Computing Large Scale Linear Least Squares with SKiLLS

Zhen Shao

January 12, 2021

1 Overview

SKiLLS (SKetchIng-Linear-Least-Sqaures) is a C++ package for finding solutions to over-determined linear least square problems. SKiLLS uses a modern dimensionality reduction technique called sketching, and is particularly suited for large scale linear least squares where the number of measurements/observations is far greater than the number of variables.

Mathematically, SkiLLS solves

$$\min_{x \in \mathbb{R}^d} f(x) := \|Ax - b\|_2^2, \tag{1}$$

where $A \in \mathbb{R}^{n \times d}$ and $b \in \mathbb{R}^n$ given. The matrix A is allowed to be rank-deficient or nearly rank-deficient.

2 Use of External Library

The location of external libraries are coded in a config.mk file, given below.

```
# External library Locations
SUITSPARSEROOT ?= /home/shaoz/SuiteSparse
LAPACKROOT ?= /home/shaoz/lapack 3.8.0/
FortranROOT ?= /home/shaoz/intel/lib/intel64
HSLROOT ?= /home/shaoz/hsledits
RCPQR_ROOT ?=/Users/zhen/Downloads/hqrrp/lapack_compatible_sources
FFTW_ROOT?= /home/shaoz/usr/lib
LIBS_MKL = Wl, start group ${MKLROOT}/lib/intel64/libmkl_intel_ilp64.a \
 ${MKLROOT}/lib/intel64/libmkl_sequential.a \
 ${MKLROOT}/lib/intel64/libmkl_core.a Wl, liomp5 lpthread lm ldl

# Optimization Level
OPTIMIZATION = O3
CXX=icc

INCLUDES = I./include I./include/impl I${SUITSPARSEROOT}/include I${BOOSTROOT} \
 I${FFTW_INCLUDE}
CXXFLAGS = Wall ${OPTIMIZATION} march=native fPIC DMKL_ILP64 I${MKLROOT}/include std=c++11
CXXFLAGS_DEBUG = Wall O0 g march=native fPIC DMKL_ILP64 I${MKLROOT}/include
```

So that it's easy to see the required dependency for the package, and use the package on different systems (I need to run the software on three different machines.).

3 Generating a sparse dimensionality reduction matrix in compressed column format

```
// Generate an array 1:n
void gen_0_to_n(long n, long *returned_array)
{
    for (long i = 0; i<n; i++){
        returned_array[i] = i;
    }
}

// Randomly shuffle an array so that the first k indices are sampled from 1:n without replacement
void select_k_from_n(
    long n /* length of array */,
    long k /*number of indices to be shuffled*/,
    long* array_to_be_shuffled)
{
    // input check
    assert(n >= k);
    for (long i=0; i<k; i++){
        long j = ( rand() % (n i) ) +i;
        long tmp = array_to_be_shuffled[j];
        array_to_be_shuffled[j] = array_to_be_shuffled[i];
        array_to_be_shuffled[i] = tmp;
    }
}

// Generate a hashing matrix, return in the compressed column sparse format
void gen_hashing_matrix(
    long m /* number of rows */,
    long n /* number of columns */,
    long nnz_per_column /* number of non zeros per column */,
    // output
    long *row_indices , long *col_array, double *values)
{
    long one_to_m[m];
    gen_0_to_n(m, one_to_m);
    for (int i=0; i<n; i++){
        select_k_from_n(m, nnz_per_column, one_to_m);
        for (int j=0; j<nnz_per_column; j++){
            row_indices[nnz_per_column*i + j] = one_to_m[j];
        }
    } // the above loop generate the row indices

    for (long i=0; i<n+1; i++){
        col_array[i] = nnz_per_column*i;
    } // generate the column array

    double scaling = sqrt(nnz_per_column);
    for (long i=0; i<n; i++){
        for (long j =0; j<nnz_per_column; j++){
            values[i*nnz_per_column +j] = 1/scaling* ((rand()%2)*2 1);
        }
    }
}
```

4 Use abstract linear operator concept for a variety of preconditioner format

```
//                SPARSE PRE                //
template <typename T>
class LinOP_sparse_preconditioner
    : public LinOp<T>
{
 public:
  typedef long idx_t;
  LinOP_sparse_preconditioner( LinOp<T>& A, cholmod_sparse* R,
  int success=1, double perturb=1e 6)
      : _A(A) , _R(R), _n(_R>ncol), _m(_A.m()),
        _success(success), _perturb(perturb)
  {
    assert( A.n() == R>nrow );
    tmp = Vec<T>(A.n());
  }
  virtual idx_t const& m() const { return _m;}
  virtual idx_t const& n() const { return _n;}

  // overwrite matrix vector multiplication by using preconditioner
  virtual void mv( const char trans, const T alpha, const Vec<T> x, const T beta, Vec<T> y )
  {
    if( trans == 'n' )
    {
      tmp = x.copy();
      cs_usolve_cholmod_structure(_R, tmp.data(), _success, _perturb);
      _A.mv( 'n', alpha, tmp, beta, y );
    }
    else
    {
      _A.mv( 't', 1.0, x, 0.0, tmp );
      cs_utsolve_cholmod_structure(_R, tmp.data(), _success, _perturb);
      scal(beta, y);
      axpy(alpha, tmp, y);
    }
  }
 private:
  Vec_d      tmp;
  LinOp<T>&  _A;
  cholmod_sparse* _R;
  idx_t _n;
  idx_t _m;
  int  _success;
  double _perturb;
};

//                                  DENSE PRE                       //
template <typename T>
class LinOP_dense_preconditioner
    : public LinOp<T>
{
 public:
  typedef long idx_t;
  LinOP_dense_preconditioner( LinOp<T>& A, Mat_d& R)
      : _A(A) , _R(R),
      _up('u'), _trans('t'), _no_trans('n'), _diag('n')
```

```
  {
    assert( A.n() == R.m());
    tmp = Vec<T>(A.n());
  }
  virtual idx_t const& m() const { return _A.m();}
  virtual idx_t const& n() const { return _R.n();}

  // overwrite matrix vector multiplication by using preconditioner
  virtual void mv( const char trans, const T alpha, const Vec<T> x, const T beta, Vec<T> y )
  {

    if( trans == 'n' )
    {
      tmp = x.copy();
      dtrsv(&_up, &_no_trans, &_diag, &_R.n(), _R.data(), &_R.ld(), tmp.data(), &tmp.inc());
      _A.mv( 'n', alpha, tmp, beta, y );
    }
    else
    {
      _A.mv( 't', 1.0, x, 0.0, tmp );
      dtrsv(&_up, &_trans, &_diag, &_R.n(), _R.data(), &_R.ld(), tmp.data(), &tmp.inc());
      scal(beta, y);
      axpy(1.0, tmp, y);
    }
  }
 private:
  Vec_d      tmp;
  LinOp<T>&   _A;
  Mat_d& _R;
  char _up;
  char _trans;
  char _no_trans;
  char _diag;
};

            IC PRE                              //
template <typename T>
class LinOP_ic_preconditioner
    : public LinOp<T>
{
 public:
  typedef long idx_t;
  LinOP_ic_preconditioner( LinOp<T>& A, void* pkeep)
      : _A(A) , _pkeep(pkeep),
      _trans(1), _no_trans(0), _ifail(0)
  {
    // assert( A.n() == R.m()); cannot do the assertion because pkeep is difficult to handle
    tmp = Vec<T>(A.n());
  }
  virtual idx_t const& m() const { return _A.m();}
  virtual idx_t const& n() const { return _A.n();} // assuming preconditioning size match

  // overwrite matrix vector multiplication by using preconditioner
  virtual void mv( const char trans, const T alpha, const Vec<T> x, const T beta, Vec<T> y )
  {

    if( trans == 'n' )
    {
      // Note IC returns Lower trianglar, so transpose or no transpose is reversed
```

```
      hsl_mi35_solve(&_trans, &_A.n(), &_pkeep, x.data() , tmp.data() , &_ifail);
      _A.mv( 'n', alpha, tmp, beta, y );
    }
    else
    {
      _A.mv( 't', 1.0, x, 0.0, tmp );
      hsl_mi35_solve(&_no_trans, &_A.n(), &_pkeep, tmp.data(), tmp.data(), &_ifail);
      scal(beta, y);
      axpy(1.0, tmp, y);
    }
  }
 private:
  Vec_d      tmp;
  LinOp<T>&  _A;
  void* _pkeep;
  fint _trans;
  fint _no_trans;
  fint _ifail;
};


//                          Vanilla LSQR                          //

void lsqr( LinOp<double>& A, const Vec_d b,
           const double tol, const long maxit,
           Vec_d& x, int& flag, long& it, int debug )
{
  assert( A.m() == b.n() );
  assert( A.n() == x.n() );

  Vec_d v( A.n() );
  Vec_d w(v.n());
  double alpha;

  // explicitly make sure it is all zero...
  for (long i=0; i<A.n(); i++){
    v(i) = 0;
  }

  Vec_d u = b.copy();
  double beta = nrm2(u);
  if(beta > 0){
    scal(1.0/beta,u);
    A.mv( 't', 1.0, u, 0.0, v );
    alpha = nrm2(v); // norm of v might be zero!!!
  }

  if (alpha >0){
    scal(1.0/alpha,v);
    w= v.copy();
  }

  x = 0.0;
  double nrm_ar = alpha*beta;
  if (nrm_ar ==0){
    it=0;
    flag=0; // converge in 0 iteration
    return;
  }
```

```

  double phi, rho;
  double cs, sn;
  double phibar = beta;
  double rhobar = alpha;
  double theta;

  double nrm_a    = 0.0;
  double nrm_r;

  // double nrm_ar_0 = alpha*beta;
   flag =1;
   it = 0;
  for( long k = 0; k < maxit; ++k )
  {
    it++;

    A.mv( 'n', 1.0, v,  alpha, u );
    beta = nrm2(u);

    if (beta>0){
      scal(1.0/beta,u);
      nrm_a = sqrt( nrm_a*nrm_a + alpha*alpha + beta*beta );
      A.mv( 't', 1.0, u,  beta, v );
      alpha = nrm2(v);
      if (alpha >0){
        scal(1.0/alpha,v);
      }
    }

    rho    = sqrt( rhobar*rhobar + beta*beta );
    cs     = rhobar/rho;
    sn     = beta/rho;
    theta  = sn*alpha;
    rhobar =  cs*alpha;
    phi    = cs*phibar;
    phibar = sn*phibar;

    axpy( phi/rho, w, x );

    scal(  theta/rho, w );
    axpy( 1.0, v,  w );

    nrm_r  = phibar;
    nrm_ar = phibar*alpha*fabs(cs);

    if( nrm_ar < tol*nrm_a*nrm_r ){
        flag = 0;
        break;
    }

  }
}

//                          Preconditioned version, sparse preconditioner
void lsqr( LinOp<double>& A, const Vec_d b,
           const double tol, const long maxit,
           Vec_d& x, int& flag, long& it, cholmod_sparse* R_11,
           int success, double perturb, int debug) // LSQR with preconditioner
```

```
{
  LinOP_sparse_preconditioner<double> Apre(A, R_11, success, perturb);
  lsqr( Apre, b, tol, maxit, x, flag, it, debug );
}


//                    Preconditioned lsqr, dense preconditioner                    //

void lsqr_dense_pre( LinOp<double>& A, const Vec_d b,
          const double tol, const long maxit,
          Vec_d& x, int& flag, long& it, Mat_d& R, int debug) // LSQR with preconditioner
{
  LinOP_dense_preconditioner<double> Apre(A, R);
  lsqr( Apre, b, tol, maxit, x, flag, it, debug);
}


                    Preconditioned lsqr, using incomplete cholesky          //
void lsqr_ic( LinOp<double>& A, const Vec_d b,
          const double tol, const long maxit,
          Vec_d& x, int& flag, long& it, void* pkeep, int debug) // LSQR with ic preconditioner
{
  LinOP_ic_preconditioner<double> Apre(A, pkeep);
  lsqr( Apre, b, tol, maxit, x, flag, it, debug);
}
```

5 One version of randomised sparse linear least squares solver

```

// C++ solver for dense linear least squares using hashing
// Solving the linear least sqaures min_x |Ax b|_2
void ls_dense_hashing_blendenpik(
    Mat_d & A, /* m by n */
    Vec_d& b, /* m by 1 */
    Vec_d& x, /* solution, n by 1*/
    long& rank, /* detected rank (in case CPQR is used, otherwise equals to n*/
    int& flag, /* LSQR convergence flag (0=not convergent, 1=convergent) */
    long& it, /* LSQR iteration count */
    double gamma, /* over sampling ratio */
    long k, /* nnz per column in the hashing matrix */
    double abs_tol, /* absolute tolerance for the residual */
    double rcond, /* control minimal diagonal entries of R_11 */
    double it_tol, /* LSQR relative tolerance */
    long max_it, /* LSQR max iteration */
    int debug, /* no use */
    int wisdom /* flag that if fftw wisdom is to be used */
    )
{
    assert( A.m() == b.n() );
    assert( A.n() == x.n() );
    if (A.m()<A.n()) {
        throw std::runtime_error( "Matrix is not overdetermined." );
    }
    long n = A.n();
    double t1;
    double t2;
    double t_tmp;

```

```
    // sketch
    Vec_d c(ceil(A.n()*gamma)); // RHS to be sketched
    Mat_d As = rand_hashing_dct(A, b, c, gamma, k, wisdom);

    // build preconditioner, test explicit sketching solution
          long* E;
          E = (long*) calloc(As.n(), sizeof(long));
          double *tau = (double*) calloc(n, sizeof(double));
          //                  Compute CPQR factorization                //
          column_pivoted_qr(As, E, tau);

          //                  Compute Q^T c, store the result in c      //
          char left= 'L';
          char right = 'R';
          char trans = 'T';
          char no_trans = 'N';
          long one = 1;
          long two = 2;
          long workspace_size, info;
          double *workspace;
          double wsize_d;

          /* Query workspace size */
          workspace_size =  1;
          workspace = &wsize_d;
          dormqr(&left, &trans, &c.n(), &one, &As.n(), As.data(), &As.ld(), tau, c.data(), &c.n(),
            workspace, &workspace_size, &info);
          /* Compute */
          workspace_size = (long)wsize_d;
          workspace = (double *)malloc(sizeof(double) * workspace_size);
          dormqr(&left, &trans, &c.n(), &one, &As.n(), As.data(), &As.ld(), tau, c.data(), &c.n(),
            workspace, &workspace_size, &info);

          //                          get the R and the rank         //
            rank=0;
            for (long i=0; i< As.n(); i++){
              if (fabs( As(i,i) ) > rcond)
              {
                rank++;
              }
              else
              {
                break;
              }
            }
            Mat_d R;
            R = As.submat(0, rank, 0, rank);

          //                    Calculate the least square solution by back substitution          /
          char up = 'u';
          char diag = 'n';
          dtrsv(&up, &no_trans, &diag, &R.n(), R.data(), &R.ld(), c.data(), &c.inc());

          //              Get the basic solution                                  //

          for (long i=0; i < rank; i++){
              x.data()[E[i] 1] = c.data()[i];
          }

```

```
            for (long i = rank; i<As.n(); i++){
                x.data()[E[i] 1] = 0;
            }
            //              Test residual                                   //
            Vec_d rs = b.copy();
            A.mv('n', 1, x, 1, rs);
            std::cout << "Explicit␣Sketching␣Residual␣is:␣"<<  nrm2(rs) << std::endl;

            if (nrm2(rs) < abs_tol){
              std::cout << "Return␣solution␣found␣by␣explicit␣sketching␣with␣residual:␣"<< nrm2(rs)
              it = 0;
              flag = 0;
              return;
            }
      Vec_d xs = x.copy();

      // subselect columns of A
      long forward = 1;
      long backward = 0;
      dlapmt(&forward, &A.m(), &A.n(), A.data(),&A.ld(), E);
      Mat_d Areduced;
      Areduced = A.submat(0, A.m(), 0, rank);
      Vec_d y(rank);

      // preconditioned lsqr (dense preconditioner)
      lsqr_dense_pre(Areduced, rs, it_tol , max_it, y, flag, it, R, debug);
      dtrsv(&up, &no_trans, &diag, &R.n(), R.data(), &R.ld(), y.data(), &y.inc());
      // recover original solution by doing the permutation

      for (long i=0; i < rank; i++){
          x.data()[E[i] 1] = y.data()[i];
      }

      for (long i = rank; i<A.n(); i++){
          x.data()[E[i] 1] = 0;
      }

      // add the explicit sketching solution xs (initial guess)
      for (long i=0; i< x.n(); i++){
        x(i) = xs(i) + x(i);
      }

      // bring A back to its original form
      dlapmt(&backward, &A.m(), &A.n(), A.data(),&A.ld(), E);
      free(E);
}
```