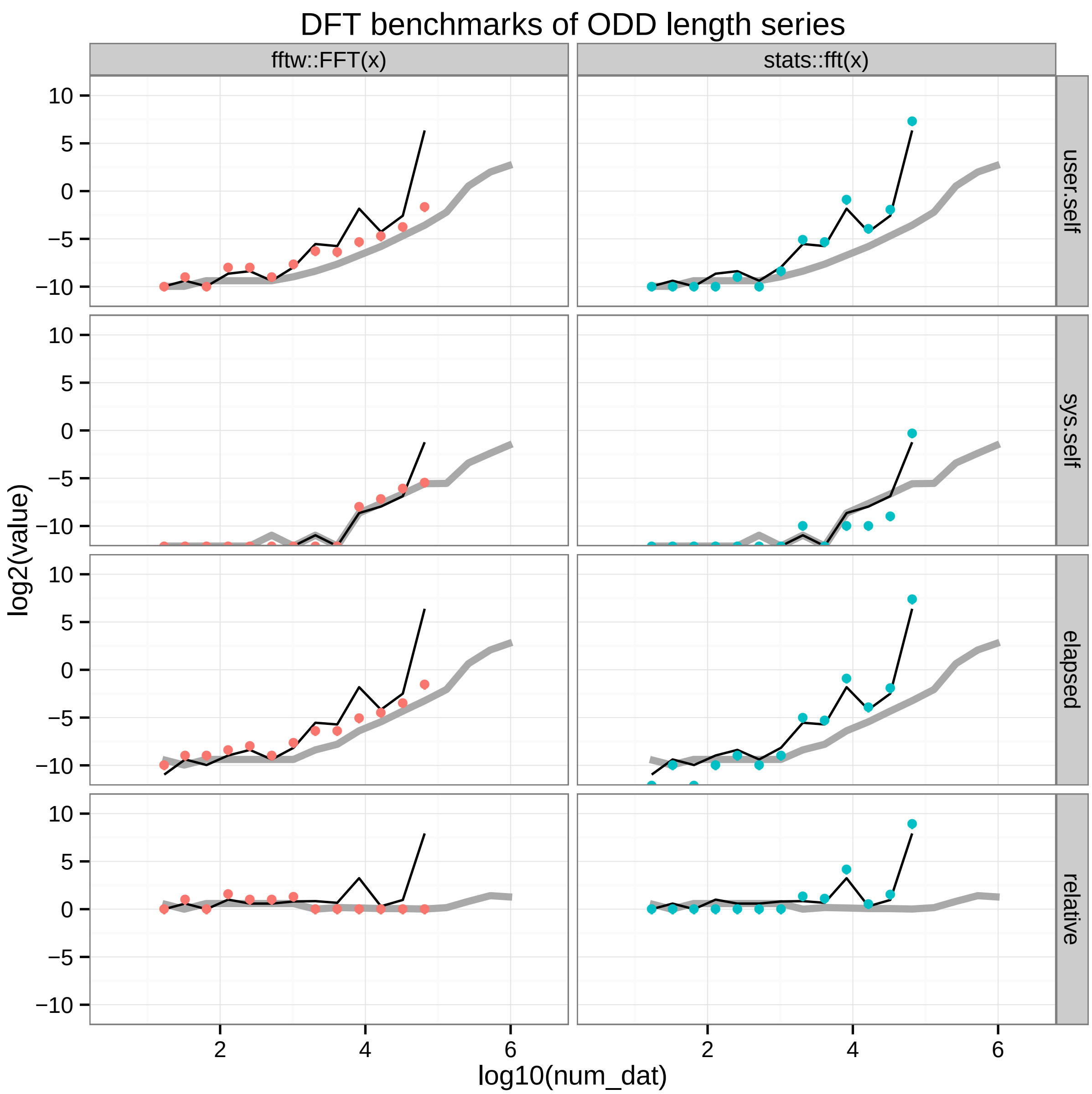

Figure 2: DFT benchmark results for NHC series lengths. We also show the summary curves for the HC results to highlight the drastic degradation in performance.