Sample Code

Solving systems of linear equations with LAPACK

Select the optimal LAPACK routine to solve a system of linear equations.

Download

macOS 13.3+ | Xcode 15.0+

Overview

The Accelerate framework provides the LAPACK library for numerical linear algebra. A basic technique of linear algebra is to solve systems of simultaneous equations. For example, the following shows three equations that contain the unknowns *x*, *y*, and *z*:

$$\begin{aligned} x + 2y + 3z &= 70 \\ 4x + 5y + 6z &= 160 \\ 7x + 8y + 9z &= 250 \end{aligned}$$

You can solve this system by rewriting the simultaneous equations as a matrix equation with the following form:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 70 \\ 160 \\ 250 \end{bmatrix}$$

This form is an *Ax* = *b* form, where *A* is the coefficient matrix, *x* is a column vector that contains the unknown values, and *b* is a column vector that contains the constant values. The number of elements in *x* is equal to the number of columns of *A*, and the number of elements in *b* is equal to the number of rows of *A*.

The process of solving this system computes the values for *x*, *y*, and *z* as −2, 24, and 8, respectively.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \cdot \begin{bmatrix} -2 \\ 24 \\ 8 \end{bmatrix} = \begin{bmatrix} 70 \\ 160 \\ 250 \end{bmatrix} \Leftrightarrow \begin{array}{r} -2 + (2 \times 24) + (3 \times 8) = 70 \\ (4 \times -2) + (5 \times 24) + (6 \times 8) = 160 \\ (7 \times -2) + (8 \times 24) + (9 \times 8) = 250 \end{array}$$

For an example of solving a linear system, see Finding an interpolating polynomial using the Vandermonde method.

LAPACK includes routines for solving systems of linear equations as *Ax* = *b*. This sample code project includes wrapper functions that simplify calling the LAPACK routines, for example, by encapsulating multiple-step workflows into a single function call.

Run the sample code app to see the results of each routine solve different example systems.

Determine the properties of the coefficient matrix

LAPACK provides different solving routines depending on the properties of the coefficient matrix, *A*:

- Is the coefficient matrix *symmetric*? A symmetric matrix is one that's equal to its transpose, that is, a matrix that's identical when swapping its row and column indices. A symmetric matrix is necessarily square. The following is an example of a symmetric matrix:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \\ 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$

- Is the coefficient matrix *positive definite*? A matrix is positive definite if all of its eigenvalues are positive. Confirm whether a matrix is positive definite by calling `spotrf_(_:_:_:_:_:)` to try a Cholesky factorization. If the factorization fails and returns a positive value, the matrix isn't positive definite. This sample code project includes the function `isPositiveDefinite(_:dimension:)` to determine whether a matrix is positive definite.

- Is the coefficient matrix *banded*? A banded matrix has all of its nonzero entries on its main diagonal and an arbitrary number of superdiagonals (above the main diagonal) and subdiagonals (below the main diagonal). The following is an example of a nonsymmetric, banded matrix with two superdiagonals and one subdiagonal:

$$\begin{bmatrix} 1 & 20 & 30 & 0 & 0 \\ 30 & 2 & 21 & 31 & 0 \\ 0 & 31 & 3 & 22 & 32 \\ 0 & 0 & 32 & 4 & 23 \\ 0 & 0 & 0 & 33 & 5 \end{bmatrix}$$

- Is the coefficient matrix *tridiagonal*? A tridiagonal matrix has all of its nonzero entries on its main diagonal, its first superdiagonal, and its first subdiagonal. The following is an example of a nonsymmetric, tridiagonal matrix:

$$\begin{bmatrix} 1 & 20 & 0 & 0 & 0 \\ 30 & 2 & 21 & 0 & 0 \\ 0 & 31 & 3 & 22 & 0 \\ 0 & 0 & 32 & 4 & 23 \\ 0 & 0 & 0 & 33 & 5 \end{bmatrix}$$

If the coefficient matrix is *sparse*, that is, most of the entries in the coefficient matrix are zero, Accelerate provides the Sparse Solvers library to help solve such systems.

Select LAPACK variants for data types

The LAPACK routines in this sample code project are all for real, single-precision matrices. All of the routines are available in single- and double-precision for real and complex values. The first character of a routine name defines the type of data the routine works on. For example:

- `sgels_(_:_:_:_:_:_:_:_:_:_:_:)` — single-precision, real values

- `dgels_(_:_:_:_:_:_:_:_:_:_:)` — double-precision, real values
- `cgels_(_:_:_:_:_:_:_:_:_:_:)` — single-precision, complex values
- `zgels_(_:_:_:_:_:_:_:_:_:_:)` — double-precision, complex values

For complex matrices, the LAPACK routine variant for real symmetric matrices requires Hermitian matrices. For example, the `cptsv_()` routine computes the solution to *Ax = b* for a complex single-precision, Hermitian, tridiagonal coefficient matrix; and `sptsv_()` computes the solution for a real single-precision, symmetric, tridiagonal coefficient matrix.

The routines in this sample code project are suitable for solving full rank systems, that is, they have a unique and exact solution.

Define values in column-major layout

The LAPACK routines in this article require the matrix data in column-major layout, which means specifying all the terms in the first column, then all of the terms in the second column, the third column, and so on. For example, if there are two columns with three row values each, the routine specifies the three row values for column one, then the three row values for column two, as the following example illustrates:

$$\begin{bmatrix} 80 & 800 \\ 180 & 1800 \\ 160 & 1600 \end{bmatrix}$$

```
let bValues: [Float] = [80, 180, 160,
                        800, 1800, 1600]
```

The routines return the result as column-major, for example, an array that contains `[10.0, 20.0, 30.0, 100.0, 200.0, 300.0]` represents the following matrix:

$$\begin{bmatrix} 10 & 100 \\ 20 & 200 \\ 30 & 300 \end{bmatrix}$$

Select the solving routine for the coefficient matrix type

This sample code project provides Swift wrapper functions to each single-precision LAPACK solving routine. Select the routine that most closely matches the coefficient matrix for the highest performance. The following shows the Swift wrapper functions and the underlying LAPACK routines to solve systems with different coefficient matrices:

- Symmetric
    - Positive definite
        - Tridiagonal
            - Swift wrapper function: `symmetric_positiveDefinite_tridiagonal()`
            - Underlying LAPACK routine: `sptsv_()`
        - Other banded
            - Swift wrapper function: `symmetric_positiveDefinite_banded()`
            - Underlying LAPACK routine: `spbsv_()`
        - General
            - Swift wrapper function: `symmetric_positiveDefinite_general()`
            - Underlying LAPACK routine: `sposv_()`
    - Indefinite
        - General
            - Swift wrapper function: `symmetric_indefinite_general()`
            - Underlying LAPACK routine: `ssysv_()`
- Nonsymmetric
    - Square
        - Tridiagonal
            - Swift wrapper function: nonsymmetric_tridiagonal()
            - Underlying LAPACK routine: `sgtsv_()`
        - Other banded
            - Swift wrapper function: `nonsymmetric_banded()`
            - Underlying LAPACK routine: `sgbsv_()`
        - General
            - Swift wrapper function: `nonsymmetric_general()`

        - Underlying LAPACK routine: `sgesv_()`
  - Nonsquare
      - QR factorization
          - Swift wrapper function: `nonsymmetric_nonsquare()`
          - Underlying LAPACK routine: `sgels_()`
      - Cholesky factorization
          - Swift wrapper function: `leastSquares_nonsquare()`
          - Underlying LAPACK routines: `sposv_()` and `ssysv_()`

Solve for a nonsquare matrix using QR factorization

A system of linear equations with a nonsquare coefficient matrix is either:

- Overdetermined — there are more equations than unknowns, that is, the coefficient matrix has more rows than columns. In this case, the system may not have a solution.
- Underdetermined — there are more unknowns than equations, that is, the coefficient matrix has more columns than rows. In this case, the system may have infinitely many solutions.

In these cases, the solution is either not exact (unless the overdetermined system is actually consistent) or not unique. In the case where LAPACK is unable to solve the system, the Swift wrapper functions return `nil`.

The Swift wrapper function `nonsymmetric_nonsquare(a:dimension:b:rightHandSideCount:)` wraps the LAPACK routine `sgels_(_:_:_:_:_:_:_:_:_:_:_:)`. This routine takes one of two approaches, depending on the system:

- When the coefficient matrix, *A*, has more rows than columns (overdetermined), the routine minimizes the error in *Ax - b* by solving the least squares problem $\| b-Ax \|_2$. The following image shows the graph of an overdetermined system with two unknowns and three equations. `nonsymmetric_nonsquare(a:dimension:b:rightHandSideCount:)` returns `[1.4615387, 0.7692307, −1.1766968]`, indicating the *x* in *Ax=b* equals `[1.4615387, 0.7692307]`, and the sum of the residuals squared (that is, $r0^2 + r1^2 + r2^2$ equals -1.1766968^2). Selecting any other point in the triangle of the three intercepts yields a larger sum of residuals squared.

- When the coefficient matrix, *A*, has more columns than rows (underdetermined), the routine finds the smallest *x* that solves the equation *min* $\| x \|_2$ such that *Ax = b*. The following image shows the graph of *y=x+1*, which is the set of solutions to the illustrated system. The closest point on the line to the origin is at x = -0.5, y = 0.5.

The `sgels_(_:_:_:_:_:_:_:_:_:_:)` routine uses QR factorization for overdetermined systems, and LQ factorization for underdetermined systems.

The following is an example of an underdetermined system with a coefficient matrix that’s nonsquare:

The following code calls `nonsymmetric_nonsquare(a:dimension:b:rightHandSideCount:)` to compute the values of *x*:

```
let aValues: [Float] = [1, 6, 11,
                        2, 7, 12,
                        3, 8, 13,
                        4, 9, 14,
                        5, 10, 15]

let dimension = (m: 3, n: 5)

/// The _b_ in _Ax = b_.
let bValues: [Float] = [355, 930, 1505]

/// Call `nonsymmetric_nonsquare` to compute the _x_ in _Ax = b_.
let x = nonsymmetric_nonsquare(a: aValues,
                               dimension: dimension,
                               b: bValues,
                               rightHandSideCount: 1)

/// Calculate _b_ using the computed _x_.
if let x = x {
    let b = matrixVectorMultiply(matrix: aValues,
                                 dimension: dimension,
                                 vector: x)

    /// Prints _b_ in _Ax = b_ using the computed _x_: `~[355, 930, 1505]`.
    print("\nnonsymmetric_nonsquare: ([355, 930, 1505]) b =", b)
}
```

Solve for a nonsquare matrix using Cholesky factorization

Where speed is more important than numerical accuracy, the sample code project provides an alternative to `sgels_(_:_:_:_:_:_:_:_:_:_:)`. The `leastSquares_nonsquare(a:dimension:b:)` function exploits the fact that the *x* in $A^TAx = A^Tb$ equals the *x* in $Ax = b$. This technique creates the square coefficient matrix A^TA and solves with either `symmetric_positiveDefinite_general(a:dimension:b:rightHandSideCount:)` or `symmetric_indefinite_general(a:dimension:b:rightHandSideCount:)`.

The `leastSquares_nonsquare(a:dimension:b:)` function uses the same problem as `nonsymmetric_nonsquare(a:dimension:b:rightHandSideCount:)`, but uses Cholesky factorization when A^TA is positive definite.

The following is an example of an overdetermined system with a coefficient matrix that's nonsquare:

The following code calls `leastSquares_nonsquare(a:dimension:b:)` to compute the values of *x*:

```swift
let aValues: [Float] = [1, 4, 7, 10,
                        2, 5, 8, 11,
                        3, 6, 9, 12]
let dimension = (m: 4, n: 3)
let bValues: [Float] = [194, 455, 716, 977]

/// Call `leastSquares_nonsquare` to compute the _x_ in _Ax = b_.
let x = leastSquares_nonsquare(a: aValues,
                               dimension: dimension,
                               b: bValues)

/// Calculate _b_ using the computed _x_.
if let x = x {
    let b = matrixVectorMultiply(matrix: aValues,
                                 dimension: dimension,
                                 vector: Array(x[0..<3]))

    /// Prints _b_ in _Ax = b_ using the computed _x_: `~[194, 455, 716, 977]`.
    print("\nleastSquares_nonsquare: b =", b)
}
```

Solve for a rank-deficient matrix

Systems with a symmetric matrix that's not full rank, *rank-deficient matrices*, don't have a single unique solution. For example, the following two multiplications contain different *x* matrices, but yield the same result in *b*:

In this case, passing matrix *A* to its most suitable function, `symmetric_indefinite_general(a:dimension:b:rightHandSideCount:)`, returns an error indicating that the routine can't compute the solution.

One option to deal with rank-deficiency is to instead solve a nearby problem of full rank by adding a small epsilon value to the matrix to regularize it. The following code adds such an epsilon to diagonal elements in matrix *A*:

```
var aValues: [Float] = [1, 2, 1,
                        2, 1, 2,
                        1, 2, 1]


let dimension = 3
let epsilon = sqrt(Float.ulpOfOne)
for i in 0 ..< dimension {
    aValues[i * dimension + i] += epsilon
}


let bValues: [Float] = [80, 100, 80]


/// Call `symmetric_indefinite_general` to compute the _x_ in _Ax = b_.
let x = symmetric_indefinite_general(a: aValues,
                                     dimension: dimension,
                                     b: bValues,
                                     rightHandSideCount: 1)


/// Calculate _b_ using the computed _x_.
if let x = x {
    let b = matrixVectorMultiply(matrix: aValues,
                                 dimension: (m: dimension, n: dimension),
                                 vector: x)


    /// Prints _b_ in _Ax = b_ using the computed _x_: `~[80, 100, 80]`.
    print("\nRank-Deficient: b =", b)
}
```

On return, *x* contains the values `[0.0, 20.0, 40.0]`:

See Also

Linear Algebra

Finding an interpolating polynomial using the Vandermonde method

Use LAPACK to solve a linear system and find an interpolating polynomial to construct new points between a series of known data points.

Compressing an image using linear algebra

Reduce the storage size of an image using singular value decomposition (SVD).

BLAS

Perform common linear algebra operations with Apple’s implementation of the Basic Linear Algebra Subprograms (BLAS).