Programming with Big Data in R

Package Examples and Demonstrations

The pbdR Core Team

Drew Schmidt, M.Sc.
Remote Data Analysis and Visualization Center,
University of Tennessee, Knoxville

Wei-Chen Chen, Ph.D.
Computer Science and Mathematics Division,
Oak Ridge National Laboratory

George Ostrouchov, Ph.D.
Computer Science and Mathematics Division,
Oak Ridge National Laboratory

Pragneskumar Patel, M.Sc.
Remote Data Analysis and Visualization Center,
University of Tennessee, Knoxville

Version 1.0

This publication was typeset using LaTeX. The illustrations were created using the **ggplot2** package (Wickham, 2009), except for Figure 1.1, which was created in Microsoft Powerpoint.

Contents

Acknowledgement

Schmidt, Ostrouchov, and Patel were supported in part by the project "NICS Remote Data Analysis and Visualization Center" funded by the Office of Cyberinfrastructure of the U.S. National Science Foundation under Award No. ARRA-NSF-OCI-0906324 for NICS-RDAV center. Chen and Ostrouchov were supported in part by the project "Visual Data Exploration and Analysis of Ultra-large Climate Data" funded by U.S. DOE Office of Science under Contract No. DE-AC05-00OR22725.

This work used resources of National Institute for Computational Sciences at the University of Tennessee, Knoxville, which is supported by the Office of Cyberinfrastructure of the U.S. National Science Foundation under Award No. ARRA-NSF-OCI-0906324 for NICS-RDAV center. This work also used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This work used resources of the Newton HPC Program at the University of Tennessee, Knoxville.

We also thank Brian D. Ripley, Kurt Hornik, and Uwe Ligges from the R Core Team for discussing package release issues and helping us solve portability problems on different platforms.

Warning: This document is written to explain the main functions of **pbdDEMO** (Schmidt *et al.*, 2012b), version 0.1-0. Every effort will be made to ensure future versions are consistent with these instructions, but features in later versions may not be explained in this document.

Information about the functionality of this package, and any changes in future versions can be found on website: “Programming with Big Data in R” at `http://r-pbd.org/`.

Part I

Preliminaries

1 Introduction

1.1 What is pbd?

The "Programming with Big Data in R" project (Ostrouchov *et al.*, 2012) (pbd or pbdR for short) is a project that aims to elevate the statistical programming language R (R Core Team, 2012) to leadership-class computing platforms. The main goal is empower data scientists by bringing flexibility and a big analytics toolbox to big data challenges, with an emphasis on productivity, portability, and performance. We achieve this in part by mapping high-level programming syntax to portable, high-performance, scalable, parallel libraries.

Figure 1.1 shows the current state of pbdR packages and how they utilize high-performance libraries. More explicitly, the current pbdR packages are:

- **pbdMPI** — an efficient interface to MPI with a focus on Single Program/Multiple Data (SPMD) parallel programming style.
- **pbdSLAP** — bundles scalable dense linear algebra libraries in double precision for **R**, based on ScaLAPACK version 2.0.2 (Blackford *et al.*, 1997)..
- **pbdNCDF4** — Interface to Parallel Unidata NetCDF4 format data files (NetCDF Group, 2008).
- **pbdBASE** — base distributed classes and methods for the pbdR Project.
- **pbdDMAT** — distributed matrix computational methods, with a focus on linear algebra.
- **pbdDEMO** — set of package demonstrations and examples, and this unifying vignette.

In this vignette, we offer many examples using the above pbdR packages. Many of the examples are high-level applications and may be commonly found in basic Statistics. The purpose is to show how to reuse the pre-existing functions and utilities of pbdR to create minor extensions which can quickly solve problems in an efficient way. The reader is encouraged to reuse and repurpose these functions.

The **pbdDEMO** package consists of two main parts. The first is a collection of roughly 20

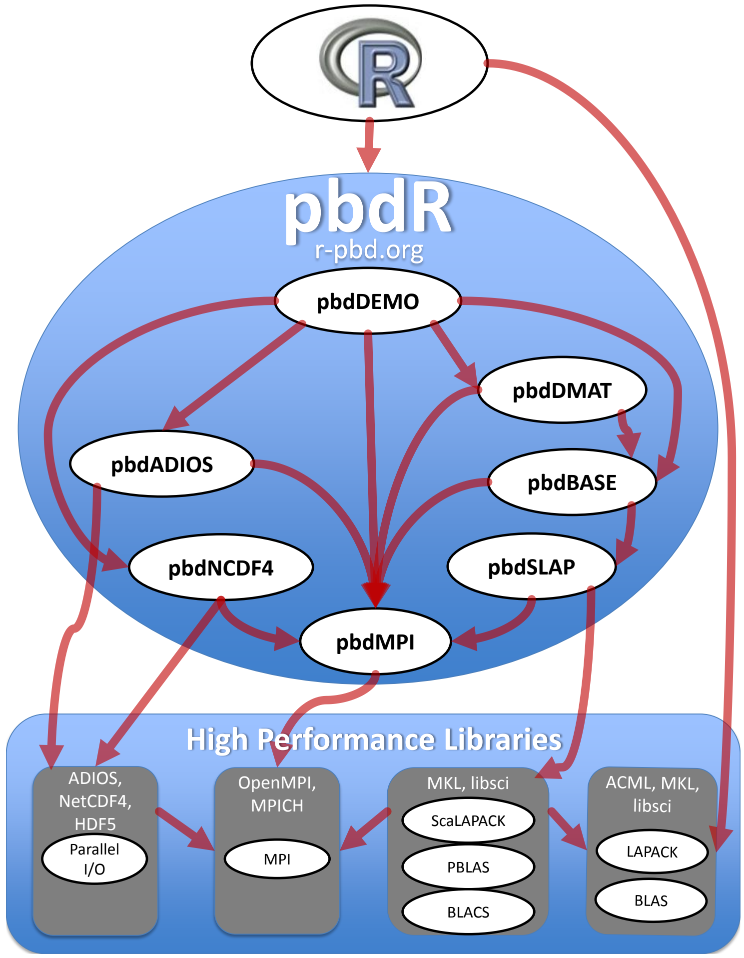

Figure 1.1: pbdR Packages and Their Relationships with Scalable Libraries

package demos. These offer example uses of the various pbdR packages. The second is this vignette, which attempts to offer detailed explanations for the demos, as well as sometimes providing some mathematical or statistical insight. A list of all of the package demos can be found in Section 1.4.

1.2 Why Parallelism? Why pbdR?

It is common, in a document such as this, to justify the need for parallelism. Generally this process goes:

> *Blah blah blah Moore's Law, blah blah Big Data, blah blah blah Concurrency.*

How about this? Parallelism is cool. Any boring nerd can use one computer, but using 10,000 at once is another story. We don't call them ***supercomputers*** for nothing.

But unfortunately, lots of people who would otherwise be thrilled to do all kinds of cool stuff with massive behemoths of computation — computers with names like **KRAKEN** and **TITAN** — are burdened by an unfortunate reality: it's really, really hard. Enter pbdR. Through our project, we put a shiny new set of clothes on high-powered compiled code, making massive-scale computation accessible to a wider audience of data scientists than ever before. Analytics in supercomputing shouldn't just be for the elites.

1.3 Installation

One can download **pbdDEMO** from CRAN at http://cran.r-project.org, and the intallation can be done with the following commands

```
tar zxvf pbdDEMO_0.1-0.tar.gz
R CMD INSTALL pbdDEMO
```

Since **pbdEMO** depends on other pbdR packages, please read the corresponding vignettes if installation of any of them is unsuccessful.

1.4 List of Demos

A full list of demos contained in the **pbdDEMO** package is provided below.

Shell Script

```
### (Use Rscript.exe for windows system)


                      # --------------------- #
                      # II Direct MPI Methods #
                      # --------------------- #

### Chapter 5
# Monte carlo simulation
mpiexec -np 4 Rscript -e "demo(monte_carlo, package='pbdDMAT', ask=F,
   echo=F)"
# Sample mean and variance
mpiexec -np 4 Rscript -e "demo(sample_stat, package='pbdDMAT', ask=F,
   echo=F)"
# Binning
mpiexec -np 4 Rscript -e "demo(binning, package='pbdDMAT', ask=F,
   echo=F)"
# Quantile
mpiexec -np 4 Rscript -e "demo(quantile, package='pbdDMAT', ask=F,
   echo=F)"
# OLS
mpiexec -np 4 Rscript -e "demo(ols, package='pbdDMAT', ask=F, echo=F)"
# Distributed Logic
```

```
mpiexec -np 4 Rscript -e "demo(comparators, package='pbdDMAT', ask=F,
   echo=F)"

                  # ---------------------------- #
                  # III Reading and Managing Data #
                  # ---------------------------- #

### Chapter 6
# Random matrix generation
mpiexec -np 4 Rscript -e "demo(randmat_global, package='pbdDMAT',
   ask=F, echo=F)"
mpiexec -np 4 Rscript -e "demo(randmat_diag_global, package='pbdDMAT',
   ask=F, echo=F)"
mpiexec -np 4 Rscript -e "demo(randmat_local, package='pbdDMAT', ask=F,
   echo=F)"

### Chapter 7
# Reading csv
mpiexec -np 4 Rscript -e "demo(read_csv, package='pbdDMAT', ask=F,
   echo=F)"
# Reading sql
mpiexec -np 4 Rscript -e "demo(read_sql, package='pbdDMAT', ask=F,
   echo=F)"
# Reading netcdf4
mpiexec -np 4 Rscript -e "demo(read_ncdf, package='pbdDMAT', ask=F,
   echo=F)"

### Chapter 8
# Loand/unload balance
mpiexec -np 4 Rscript -e "demo(balance, package='pbdDMAT', ask=F,
   echo=F)"
# SPMD to DMAT
mpiexec -np 4 Rscript -e "demo(spmd_dmat, package='pbdDMAT', ask=F,
   echo=F)"
# Distributed matrix redistributions
mpiexec -np 4 Rscript -e "demo(reblock, package='pbdDMAT', ask=F,
   echo=F)"

                  # ---------------------------- #
                  # IV Distributed Matrix Methods #
                  # ---------------------------- #

### Chapter 9
# Sample statistics revisited
mpiexec -np 4 Rscript -e "demo(sample_stat_dmat, package='pbdDMAT',
   ask=F, echo=F)"
# Verify solving Ax=b at scale
mpiexec -np 4 Rscript -e "demo(verify, package='pbdDMAT', ask=F,
   echo=F)"
# PCA compression
mpiexec -np 4 Rscript -e "demo(pca, package='pbdDMAT', ask=F, echo=F)"
# OLS and predictions
mpiexec -np 4 Rscript -e "demo(ols_dmat, package='pbdDMAT', ask=F,
```

```
    echo=F)"
```

2 Background

2.1 Notation

Note that we tend to use suffix `.spmd` for an object when we wish to indicate that the object is distributed. This is purely for pedagogical convenience, and has no semantic meaning. Since the code is written in SPMD style, you can think of such objects as referring to either a large, global object, or to a processor's local piece of the whole (depending on context). This is less confusing than it might first sound.

We will not use this suffix to denote a global object common to all processors. As a simple example, you could imagine having a large matrix with (global) dimensions $m \times n$ with each processor owning different collections of rows of the matrix. All processors might need to know the values for m and n; however, m and n do not depend on the local process, and so these do not receive the `.spmd` suffix. In many cases, it may be a good idea to invent an S4 class object and a corresponding set of methods. Doing so can greatly improve the usability and readability of your code, but is never strictly necessary. However, these constructions are the foundation of the **pbdBASE** (Schmidt *et al.*, 2012a) and **pbdDMAT** (Schmidt *et al.*, 2012c) packages.

On that note, depending on your requirements in distributed computing with R, it may be beneficial to you to use higher pbdR toolchain. If you need to perform dense matrix operations, or statistical analyses which depend heavily on matrix algebra (linear modeling, principal components analysis, ...), then the **pbdBASE** and **pbdDMAT** packages are a natural choice. The major hurdle to using these tools is getting the data into the appropriate `ddmatrix` format, although we provide many tools to ease the pains of doing so. Learning how to use these packages can greatly improve code performance, and take your statistical modeling in R to previously unimaginable scales.

Again for the sake of understanding, we will at times append the suffix `.dmat` to objects of class `ddmatrix` to differentiate them from the more general `.spmd` object. As with `.spmd`, this carries no semantic meaning, and is merely used to improve the readability of example code (especially when managing both "`.spmd`" and `ddmatrix` objects).

2.2 SPMD Programming with R

Throughout this document, we will be using the "Single Program/Multiple Data", or SPMD, paradigm for distributed computing. Writing programs in the SPMD style is a very natural way of doing computations in parallel, but can be somewhat difficult to properly describe. As the name implies, only one program is written, but the different processors involved in the computation all execute the code independently on different portions of the data. The process is arguably the most natural extension of running serial codes in batch.

Unfortunately, executing jobs in batch is a somewhat unknown way of doing business to the typical R user. While details and examples about this process will be provided in the chapters to follow, the reader is encouraged to see the **pbdMPI** package's vignette (Chen *et al.*, 2012b) first. Ideally, readers should run the demos of the **pbdMPI** package, going through the code step by step.

Part II

Direct MPI Methods

3

MPI for the R User

Cicero once said that "If you have a garden and a library, you have everything you need." So in that spirit, for the next two chapters we will use the MPI library to get our hands dirty and root around in the dirt of low-level MPI programming.

3.1 MPI Basics

In a sense, Cicero (in the above tortured metaphor) was quite right. MPI is all that we *need* in the same way that I might only *need* bread and cheese, but really what I *want* is a pizza. MPI is somewhat low-level and can be quite fiddley, but mastering it adds a very powerful tool to the repertoire of the parallel R programmer, and is essential for anyone who wants to do large scale development of parallel codes.

"MPI" stands for "Message Passing Interface". How it really works goes *well* beyond the scope of this document. But at a basic level, the idea is that the user is running a code on different compute nodes that (usually) can not directly modify objects in each others' memory. In order to have all of the nodes working together on a common problem, data and computation directives are passed around over the network (often over a specialized link called infiniband).

The general process for directly — or indirectly — utilizing MPI goes something like this:

1. Initialize communiator(s).
2. Have each process read in its portion of the data.
3. Perform computations.
4. Communicate results.
5. Shut down the communicator(s).

Some of the above steps may be swept away under a layer of abstraction for the user, but the need may arise where directly interfacing with MPI is not only beneficial, but necessary.

More details and numerous examples using MPI with R are available in the sections to follow, as well as in the **pbdMPI** vignette.

3.2 pbdMPI vs Rmpi

There is another package on the CRAN which the R programmer may use to interface with MPI, namely **Rmpi** (Yu, 2012). There are several issues one must consider when choosing which package to use if one were to only use one of them.

1. (+) **pbdMPI** is easier to install than **Rmpi**
2. (+) **pbdMPI** is easier to use than **Rmpi**
3. (+) **pbdMPI** can often outperform **Rmpi**
4. (+) **pbdMPI** integrates with the rest of pbd
5. (−) **Rmpi** can be used with **foreach** (Analytics, 2012) via **doMPI** (Weston, 2010)
6. (−) **Rmpi** can be used in the master/worker paradigm

We do not believe that the above can be reduced to a zero-sum game with unambiguous winner and loser. Ultimately the needs of the user (or developer) are paramount. We believe that pbd makes a very good case for itself, especially in the SPMD style, but it can not satisfy everyone. However, for the remainder of this section, we will present the case for several of the, as yet, unsubstantiated pluses above.

In the case of ease of use, **Rmpi** uses bindings very close to the level as they are used in C's MPI API. Specifically, whenever performing, for example, a reduction such as allreduce, you must specify the type of your data. For example, using **Rmpi**'s API

```
mpi.allreduce(x, type = 1)
```

would perform the sum allreduce if the object `x` consists of integer data, while

```
mpi.allreduce(x, type = 2)
```

would be used if `x` consists of doubles. However, with **pbdMPI**

```
allreduce(x)
```

is used for both by making use of R's S4 system of object oriented programming. This is not mere code golfing[1] that we are engaging in. The concept of what "type" your data is in R is

[1]See https://en.wikipedia.org/wiki/Code_golf

fairly foreign to most R users, and misusing the `type` argument in **Rmpi** is a very easy way to crash your program. Instead, we take the approach of adding a small abstraction layer on top (which we intend to show does not negatively impact performance in general) so that the user need not worry about such details.

In terms of performance, **pbdMPI** can greatly outperform **Rmpi**. We present here the results of a benchmark we performed comparing the allgather operation between the two packages (Schmidt *et al.*, 2012e). The benchmark consisted of calling the respective allgather function from each package on a randomly generated $10,000 \times 10,000$ distributed matrix with entries coming from the standard normal distribution, using different numbers of processors. Table 3.1 shows the results for this test, and in each case, **pbdMPI** is the clear victor.

Table 3.1: Runtimes (seconds) for **Rmpi** and **pbdMPI**.

Cores	**Rmpi**	**pbdMPI**	Speedup
32	24.6	6.7	3.67
64	25.2	7.1	3.55
128	22.3	7.2	3.10
256	22.4	7.1	3.15

Whichever package you choose, whichever your favorite, for the remainder of this document we will be using (either implicitly or explicitly) **pbdMPI**.

3.3 Timing MPI Tasks

Measuring run time is a fundamental performance measure in computing. However, in parallel computing, not all "parallel components" (e.g. threads, or MPI processes) will take the same amount of time to complete a task, even when all tasks are given completely identical jobs. So measuring "total run time" begs the question, run time of what?

To help, we offer a timing function `demo.timer()` which can wrap segments of code much in the same way that `system.time()` does. However, the three numbers reported by `demo.timer()` are: (1) the minimum elapsed time measured across all processes, (2) the average elapsed time measured across all processes, and (3) the maximum elapsed time across all processes. The code for this function is listed below:

Timer Function

```
demo.timer <- function(timed)
{
  ltime <- system.time(timed)[3]

  mintime <- allreduce(ltime, op='min')
  maxtime <- allreduce(ltime, op='max')

  meantime <- allreduce(ltime, op='sum') / comm.size()

```

```
  return( c(min=mintime, mean=meantime, max=maxtime) )
}
```

4 Basic Statistics Examples

This section introduces a few simple examples and explains a little about computing with distributed data directly over MPI. These implemented examples/functions are partly selected from the Cookbook of HPSC website (Chen and Ostrouchov, 2011) at http://thirteen-01.stat.iastate.edu/snoweye/hpsc/?item=cookbook. Please see more details there.

4.1 Monte Carlo Simulation

Example: Compute a numerical approximation for π.

The demo command is

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(monte_carlo,'pbdDEMO',ask=F,echo=F)"
```

This is a simple Monte Carlo simulation example for numerically estimating π. Suppose we sample N uniform observations (x_i, y_i) inside (or perhaps on the border of) the unit square $[0, 1] \times [0, 1]$, where $i = 1, 2, \ldots, N$. Then

$$\pi \approx 4\frac{L}{N} \tag{4.1}$$

where $0 \leq L \leq N$ is the number of observations sampled satisfying

$$x_i^2 + y_i^2 \leq 1 \tag{4.2}$$

The intuitive explanation for this is strategy which is sometimes given belies a misunderstanding of infinite cardinalities, and infinite processes in general. We are not *directly* approximating an area through this sampling scheme, because to do so with a finite-point sampling scheme would be madness requiring a transfinite process. Indeed, let S_N be the collection of elements satisfying inequality (4.2). Then note that for each $N \in \mathbb{N}$ that the area of S_N is precisely 0. Whence,

$$\lim_{N\to\infty} Area(S_N) = 0$$

This bears repeating. Finite sampling of an uncountable space requires uncountably many such sampling operations to "fill" the infinite space. For a proper treatment of set theoretic constructions, including infinite cardinals, see (Kunen, 1980).

One could argue that we are evaluating a ratio of integrals with each using the counting measure, which satisfies technical correctness but is far from clear. Now indeed, certain facts of area are vital here, but some care should be taken in the discussion as to what exactly justifies our claim in (4.1).

In reality, we are evaluating the probability that someone throwing a 0-dimensional "dart" at the unit square will have that "dart" also land below the arc of the unit circle contained within the unit square. Formally, let U_1 and U_2 be random uniform variables, each from the closed unit interval $[0, 1]$. Define the random variable

$$X := \begin{cases} 1, & U_1^2 + U_2^2 \leq 1 \\ 0, & \text{otherwise} \end{cases}$$

Let $V_i = U_i^2$ for $i = 1, 2$. Then the expected value

$$\begin{aligned}
E[X] &= P(V_1 + V_2 \leq 1) \\
&= \int_0^1 \int_0^{1-V_1} p(V_1, V_2) dV_2 dV_1 \\
&= \int_0^1 \int_0^{1-V_1} \left(\frac{1}{2\sqrt{V_1}}\right)\left(\frac{1}{2\sqrt{V_2}}\right) dV_2 dV_1 \\
&= \frac{1}{2}\int_0^1 \left(\frac{1-V_1}{V_1}\right)^{1/2} dV_1 \\
&= \frac{1}{2}\left[V_1\left(\frac{1-V_1}{V_1}\right)^{1/2} - \frac{1}{2}\arctan\left(\frac{\left(\frac{1-V_1}{V_1}\right)^{1/2}(2V_1 - 1)}{2(V_1 - 1)}\right)\right]_{V_1 \to 0}^{V_1 \to 1} \\
&= \frac{1}{2}\left[\frac{\pi}{4} + \frac{\pi}{4}\right]
\end{aligned}$$

and by sampling observations X_i for $i = 1, \ldots, N$, by the Strong Law of Large Numbers

$$\bar{X}_N \xrightarrow{a.s.} \frac{\pi}{4} \quad \text{as } N \to \infty$$

In other words,

$$P\left(\lim_{N\to\infty} \bar{X}_N = \frac{\pi}{4}\right) = 1$$

Whence,

$$\frac{L}{N} \xrightarrow{a.s.} \frac{\pi}{4} \quad \text{as } N \to \infty$$

But because no one is going to read that, and if they do they'll just call the author a grumpy old man, the misleading picture you desire can be found in Figure 4.1. And to everyone who found this looking for a homework solution, you're welcome.

The key step of the demo code is in the following block:

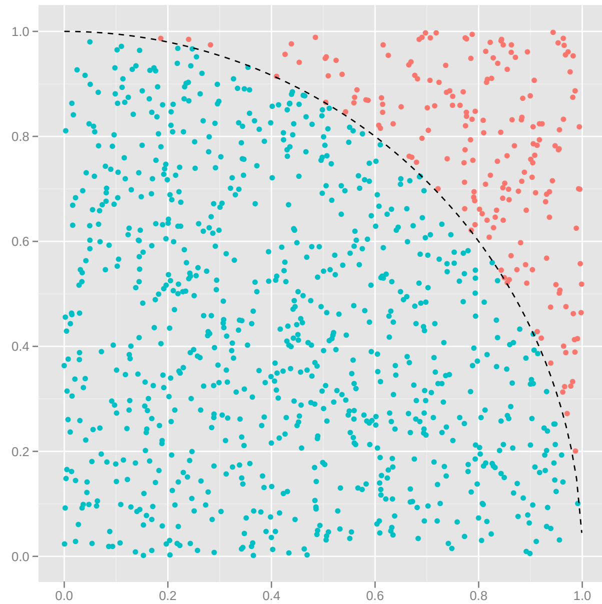

Figure 4.1: Approximating π by Monte Carlo methods

R Code

```r
1 N.spmd <- 1000
2 X.spmd <- matrix(runif(N.spmd * 2), ncol = 2)
3 r.spmd <- sum(rowSums(X.spmd^2) <= 1)
4 ret <- allreduce(c(N.spmd, r.spmd), op = "sum")
5 PI <- 4 * ret[2] / ret[1]
6 comm.print(PI)
```

In line 1, we specify sample size in `N.spmd` for each processor, and $N = D \times$ `N.spmd` if D processors are executed. In line 2, we generate samples in `X.spmd` for every processor. In line 3, we compute how many of the "radii" are less than or equal to 1 for each processors. In line 4, we call `allreduce()` to obtain total numbers across all processors. In line 5, we use the Equation (4.1). Since SPMD, `ret` is common on all processors, and so is `PI`.

4.2 Sample Mean and Sample Variance

Example: Compute sample mean/variance for distributed data.

The demo command is

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(sample_stat,'pbdDEMO',ask=F,echo=F)"
```

Suppose $\boldsymbol{x} = \{x_1, x_2, \ldots, x_N\}$ are observed samples, and N is potentially very large. We can distribute $\boldsymbol{x}$ in 4 processors, and each processor receives a proportional amount of data. One simple way to compute sample mean $\bar{x}$ and sample variance s_x is based on the formulas:

$$
\begin{aligned}
\bar{x} &= \frac{1}{N}\sum_{n=1}^{N} x_n \\
&= \sum_{n=1}^{N} \frac{x_n}{N}
\end{aligned}
\tag{4.3}
$$

and

$$
\begin{aligned}
s_x &= \frac{1}{N-1}\sum_{n=1}^{N}(x_n - \bar{x})^2 \\
&= \frac{1}{N-1}\sum_{n=1}^{N} x_n^2 - \frac{2\bar{x}}{N-1}\sum_{n=1}^{N} x_n + \frac{1}{N-1}\sum_{n=1}^{N}\bar{x}^2 \\
&= \sum_{n=1}^{N}\left(\frac{x_n^2}{N-1}\right) - \frac{N\bar{x}^2}{N-1}
\end{aligned}
\tag{4.4}
$$

where expressions (4.3) and (4.4) are one-pass algorithms, which are potentially faster than the first expressions, especially for large N. However, this method of computing the variance in one pass can suffer from round-off errors, and so in general is not numerically stable. We provide this here for demonstration purposes only. Additionally, only the first and second moments are implemented, while the extension of one-pass algorithms to higher order moments is also possible.

The demo generates random data on 4 processors, then utilizes the `mpi.stat()` function:

R Code

```r
mpi.stat <- function(x.spmd){
  ### For mean(x).
  N <- allreduce(length(x.spmd), op = "sum")
  bar.x.spmd <- sum(x.spmd / N)
  bar.x <- allreduce(bar.x.spmd, op = "sum")

  ### For var(x).
  s.x.spmd <- sum(x.spmd^2 / (N - 1))
  s.x <- allreduce(s.x.spmd, op = "sum") - bar.x^2 * (N / (N -
     1))

  list(mean = bar.x, s = s.x)
} # End of mpi.stat().
```

where `allreduce()` in **pbdMPI** (Chen *et al.*, 2012a) can be utilized in this examples to aggregate local information across all processors.

4.3 Binning

Example: Find binning counts for distributed data.

The demo command is

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(binning,'pbdDEMO',ask=F,echo=F)"
```

Binning is a classical problem in statistics which helps to quickly summarize the data structure by setting some "breaks" between the minimum and maximum values. This is a particularly useful tool for constructing histograms, as well as categorical data analysis.

The demo generates random data on 4 processors, then utilizes the `mpi.bin()` function:

R Code

```r
mpi.bin <- function(x.spmd, breaks = pi / 3 * (-3:3)){
  bin.spmd <- table(cut(x.spmd, breaks = breaks))
  bin <- as.array(allreduce(bin.spmd, op = "sum"))
  dimnames(bin) <- dimnames(bin.spmd)
  class(bin) <- class(bin.spmd)
  bin
} # End of mpi.bin().
```

This simple implementation utilizes R's own `table()` function to obtain local counts, then calls `allreduce()` to obtain global counts on all processors.

4.4 Quantile

Example: Compute sample quantile order statistics for distributed data.

The demo command is

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(quantile,'pbdDEMO',ask=F,echo=F)"
```

Another fundamental tool in the statistician's toolbox is finding quantiles. Quantiles are points taken from the cumulative distribution function. Formally, a *q-quantile* (or *q-tile*) with $q \in [0, 1]$

of a random variable X is any value θ_q such that[1]

$$P(X \leq \theta_q) \geq q \qquad \text{and}$$
$$P(X \geq \theta_q) \leq 1 - q$$

Note that by this definition, a quantile neither need exist or be unique. Indeed, for the former, consider the standard normal distribution with $q = 1$, and for the latter consider the probability 0 values of a uniform distribution. Perhaps to narrow the scope of these problems, another common definition is

$$\theta_q = \inf\{x \mid P(X \leq x) \geq q\}$$

In this example, we will be estimating quantiles from a sample. Doing so requires sub-dividing the data into q (almost) evenly sized subsets, giving rise to the language k'th q-tile, for integers $0 < k < \frac{1}{q}$.

Before proceeding, we wish to make very clear the distinction between a theoretical quantile and quantile estimation, as many web pages confuse these two topics. A quantile's estimation from a sample requires ordering and can take many forms; in fact, there are nine possible such forms in R's own `quantile()` function (see `help(quantile)` in R). The definitions of Kendall and Cramer may be the source of all the confusion (Benson, 1949). Kendall's definition, conflating the term "quantile" with the act of quantile estimation, seems to have entered most dictionaries (and Wikipedia), whereas mathematical statistics favors the more general and simple definition of Cramer.

This example can be extended to construct Q-Q plots, compute cumulative density function estimates and nonparametric statistics, as well as solve maximum likelihood estimators. This is perhaps an inefficient implementation to approximate a quantile and is not equivalent to the original `quantile()` function in R. But in some sense, it should work well at a large scale. The demo generates random data on 4 processors, then utilizes the `mpi.quantile()`:

R Code

```r
mpi.quantile <- function(x.spmd, prob = 0.5){
  if(sum(prob < 0 | prob > 1) > 0){
    stop("prob should be in (0, 1)")
  }

  N <- allreduce(length(x.spmd), op = "sum")
  x.max <- allreduce(max(x.spmd), op = "max")
  x.min <- allreduce(min(x.spmd), op = "min")

  f.quantile <- function(x, prob = 0.5){
    allreduce(sum(x.spmd <= x), op = "sum") / N - prob
  }

  uniroot(f.quantile, c(x.min, x.max), prob = prob[1])$root
} # End of mpi.quantile().
```

[1] This definition is due to the mathematical statistician Herman Rubin: http://mathforum.org/kb/message.jspa?messageID=406278

Here, a numerical function is solved by using `uniroot()` to find out the appropriate value where the cumulative probability is less than or equal to the specified quantile. Specifically, it finds the zero, or root, of the monotone `f.quantile()` function. This simple example shows that with just a little effort, direct MPI methods are greatly applicable on large scale data analysis and likelihood computing.

Note that in the way that the `uniroot()` call is used above, we are legitimately operating in parallel and on distributed data. Other optimization functions such as `optim()` and `nlm()` can be utilized in the same way.

4.5 Ordinary Least Squares

Example: Compute ordinary least square solutions for SPMD distributed data.

The demo command is

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(ols,'pbdDEMO',ask=F,echo=F)"
```

Ordinary least squares (OLS) is perhaps *the* fundamental tool of the statistician. The goal is to find a solution $\boldsymbol{\beta}$ such that

$$||\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{y}||_2^2 \tag{4.5}$$

is minimized. In statistics, we tend to prefer to think of the problem as being of the form

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon} \tag{4.6}$$

where $\boldsymbol{y}$ is $N \times 1$ observed vector, $\boldsymbol{X}$ is $N \times p$ (possibly designed) matrix which is often assumed to have full rank (more on that later), and $N >> p$, $\boldsymbol{\beta}$ is the unknown parameter to be estimated, and $\boldsymbol{\epsilon}$ is errors and to be minimized in norm.

Note that above, we do indeed mean (in fact, stress) *a* solution to the linear least squares problem. For many applications a statistician will face, expression (4.5) will actually have a unique solution. But this is not always the case, and trouble often arises when the model matrix is rank-deficient. Indeed, in this case it may occur that there is an infinite family of solutions. So typically we go further and demand that a solution $\boldsymbol{\beta}$ be such that $||\boldsymbol{\beta}||_2$ is at least as small as the corresponding norm of any other solution (although even this may not guarantee uniqueness).

A properly thorough treatment of the problems involved here go beyond the scope of this document, and require the reader have in-depth familiarity with linear algebra. For our purposes, the concise explanation above will suffice.

In the full rank case, we can provide an analytical, "closed-form" solution to the problem. In this case, the classical Maximum Likelihood solution is given by:

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{4.7}$$

This example can be also generalized to weighted least square (WLS), and linear mixed effect models (LME).

The implementation is straight forward:

R Code

```
if(length(y.spmd) != nrow(X.spmd)){
  stop("length(y.spmd) != nrow(X.spmd)")
}

t.X.spmd <- t(X.spmd)
A <- allreduce(t.X.spmd %*% X.spmd, op = "sum")
B <- allreduce(t.X.spmd %*% y.spmd, op = "sum")

solve(matrix(A, ncol = ncol(X.spmd))) %*% B
```

While this is a fine demonstration of the power of "getting your hands dirty", this approach is only efficient for small N and small p. This is, in large part, because the operation is not "fully parallel", in that the solution is serial and replicated on all processors. Worse, directly computing

$$\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}$$

has numerical stability issues. Instead, it is generally better (although much slower) to take an orthogonal factorization of the data matrix. See Appendix A for details.

Finally, all of the above assumes that the model matrix $\boldsymbol{X}$ is full rank. However, we have implemented an efficient method of solving linear least squares problems in **pbdDMAT**'s `lm.fit()` method for distributed matrices. This method uses a fully parallel rank-revealing QR decomposition to find the least squares solution. So for larger problems, and especially those where numerical accuracy is important or rank-degeneracy is a possibility, it is much better to simply convert `y.spmd` and `X.spmd` into the block-cyclic format as in Part IV and utilize **pbdBASE** and **pbdDMAT** for all matrix computations.

4.6 Distributed Logic

Example: Manage comparisons across all MPI processes.

The demo command is

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(comparators,'pbdDEMO',ask=F,echo=F)"
```

This final MPI example is not statistical in nature, but is very useful all the same, and so we include it here. The case frequently arises where the MPI programmer will need to do logical comparisons across all processes. The idea is to extend the very handy `all()` and `any()` base R functions to operate similarly on distributed logicals.

You could do this directly. Say you want to see if any processes have `TRUE` stored in the variable `localLogical`. This amounts to something on the order of:

R Code

```
globalLogical <- as.logical(allreduce(localLogical, op='max')
```

Or you can use the function `comm.any()` from **pbdMPI**:

R Code

```
globalLogical <- comm.any(localLogical)
```

which essentially does the same thing, but is more concise. Likewise, there is a `comm.all()` function, which in the equivalent "long-form" above would use `op='min'`.

The demo for these functions consists of two parts. For the first, we do a simple demonstration of how these functions behave:

R Code

```
rank <- comm.rank()

comm.cat("\ntest value:\n", quiet=T)
test <- (rank > 0)
comm.print(test, all.rank=T, quiet=T)

comm.cat("\ncomm.all:\n", quiet=T)
test.all <- comm.all(test)
comm.print(test.all, all.rank=T, quiet=T)

comm.cat("\ncomm.any:\n", quiet=T)
test.any <- comm.any(test)
comm.print(test.any, all.rank=T, quiet=T)
```

which should have the output:

```
test value:
[1] FALSE
[1] TRUE
[1] TRUE
[1] TRUE

comm.all:
[1] FALSE
[1] FALSE
[1] FALSE
[1] FALSE
```

```
comm.any:
[1] TRUE
[1] TRUE
[1] TRUE
[1] TRUE
```

The demo also has another use case which could be very useful to a developer. You may be interested in trying something on only one processor and then shutting down all MPI processes if problems are encountered. To do this in SPMD style, you can create a variable on all processes to track whether a problem has been encountered. Then after critical code sections, use `comm.any()` to update and act appropriately. A very simple example is provided below.

R Code

```r
need2stop <- FALSE

if (rank==0){
  need2stop <- TRUE
}

need2stop <- comm.any(need2stop)

if (need2stop)
  stop("Problem :[")
```

Part III

Reading and Managing Data

5 Random Distributed Matrices

The **pbdBASE** and **pbdDMAT** packages offer a distributed matrix class, `ddmatrix`, as well as a collection of high-level methods for performing common matrix operations. For example, if you want to compute the mean of an R matrix `x`, you would call

```
mean(x)
```

That's exactly the same command you would issue if `x` is no longer an ordinary R matrix, but a distributed matrix. These methods range from simple, embarrassingly parallel operations like sums and means, to tightly coupled linear algebra operations like matrix-matrix multiply and singular value decomposition.

Unfortunately, these higher methods come with a different cost: getting the data into the right format, namely the distributed matrix class. This can be especially frustrating because we assume that the any object of class `ddmatrix` is *block cyclically distributed*. This concept is discussed at length in the **pbdBASE** vignette (Schmidt *et al.*, 2012d), and we do not intend to discuss the concept of a block cyclic data distribution at length herein. However, we will demonstrate several examples of getting data into and out of the distributed block cyclic matrix format.

Once the hurdle of getting the data into the "right format" is out of the way, these methods offer very simple syntax (designed to mimic R as closely as possible) with the ability to scale computations on very large distributed machines. So the process of getting the data into the correct format must be addressed. We begin dealing with this issue in the simplest way possible, namely by using randomly generated data.

5.1 Fixed Global Dimension

Example: randomly generate distributed matrices with random normal data of specificed global dimension.

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(randmat_global,'pbdDEMO',ask=F,echo=F)"
```

This demo shows 3 separate ways that one can generate a random normal matrix with specified global dimension. The first two generate the matrix in full on at least one processor and distribute(s) the data, while the last method generates locally only what is needed. As such, the first two can be considered demonstrations with what to do when you have data read in on one processor and need to distribute it out to the remaining processors, but for the purposes of building a randomly generated distributed matrix, they are not particularly efficient strategies.

The basic idea is as follows. If we have a matrix `x` stored on processor 0 and `NULL` on the others, then we can distribute it out as an object of class `ddmatrix` via the command `as.ddmatrix()`. For example

```r
if (comm.rank()==0){
  x <- matrix(rnorm(100), nrow=10, ncol=10)
} else {
  x <- NULL
}

dx <- as.ddmatrix(x)
```

will distribute the required data to the remaining processors. We note for clarity that this is not equivalent to sending the full matrix to all processors and then throwing away all but what is needed. Only the required data is communicated to the processors.

That said, having all of the data on all processors can be convenient while testing, if only for being more minimalistic in the amount of code/thinking required. To do this, one need only do the following:

```r
x <- matrix(rnorm(100), nrow=10, ncol=10)

dx <- as.ddmatrix(x)
```

Here, each processor generates the full, global matrix, then throws away what is not needed. Again, this is not efficient, but the code is concise, so it is extremely useful in testing. Now, this assumes you are using the same seed on each processor. This can be managed using the **pbdMPI** function `comm.set.seed()`, as in the demo script. For more information, see that package's documentation.

Finally, you can generate locally only what you need. The demo script does this via the **pbdDEMO** package's `Hnorm()` or "huge normal" function. There are two others provided, namely

`Hconst()` and `Hunif()`. The naming convention was chosen because the latter most function name makes the author laugh. Internally, these "huge" functions rely on a much stronger working knowledge of the underlying data structure than most will be comfortable with. However, for the sake of completeness, we will briefly examine `Hnorm()`.

Hnorm()

```
Hnorm <- function(dim, bldim, mean=0, sd=1, ICTXT=0)
{
  if (length(bldim)==1L)
    bldim <- rep(bldim, 2L)

  ldim <- base.numroc(dim=dim, bldim=bldim, ICTXT=ICTXT, fixme=FALSE)

  if (any(ldim < 1L)){
    xmat <- matrix(0)
    ldim <- c(1, 1)
  }
  else
    xmat <- matrix(rnorm(prod(ldim), mean=mean, sd=sd), nrow=ldim[1L], ncol=ldim[2L])

  dx <- new("ddmatrix", Data=xmat,
            dim=dim, ldim=ldim, bldim=bldim, CTXT=ICTXT)

  return(dx)
}
```

The concise explanation is that the `base.numroc()` utility determines the size of the local storage. This is all very well documented in the **pbdBASE** documentation, but since no one even pretends to read that stuff, `NUMROC` is a ScaLAPACK tool, which means "`NUM`ber of `R`ows `O`r `C`olumns." The function `base.numroc()` is an implementation in R which calculates the number of rows *and* columns at the same time (so it is a bit of a misnomer, but preserved for historical reasons).

More precisely, it calculates the local storage requirements given a global dimension `dim`, a blocking factor `bldim`, and a BLACS context number `ICTXT`. The extra argument `fixme` determines whether or not the lowest value returned should be 1. If `fixme==FALSE` and any of the returned local dimensions are less than 1, then that processor does not actually own any of the global matrix — it has no local storage. But something must be stored, and so we default this to `matrix(0)`, the 1×1 matrix with single entry 0.

5.2 Diagonal, Fixed Global Dimension

*Example: randomly generate **diagonal** distributed matrices with random normal data of specificed global dimension.*

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e
    "demo(randmat_diag_global,'pbdDEMO',ask=F,echo=F)"
```

In R, the `diag()` function serves two purposes. If given a matrix, it produces a vector containing the diagonal entries of that matrix; but if given a vector, it constructs a diagonal matrix whose diagonal is that vector. And so for example, the zero and identity matrices of any dimension can quickly be constructed via:

Diagonal Matrices in R

```r
diag(x=0, nrow=10, ncol=10) # zero matrix
diag(x=1, nrow=10, ncol=10) # identity matrix
```

Both of the above functionalities of `diag()` are available for distributed matrices; however we will only focus on the latter.

When you wish to construct a diagonal distributed matrix, you can easily do so by using the additional `type=` argument to our `diag()` method. By default, `type="matrix"`, though the user may specify `type="ddmatrix"`. If so, then as one might expect, the optional `bldim=` and `ICTXT=` arguments are available. So with just a little bit of tweaking, the above example becomes:

Diagonal Matrices in pbdR

```r
diag(x=0, nrow=10, ncol=10, type="ddmatrix") # zero
    (distributed) matrix
diag(x=1, nrow=10, ncol=10, type="ddmatrix") # identity
    (distributed) matrix
```

In fact, the `type=` argument employs partial matching, so if we really want to be lazy, then we could simply do the following:

Diagonal Matrices in pbdR

```r
diag(x=0, nrow=10, ncol=10, type="d") # zero (distributed) matrix
diag(x=1, nrow=10, ncol=10, type="d") # identity (distributed)
    matrix
```

Beyond the above brief explanation, the demo for this functionality is mostly self-contained, although we do employ the `redistribute()` function to fully show off local data storage. This function is explained in detail in Chapter 7.

5.3 Fixed Local Dimension

Example: randomly generate distributed matrices with random normal data of specificed local dimension.

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(randmat_local,'pbdDEMO',ask=F,echo=F)"
```

This is similar to the above, but with a critical difference. Instead of specifying a fixed *global* dimension and then go determine what the local storage space is, instead we specify a fixed *local* dimension and then go figure out what the global dimension should be. This can be useful for testing weak scaling of an algorithm, where different numbers of cores are used with the same local problem size.

To this end, the demo script utilizes the `Hnorm.local()` function, which has the user specify a local dimension size that all the processors should use, as well as a blocking factor and BLACS context value.

Hnorm.local()

```
Hnorm.local <- function(ldim, bldim, mean=0, sd=1, ICTXT=0)
{
  if (length(bldim)==1L)
    bldim <- rep(bldim, 2L)

  blacs_ <- base.blacs(ICTXT=ICTXT)
  nprows <- blacs_$NPROW
  npcols <- blacs_$NPCOL

  dim <- c(nprows*ldim[1L], npcols*ldim[2L])

  if (any( (dim %% bldim) != 0 )){
    comm.cat("WARNING : at least one margin of 'bldim' does not
       divide the global dimension.\n", quiet=T)

    bldim[1L] <- nbd(dim[1L], bldim[1L])
    bldim[2L] <- nbd(dim[2L], bldim[2L])
    comm.cat(paste("Using bldim of ", bldim[1L], "x", bldim[2L],
       "\n\n", sep=""), quiet=T)
  }

  Data <- matrix(rnorm(prod(ldim), mean=mean, sd=sd),
      nrow=ldim[1L], ncol=ldim[2L])

```

```
  dx <- new("ddmatrix", Data=Data,
            dim=dim, ldim=ldim, bldim=bldim, CTXT=ICTXT)

  return(dx)
}
```

Now here things get somewhat tricky, because in order for this matrix to exist at all, each margin of the blocking factor must divide (as an integer) the corresponding margin of the global dimension. To better understand why this is so, the reader is suggested to read the **pbdBASE** vignette. But if you already understand or are merely willing to take it on faith, then you surely grant that this is a problem.

So here, we assume that the local dimension is chosen appropriately, but it is possible that a bad blocking factor is supplied by the user. Rather than halt with a stop error, we attempt to find the next best blocking factor possible. We do this with a simple "next best divisor" function:

nbd()

```
nbd <- function(n, d)
{
  if (n < d)
    stop("'n' may not be smaller than 'd'")

  ret <- .Fortran("NBD",
                  as.integer(n), as.integer(d),
                  PACKAGE="pbdDEMO")$D

  return( ret )
}
```

which is just a shallow wrapper on the Fortran code:

NBD

```
      SUBROUTINE NBD(N, D)
      INTEGER N, D, I, TEST

      DO 10 I = D+1, N-1, 1
        TEST = MOD(N, I)
        IF (TEST.EQ.0) THEN
          D = I
          RETURN
        END IF
   10 CONTINUE

      D = N
      RETURN
      END
```

Even those who don't know Fortran should easily be able to see what is going on here. We are given integers `N` and `D`, and we loop over the integers inbetween these two until we find one which divides `N`.

So going back to the `Hnorm.local()` function, the second `if` block contains the readjusting (as necessary) of the blocking factors. Then the local data matrix is generated and wrapped up in its class before being returned — everything else is just sugar.

6 Reading Data

As we mentioned at the beginning of the discussion on distributed matrix methods, most of the hard work in using these tools is getting the data into the right format. Once this hurdle has been overcome, the syntax will magically begin to look like native R syntax. Some insights into this difficulty can be seen in the previous section, but now we tackle the problem head on: how do you get real data into the distributed matrix format?

6.1 CSV Files

Example: Read data from a csv directly into a distributed matrix.

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(read_csv,'pbdDEMO',ask=F,echo=F)"
```

It is simple enough to read in a csv file serially and then distribute the data out to the other processors. This process is essentially identical to one of the random generation methods in Section 5.1. For the sake of completeness, we present a simple example here:

```r
if (comm.rank()==0){ # only read on process 0
  x <- read.csv("myfile.csv")
} else {
  x <- NULL
}

dx <- as.ddmatrix(x)
```

However, this is inefficient, especially if the user has access to a parallel file system. In this case, several processes should be used to read parts of the file, and then distribute that data out to the larger process grid. Although really, the user should not be using csv to store large amounts of data because it always requires a sort of inherent "serialness". Regardless, a demonstration of how this is done is useful. We can do so via the **pbdDEMO** package's function `read.csv.ddmatrix` on an included dataset:

Reading a CSV with Multiple Readers

```
dx <- read.csv.ddmatrix("../extra/data/x.csv",
                        sep=",", nrows=10, ncols=10,
                        header=TRUE, bldim=4,
                        num.rdrs=2, ICTXT=0)

print(dx)
```

The code powering the function itself is quite complicated, going well beyond the scope of this document. To understand it, the reader should see the advanced sections of the **pbdBASE** vignette.

6.2 SQL Databases

Example: Read data from a sql database directly into a distributed matrix.

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(read_sql,'pbdDEMO',ask=F,echo=F)"
```

Just as above, we can use a SQL database to read in our data, powered by the **sqldf** package (Grothendieck, 2012). Here it is assumed that the data is stored in the database in a structure that is much the same as a csv is stored on disk. Internally, the query performed is:

```
sqldf(paste("SELECT * FROM ", table, " WHERE rowid  = 1"),
   dbname=dbname)
```

To use a more complicated query for a database with differing structure, it should be possible (no promises) to substitute this line of the `read.sql.ddmatrix()` function for the desired query. However, as before, much of the rest of the tasks performed by this function go beyond the scope of this document. However, they are described in the **pbdBASE** package vignette.

6.3 NetCDF4 Files

Example: Read data from a netcdf4 file, perform matrix computations, and write results to disk.

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(red_ncdf,'pbdDEMO',ask=F,echo=F)"
```

WORK IN PROGRESS

7 Redistribution Methods

One final challenge similar to, but distinct from reading in data is managing data which has already been read into the R processes. Throughout this chapter, we will be making reference to several particulars to the block-cyclic data type used for objects of class `ddmatrix`. In particular, a working knowledge of the block-cyclic data structure and their relationship with BLACS contexts is most useful for the content to follow. As such, the reader is *strongly* encouraged to be familiar with the content of the **pbdBASE** vignette before proceeding.

7.1 Distributed Matrix Redistributions

Example: Convert between different distributed matrix distributions.

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(reblock,'pbdDEMO',ask=F,echo=F)"
```

The distributed matrix class `ddmatrix` has two components which can be specified, and modified, by the user to drastically affect the composition of the distributed matrix. In particular, these are the object's block-cyclic blocking factor `bldim`, and the BLACS communicator number `CTXT` which sets the 2-dimensional processor grid.

Thankfully, redistributing is a fairly simple process; though we would emphasize that **this is not free of cost**. Reshaping data, especially at scale, can be much more expensive in total than even computation time. That said, sometimes data must move. It is better to get the job done slowly than to "take your ball and go home" with no results. But we caution that if redistribution can be avoided, then it should, at all costs.

The demo relies on a utility from the **pbdBASE** package, namely `redistribute()`. As the name implies, this method is for "reshaping" a block-cyclically distributed matrix of one kind to another. Specifically, this takes an object of class `ddmatrix` as both an input and an output; i.e., and to emphasize the title of the chapter, this is not a method of *distribution* but *redistribution*.

For example, if I have a distributed matrix `dx` and I wish to reshape the distributed matrix so that it now has blocking dimension `newbldim` and is distributed across BLACS context `newCTXT`, then I need merely call:

```
dy <- redistribute(dx, bldim=newbldim, ICTXT=newCTXT)
```

Assuming the data is block cyclic of *any* kind, including degenerate cases, we can convert it to a block cyclic format of any other kind we wish via this `redistribute()` function. The only requirement is that the two different distributions have at least 1 processor in common, and so using the default BLACS contexts (0, 1, and 2) is always acceptable.

7.2 Implicit Redistributions

There are several useful functions which apply to distributed matrices, but require a data redistribution as in Section 7, whether the user realizes it or not. These functions are listed in

Function	**Example**	**Package**	**Effect**
`‘[‘`	`dx[, -1]`	**pbdBASE**	Row/Column extraction and subsetting
`na.exclude()`	`na.exclude(dx)`	**pbdBASE**	Drop rows with `NA`’s
`apply()`	`apply(dx, 2, sd)`	**pbdDMAT**	Applies function to margin

Table 7.1: Distributed Matrix Methods with Implicit Data Redistributions

Table 7.1. By default, these functions will re-distribute back to the original data distribution after having performed the initial (necessary) redistribution and performed the requested operations. That is, by default, the problem of managing different data distributions is hidden from the user and entirely implicit. However, there are advantages to becoming familiar with managing these data distributions, because each of these functions has the option to have redistribution directly managed. Now, a data redistribution must occur to use these functions, but understanding which and why can help minimize the number of redistributions performed.

Many of the full details, such as *why* the redistributions need occur in the first place, are outlined in the **pbdBASE** vignette, but we provide a simple example here. Suppose we have a distributed matrix `dx` distributed on the default grid (i.e., BLACS context 0) and we wish to drop the first column and then use the `apply()` function to extract the p-values, column-wise, of the result of running the Shapiro-Wilk normality test independently on the columns. No one is claiming that this is a wise thing to do, but it is useful for the purpose of demonstration.

To achieve this, we could execute the following:

Implicit Redistributions

```
dx <- dx[-1, ]
```

```

result <- apply(dx, MARGIN=2, FUN=function(col)
   shapiro.test(col)$p, reduce=TRUE)
```

In reality, underneath this is actually performing the following sequence of operations:

Implicit Redistributions

```
dx <- redistribute(dx, ICTXT=2)
dx <- dx[, -1]
dx <- redistribute(dx, ICTXT=0)

dx <- redistribute(dx, ICTXT=2)
result <- apply(dx, MARGIN=2, FUN=function(col)
   shapiro.test(col)$p, reduce=TRUE)
```

Or suppose we wanted instead to drop the first column; then this is equivalent to

Implicit Redistributions

```
dx <- redistribute(dx, ICTXT=1)
dx <- dx[, -1]
dx <- redistribute(dx, ICTXT=0)

dx <- redistribute(dx, ICTXT=2)
result <- apply(dx, MARGIN=2, FUN=function(col)
   shapiro.test(col)$p, reduce=TRUE)
```

The problem should be obvious. However, thoroughly understanding the problem, we can easily manage the data redistributions using the `ICTXT=` option in these function. So for example, we can minimize the redistributions to only the minimal necessary amount with the following:

Implicit Redistributions

```
dx <- dx[, -1, ICTXT=2]

result <- apply(dx, MARGIN=2, FUN=function(col)
   shapiro.test(col)$p, reduce=TRUE)
```

This is equivalent to explicitly calling:

Implicit Redistributions

```
dx <- redistribute(dx, ICTXT=2)
dx <- dx[, -1, ICTXT=2]

result <- apply(dx, MARGIN=2, FUN=function(col)
   shapiro.test(col)$p, reduce=TRUE)
```

This is clearly preferred. For more details, see the relevant function documentation.

7.3 Load Balance and Unload Balance

Example: Load balancing (and unbalancing) distributed data.

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(balance,'pbdDEMO',ask=F,echo=F)"
```

Suppose we have an unbalanced, distributed input matrix `X.spmd`. We can call `balance.info()` on this object to store some information about how to balance the data load across all processors. This can be useful for tracking data movement, as well as for "unbalancing" later, if we so choose. Next, we call `load.balance()` to obtain a load-balanced object `new.X.spmd`. We can also now undo this entire process and get back to `X.spmd` by calling `unload.balance()` on `new.X.spmd`.

All together, the code looks something like:

R Code

```
bal.info <- balance.info(X.spmd)
new.X.spmd <- load.balance(X.spmd, bal.info)
org.X.spmd <- unload.balance(new.X.spmd, bal.info)
```

The details of this exchange are depicted in the example in Figure 7.3. Here, `X.spmd` is unbalanced, and `new.X.spmd` is a balanced version of `X.spmd`.

$$
\begin{array}{ccc}
\texttt{X.spmd(org.X.spmd)} & & \texttt{new.X.spmd} \\
\left[\begin{array}{ccc}
x_{1,1} & x_{1,2} & x_{1,3} \\ \hline
x_{2,1} & x_{2,2} & x_{2,3} \\
x_{3,1} & x_{3,2} & x_{3,3} \\ \hline
x_{4,1} & x_{4,2} & x_{4,3} \\
x_{5,1} & x_{5,2} & x_{5,3} \\
x_{6,1} & x_{6,2} & x_{6,3} \\ \hline
x_{7,1} & x_{7,2} & x_{7,3} \\
x_{8,1} & x_{8,2} & x_{8,3} \\
x_{9,1} & x_{9,2} & x_{9,3} \\
x_{10,1} & x_{10,2} & x_{10,3}
\end{array}\right]
&
\begin{array}{c}
\texttt{load.balance()} \\ \longrightarrow \\ \\ \longleftarrow \\ \texttt{unload.balance()}
\end{array}
&
\left[\begin{array}{ccc}
x_{1,1} & x_{1,2} & x_{1,3} \\
x_{2,1} & x_{2,2} & x_{2,3} \\
x_{3,1} & x_{3,2} & x_{3,3} \\ \hline
x_{4,1} & x_{4,2} & x_{4,3} \\
x_{5,1} & x_{5,2} & x_{5,3} \\
x_{6,1} & x_{6,2} & x_{6,3} \\ \hline
x_{7,1} & x_{7,2} & x_{7,3} \\
x_{8,1} & x_{8,2} & x_{8,3} \\
x_{9,1} & x_{9,2} & x_{9,3} \\ \hline
x_{10,1} & x_{10,2} & x_{10,3}
\end{array}\right]
\end{array}
$$

Figure 7.1: $\boldsymbol{X}$ is distributed in `X.spmd(org.X.spmd)` and `new.X.spmd`. Both are distributed row-wise in 4 processors. The colors represent processors 0, 1, 2, and 3, respectively.

The function `balance.info()` is extremely useful, because it will return the information used to load balance the given data `X.spmd`. The return of `balance.info()` is a list consisting of two dataframes, `send` and `recv`, as well as two vectors, `N.allspmd` and `new.N.allspmd`.

Here, `send` records the original processor rank and the destination processor rank of the unbalanced data (that which is to be transmitted by that processor). The `load.balance()` function uses this table to move the data via **pbdMPI**'s `isend()` function. If any "destination rank" is not the "original rank", then the corresponding data row will be moved to the appropriate processor. On the other hand, `recv` records the original processor rank and the destination rank of balanced data (that which is received by that processor).

The `N.allspmd` and `new.N.allspmd` objects both have length equal to the communicator containing all numbers of rows of `X.spmd` before and after the balancing, respectively. This is for double checking and avoiding a 0-row matrix issue.

For `unload.balance`, the process amounts to reversing `bal.info` and passing it to `load.balance`.

Finally, note that the "balanced" data is chosen to be balanced in a very particular way; it is arguably not "balanced", since 3 processors own 3 rows while 1 owns 1 row, and it is perhaps more balanced to have 2 processors own 3 rows and 2 own 2. However, we make this choice for the reason that our "balanced" data will always be a certain kind of degenerate block-cyclic structure. We will discuss this at length in the following section.

7.4 Convert Between SPMD and DMAT

Example: Convert between SPMD and DMAT formats.

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(spmd_dmat,'pbdDEMO',ask=F,echo=F)"
```

The final redistribution challenge we will present is taking an object in SPMD format and putting it in the DMAT format. More precisely, we assume the input object `X.spmd` is in SPMD and transfer the convert the object into an object of class `ddmatrix` which we will call `X.dmat`.

, then convert again to a R object in `X` which is common on all processors, as in the next.

The Figure 7.4 illustates an example `X.spmd` and `X.dmat` conversion. For full details about the block-cyclic data format used for class `ddmatrix`, see the **pbdBASE** vignette.

To perform such a redistribution, one simply needs to call:

R Code

```
X.dmat <- spmd2dmat(X.spmd)
```

or

R Code

```
X.spmd <- dmat2spmd(X.dmat)
```

Here, the `spmd2dmat` function does the following:

$$
\begin{array}{c}
\texttt{X.spmd} \\
\left[\begin{array}{ccc}
x_{1,1} & x_{1,2} & x_{1,3} \\ \hline
x_{2,1} & x_{2,2} & x_{2,3} \\
x_{3,1} & x_{3,2} & x_{3,3} \\ \hline
x_{4,1} & x_{4,2} & x_{4,3} \\
x_{5,1} & x_{5,2} & x_{5,3} \\
x_{6,1} & x_{6,2} & x_{6,3} \\ \hline
x_{7,1} & x_{7,2} & x_{7,3} \\
x_{8,1} & x_{8,2} & x_{8,3} \\
x_{9,1} & x_{9,2} & x_{9,3} \\
x_{10,1} & x_{10,2} & x_{10,3}
\end{array}\right]
\end{array}
\quad
\begin{array}{c}
\texttt{spmd2dmat} \\
\longrightarrow \\[2em]
\longleftarrow \\
\texttt{dmat2spmd}
\end{array}
\quad
\begin{array}{c}
\texttt{X.dmat} \\
\left[\begin{array}{cc|c}
x_{1,1} & x_{1,2} & x_{1,3} \\
x_{2,1} & x_{2,2} & x_{2,3} \\ \hline
x_{3,1} & x_{3,2} & x_{3,3} \\
x_{4,1} & x_{4,2} & x_{4,3} \\ \hline
x_{5,1} & x_{5,2} & x_{5,3} \\
x_{6,1} & x_{6,2} & x_{6,3} \\ \hline
x_{7,1} & x_{7,2} & x_{7,3} \\
x_{8,1} & x_{8,2} & x_{8,3} \\ \hline
x_{9,1} & x_{9,2} & x_{9,3} \\
x_{10,1} & x_{10,2} & x_{10,3}
\end{array}\right]
\end{array}
$$

Figure 7.2: $\boldsymbol{X}$ is distributed in `X.spmd` and `X.dmat`. Both are distributed in 4 processors where colors represents processor 0, 1, 2, and 3. Note that `X.dmat` is in block-cyclic format of 2×2 grid with 2×2 block dimension.

1. Check number of columns of `X.spmd`. All processors should be the same.
2. Row balance the SPMD matrix as necessary via `load.balance()` as in Section 7.3.
3. Call construct a new ddmatrix object (via the `new()` constructor) on the balanced matrix, say `X.dmat`, in BLACS context 2 (`ICTXT = 2`).
4. Redistribute `X.dmat` to another BLACS context as needed (default `ICTXT = 0`) via the `base.reblock()` function as in Section 7.1.

Note that the `load.balance()` function, as used above, is legitimately necessary here. Indeed, this function takes a collection of distributed data and converts it into a degenerate block cyclic distribution; namely, this places the data in block "1-cycle" format, distributed across an $n \times 1$ processor grid. In the context of Figure 7.4 (where the aforementioned process is implicit), this is akin to first moving the data into a distributed matrix format with `bldim=c(3,3)` and `CTXT=2`. Finally, we can take this degenerate block-cyclic distribution and again to Figure 7.4 as our motivating example, we convert the data balanced data so that it has `bldim=c(2,2)` and `CTXT=0`.

Part IV

Distributed Matrix Methods

8 Advanced Statistics Examples

The **pbdDMAT** package contains many useful methods for doing computations with distributed matrices. For comprehensive (but shallow) demonstrations of the distributed matrix methods available, see the **pbdDMAT** package's vignette and demos.

Here we showcase a few more advanced things that can be done by chaining together R and pbdR code seamlessly.

8.1 Sample Mean and Variance Revisited

Example: Get summary statistics from a distributed matrix.

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(sample_stat_dmat,'pbdDEMO',ask=F,echo=F)"
```

Returning to the sample statistics problem from Section 4.2, we can solve these same problems — and then some — using distributed matrices. For the remainder, suppose we have a distributed matrix `dx`.

Computing a mean is simple enough. We need only call

Summary Statistics

```
mean(dx)
```

We also have access to the other summary statistics methods for matrices, however, such as `rowSums()`, `rowMeans()`, etc. Furthermore, we can calculate variances for distributed matrices. Constructing the variance-covariance matrix is as simple as calling

Summary Statistics

```r
cov(dx)
```

Or we could generate standard deviations column-wise, using the method R suggests for ordinary matrices using `apply()`

Summary Statistics

```r
apply(dx, MARGIN=2, FUN=sd)
```

or we could simply call

Summary Statistics

```r
sd(dx)
```

In R, calling `sd()` on a matrix issues a warning, telling the user to instead use `apply()`. Either of these approaches works with a distributed matrix (with the code as above), but for us, using `sd()` is preferred. This is because, as outlined in Section 7.2, our `apply()` method carries an implicit data redistribution with it, while the `sd()` method is fast, ad-hoc code which requires no redistribution of the data.

8.2 Verification of Distributed System Solving

Example: Solve a system of equations and verify that the solution is correct.

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(verify,'pbdDEMO',ask=F,echo=F)"
```

The **pbdDEMO** contains a set of verification routines, designed to test for validity of the numerical methods at any scale. Herein we will discuss the verification method for solving systems of linear equations, `verify.solve()`.

The process is simple. The goal is to solve the equation (in matrix notation)

$$Ax = b$$

for $n \times n$ matrix A and $n \times 1$ matrix b. However, here we start with A and x and use these to produce b. We then forget we ever knew what x was and solve the system. Finally, we remember what x really should be and compare that with our numerical solution.

More specifically, we take the matrix A to be random normal generated data and the true solution x to be a constant vector. We then calculate

$$b := Ax$$

and finally the system is solve for a now (pretend) unknown x, so that we can compare the numerically determined x to the true constant x. All processes are timed, and both success/failure and timing results are printed for the user at the completion of the routine. This effectively amounts to calling:

Verifying Distributed System Solving

```
# generating data
timer({
  x <- Hnorm(dim=c(nrows, nrows))
  truesol <- Hconst(dim=c(nrows, 1))
})

timer({
  rhs <- x %*% truesol
})

# solving
timer({
  sol <- solve(x, rhs)
})

# verifying
timer({
  iseq <- all.equal(sol, truesol)
  iseq <- as.logical(allreduce(iseq, op='min'))
})
```

with some added window dressing.

8.3 Compression with Principal Components Analysis

Example: Take PCA and retain only a subset of the rotated data.

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(pca,'pbdDEMO',ask=F,echo=F)"
```

Suppose we wish to perform a principal components analysis and retain only some subset of the columns of the rotated data. One of the ways this is often done is by using the singular values — the standard deviations of the components — as a measure of variation retained by a component. However, the first step is to get the principal components data. Luckily this is trivial. If our data is stored in the distributed matrix object `dx`, then all we need to is issue the command:

```
pca <- prcomp(x=dx, retx=TRUE, scale=TRUE)
```

Now that we have our PCA object (which has the same structure as that which comes from calling `prcomp()` on an ordinary R matrix), we need only decide how best to throw away what we do not want. We might want to retain at least as many columns as would be needed to retain 90% of the variation of the original data:

```
prop_var <- cumsum(pca$sdev)/sum(pca$sdev)
i <- min(which(prop_var > 0.9))

new_dx <- pca$x[, 1:i]
```

Or we might want one fewer column than the number that would give us 90%:

```
prop_var <- cumsum(pca$sdev)/sum(pca$sdev)
i <- max(min(which(prop_var > 0.9)) - 1, 1)

new_dx <- pca$x[, 1:i]
```

8.4 Predictions with Linear Regression

Example: Fit a linear regression model and use it to make a prediction on new data.

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(ols_dmat,'pbdDEMO',ask=F,echo=F)"
```

Suppose we have some predictor variables stored in the distributed $n \times p$ matrix `dx` and a response variable stored in the $n \times 1$ distributed matrix `dy`, and we wish to use the ordinary least squares model from (4.6) to make a prediction about some new data, say `dy.new`. Then this really amounts to just a few simple commands, namely:

```
mdl <- lm.fit(dx, dy)

pred <- dx.new %*% mdl$coefficients

comm.print(submatrix(pred), quiet=T)
```

A

Numerical Linear Algebra and Linear Least Squares Problems

For the remainder, assume that all matrices are real-valued.

Let us revisit the problem of solving linear least squares problems, introduced in Section 4.5. Recall that we wish to find a solution $\boldsymbol{\beta}$ such that

$$||\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{y}||_2^2$$

In the case that $\boldsymbol{X}$ is full rank (which is often assumed, whether reasonable or not), this has analytical solution

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{A.1}$$

However, even with this nice closed form, implementing this efficiently on a computer is not entirely straightforward. Herein we discuss several of the issues in implementing the linear least squares solution efficiently. For simplicity, we will assume that $\boldsymbol{X}$ is full rank, although this is not necessary — although rank degeneracy does complicate things. For more details on the rank degeneracy problem, and linear least squares problems in general, see the classic *Matrix Computations* (Golub and Van Loan, 1996).

A.1 Forming the Normal Equations

If we wish to implement this numerically, then rather than directly computing the inverse of $\boldsymbol{X}^T\boldsymbol{X}$ directly, we would instead compute the Cholesky factorization

$$\boldsymbol{X}^T\boldsymbol{X} = LL^T$$

where L is lower triangular. Then turning to the so-called "normal equations"

$$\left(\boldsymbol{X}^T\boldsymbol{X}\right)\boldsymbol{\beta} = \boldsymbol{X}^T\boldsymbol{y} \tag{A.2}$$

by simple substitution and grouping, we have

$$L\left(L^T\boldsymbol{\beta}\right) = \boldsymbol{X}^T\boldsymbol{y}$$

Now, since L is triangular, these two triangular systems (one forward and one backward substitution found by careful grouping of terms above) can be solved in a numerically stable way (Higham, 2002). However, forming the Cholesky factorization itself suffers from the effects of roundoff error in having to form the product $\boldsymbol{X}^T\boldsymbol{X}$. We elaborate on this to a degree in the following section.

A.2 Using the QR Factorization

Directly computing the normal equations is ill advised, because it is often impossible to do so with adequate numerical precision. To fully appreciate this problem, we must entertain a brief discussion about condition numbers.

By definition, if a matrix has finite condition number, then it must have been invertible. However, for numerical methods, a condition number which is "big enough" is essentially infinite (loosely speaking). And observe that forming the product $\boldsymbol{X}^T\boldsymbol{X}$ squares the condition number of $\boldsymbol{X}$:

$$\begin{aligned}\kappa\left(\boldsymbol{X}^T\boldsymbol{X}\right) &= \left|\left|\boldsymbol{X}^T\boldsymbol{X}\right|\right| \left|\left|\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\right|\right| \\ &= \left|\left|\boldsymbol{X}^T\boldsymbol{X}\right|\right| \left|\left|\boldsymbol{X}^{-1}\left(\boldsymbol{X}^T\right)^{-1}\right|\right| \\ &= \left|\left|\boldsymbol{X}^T\right|\right| \left|\left|\boldsymbol{X}\right|\right| \left|\left|\boldsymbol{X}^{-1}\right|\right| \left|\left|\boldsymbol{X}^{-T}\right|\right| \\ &= \left|\left|\boldsymbol{X}\right|\right| \left|\left|\boldsymbol{X}\right|\right| \left|\left|\boldsymbol{X}^{-1}\right|\right| \left|\left|\boldsymbol{X}^{-1}\right|\right| \\ &= \left|\left|\boldsymbol{X}\right|\right|^2 \left|\left|\boldsymbol{X}^{-1}\right|\right|^2 \\ &= \kappa(\boldsymbol{X})^2\end{aligned}$$

So if $\kappa\left(\boldsymbol{X}\right)$ is "large", then forming this product can lead to large numerical errors when attempting to numerically invert or factor a matrix, or solve a system of equations.

To avoid this problem, the orthogonal QR-decomposition is typically used. Here we take

$$\boldsymbol{X} = QR$$

where Q is orthogonal and R is upper trapezoidal (n the overdetermined case, R is triangular). This is beneficial, because orthogonal matrices are norm-preserving, i.e. Q is an isometry, and whence

$$\begin{aligned}\left|\left|\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{y}\right|\right|_2 &= \left|\left|QR\boldsymbol{\beta} - \boldsymbol{y}\right|\right|_2 \\ &= \left|\left|Q^TQR\boldsymbol{\beta} - Q^T\boldsymbol{y}\right|\right|_2 \\ &= \left|\left|R\boldsymbol{\beta} - Q^T\boldsymbol{y}\right|\right|_2\end{aligned}$$

This amounts to solving the triangular system

$$R\boldsymbol{\beta} = Q^T\boldsymbol{y}$$

As noted in Section A.1, solving this system can be done in a numerically stable fashion (and the high performance libraries employed by both R and pbdR use stable implementations). The key difference here is that the QR factorization is of $\boldsymbol{X}$, not $\boldsymbol{X}^T\boldsymbol{X}$, and so we need only worry about the conditioning of $\boldsymbol{X}$ (as opposed to its squared condition number).

While this method is much less prone to the numerical issues discussed above, it is much slower computationally. Additionally, we note that unlike the method in forming the normal equations, this method can be extended to the rank degenerate case.

A.3 Using the Singular Value Decomposition

There is another, arguably much more well-known matrix factorization which we can use to develop yet another analytically equivalent solution to the least squares problem, namely the Singular Value Decomposition (SVD). Using this factorization leads to a very elegant solution, as is so often the case with the SVD.

Note that in (A.1), the quantity

$$(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T$$

is the Moore-Penrose inverse of $\boldsymbol{X}$. So if we take

$$\boldsymbol{X} = U\Sigma V^T$$

to be the SVD of $\boldsymbol{X}$, then we have

$$\begin{aligned}
\boldsymbol{X}^+ &= \left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T \\
&= \left(\left(U\Sigma V^T\right)^T\left(U\Sigma V^T\right)\right)^{-1} U\Sigma V^T \\
&= \left(V\Sigma^T\Sigma V^T\right)^{-1} V\Sigma^T U^T \\
&= V\left(\left(\Sigma^T\Sigma\right)^{-1}\Sigma^T\right)U^T \\
&= V\Sigma^+ U^T
\end{aligned}$$

Whence,

$$\boldsymbol{\beta} = V\Sigma^+ U^T\boldsymbol{y}$$

Conceptually, this is arguably the most elegant method of solving the linear least squares problem. Additionally, as with the QR method above, with slight modification the above argument can extend to the rank degenerate case; however, we suspect that the SVD is much more well known to mathematicians and statisticians than is the QR decomposition. This abstraction comes at a great cost, though, as this approach is handily the most computationally intensive of the three presented here.

B

Linear Regression and Rank Degeneracy in R

In the case that $\boldsymbol{X}$ is rank deficient, then $\boldsymbol{X}$ (and whence $\boldsymbol{X}^T\boldsymbol{X}$) is not invertible, so the problem can not be solved by the method of Section A.1. Both R and pbdR use a QR factorization as in Section A.2, although the two systems use a slightly different approach. While most of the linear algebra in R is handled by LAPACK (Anderson *et al.*, 1999), arguably the most important numerical function in all of R, namely `lm.fit()` used by `lm()` to fit linear regression models, uses LINPACK (Dongarra *et al.*, 1979). By comparison to LAPACK, LINPACK is much less sophisticated. However, pbdR uses level 3 PBLAS and ScaLAPACK (the distributed equivalent of using level 3 BLAS and LAPACK) to fit linear regression models.

The LINPACK routines used by R are `DQRLS`, which calls a modified `DQRDC2` to compute a rank-revealing QR factorization with a "limited pivoting strategy" (more on this later). Finally, `DQRSL` is called to apply the output of the QR factorization to compute the least squares soluations. By contrast, pbdR uses a modified `PDGELS` routine, which uses a version of `PDGEQPF` modified to use R's "limited pivoting strategy", and then calls `PDORMQR` to fit the least squares solution.

Neither approach assumes that the model matrix is full rank. Instead, the methods are *rank-revealing*, in that they attempt to discover the numerical rank while computing the orthogonal factorization. However, both R and (for the sake of consistency) pbdR use a "limited pivoting strategy" (with language, we believe, due to Ross Ihaka) in determining numerical rank. Generally when computing a QR with pivoting, for the sake of numerical stability one chooses the column with largest partial norm while forming the Householder reflections. However, in doing so it is possible to permute the columns in such a way that a desired statistical interpretation (such as in an ANOVA) is destroyed. The solution employed by R is to merely iterate over the columns and choose the current column as the pivot each time. When a column is detected to have "small" partial norm, it is pushed to the back. The author of these modification writes:

> a limited column pivoting strategy based on the 2-norms of the reduced columns moves columns with near-zero norm to the right-hand edge of the x matrix. this strategy means that sequential one degree-of-freedom effects can be computed in a natural way.
>
> i am very nervous about modifying linpack code in this way. if you are a compu-

> tational linear algebra guru and you really understand how to solve this problem please feel free to suggest improvements to this code.

So in this way, if a model matrix is full rank, then the estimates coming from R should be considered at least as trustworthy as probably every other statistical software package of note. If it is not, then this method presents a possible numerical stability issue; although to what degree, if any at all, this is actually a problem, the authors at present have no real knowledge. If numerical precision is absolutely paramount, consider using the SVD to solve the least squares problem; though do be aware that this is hands down the slowest possible approach.

We again note that the limited pivoting strategy of R is employed by pbdR in the `lm.fit()` method for class `ddmatrix`.

Bibliography

Analytics R (2012). *foreach: Foreach looping construct for R*. R package version 1.4.0, URL http://CRAN.R-project.org/package=foreach.

Anderson E, Bai Z, Bischof C, Blackford LS, Demmel J, Dongarra JJ, Du Croz J, Hammarling S, Greenbaum A, McKenney A, Sorensen D (1999). *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. ISBN 0-89871-447-8.

Benson F (1949). "A Note on the Estimation of Mean and Standard Deviation from Quantiles." *Journal of the Royal Statistical Society. Series B (Methodological)*, **11**(1), pp. 91–100. ISSN 00359246. URL http://www.jstor.org/stable/2983699.

Blackford LS, Choi J, Cleary A, D'Azevedo E, Demmel J, Dhillon I, Dongarra J, Hammarling S, Henry G, Petitet A, Stanley K, Walker D, Whaley RC (1997). *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA. ISBN 0-89871-397-8 (paperback). URL http://netlib.org/scalapack/slug/scalapack_slug.html/.

Chen WC, Ostrouchov G (2011). "HPSC – High Performance Statistical Computing for Data Intensive Research." URL http://thirteen-01.stat.iastate.edu/snoweye/hpsc/.

Chen WC, Ostrouchov G, Schmidt D, Patel P, Yu H (2012a). "pbdMPI: Programming with Big Data – Interface to MPI." R Package, URL http://cran.r-project.org/package=pbdMPI.

Chen WC, Ostrouchov G, Schmidt D, Patel P, Yu H (2012b). "A Quick Guide for the pbdMPI package." R Vignette, URL http://cran.r-project.org/package=pbdMPI.

Dongarra JJ, Moler CB, Bunch JR, Stewart GW (1979). *LINPACK User's Guide*. SIAM.

Golub GH, Van Loan CF (1996). *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)(3rd Edition)*. 3rd edition. The Johns Hopkins University Press.

Grothendieck G (2012). *sqldf: Perform SQL Selects on R Data Frames*. R package version 0.4-6.4, URL http://CRAN.R-project.org/package=sqldf.

Higham NJ (2002). *Accuracy and Stability of Numerical Algorithms*. 2nd edition. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. ISBN 0898715210.

Kunen K (1980). *Set Theory: An Introduction to Independence Proofs*. North-Holland.

NetCDF Group (2008). "Network Common Data Form." Software package, URL http://www.unidata.ucar.edu/software/netcdf/.

Ostrouchov G, Chen WC, Schmidt D, Patel P (2012). "Programming with Big Data in R." URL http://r-pbd.org/.

R Core Team (2012). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.r-project.org/.

Schmidt D, Chen WC, Ostrouchov G, Patel P (2012a). "pbdBASE: Programming with Big Data – Core pbd Classes and Methods." R Package, URL http://cran.r-project.org/package=pbdBASE.

Schmidt D, Chen WC, Ostrouchov G, Patel P (2012b). "pbdDEMO: Programming with Big Data – Demonstrations of pbd Packages." R Package, URL http://cran.r-project.org/package=pbdDEMO.

Schmidt D, Chen WC, Ostrouchov G, Patel P (2012c). "pbdDMAT: Programming with Big Data – Distributed Matrix Algebra Computation." R Package, URL http://cran.r-project.org/package=pbdDMAT.

Schmidt D, Chen WC, Ostrouchov G, Patel P (2012d). "A Quick Guide for the pbdBASE package." R Vignette, URL http://cran.r-project.org/package=pbdBASE.

Schmidt D, Ostrouchov G, Chen WC, Patel P (2012e). "Tight Coupling of R and Distributed Linear Algebra for High-Level Programming with Big Data." In P Kellenberger (ed.), *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE Computer Society.

Weston S (2010). *doMPI: Foreach parallel adaptor for the Rmpi package*. R package version 0.1-5, URL http://CRAN.R-project.org/package=doMPI.

Wickham H (2009). *ggplot2: elegant graphics for data analysis*. Springer New York. ISBN 978-0-387-98140-6. URL http://had.co.nz/ggplot2/book.

Yu H (2012). *Rmpi: Interface (Wrapper) to MPI (Message-Passing Interface)*. R package version 0.6-1, URL http://CRAN.R-project.org/package=Rmpi.