The `irlba` Package

Bryan W. Lewis
blewis@illposed.net,

adapted from the work of:
Jim Baglama (University of Rhode Island)
and Lothar Reichel (Kent State University).

December 24, 2016

1 Introduction

The `irlba` package provides a fast way to compute partial singular value decompositions (SVD) of large sparse or dense matrices. Recent additions to the package can also compute fast partial symmetric eigenvalue decompositions and principal components. The package is an R implementation of the *augmented implicitly restarted Lanczos bidiagonalization algorithm* of Jim Baglama and Lothar Reichel[1]. Source code is maintained at https://github.com/bwlewis/irlba.

The `irlba` package works with real- and complex-valued dense R matrices and real-valued sparse matrices from the `Matrix` package. It provides several easy ways to define custom matrix arithmetic that works with other matrix classes including `big.matrix` from the `bigmemory` package and others. The `irlba` is both faster and more memory efficient than the usual R `svd` function for computing a few of the largest singular vectors and corresponding singular values of a matrix. It takes advantage of available high-performance linear algebra libraries if R is compiled to use them. In particular, the package uses the same BLAS and LAPACK libraries that R uses (see https://cran.r-project.org/doc/manuals/R-admin.html#BLAS), or the CHOLMOD library from R's Matrix package for sparse matrix problems.

A whirlwind summary of the algorithm follows, along with a few basic examples. A much more detailed description and discussion of the algorithm may be found in the cited Baglama-Reichel reference.

[1] Augmented Implicitly Restarted Lanczos Bidiagonalization Methods, J. Baglama and L. Reichel, SIAM J. Sci. Comput. 2005.

2 Partial Singular Value Decomposition

Let $A \in \mathbf{R}^{\ell \times n}$ and assume $\ell \geq n$. These notes simplify the presentation by considering only real-valued matrices and assuming without losing generality that there are at least as many rows as columns (the method works more generally). A singular value decomposition of A can be expressed as:

$$A = \sum_{j=1}^{n} \sigma_j u_j v_j^T, \qquad v_j^T v_k = u_j^T u_k = \begin{cases} 1 & \text{if } j = k, \\ 0 & \text{o.w.}, \end{cases}$$

where $u_j \in \mathbf{R}^{\ell}$, $v_j \in \mathbf{R}^{n}$, $j = 1, 2, \ldots, n$, and $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$.

Let $1 \leq k < n$. A rank k partial SVD of A is defined as:

$$A_k \;:=\; \sum_{j=1}^{k} \sigma_j u_j v_j^T.$$

The following simple example shows how to use `irlba` to compute the five largest singular values and corresponding singular vectors of a 5000×5000 matrix. We compare to the usual R `svd` function and report timings for our test machine, a 4-CPU core, 3.0 GHz AMD A10-7850K personal computer with 16 GB RAM, using R version 3.3.1 using the high performance AMD ACML core math library BLAS and LAPACK.

```
> library('irlba')
> set.seed(1)
> A <- matrix(rnorm(5000*5000), 5000)
> t1 <- proc.time()
> L <- irlba(A, 5)
> print(proc.time() - t1)
   user  system elapsed
 17.440   0.192   4.417
> gc()
          used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells 1096734  58.6    1770749  94.6  1442291  77.1
Vcells 26685618 203.6 62229965 474.8 52110704 397.6
```

Compare with the standard `svd` function:

```
> t1 <- proc.time()
> S <- svd(A, nu=5, nv=5)
> print(proc.time() - t1)
   user  system elapsed
277.092  11.552  74.425
> gc()
          used  (Mb) gc trigger  (Mb)  max used  (Mb)
Ncells 1097441  58.7    1770749  94.6   1442291  77.1
Vcells 26741910 204.1 169891972 1296.2 176827295 1349.1
```

The `irlba` method uses about 1/20 elapsed time as the `svd` method in this example and less than one third the peak memory. The defalut tolerance value yields the following relative error in the estimated singular values:

```
> sqrt (crossprod(S$d[1:5]-L$d)/crossprod(S$d[1:5]))
            [,1]
[1,] 4.352641e-10
```

2.1 Convergence tolerance

IRLBA is an iterative method that estimates a few largest singular values and associated singular vectors. A sketch of the algorithm is outlined in Section 3 below. The R `tol` argument controls when the algorithm converges. Convergence occurs when

$$\|AV_k - US_k\| < \text{tol} \cdot \|A\|,$$

where $\|\cdot\|$ means spectral matrix norm, A is the matrix, V_k and U_k are the *estimated* right and left k singular vectors computed by the algorithm, and $\|A\|$ is the *estimated* spectral norm of the matrix defined by the largest singular value computed by the algorithm. Using R notation, the algorithm stops when

```
L <- irlba(A, k, tol)
svd(A %*% L$v - L$u %*% diag(L$d))$d[1] < tol * L$d[1]
```

It's possible, but unlikely, to encounter problems that fail to converge before the maximum number of algorithm iterations specified by the `maxit` argument.

2.2 Differences with `svd`

The `irlba` function is designed to compute a *partial* singular value decomposition. It is largely compatible with the usual R `svd` function but there are some differences. In particular:

1. The irlba function only computes the number of singular values corresponding to the maximum of the desired singular vectors, max(nu, nv). For example, if 5 singular vectors are desired (nu=nv=5), then only the five corresponding singular values are computed. The standard R svd function always returns the *total* set of singular values for the matrix, regardless of how many singular vectors are specified.

2. The irlba function is an iterative method that continues until either a tolerance or maximum number of iterations is reached. Problems with difficult convergence properties are not likely to be encountered, but the method will fail with an error after the iteration limit is reached in those cases.

Watch out especially for the first difference noted above!

2.3 Principal Components

Version 2.1.0 of the package introduces optional arguments and prcomp-like function syntax for efficiently computing partial SVDs of matrices after centering and scaling their columns and other adjustments. Use the following arguments to the irlba function, or the new irlba_prcomp function for PCA:

- center: if center is a numeric vector with length equal to the number of columns of the matrix, then each column of the matrix has the corresponding value from center subtracted from it.

- scale: if 'scale' is a numeric vector with length equal to the number of columns of the matrix, then each column is divided by the corresponding value from scale.

Both centering and scaling options are performed implicitly in the algorithm and, for instance, do not affect sparsity of the input matrix or increase storage requirements. The following example compares the output of the usual prcomp function with output from irlba. Note that in general, singular vectors and principal component vectors are only unique up to sign!

```
>     set.seed(1)
>     x  <- matrix(rnorm(200), nrow=20)
>     p1 <- prcomp_irlba(x, n=3)
>     summary(p1)
Importance of components:
                         PC1    PC2    PC3
Standard deviation     1.541 1.2513 1.1916
Proportion of Variance 0.443 0.2921 0.2649
Cumulative Proportion  0.443 0.7351 1.0000
```

```
>     # Compare with
>     p2 <- prcomp(x, tol=0.7)
>     summary(p2)
Importance of components:
                       PC1    PC2    PC3
Standard deviation   1.541 1.2513 1.1916
Proportion of Variance 0.443 0.2921 0.2649
Cumulative Proportion 0.443 0.7351 1.0000
```

Alternatively, you can compute principal components directly using the singular value decomposition and the `center` option:

```
> p3 <- svd(scale(x, center=colMeans(x), scale=FALSE))
> p4 <- irlba(x, 3, center=colMeans(x))

> # compare with prcomp
> sqrt(crossprod(p1$rotation[,1] - p3$v[,1]))
            [,1]
[1,] 9.773228e-13
> sqrt(crossprod(p1$rotation[,1] + p4$v[,1]))
            [,1]
[1,] 1.652423e-12
```

The implementation of the `center` function argument takes advantage of computational efficiencies in the IRLB algorithm that result in a modest savings of a few vector inner products per iteration compared to a naive implementation using a custom matrix vector product to center the matrix.

2.4 Truncated symmetric eigenvalue decomposition

Use the `partial_eigen` function to estimate a subset of the largest (most positive) eigenvalues and corresponding eigenvectors of a symmetric dense or sparse real-valued matrix. The function is particularly well-suited to estimating the largest eigenvalues and corresponding eigenvectors of symmetric positive semi-definite matrices of the form $A^T A$.

2.5 User-Defined Matrix Multiplication

The `irlba` function includes options for specifying a custom matrix multiplication function. Custom matrix multiplication functions can be used, for example, with the `big.matrix` class from the `bigmemory`/`bigalgebra` packages, or to compute the partial SVD of matrix-free linear operators.

User-defined matrix operations may specified using the optional `mult` parameter. If defined, it must be a function of two arguments that computes matrix vector products. Either argument can

be a vector, and the mult function must deal with that. The following example illustrates a simple custom matrix function that scales the columns of the matrix, and then compares it with other ways of doing the same thing.

```
> set.seed(1)
> A <- matrix(runif(200), nrow=20)
> col_scale <- 1:10
> mult <- function(x,y)
+        {
+          # check if x is a plain vector
+          if(is.vector(x))
+          {
+            return((x %*% y)/col_scale)
+          }
+          # else x is the matrix
+          x %*% (y/col_scale)
+        }

> irlba(A, 3, mult=mult)$d
[1] 3.1384503 0.9477628 0.4313855

> # Compare with:
> irlba(A, 3, scale=col_scale)$d
[1] 3.1384503 0.9477628 0.4313855

> # Compare with:
> svd(scale(A, scale=col_scale, center=FALSE))$d[1:3]
[1] 3.1384503 0.9477628 0.4313855
```

Alternatively, simply use R's operator overloading methods to define customized matrix vector products. This is the approach taken in the http://bwlewis.github.io/1000_genomes_examples/PCA_whole_genome.html vignette to work with large out of core genomics data. I prefer the simple operator overloading approach and may consider retiring the mult function argument in some future package version.

NOTE! When a user-defined matrix multiplication function is used, either using the mult argument or through operator overloading, the R reference IRLBA implementation is used instead of the faster C code path in newer package versions.

3 A Quick Summary of the IRLBA Method

3.1 Partial Lanczos Bidiagonalization

Start with a given vector p_1. Compute m steps of the Lanczos process:

$$\begin{aligned} AP_m &= Q_m B_m \\ A^T Q_m &= P_m B_m^T + r_m e_m^T, \end{aligned}$$

$B_m \in \mathbf{R}^{m \times m}, P_m \in \mathbf{R}^{n \times m}, Q_m \in \mathbf{R}^{\ell \times m},$

$P_m^T P_m = Q_m^T Q_m = I_m,$

$r_m \in \mathbf{R}^n, P_m^T r_m = 0,$

$P_m = [p_1, p_2, \ldots, p_m].$

3.2 Approximating Partial SVD with A Partial Lanczos bidiagonalization

$$\begin{aligned} A^T A P_m &= A^T Q_m B_m \\ &= P_m B_m^T B_m + r_m e_m^T B_m, \end{aligned}$$

$$\begin{aligned} A A^T Q_m &= A P_m B_m^T + A r_m e_m^T, \\ &= Q_m B_m B_m^T + A r_m e_m^T. \end{aligned}$$

Compute the SVD of B_m:

$$B_m = \sum_{j=1}^{m} \sigma_j^B u_j^B \left(v_j^B\right)^T.$$

$$\left(\text{i.e., } B_m v_j^B = \sigma_j^B u_j^B, \text{ and } B_m^T u_j^b = \sigma_j^B v_j^B.\right)$$

Define: $\tilde{\sigma}_j := \sigma_j^B, \qquad \tilde{u}_j := Q_m u_j^B, \qquad \tilde{v}_j := P_m v_j^B.$

Then:

$$\begin{aligned} A\tilde{v}_j &= AP_m v_j^B \\ &= Q_m B_m v_j^B \\ &= \sigma_j^B Q_m u_j^B \\ &= \tilde{\sigma}_j \tilde{u}_j, \end{aligned}$$

and

$$\begin{aligned} A^T\tilde{u}_j &= A^T Q_m u_j^B \\ &= P_m B_m^T u_j^B + r_m e_m^T u_j^B \\ &= \sigma_j^B P_m v_j^B + r_m e_m^T u_j^B \\ &= \tilde{\sigma}_j \tilde{v}_j + r_m e_m^T u_j^B. \end{aligned}$$

The part in red above represents the error with respect to the exact SVD. The IRLBA strategy is to iteratively reduce the norm of that error term by augmenting and restarting.

Here is the overall method:

1. Compute the Lanczos process up to step m.
2. Compute $k < m$ approximate singular vectors.
3. Orthogonalize against the approximate singular vectors to get a new starting vector.
4. Continue the Lanczos process with the new starting vector for m more steps.
5. Check for convergence tolerance and exit if met.
6. GOTO 1.

3.3 Sketch of the augmented process...

$$\begin{aligned} \bar{P}_{k+1} &:= [\tilde{v}_1, \tilde{v}_2, \ldots, \tilde{v}_k, p_{m+1}], \\ A\bar{P}_{k+1} &= [\tilde{\sigma}_1\tilde{u}_1, \tilde{\sigma}_1\tilde{u}_2, \ldots, \tilde{\sigma}_k\tilde{u}_k, Ap_{m+1}] \end{aligned}$$

Orthogonalize Ap_{m+1} against $\{\tilde{u}_j\}_{j=1}^k$: $Ap_{m+1} = \sum_{j=1}^k \rho_j \tilde{u}_j + r_k$.

$$\begin{aligned} \bar{Q}_{k+1} &:= [\tilde{u}_1, \tilde{u}_2, \ldots, \tilde{u}_k, r_k/\|r_k\|], \\ \bar{B}_{k+1} &:= \begin{bmatrix} \tilde{\sigma}_1 & & & \rho_1 \\ & \tilde{\sigma}_2 & & \rho_2 \\ & & \ddots & \rho_k \\ & & & \|r_k\| \end{bmatrix}. \end{aligned}$$

$$A\bar{P}_{k+1} = \bar{Q}_{k+1}\bar{B}_{k+1}.$$