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).

August 1, 2015

1 Introduction

The irlba package provides a fast way to compute partial singular value decompositions (SVD) of large matrices. It is an R implementation of the *augmented implicitly restarted Lanczos bidiagonalization algorithm* of Jim Baglama and Lothar Reichel[1]. The irlba package source code is maintained at https://github.com/bwlewis/IRL. An old introductory example using the Netflix prize data set can be found at http://goo.gl/fRech.

The irlba package works with real- and complex-valued dense R matrices and real-valued sparse matrices from the Matrix package. The package provides a simple way 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. The package takes advantage of available high-performance linear algebra libraries if R is compiled to use them.

A whirlwind summary of the algorithm follows, along with a few basic examples. See the companion package vignette for more substantial applications. 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. 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, an 8-CPU core, 2.0 GHz AMD Opteron server with 16 GB RAM, using R version 2.13.0 compiled with the high performance AMD ACML core math libraries.

```
> library('irlba')
> A <- matrix(rnorm(5000*5000), 5000)
> t1 <- proc.time()
> L <- irlba(A, nu=5, nv=5)
> print(proc.time() - t1)
   user  system elapsed
 41.640   0.470  36.985
> gc()
           used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells   137098   7.4     350000  18.7   350000  18.7
Vcells 25180235 192.2 52881183 403.5 52881005 403.5
```

Now, compare with the standard svd function:

```
> t1 <- proc.time()
> S <- svd(A, nu=5, nv=5)
> print(proc.time() - t1)
   user  system elapsed
616.035   4.396 187.371
> gc()
           used  (Mb) gc trigger  (Mb)  max used  (Mb)
```

```
Ncells  137109   7.4    350000   18.7   350000   18.7
Vcells 25235234 192.6 168397903 1284.8 200272760 1528.0

# Compare the singular values computed by each method:
> sqrt (crossprod(S$d[1:5]-L$d)/crossprod(S$d[1:5]))
             [,1]
[1,] 1.56029e-12
```

The `irlba` method uses less than one tenth total CPU time as the `svd` method in this example, less than one fifth the total run time, and about one fourth the peak memory.

2.1 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. There exists pathological problems for which `irlba` does not converge (see the references for more information). Such problems 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 for the first difference noted above!

2.2 Principal Components

Version 2.0.0 of the package introduces simple syntax for efficiently computing partial SVDs of matrices after centering and scaling their columns and other adjustments. These options are useful, for example, to compute principal components analysis (PCA). Three categories of options are available:

- `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`.
- `dU, dS, dV`: Optional real-valued deflation parameters to compute the rank-one deflated partial SVD of `A - ds*du %*% t(dv)`, where `A` is the data matrix, `dS` a real-valued scalar, and `dU` and `dV` real-valued numeric vectors defining the subspace.

In particular, you can use the `center` option for PCA. The following example compares the output of `irlba` with the `prcomp` function. Note that in general, singular vectors and principal component vectors are only unique up to sign!

```
> set.seed(1)
> A <- matrix(runif(200),nrow=20)
> P <- irlba(A, nv=1, center=colMeans(A))
> cbind(P$v, prcomp(A)$rotation[,1])
             [,1]        [,2]
 [1,] -0.46776256  0.46776158
 [2,]  0.26335590 -0.26335688
 [3,]  0.40484529 -0.40484233
 [4,]  0.20867236 -0.20867277
 [5,] -0.35983641  0.35983677
 [6,]  0.47980186 -0.47980195
 [7,] -0.04156462  0.04156139
 [8,]  0.34337641 -0.34338070
 [9,]  0.07680945 -0.07680577
[10,] -0.13846032  0.13846091
```

The implementation of the `center` and deflation options take 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 that replaces the matrix product `A %*% x` with `A %*% x - ds*du %*% t(dv) %*% x`.

2.3 User-Defined Matrix Multiplication

The `irlba` function includes an option for specifying a custom matrix multiplication function. Use this option, for example, the `big.matrix` class from the `bigmemory/bigalgebra` packages, or to compute the partial SVD of matrix-free linear operators, for example.

User-defined matrix operations are 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)
> mult <- function(x,y)
+         {
+           # check if x is a plain, row or column vector
+           if(is.vector(x) || ncol(x)==1 || nrow(x)==1)
+           {
+             return((x %*% y)/col_scale)
+           }
+           # else x is the matrix
+           x %*% (y/col_scale)
+         }

> irlba(A, 3, mult=mult)$d
[1] 2.8383609 0.7442858 0.6470490

> # Compare with:
> irlba(A, 3, scale=col_scale)$d
[1] 2.8383609 0.7442858 0.6470492

> # Compare with:
> svd(sweep(A,2,col_scale,FUN='/'))$d[1:3]
[1] 2.8383609 0.7442858 0.6470492
```

Due to technical implementation details, you are prohibited from using custom matrix product functions together with the rank 1 deflation options. However, custom matrix products are powerful and you can easily craft such a function to perform arbitrary subspace deflation within the function itself.

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:

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

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

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

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