API Collection

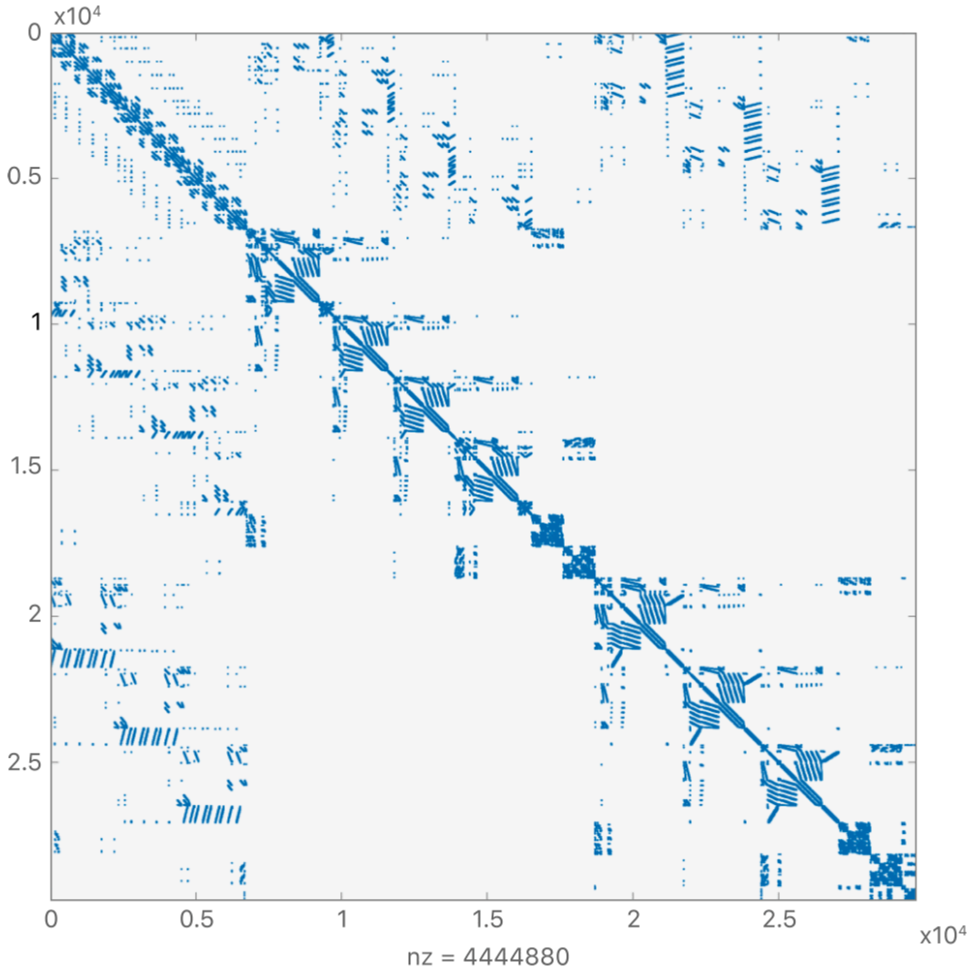
Sparse Solvers

Solve systems of equations where the coefficient matrix is sparse.

Overview

The Sparse Solvers library in the Accelerate framework handles the solution of systems of equations where the coefficient matrix is sparse. That is, most of the entries in the matrix are zero. The Sparse Solvers library provides a sparse counterpart to the dense factorizations and linear solvers that LAPACK provides.

Many problems in science and technology require the solution of large systems of simultaneous equations. When these equations are linear, you usually represent them as the matrix equation $Ax = b$. Even when the equations are nonlinear, you often solve the problem as a sequence of linear approximations.



Sparse matrices

Routines from libraries such as BLAS and LAPACK work with matrices that you store as a 2D dense array of floating-point values. However, the algorithms you use to manipulate matrices and solve equations normally require $O(n\_^2)\_$ data and $O(n\_^3)\_$ operations. As a result, scaling to a large *n* is prohibitive.

To avoid the expense of these algorithms, you can leverage the fact that in many real-world applications, matrices can contain many entries that are zero. Such matrices are called *sparse* (conversely, nonsparse matrices are called *dense*).

These zeros arise naturally in these types of situations:

Sparse data sets
: For example, each user buys only a small fraction of the products from a retailer.

Limited connectivity
: For example, most people on social networks only connect to a tiny proportion of the entire user base.

Physical properties
: For example, points on structural meshes only connect to locally adjacent points.

By exploiting these zero entries, you can often reduce the storage and computational requirements to $O(\tau*n)$ and $O(\tau*n\_^2)$,_ respectively, where τ is the average number of entries in each column. This reduction makes the solution of large problems (*n* in the millions or larger) tractable on most computers.

For example, the sparse benchmark matrix *ldoor*, which arises from structural modeling, has 952,200 x 952,200 entries with an average of 25 nonzero `Double` entries per column. The following table shows the number of floating-point operations (1 Tflop is 10^{12} floating-point operations) and the memory necessary to perform Cholesky factorization on that matrix:

	Floating-point operations	**Memory**
Dense	287,782 Tflop	6800 GB
Sparse	0.0783 Tflop	1.2 GB

Solution approaches

The Accelerate framework offers two solution approaches:

- **Direct methods** perform a factorization such as Cholesky ($A = LL^T$) or QR. These methods provide a fast and accurate opaque solution.
- **Iterative methods** find an approximate solution requiring only repeated multiplication by *A* or A^T. Although they require less memory than direct methods, and can be faster for very large problems, they typically require problem-specific preconditioners to be effective.

The following table summarizes the differences between direct methods and iterative methods:

	Direct methods	**Iterative methods**
Ease of use	Simple	Complex
Accuracy	Machine precision	Square root of machine precision
Speed	Fast for small problems Quite fast for larger problems	Fastest for large problems, but only with a suitable problem-specific preconditioner
Memory requirements	High	Low

In contrast to direct methods, iterative methods provide a way for expert users to find approximate solutions faster using less memory. You can also use iterative methods when forming the explicit matrix is prohibitively expensive, but performing matrix-vector multiplications is performant. However, to achieve these gains, you need to select an appropriate preconditioner (an operator that approximates the inverse of *A*) that's specific to your problem. It's best to try a direct method before trying to use iterative methods.

Iterative refinement

It's sometimes possible to improve the accuracy of the solution to *Ax* = *b* using *iterative refinement*. After finding an initial solution, iterative refinement reuses the factorization to find a series of small corrections with the aim of reducing the backward error.

The following code shows how to refine the values in the unknowns vector, *x*, over a fixed number of iterations:

```
// The coefficient matrix, `A`.
let A: SparseMatrix_Double! = [ ... ]

// The right-hand-side vector, `b`.
let b: DenseVector_Double! = [ ... ]

// The factorization of `A` computed by `SparseFactor()`.
let factorization: SparseOpaqueFactorization_Double! = [ ... ]

// The unknowns vector, `x` computed by `SparseSolve()`.
let x: DenseVector_Double! = [ ... ]

let count = Int(x.count)
let n = vDSP_Length(count)

let residualData = UnsafeMutableBufferPointer<Double>.allocate(capacity: count)
let residual = DenseVector_Double(count: Int32(count), data: residualData.baseAddres

let correctionData = UnsafeMutableBufferPointer<Double>.allocate(capacity: count)
let correction = DenseVector_Double(count: Int32(count), data: correctionData.baseAd

defer {
    residualData.deallocate()
    correctionData.deallocate()
}
```

```
let maximumIteratons = 3

for _ in 0 ..< maximumIteratons {

    // Calculate residual r = Ax - b.
    vDSP_vnegD(b.data, 1,
               residual.data, 1,
               n)
    SparseMultiplyAdd(A, x, residual);

    // Solve for correction and update x.
    SparseSolve(factorization, residual, correction);

    // vDSP operation that calculates `x[i] -= correction[i]` for
    // `i` in `0 ..< n`.
    vDSP_vsubD(correction.data, 1,
               x.data, 1,
               x.data, 1,
               n)
}
```

Sparse Solvers and multithreading

By default, the Sparse Solvers library runs in multithreaded mode. Because multithreaded mode may sum child nodes and their ancestors in different orders, the solutions that the library provides may be different — although equally valid — across different runs.

To ensure that results are deterministic, set `VECLIB_MAXIMUM_THREADS=1` to specify single-threaded mode.

Topics

Creating sparse matrices

Creating sparse matrices

Create sparse matrices for factorization and solving systems.

`struct SparseMatrix_Double`

A structure that contains a sparse matrix of double-precision, floating-point values.

`struct SparseMatrix_Float`

A structure that contains a sparse matrix of single-precision, floating-point values.

Conversion from Other Formats

Create sparse matrices from coordinate format arrays and BLAS opaque matrices.

Creating dense matrices and dense vectors

`struct DenseMatrix_Double`

A structure that contains a dense matrix of double-precision, floating-point values.

`struct DenseMatrix_Float`

A structure that contains a dense matrix of single-precision, floating-point values.

`struct DenseVector_Double`

A structure that contains a dense vector of double-precision, floating-point values.

`struct DenseVector_Float`

A structure that contains a dense vector of single-precision, floating-point values.

Creating sparse complex matrices

`struct SparseMatrix_Complex_Double`

A type representing a sparse complex matrix.

`struct SparseMatrix_Complex_Float`

A type representing a sparse complex matrix.

`struct SparseAttributesComplex_t`

A type representing the attributes of a matrix.

`struct SparseMatrixStructureComplex`

A type representing the sparsity structure of a sparse complex matrix.

Creating dense complex matrices and dense complex vectors

`struct DenseMatrix_Complex_Double`

Contains a dense `rowCount` x `columnCount` matrix of complex double values stored in column-major order.

`struct DenseMatrix_Complex_Float`

Contains a dense `rowCount` x `columnCount` matrix of complex float values stored in column-major order.

`struct DenseVector_Complex_Double`

Contains a dense vector of double complex values.

`struct DenseVector_Complex_Float`

Contains a dense vector of float complex values.

Solving systems with direct sparse methods

Solving systems using direct methods

Use direct methods to solve systems of equations where the coefficient matrix is sparse.

`struct SparseOpaqueFactorization_Double`

A structure that represents the factorization of a matrix of double-precision, floating-point values.

`struct SparseOpaqueFactorization_Float`

A structure that represents the factorization of a matrix of single-precision, floating-point values.

`struct SparseOpaqueFactorization_Complex_Double`

A semi-opaque type representing a matrix factorization in complex double.

`struct SparseOpaqueFactorization_Complex_Float`

A semi-opaque type representing a matrix factorization in complex float.

Sparse Matrix Factor Functions

Compute the factorization of a matrix.

Sparse Direct Solving Functions (Matrix RHS)

Solve a system with a right-hand-side dense matrix using a factored sparse coefficient matrix.

Sparse Direct Solving Functions (Vector RHS)

Solve a system with a right-hand-side dense vector using a factored sparse coefficient matrix.

Sparse Symbolic Factorization Functions

Calculate the symbolic factorization of a matrix, and solve systems using precalculated symbolic factorizations.

Sparse Refactor Functions

Recompute a factorization using the numerical data from a matrix.

Subfactor Functions

Extract and work with subfactors.

Solving systems with iterative sparse methods

Solving systems using iterative methods

Use iterative methods to solve systems of equations where the coefficient matrix is sparse.

Sparse Iterative Solving Functions (Matrix RHS)

Solve a system with a right-hand-side dense matrix using iterative methods.

Sparse Iterative Solving Functions (Vector RHS)

Solve a system with a right-hand-side dense vector using iterative methods.

Sparse Iterate Functions

Perform a single iteration of the specified iterative method.

Sparse Iterative Methods

Select a suitable iterative method to solve a system.

Preconditioners

Create preconditioners for iterative solves.

Multiplying and transposing sparse matrices

Sparse Matrix and Dense Matrix Multiplication

Multiply sparse and dense matrices.

Sparse Matrix and Dense Vector Multiplication

Multiply sparse matrices and dense vectors.

Transposition

Transpose matrices, factorizations, and subfactors.

Retaining and releasing resources

Memory Management

Retain and release sparse objects.

Macros

`SPARSE_CHECK_CONSISTENT_DS_MAT_IN_PLACE`

`SPARSE_CHECK_CONSISTENT_DS_MAT_OUT_PLACE`

`SPARSE_CHECK_CONSISTENT_DS_VEC_IN_PLACE`

`SPARSE_CHECK_CONSISTENT_DS_VEC_OUT_PLACE`

`SPARSE_CHECK_CONSISTENT_MAT_OUT_PLACE`

`SPARSE_CHECK_MATCH_SYMB_FACTOR`

`SPARSE_CHECK_VALID_MATRIX_STRUCTURE`

`SPARSE_CHECK_VALID_NUMERIC_FACTOR`

`SPARSE_CHECK_VALID_SYMBOLIC_FACTOR`

`SPARSE_CLOSED_ENUM`

`SPARSE_ENUM`

`SPARSE_PARAMETER_CHECK`

`SPARSE_PUBLIC_INTERFACE`

`SPARSE_INCLUDED_VIA_ACCELERATE`

See Also

Sparse Matrices

Creating sparse matrices

Create sparse matrices for factorization and solving systems.

Solving systems using direct methods

Use direct methods to solve systems of equations where the coefficient matrix is sparse.

Solving systems using iterative methods

Use iterative methods to solve systems of equations where the coefficient matrix is sparse.

Creating a sparse matrix from coordinate format arrays

Use separate coordinate format arrays to create sparse matrices.