Golub-Kahan-Lanczos bidiagonalization

Qingyu Huang
201703956

June 13, 2018

Introduction

The Golub-Kahan-Lanczos bidiagonalization factorization can be used on its own to solve linear systems and ordinary least squares problems, calculate the determinant and the (pseudo-)inverse of a matrix. But it mostly is used as the first step in the QR-like singular value decomposition (SVD) method, it also provides a powerful tool for solving large-scale singular value and related eigenvalue problems, as well as least-squares and saddle-point problems.

GKL bidiagonalization

The Golub-Kahan-Lanczos bidiagonalization is

$$U^*AV = B = \begin{bmatrix} \alpha_1 & & & & & \\ \beta_1 & \alpha_2 & & & & \\ & \beta_2 & \alpha_3 & & & \\ & & \beta_3 & \ddots & & \\ & & & \ddots & \alpha_{n-1} & \\ & & & & \beta_{n-1} & \alpha_n \end{bmatrix}$$

U and V are unitary matrices, and B is bidiagonal matrix. We take any $m \times 1$ column vector b as a starting vector, and choose $\beta_1 = \|b\|$, $u_1 = b/\beta_1$, $\alpha_1 = \|A^T u_1\|$ and $v_1 = (A^T u_1)/\alpha_1$.

Algorithm 1 Golub-Kahan-Lanczos Bidiagonalization procedure

```
1: Starting vector b, Choose $\beta_1 = \|b\|$, $u_1 = b/\beta_1$, $\alpha_1 = \|A^T u_1\|$ and
   $v_1 = (A^T u_1)/\alpha_1$.
2: for each $j = 1, 2, \cdots$ do
3:     $u_{j+1} = Av_j - \alpha_j u_j$
4:     $\beta_{j+1} = \|u_{j+1}\|_2$
5:     $u_{j+1} = u_{j+1}/\beta_{j+1}$
6:     $v_{j+1} = A^T u_{j+1} - \beta_{j+1} v_j$
7:     $\alpha_{j+1} = \|v_{j+1}\|_2$
8:     $v_{j+1} = v_{j+1}/\alpha_{j+1}$
9: end for
```

Implementation

We use GNU Scientific Library to implement the algorithm, suppose the size of matrix A is $m \times n$, m is the number of rows and n is the number of columns, so matrix A has three cases: **m=n**, **m<n** and **m>n**. We will test these three cases in our c language program.

The program include two files: **main.c** and **gkl_bidiag.c**. **main.c** include all the test code, **gkl_bidiag.c** include the Golub-Kahan-Lanczos bidiagonalization factorization function **gkl_bidiag**, the **print_matrix** function for printing the matrix out, and the **check_matrix_equal** function used for checking whetheer two matrices equal or not.

When the size of matrix A is $m \times n$, suppose the minimum of m and n is p, the size of matrix U should be $m \times p$, the size of matrix V should be $n \times p$, and the size of bidiagnoal matrix B should be $p \times p$. First assign the start vector b to an arbitrary unit 2-norm vector, in our program we assign vector b as $[1, 0, 0, \cdots, 0]^T$, then do every step exactly as the algorithm above and finally get the matrix U and matrix V, then group vector alpha and beta together to make the matrix B. The alpha vector is in the main diagonal of the matrix B, and beta vector is in the first diagonal below the main diagonal of the matrix B.

In the main program we randomly generate three different matrix with different size to represent m=n, m<n and m=n, we get matrix U,V and B of each matrix and then test U^*AV with the B, the difference between these two matrices are very small, the error is less than 1e-10.

References

Galassi, M., et al. "GNU Scientific Library Reference Manual, ISBN 0954612078." Library available online at http://www.gnu.org/software/gsl (2015).

Sikurajapathi, Indunil. Computing the Leading Singular Values of a Large Matrix by Direct and Inverse Iteration. Skolan för datakunskap och kommunikation, Kungliga Tekniska högskolan, 2007.

Appendix

```

void gkl_bidiag(gsl_matrix* A,gsl_matrix* U,
                gsl_matrix* B,gsl_matrix* V)
{
    int m=A->size1;
    int n=A->size2;
    int p=(m<n)? m:n;

    gsl_vector* start=gsl_vector_alloc(m);
    gsl_vector* alpha=gsl_vector_alloc(p);
    gsl_vector* beta=gsl_vector_alloc(p);
    gsl_vector* tmp=gsl_vector_alloc(n);
    gsl_vector* Vj=gsl_vector_alloc(n);
    gsl_vector* Uj=gsl_vector_alloc(m);

    gsl_vector_set(start,0,1);
    gsl_vector_set(beta,0,gsl_blas_dnrm2(start));

    gsl_vector_scale(start,
                     1.0/gsl_blas_dnrm2(start));

    gsl_matrix_set_col(U,0,start);
    gsl_blas_dgemv (CblasTrans,1.0,A,start,0,tmp);

    gsl_vector_set(alpha,0,gsl_blas_dnrm2(tmp));
    gsl_vector_scale(tmp,1.0/gsl_blas_dnrm2(tmp));
    gsl_matrix_set_col(V,0,tmp);


    for(int j=0; j<p-1; j++)
    {
        gsl_matrix_get_col(Vj, V, j);
        gsl_matrix_get_col(Uj, U, j);

        gsl_blas_dgemv (CblasNoTrans, 1.0, A, Vj,
                      (-1)*gsl_vector_get(alpha,j),Uj);

        gsl_vector_set(beta,j+1,gsl_blas_dnrm2(Uj));
        gsl_vector_scale(Uj,1.0/gsl_blas_dnrm2(Uj));
        gsl_matrix_set_col(U,j+1,Uj);
        gsl_blas_dgemv (CblasTrans, 1.0, A, Uj,
                      (-1)*gsl_vector_get(beta,j+1),Vj);
        gsl_vector_set(alpha,j+1,gsl_blas_dnrm2(Vj));
        gsl_vector_scale(Vj,1.0/gsl_blas_dnrm2(Vj));

        gsl_matrix_set_col(V,j+1,Vj);

    }

    for(int i=0; i<p; i++)
    {
        gsl_matrix_set(B,i,i,
                              gsl_vector_get(alpha,i));
    }
    for(int i=1; i<p; i++)
    {
        gsl_matrix_set(B,i,i-1,
                              gsl_vector_get(beta,i));
    }

    gsl_vector_free(start);
    gsl_vector_free(alpha);
    gsl_vector_free(beta);
    gsl_vector_free(tmp);
    gsl_vector_free(Vj);
    gsl_vector_free(Uj);

}
```