QRkit: Sparse, Composable QR Decompositions for Efficient and Stable Solutions to Problems in Computer Vision
Supplementary material

Jan Svoboda
USI Lugano, Switzerland
jan.svoboda@usi.ch

Thomas Cashman, Andrew Fitzgibbon
Microsoft HoloLens, Cambridge, UK
tcashman,awf@microsoft.com

This supplementary material contains some implementation details and code examples.

1. "As-Banded-As-Possible" row reordering

The current implementation of "As-Banded-As-Possible" row reordering is a simple greedy algorithm (see Algorithm 1). Given a sparse matrix $A \in \mathbb{R}^{n \times m}$, we create permutation matrix $P_r \in \{0,1\}^{n \times n}$ such that rows with lower first nonzero index are ordered first. Such reordering is not guaranteed to be fill-in reducing. It only transforms A into a form that we may be able to factorize efficiently (Figure 1).

Algorithm 1 Algorithm that creates row reordering permutation based on indices of the first non-zero value in each row.

for $i \leftarrow 1 \dots n$ **do**
  $\text{rinfo}(i) \leftarrow (\textsc{IndexOfFirstNonZero}(A(i,:))$
end for
$(\text{rinfo}, \text{indices}) \leftarrow \textsc{Sort}(\text{rinfo})$
$A \leftarrow A(\text{indices}, :)$

2. Matrix Q representation

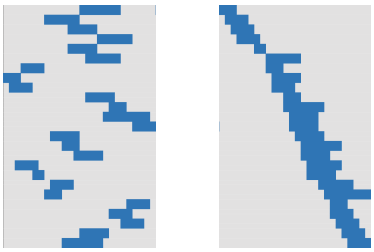
Matrix $\mathbf{Q}$ of the QR decomposition $A = \mathbf{QR}$, $A, \mathbf{R} \in \mathbb{R}^{n \times m}$, $\mathbf{Q} \in \mathbb{R}^{n \times n}$ is typically not stored explicitly, but rather expressed in terms of Householder reflectors[3].

A Householder reflector $v \in \mathbb{R}^n$ can be used to eliminate a single column j of A as

$$A(:,j) = (I - 2vv^T)A(:,j), \tag{1}$$

where $I \in \mathbb{R}^{n \times n}$ is an identity matrix. Equation 1 however forms $(I - 2vv^T)$ of size $n \times n$, which would be very inefficient, especially as the dimension n scales up. It is thus common practice to rewrite Equation 1 as

$$A(:,j) = A(:,j) - v\left(2v^\top A(:,j)\right). \tag{2}$$



(a) A (b) $P_r A$

Figure 1. Greedy row permutation P_r discovering banded structure in the matrix A.

Since $v^\top A(:,j)$ is just a scalar value, no $n \times n$ matrix is formed explicitly. Using Equation 2, $\mathbf{R}$ can be formed from A by performing a sequence of multiplications by Householder vectors, which can be implemented in terms of Algorithm 2.

Algorithm 2 Evaluation of $R = \mathbf{Q}^T A$ as multiplication by sequence of Householder vectors $V = \{v_1, \dots, v_m\}, v_k \in \mathbb{R}^n, k \in \{1, \dots, m\}$.

for $j \leftarrow 1 \dots m$ **do**
  for $k \leftarrow 1 \dots m$ **do**
    $\tau \leftarrow 2v_k^T A(:,j)$
    $A(:,j) \leftarrow A(:,j) - \tau v_k$
  end for
end for

3. Sparse Blocked Householder representation

The technique described in the previous section can be well adapted also for use with the blocked representation[1, 6]. In particular, we use the compressed WY representation[7] throughout this work. Similarly to Equation 1, elimination of a block of r columns starting at the jth column of A is expressed as

$$A(:, j:j+r) = (I + Y_k T_k Y_k^\top)^\top A(:, j:j+r), \tag{3}$$

where $Y \in \mathbb{R}^{n \times r}$ is lower trapezoidal, $T \in \mathbb{R}^{r \times r}$ is upper triangular and $I \in \mathbb{R}^{n \times n}$ is an identity matrix. Following the same principle as in Equation 2, to avoid forming the $(I + YTY^\top)$ explicitly, we can rewrite Equation 3 as

$$A(:, j : j + r) += Y_k(T_k^\top (Y_k^\top A(:, j : j + r))). \quad (4)$$

The attentive reader will notice that $Y(T^\top (Y^\top A(:, j : j + r))) \in \mathbb{R}^{n \times r}$ and therefore the full $n \times n$ matrix is never formed. Typically $r \ll n$ and so all the operations are performed on very small blocks compared to the full size of A.

Leveraging the banded sparsity structure of A, we can further reduce the size of the blocks greatly. Given A, Y, T as in Figure 2, notice the sparsity structure of Y: a block diagonal matrix of similar structure to A, augmented with blocks on the main diagonal. Put another way, each column of Y has two blocks of contiguous nonzeros. Any reasonably banded sparse matrix A will yield block Y with similar sparsity pattern. Given kth block with Y_k, T_k, it holds that the sub-block on the upper diagonal is $r \times r$, while the lower diagonal sub-block is of size $n_k - r \times r$, where n_k is the number of nonzero rows in the kth block. We consider this during the implementation and instead of storing Y as a sparse matrix, we store it as two dense sub-blocks, remembering offset of each block, as follows:

```
class BlockYTY {
  int id;  // Pos. on the upper diagonal
  int il;  // Pos. on the lower diagonal
  Eigen::Matrix<Scalar> Yd; // In practice,
  Eigen::Matrix<Scalar> Yl; // both are stored
                            // in one Matrix
  Eigen::Matrix<Scalar> T;

  ...
};
```

Matrix-matrix operations on the small dense blocks are much more efficient than for the general sparse representation. We describe the whole procedure by Algorithm 3.

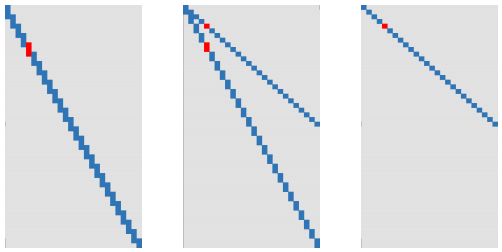
It is the case that constructing dense blocks from sparse sub-blocks, performing the matrix product and filling the result back is much more efficient than performing sparse matrix operations on the full-size blocks.

We group the YT blocks together using a custom container for block sparse matrices which behaves similarly to `std::vector`. This container provides expression templates for applying the sequence of YT blocks to a matrix, as described by Algorithm 3. We therefore allow the user to write the rather complicated Equation 4 in a simplified form as shown in the following code example.

```
template <typename BlockType, typename IdxType>
class SparseBlockCOO {
  struct Element {
    IdxType row;
    IdxType col;
    BlockType value;
    ...
  };

  typedef std::vector<Element> ElementsVec;
  ElementsVec elems;

  // Methods definition
  ...
};

class SparseBlockYTY : SparseBlockCOO {
  // sequenceYTY() expr template definition
};

SparseBlockYTY blks;
// Fill in blks here
...
vec = blks.sequenceYTY().transpose() * vec;
```

Algorithm 3 Evaluation of $R = \mathbf{Q}^T A$ as multiplication by sequence of YTY^T blocks.

for $j \leftarrow 1 \dots m$ **do**
  for $k \leftarrow 1 \dots p$ **do**
    GETSUBVEC$(A, id, il, n_k, r, j, a_{jk})$
    $a_{jk} \leftarrow a_{jk} - (Y_k(T_k^T(Y_k^T a_{jk})))$
    SETSUBVEC$(a_{jk}, id, il, n_k, r, j, A)$
  end for
end for
function GETSUBVEC$(A, id, il, n_k, r, j, a_{jk})$
  $a_{jk}(0 : r) \leftarrow A(id : id + r, j)$
  $a_{jk}(r : n_k) \leftarrow A(il : il + n_k - r, j)$
end function
function SETSUBVEC$(a_{jk}, id, il, n_k, r, j, A)$
  $A(id : id + r, j) \leftarrow a_{jk}(0 : r)$
  $A(il : il + n_k - r, j) \leftarrow a_{jk}(r : n_k)$
end function



(a) A (b) Y (c) T

Figure 2. Block banded matrix A and dense sub-blocks Y and T as columns of sparse matrices. The blocks Y_k and T_k are grouped together in the form of a sparse matrix only for visualization purposes. The k^{th} block of A together with its Y_k and T_k dense sub-blocks are emphasized in red color.

4. BacktrackLevMarq

Throughout this work, we use our reimplementation of Levenberg Marquardt based on Matlab code from AWFUL[2], which combines ideas from the original implementation of Moré[5] together with modifications introduced by Lourakis[4].

This allows us to perform non-linear optimizations on large-scale problems that cannot be handled by the implementation currently available in Eigen. Besides that, using BacktrackLevMarq, we can compare to the SSBA bundle adjustment software[8], as it internally uses very similar Levenberg-Marquardt based optimizer.

The main difference between the classical implementation of Moré and ours is the approach to updating the damping parameter λ. Instead of using a trust-region framework[5], we follow ideas of Lourakis[4] and update λ as follows:

$$\rho = \frac{\|\epsilon_p\|^2 - \|x - f(p_{new})\|^2}{\delta_p^T(\lambda\delta_p + J^T\epsilon_p)}, \tag{5}$$

$$\lambda = \begin{cases} \rho > 0 & \lambda \cdot \max(\frac{1}{3}, 1 - (2\rho - 1)^3) \\ otherwise & \lambda v \end{cases}, \tag{6}$$

where $\epsilon_p = x - f(p)$ is the residual vector for the current parameter vector p, p_{new} is the updated parameter vector using the step δ_p and J is the Jacobian of function f. The parameter v is the multiplicative factor of λ, which is updated in every step together with λ by the following acceleration rule:

$$v = \begin{cases} \rho > 0 & v = 2 \\ otherwise & v = 2v \end{cases} \tag{7}$$

5. Computational complexity

Throughout our work, we use the compressed representation of Householder products WY described above in Section 3. Schreiber and van Loan [7] show that this yields computational complexity $O(n_k m_k r_k)$ for factorization of a matrix block $B \in \mathbb{R}^{n_k \times m_k}$ with rank r_k. We list algorithmic complexity for our four composite QR solvers below.

The most simple block diagonal case is straightforward: its complexity is $O(K n_k m_k r_k)$, where K is the number of non-overlapping diagonal blocks, each of size $n_k \times m_k$ with rank r_k.

For the block banded case, if the number of overlapping columns r_c is a fixed constant independent of the column dimension m of A, then asymptotic complexity is again $O(K n_k m_k r_k)$ for the number of blocks K. Note, however, that the sizes of the matrices involved in the block QR decompositions are larger: for an overlap of two neighbouring blocks, we would expect an eight-fold increase in the computational cost of the individual QR decompositions of size $2n_k \times 2m_k$ and rank $2r_k$. The constant used to bound the complexity is therefore higher in this case, and depends strongly on the amount of overlap between blocks.

Another case is horizontal concatenation, where the computational complexity depends on the method used to factorize the right and left superblocks. The full process involves factorization of the left block followed by matrix multiplication and factorization of the right block. Since the computational complexity of the general QR decomposition algorithm is $O(n^3)$, and this expression bounds the complexity of both the matrix multiplication and subsequent factorization, we expect asymptotic complexity of $O(n^3)$.

Lastly, the vertical concatenation, is composed of row permutation followed by application of either a block diagonal or block banded solver. Since the row permutation complexity is merely $O(n^2)$ in the number of matrix rows n, which we expect to be larger than m, the complexity is again bounded by $O(K n_k m_k r_k)$, assuming that the permutation yields a favorable sparsity pattern which reduces the block overlap to zero (i.e. the block diagonal case) or to a fixed constant independent of matrix dimension (as analysed above for the block banded case).

6. Code example

Appendix A contains full code example for the ellipse fitting problem. It illustrates how to describe a problem at hand using the `SparseFunctor` object, which holds the definitions of $f(x)$ and Jacobian J together with the definition of the QR solver to be used.

7. Repository

QRkit implementation is available as pull-request repository at the moment: `https://bitbucket.org/jasvob/eigen_sparse_qr`. This repository contains also the ellipse fitting benchmark, which is included as a part of the Eigen unit tests. Our updates to the Eigen official repository have been very positively recevied, and we believe they will become part of the official Eigen release in the near future.

The bundle adjustment benchmarks presented in the paper are available at `https://github.com/jasvob/BundleAdjustment_Benchmarks`.

References

[1] C. Bischof and C. V. Loan. The WY representation for products of householder matrices. *SIAM Journal on Scientific and Statistical Computing*, 8(1):s2–s13, 1987.

[2] A. W. Fitzgibbon. AWF utility library. https://github.com/awf/awful.

[3] A. S. Householder. A class of methods for inverting matrices. *Journal of the Society for Industrial and Applied Mathematics*, 6(2):189–195, 1958.

[4] M. I. A. Lourakis and A. A. Argyros. SBA: A software package for generic sparse bundle adjustment. *ACM Trans. Math. Softw.*, 36(1):2:1–2:30, 2009.

[5] J. J. Moré. *Levenberg–Marquardt algorithm: implementation and theory*. 1977.

[6] R. Schreiber and B. Parlett. Block reflectors: Theory and computation. *SIAM Journal on Numerical Analysis*, 25(1):189–205, 1988.

[7] R. Schreiber and C. van Loan. A storage-efficient WY representation for products of householder transformations. *SIAM J. Sci. Stat. Comput.*, 10(1):53–57, 1989.

[8] C. Zach. Robust bundle adjustment revisited. In *Computer Vision–ECCV 2014*, pages 772–787. Springer, 2014.

A. Ellipse fitting problem

```
// Let's use modern-as-possible C++, so elide ellipse ctors etc
using namespace Eigen;
typedef int IndexType;

typedef SparseMatrix<Scalar, ColMajor, IndexType> JacobianType;

// Define ellipse fitting problem functor
struct EllipseFittingFunctor : SparseFunctor<double, IndexType>
{
  // Class data: 2 x N matrix with each column a 2D point
  Matrix2Xd points;

  // Constructor initializes points, and tells the base class how many parameters there are in total
  EllipseFittingFunctor(const Matrix2Xd& points) :
    SparseFunctor<double, IndexType>(nParamsModel + points.cols(), points.cols() * 2),
    points(points) {
  }

  typedef VectorXd InputType; // Or is this defined in base?

  // Functor function f(x)
  int operator()(const InputType& x, ValueType& fx) const {
    // Ellipse parameters are the last 5 entries
    auto params = params.tail(5);
    double a = params[0];
    double b = params[1];
    double x0 = params[2];
    double y0 = params[3];
    double r = params[4];

    // Correspondences (t values) are the first N
    for (int i = 0; i < points.cols(); i++) {
      double t = x[i];
      double x = a*cos(t)*cos(r) - b*sin(t)*sin(r) + x0;
      double y = a*cos(t)*sin(r) + b*sin(t)*cos(r) + y0;
      fx[2 * i + 0] = points[0, i] - x;
      fx[2 * i + 1] = points[1, i] - y;
    }

    return 0;
  }

  // Functor jacobian J
  int df(const InputType& uv, JacobianType& fjac) {
    // Ellipse parameters are the last 5 entries
    auto params = uv.tail(5);
    double a = params[0];
    double b = params[1];
    double r = params[4];

    int npoints = points.cols();
    // Triplet entries are (row_index, col_index, value), and there are
    npoints * rows_per_point * nonzeros_per_row
    TripletArray<JacobianType::Scalar, IndexType> triplets(npoints * 2 * 5);
    for (int i = 0; i < npoints; i++) {
      double t = uv(i);
      triplets.add(2 * i, i, +a*cos(r)*sin(t) + b*sin(r)*cos(t));
      triplets.add(2 * i, npoints + 0, -cos(t)*cos(r));
      triplets.add(2 * i, npoints + 1, +sin(t)*sin(r));
      triplets.add(2 * i, npoints + 2, -1);
      triplets.add(2 * i, npoints + 4, +a*cos(t)*sin(r) + b*sin(t)*cos(r));

      triplets.add(2 * i + 1, i, +a*sin(r)*sin(t) - b*cos(r)*cos(t));
      triplets.add(2 * i + 1, npoints + 0, -cos(t)*sin(r));
      triplets.add(2 * i + 1, npoints + 1, -sin(t)*cos(r));
```

```
      triplets.add(2 * i + 1, npoints + 3, -1);
      triplets.add(2 * i + 1, npoints + 4, -a*cos(t)*cos(r) + b*sin(t)*sin(r));
    }

    fjac.setFromTriplets(triplets.begin(), triplets.end());
    return 0;
  }

  // QR solver for dense sub-blocks of the block diagonal block (here they are just 2x1 matrices)
  typedef ColPivHouseholderQR<Matrix2x1d> DenseQRSolver;
  // QR solver for the block diagonal block J1
  typedef BlockDiagonalSparseQR<JacobianType, DenseQRSolver> LeftSuperBlockSolver;
  // QR solver for Q1'J2 is general dense (faster than general sparse by about 1.5x for n=500K)
  typedef ColPivHouseholderQR<MatrixXXd> RightSuperBlockSolver;
  // QR solver for horizontal concatenation of the above.
  typedef BlockAngularSparseQR<JacobianType, LeftSuperBlockSolver, RightSuperBlockSolver> QRSolver;

  // Tell the algorithm how to set the QR solver parameters.
  void initQRSolver(QRSolver &qr) {
    // We know the pattern of the block diagonal part, set it in advance
    // 3x1 dense blocks - 2 per residual plus 1 for damping
    qr.getLeftSolver().setPattern(points.cols() * 2 + points.cols(),  points.cols(), 3, 1);
    // Tell the solver the size of the block diagonal part
    qr.setSparseBlockParams(points.cols() * 2 + points.cols(), points.cols());
  }
};

void run()
{
  ...
  // Initial parameters
  VectorXd params(5 + npoints);
  {
    // fill params with initial estimates of ellipse parameters and t values
  }

  // Run Levenberg Marquardt with the defined QR solver
  EllipseFittingFunctor functor(points);
  Eigen::BacktrackLevMarq<EllipseFittingFunctor> lm(functor);
  auto info = lm.minimize(params);

   ...
}
```