```
// A simple quickref for Eigen. Add anything that's missing.
// Main author: Keir Mierle

#include <Eigen/Dense>

Matrix<double, 3, 3> A;               // Fixed rows and cols. Same as Matrix3d.
Matrix<double, 3, Dynamic> B;         // Fixed rows, dynamic cols.
Matrix<double, Dynamic, Dynamic> C;   // Full dynamic. Same as MatrixXd.
Matrix<double, 3, 3, RowMajor> E;     // Row major; default is column-major.
Matrix3f P, Q, R;                     // 3x3 float matrix.
Vector3f x, y, z;                     // 3x1 float matrix.
RowVector3f a, b, c;                  // 1x3 float matrix.
VectorXd v;                           // Dynamic column vector of doubles
double s;

// Basic usage
// Eigen          // Matlab           // comments
x.size()          // length(x)        // vector size
C.rows()          // size(C,1)        // number of rows
C.cols()          // size(C,2)        // number of columns
x(i)              // x(i+1)           // Matlab is 1-based
C(i,j)            // C(i+1,j+1)       //

A.resize(4, 4);   // Runtime error if assertions are on.
B.resize(4, 9);   // Runtime error if assertions are on.
A.resize(3, 3);   // Ok; size didn't change.
B.resize(3, 9);   // Ok; only dynamic cols changed.

A << 1, 2, 3,     // Initialize A. The elements can also be
     4, 5, 6,     // matrices, which are stacked along cols
     7, 8, 9;     // and then the rows are stacked.
B << A, A, A;     // B is three horizontally stacked A's.
A.fill(10);       // Fill A with all 10's.

// Eigen                                // Matlab
MatrixXd::Identity(rows,cols)           // eye(rows,cols)
C.setIdentity(rows,cols)                // C = eye(rows,cols)
MatrixXd::Zero(rows,cols)               // zeros(rows,cols)
C.setZero(rows,cols)                    // C = ones(rows,cols)
MatrixXd::Ones(rows,cols)               // ones(rows,cols)
C.setOnes(rows,cols)                    // C = ones(rows,cols)
MatrixXd::Random(rows,cols)             // rand(rows,cols)*2-1        // MatrixXd::Random returns uniform random numbers in
(-1, 1).
C.setRandom(rows,cols)                  // C = rand(rows,cols)*2-1
```

```
VectorXd::LinSpaced(size,low,high)   // linspace(low,high,size)'
v.setLinSpaced(size,low,high)        // v = linspace(low,high,size)'


// Matrix slicing and blocks. All expressions listed here are read/write.
// Templated size versions are faster. Note that Matlab is 1-based (a size N
// vector is x(1)...x(N)).
// Eigen                           // Matlab
x.head(n)                          // x(1:n)
x.head<n>()                        // x(1:n)
x.tail(n)                          // x(end - n + 1: end)
x.tail<n>()                        // x(end - n + 1: end)
x.segment(i, n)                    // x(i+1 : i+n)
x.segment<n>(i)                    // x(i+1 : i+n)
P.block(i, j, rows, cols)          // P(i+1 : i+rows, j+1 : j+cols)
P.block<rows, cols>(i, j)          // P(i+1 : i+rows, j+1 : j+cols)
P.row(i)                           // P(i+1, :)
P.col(j)                           // P(:, j+1)
P.leftCols<cols>()                 // P(:, 1:cols)
P.leftCols(cols)                   // P(:, 1:cols)
P.middleCols<cols>(j)              // P(:, j+1:j+cols)
P.middleCols(j, cols)              // P(:, j+1:j+cols)
P.rightCols<cols>()                // P(:, end-cols+1:end)
P.rightCols(cols)                  // P(:, end-cols+1:end)
P.topRows<rows>()                  // P(1:rows, :)
P.topRows(rows)                    // P(1:rows, :)
P.middleRows<rows>(i)              // P(i+1:i+rows, :)
P.middleRows(i, rows)              // P(i+1:i+rows, :)
P.bottomRows<rows>()               // P(end-rows+1:end, :)
P.bottomRows(rows)                 // P(end-rows+1:end, :)
P.topLeftCorner(rows, cols)        // P(1:rows, 1:cols)
P.topRightCorner(rows, cols)       // P(1:rows, end-cols+1:end)
P.bottomLeftCorner(rows, cols)     // P(end-rows+1:end, 1:cols)
P.bottomRightCorner(rows, cols)    // P(end-rows+1:end, end-cols+1:end)
P.topLeftCorner<rows,cols>()       // P(1:rows, 1:cols)
P.topRightCorner<rows,cols>()      // P(1:rows, end-cols+1:end)
P.bottomLeftCorner<rows,cols>()    // P(end-rows+1:end, 1:cols)
P.bottomRightCorner<rows,cols>()   // P(end-rows+1:end, end-cols+1:end)

// Of particular note is Eigen's swap function which is highly optimized.
// Eigen                           // Matlab
R.row(i) = P.col(j);               // R(i, :) = P(:, i)
R.col(j1).swap(mat1.col(j2));      // R(:, [j1 j2]) = R(:, [j2, j1])
```

```
// Views, transpose, etc; all read-write except for .adjoint().
// Eigen                                // Matlab
R.adjoint()                            // R'
R.transpose()                          // R.' or conj(R')
R.diagonal()                           // diag(R)
x.asDiagonal()                         // diag(x)
R.transpose().colwise().reverse(); // rot90(R)
R.conjugate()                          // conj(R)

// All the same as Matlab, but matlab doesn't have *= style operators.
// Matrix-vector.  Matrix-matrix.   Matrix-scalar.
y  = M*x;          R  = P*Q;        R  = P*s;
a  = b*M;          R  = P - Q;      R  = s*P;
a *= M;            R  = P + Q;      R  = P/s;
                   R *= Q;          R  = s*P;
                   R += Q;          R *= s;
                   R -= Q;          R /= s;

// Vectorized operations on each element independently
// Eigen                  // Matlab
R = P.cwiseProduct(Q);    // R = P .* Q
R = P.array() * s.array();// R = P .* s
R = P.cwiseQuotient(Q);   // R = P ./ Q
R = P.array() / Q.array();// R = P ./ Q
R = P.array() + s.array();// R = P + s
R = P.array() - s.array();// R = P - s
R.array() += s;           // R = R + s
R.array() -= s;           // R = R - s
R.array() < Q.array();    // R < Q
R.array() <= Q.array();   // R <= Q
R.cwiseInverse();         // 1 ./ P
R.array().inverse();      // 1 ./ P
R.array().sin()           // sin(P)
R.array().cos()           // cos(P)
R.array().pow(s)          // P .^ s
R.array().square()        // P .^ 2
R.array().cube()          // P .^ 3
R.cwiseSqrt()             // sqrt(P)
R.array().sqrt()          // sqrt(P)
R.array().exp()           // exp(P)
R.array().log()           // log(P)
R.cwiseMax(P)             // max(R, P)
R.array().max(P.array())  // max(R, P)
R.cwiseMin(P)             // min(R, P)
```

```
R.array().min(P.array())  // min(R, P)
R.cwiseAbs()              // abs(P)
R.array().abs()           // abs(P)
R.cwiseAbs2()             // abs(P.^2)
R.array().abs2()          // abs(P.^2)
(R.array() < s).select(P,Q);  // (R < s ? P : Q)

// Reductions.
int r, c;
// Eigen                  // Matlab
R.minCoeff()              // min(R(:))
R.maxCoeff()              // max(R(:))
s = R.minCoeff(&r, &c)    // [s, i] = min(R(:)); [r, c] = ind2sub(size(R), i);
s = R.maxCoeff(&r, &c)    // [s, i] = max(R(:)); [r, c] = ind2sub(size(R), i);
R.sum()                   // sum(R(:))
R.colwise().sum()         // sum(R)
R.rowwise().sum()         // sum(R, 2) or sum(R')'
R.prod()                  // prod(R(:))
R.colwise().prod()        // prod(R)
R.rowwise().prod()        // prod(R, 2) or prod(R')'
R.trace()                 // trace(R)
R.all()                   // all(R(:))
R.colwise().all()         // all(R)
R.rowwise().all()         // all(R, 2)
R.any()                   // any(R(:))
R.colwise().any()         // any(R)
R.rowwise().any()         // any(R, 2)

// Dot products, norms, etc.
// Eigen                  // Matlab
x.norm()                  // norm(x).    Note that norm(R) doesn't work in Eigen.
x.squaredNorm()           // dot(x, x)   Note the equivalence is not true for complex
x.dot(y)                  // dot(x, y)
x.cross(y)                // cross(x, y) Requires #include <Eigen/Geometry>

//// Type conversion
// Eigen                           // Matlab
A.cast<double>();                  // double(A)
A.cast<float>();                   // single(A)
A.cast<int>();                     // int32(A)
A.real();                          // real(A)
A.imag();                          // imag(A)
// if the original type equals destination type, no work is done
```

```
// Note that for most operations Eigen requires all operands to have the same type:
MatrixXf F = MatrixXf::Zero(3,3);
A += F;                // illegal in Eigen. In Matlab A = A+F is allowed
A += F.cast<double>(); // F converted to double and then added (generally, conversion happens on-the-fly)

// Eigen can map existing memory into Eigen matrices.
float array[3];
Vector3f::Map(array).fill(10);            // create a temporary Map over array and sets entries to 10
int data[4] = {1, 2, 3, 4};
Matrix2i mat2x2(data);                    // copies data into mat2x2
Matrix2i::Map(data) = 2*mat2x2;           // overwrite elements of data with 2*mat2x2
MatrixXi::Map(data, 2, 2) += mat2x2;      // adds mat2x2 to elements of data (alternative syntax if size is not know at
compile time)

// Solve Ax = b. Result stored in x. Matlab: x = A \ b.
x = A.ldlt().solve(b));  // A sym. p.s.d.    #include <Eigen/Cholesky>
x = A.llt() .solve(b));  // A sym. p.d.      #include <Eigen/Cholesky>
x = A.lu()  .solve(b));  // Stable and fast. #include <Eigen/LU>
x = A.qr()  .solve(b));  // No pivoting.     #include <Eigen/QR>
x = A.svd() .solve(b));  // Stable, slowest. #include <Eigen/SVD>
// .ldlt() -> .matrixL() and .matrixD()
// .llt()  -> .matrixL()
// .lu()   -> .matrixL() and .matrixU()
// .qr()   -> .matrixQ() and .matrixR()
// .svd()  -> .matrixU(), .singularValues(), and .matrixV()

// Eigenvalue problems
// Eigen                          // Matlab
A.eigenvalues();                  // eig(A);
EigenSolver<Matrix3d> eig(A);     // [vec val] = eig(A)
eig.eigenvalues();                // diag(val)
eig.eigenvectors();               // vec
// For self-adjoint matrices use SelfAdjointEigenSolver<>
```