Implementation of Tukey's median polish in C++ and a comparison to an existing R implementation

Henry Kenlay - WORC5011

December 7, 2017

In this report I Implement Tukey's median polish algorithm in `C++` and compare the running time to the base `R` implementation over randomly generated datasets of various sizes. The relevant code for this report can be found at `https://github.com/HenryKenlay/MedianPolish`.

1 Median Polish

The median polish is an exploratory data analysis procedure proposed by John Tukey in 1977 [1]. Given an $n \times m$ matrix A containing experimental data, we may want to model the data as

$$A_{i,j} = \mu + \alpha_i + \beta_j + \epsilon_{i,j} \tag{1}$$

where μ is a global effect, α_i is a row effect, β_j is a column effect and $\epsilon_{i,j}$ is observational noise. A two way linear regression minimizes the mean of residuals and is sensitive to outliers in the data (the result is closely related to a two-way ANOVA model). The median polish algorithm is an iterative procedure to fit these parameters such that the median of the row effects, column effects and residuals are zero. Since median is used rather than mean it is robust to outliers in the data. Fitting data in this way is used often in high throughput drug discovery [2, 3, 4]. In this setting outliers are common due to experimental failures from vast numbers of cheap experiments. Experiments are done on grid shaped assays where row and column effects are a suitable model.

To obtain an additive fit of this form we can operate iteratively on the data matrix, finding and subtracting row medians and column medians. For example we could begin with the rows, calculating the median of each row and then subtracting this value from every observation in the row. Then we would continue with the columns of the resulting matrix, finding the median of each and subtracting it from the entries in its column. If a row or column has its median equal to zero then we will make no change in that row or column. Continuing to repeating this process until all rows and columns have zero median is known as median polish. For more details see [5].

2 Implementation

My code was written in `C++` and originally compiled using `g++` which is installed via `Xcode`. However, for the 3rd version of my implementation I required `openmp` which I had difficulty installing. Therefore, I installed and used `g++-7 (Homebrew GCC 7.2.0) 7.2.0` compiler which includes `openmp` (enabled by using the flag `-fopenmp`).

My code has an option to display convergence information which I used to check correctness but this was turned off during benchmarking. Similarly, in the `R` implementation I set the `trace.iter` argument to false so that convergence details were not printed and used `invisible` which stops `R` from printing the output once calculated.

2.1 First version

In the R programming language it is possible to view source code of functions. I ported the median polish function as close as possible into `C++` code. I used the Eigen package to help me accomplish this. The `R` implementation of median uses `C` under the hood, and calculates the median by doing a partial quick sort. I therefore implemented my median function in the same way.

2.2 Second version

After implementing the median polish algorithm making use of the Eigen library I found that when passing an Eigen object to a function it is passed by value. Functions also return by value in `C++`. To speed up my implementation I made functions take references to Eigen arrays, rather than the copied values. I also initialized the input matrix in the `main` before passing a pointer to a function to populate it from disk.

2.3 Third version

To further speed up the implementation, its easily noticeable that the sub routines in the median polish algorithm are embarrassingly parallel. When calculating the median of each row (or column) of the input array I did this in a parallel for loop making use of the `openmp` library.

3 Results

To benchmark the algorithms I wrote a python script. I used the `call` routine from the `subprocess` library and the `time` library. The running time of the program can be described as

$$T = T_{\text{OH}} + T_{\text{R}} + T_{\text{MP}} \tag{2}$$

where T_{OH} is the overhead from calling from python and starting the R/C++ program, T_{R} is the time it takes for the program to read data from disk and T_{MP} is the time it takes for the median polish algorithm to run.

The `C++` and `R` implementations take command arguments where the first two arguments are the size of the input matrix n, m. For this analysis I assume $n = m$ and made dummy data using a separate python script. If a third command argument is given then only the data is read from disk and the median polish algorithm is not run on the matrix. When we just read from disk then the running time is $T_0 = T_{\text{OH}} + T_{\text{R}}$ and $T_{\text{MP}} = T - T_0$ which is what I want to measure.

Each program is ran 60 times, 30 runs are reading the data from disk and calculating the median polish and another 30 runs are just reading the data from disk. Unfortunately, I couldn't figure out a way to calculate T and T_0 for the same instance of a program (i.e. get the two values from each run, rather than do separate runs to estimate each), so instead I present $\mathbb{E}(T) - \mathbb{E}(T_0)$ rather than $\mathbb{E}(T - T_0)$ and the variance is given by $\mathbb{V}(T - T_0) = \mathbb{V}(T) + \mathbb{V}(T_0)$ as the random variables were measured independently. I also present T_0. Under the assumption that $T_{\text{OH}} \approx 0$ which measures the speed of reading from disk between the implementations. I also report T. The implementations are tested on benchmarks of matrices with size $100 \times 100, 200 \times 200, \ldots 1000 \times 1000$. The test matrices are generated by generating global, row and column effects from a uniform distribution on the unit interval and the noise term is from a standard normal. The results are shown in figure 1. The 3rd `C++` implementation outperforms `R` for algorithm running time, all implementations beat `R` for reading data and for reading data and running the algorithm.

4 Discussion

As far as I can tell looking at the source code the R implementation does not do any parallel computing. I was surprised the second version of my code wasn't faster than the R implementation. To improve performance one could play around with the compilers optimization options. For the random datasets I created, the algorithm converges in a single iteration, however even for real datasets its typical for the algorithm to converge in just a few iterations (from [5]: "In practice, two complete iterations are often enough").

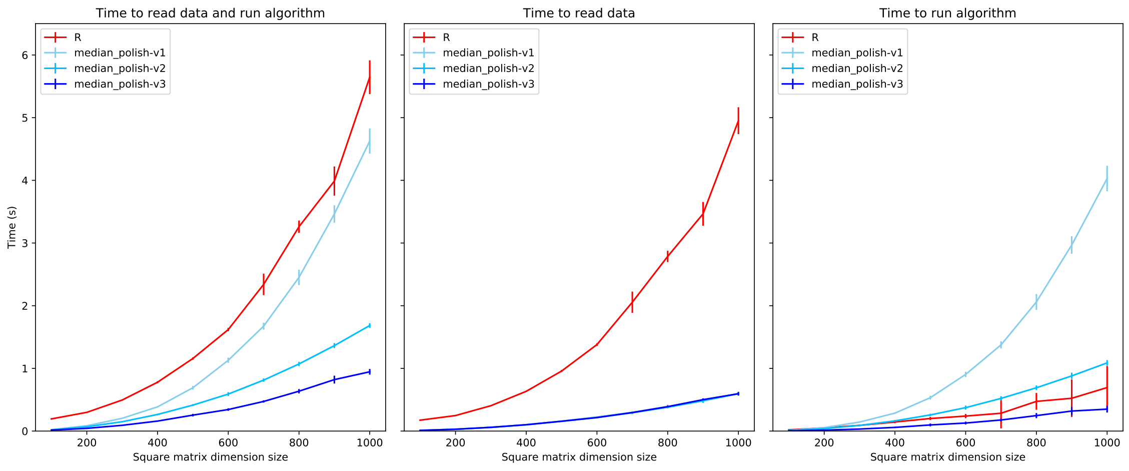

Figure 1: Running time of the `R` and three `C++` implementations. The error bars are given by a single standard deviation over 30 runs.

References

[1] John W Tukey. Exploratory data analysis. 1977.

[2] Christine Brideau, Bert Gunter, Bill Pikounis, and Andy Liaw. Improved statistical methods for hit selection in high-throughput screening. *Journal of biomolecular screening*, 8(6):634–647, 2003.

[3] Xueping Liu, Ingo Vogt, Tanzeem Haque, and Mónica Campillos. Hitpick: a web server for hit identification and target prediction of chemical screenings. *Bioinformatics*, 29(15):1910–1912, 2013.

[4] Nathalie Malo, James A Hanley, Sonia Cerquozzi, Jerry Pelletier, and Robert Nadon. Statistical practice in high-throughput screening data analysis. *Nature biotechnology*, 24(2):167–175, 2006.

[5] John D Emerson and David C Hoaglin. Analysis of two-way tables by medians. *Understanding robust and exploratory data analysis*, pages 165–210, 1983.