Article

Finding an interpolating polynomial using the Vandermonde method

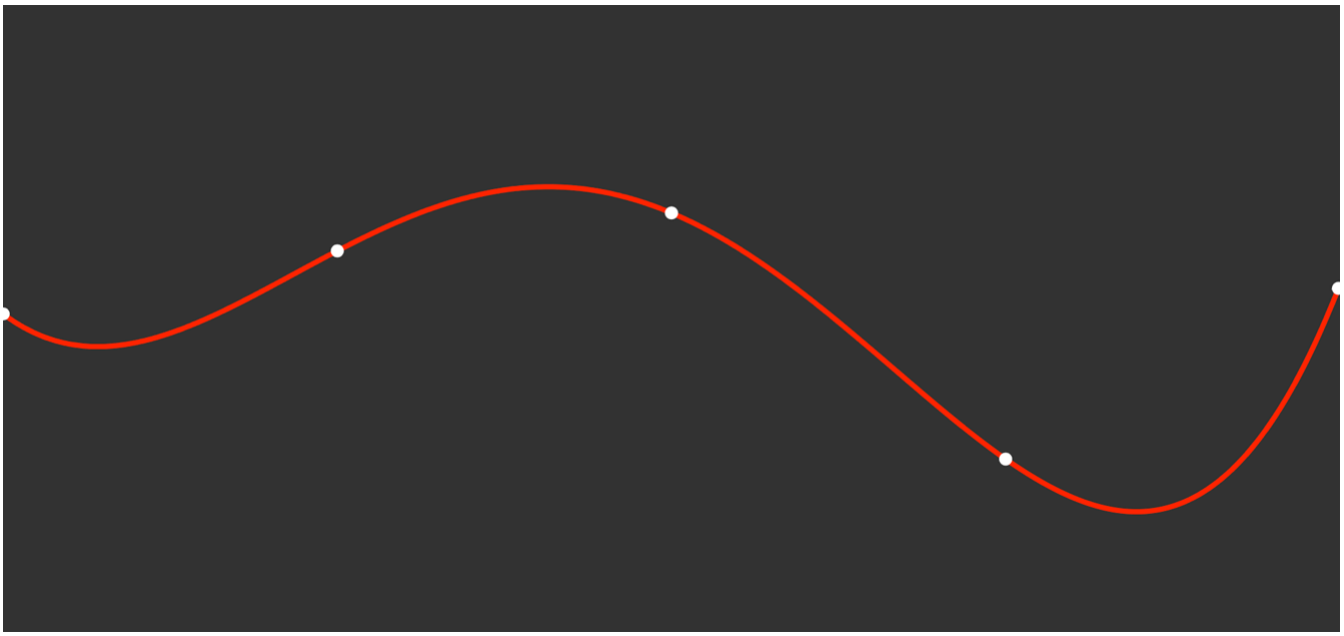
Use LAPACK to solve a linear system and find an interpolating polynomial to construct new points between a series of known data points.

Overview

This article demonstrates how you can generate a continuous curve that passes through a small set of points by computing an *interpolating polynomial*. A polynomial is the sum of a series of terms constructed from variables, coefficients, and exponents (for example, $6x^3 + 7x^2$, where *6* and *7* and the coefficients, and *x* is the variable); and an interpolating polynomial fills in the gaps between the supplied variables and coefficients.

For any number of data points, there is a unique interpolating polynomial of order (that is, the largest exponent) which is the number of data points minus one. However, for large numbers of data points, this solution can become numerically unstable.

The image below shows five known points, as white dots, and the values generated by evaluating the found interpolating polynomial, as a red line:



The code in this article determines the polynomial coefficients using a Vandermonde matrix based on the x-components of the known points. The coefficients are the solution to *Ax=b*, where *A* is the Vandermonde matrix and *b* is a vector of the y-components of the known points. You'll use LAPACK to solve *Ax=b*. LAPACK is an acronym for Linear Algebra Package and is a standard software library for numerical linear algebra.

Generate known data

Create an array containing five two-element vectors that describe the known data points between which the code interpolates.

In a real-world app, you will most likely acquire data points from an external source such as a meteorological or financial data source. For this example, specify x-components that are evenly distributed between 0 and 1023, and generate random y-components:

```
import simd

let points: [simd_double2] = [
    simd_double2(0,    Double.random(in: -10...10)),
    simd_double2(256,  Double.random(in: -10...10)),
    simd_double2(512,  Double.random(in: -10...10)),
    simd_double2(768,  Double.random(in: -10...10)),
    simd_double2(1023, Double.random(in: -10...10)),
    ]
```

Create a Vandermonde matrix

Construct a Vandermonde matrix where the rows are defined by the elements in a source vector that are successively raised to each integer power up to the source vector's element count, minus one. For example, in the case of a five-element source vector, *x*, the Vandermonde matrix is of the form:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & x_0^3 & x_0^4 \\ 1 & x_1 & x_1^2 & x_1^3 & x_1^4 \\ 1 & x_2 & x_2^2 & x_2^3 & x_2^4 \\ 1 & x_3 & x_3^2 & x_3^3 & x_3^4 \\ 1 & x_4 & x_4^2 & x_4^3 & x_4^4 \end{pmatrix}$$

The Vandermonde matrix used in this article derives from the x-components of the points you're interpolating. For example, given the following points:

$$\begin{pmatrix} 0 & 10 \\ 1 & -2 \\ 2 & 3 \\ 3 & -10 \\ 4 & 16 \end{pmatrix}$$

The resulting Vandermonde matrix contains the following values:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & 3 & 9 & 27 & 81 \\ 1 & 4 & 16 & 64 & 256 \end{pmatrix}$$

The following code constructs a Vandermonde matrix from the `points` array:

```
import Accelerate

let exponents = (0 ..< points.count).map {
    return Double($0)
}

let vandermonde: [[Double]] = points.map { point in
    let bases = [Double](repeating: point.x,
                         count: points.count)
    return vForce.pow(bases: bases,
                      exponents: exponents)
}
```

Calculate coefficients

The coefficients for the polynomial are the solution to *Ax=b*, where *A* is the Vandermonde matrix and *b* is the y-components of the known points. For example, using the matrix created in Finding an interpolating polynomial using the Vandermonde method, the coefficients are the *x* in the following:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & 3 & 9 & 27 & 81 \\ 1 & 4 & 16 & 64 & 256 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 10 \\ -2 \\ 3 \\ -10 \\ 16 \end{pmatrix}$$

Create the function `solveLinearSystem(a:a_rowCount:a_columnCount:b:b_count:)` to encapsulate the LAPACK routines that solve *Ax=b*. Note that LAPACK overwrites *b* with the solution vector, *x*:

```
let coefficients: [Double] = {
    var a = vandermonde.flatMap { $0 }
    var b = points.map { $0.y }
```

```
    do {
        try ViewController.solveLinearSystem(matrixA: &a,
                                              matrixB: &b,
                                              count: points.count)
    } catch {
        fatalError("Unable to solve linear system.")
    }

    vDSP.reverse(&b)

    return b
}()
```

On return, `coefficients` contains the polynomial coefficients.

Use LAPACK to solve a linear system

> **Important**
>
> This sample uses the LAPACK library under the Accelerate framework that's in line with LAPACK version 3.9.1. Specify `ACCELERATE_NEW_LAPACK=1` and `ACCELERATE_LAPACK_ILP64=1` as preprocessor macros in Xcode build settings.

Use the LAPACK `dgels` routine to perform the solve. The `dgels` name derives from **d**ouble-precision, **ge**neral-matrix, **l**east-**s**quares.

```
static func solveLinearSystem(matrixA: inout [Double],
                              matrixB: inout [Double],
                              count: Int) throws {

    /// By default, LAPACK expects matrices in column-major format. Specify transpos
    /// the row-major Vandermonde matrix.
    let trans = Int8("T".utf8.first!)

    /// Pass `-1` to the `lwork` parameter of `dgels_` to calculate the optimal size
    /// workspace array. The function writes the optimal size to the `workDimension`
    var workspaceCount = Double(0)
    let err = dgels(transpose: trans,
                    rowCount: count,
                    columnCount: count,
                    rightHandSideCount: 1,
```

```
                    matrixA: &matrixA, leadingDimensionA: count,
                    matrixB: &matrixB, leadingDimensionB: count,
                    workspace: &workspaceCount,
                    workspaceCount: -1)

    if err != 0 {
        throw LAPACKError.internalError
    }

    ///  Create the workspace array based on the workspace query result.
    var workspace = UnsafeMutablePointer<Double>.allocate(
        capacity: Int(workspaceCount))
    defer {
        workspace.deallocate()
    }

    /// Perform the solve by passing the workspace array size to the `lwork` paramet
    let info = dgels(transpose: trans,
                     rowCount: count,
                     columnCount: count,
                     rightHandSideCount: 1,
                     matrixA: &matrixA, leadingDimensionA: count,
                     matrixB: &matrixB, leadingDimensionB: count,
                     workspace: workspace,
                     workspaceCount: Int(workspaceCount))

    if info < 0 {
        throw LAPACKError.parameterHasIllegalValue(parameterIndex: abs(Int(info)))
    } else if info > 0 {
        throw LAPACKError.diagonalElementOfTriangularFactorIsZero(index: Int(info))
    }
}

public enum LAPACKError: Swift.Error {
    case internalError
    case parameterHasIllegalValue(parameterIndex: Int)
    case diagonalElementOfTriangularFactorIsZero(index: Int)
}
```

This example calls the `dgels` function that's a wrapper around the underlying LAPACK function `dgels_(_:_:_:_:_:_:_:_:_:_:_:)`. The wrapper provides a more Swift-friendly way of calling the LAPACK function.

```
/// A wrapper around `dgels_` that accepts values rather than pointers to values.
static func dgels(transpose trans: CChar,
                  rowCount m: Int,
                  columnCount n: Int,
                  rightHandSideCount nrhs: Int,
                  matrixA a:  UnsafeMutablePointer<Double>,
                  leadingDimensionA lda: Int,
                  matrixB b:  UnsafeMutablePointer<Double>,
                  leadingDimensionB ldb: Int,
                  workspace work:  UnsafeMutablePointer<Double>,
                  workspaceCount lwork: Int) -> Int32 {

    var info = Int32(0)

    withUnsafePointer(to: trans) { trans in
        withUnsafePointer(to: __LAPACK_int(m)) { m in
            withUnsafePointer(to: __LAPACK_int(n)) { n in
                withUnsafePointer(to: __LAPACK_int(nrhs)) { nrhs in
                    withUnsafePointer(to: __LAPACK_int(lda)) { lda in
                        withUnsafePointer(to: __LAPACK_int(ldb)) { ldb in
                            withUnsafePointer(to: __LAPACK_int(lwork)) { lwork in
                                dgels_(trans, m, n,
                                       nrhs,
                                       a, lda,
                                       b, ldb,
                                       work, lwork,
                                       &info)
                            }
                        }
                    }
                }
            }
        }
    }

    return info
}
```

Evaluate the polynomial

The vDSP `evaluatePolynomial(usingCoefficients:withVariables:)` function evaluates a polynomial. For example, the following code evaluates a simple polynomial that

consists of three variables and three coefficients:

```
let coefficients: [Float] = [5, 6, 7]
let variables: [Float] = [1, 2, 3]

var c = [Float](repeating: .nan,
                count: variables.count)

let result = vDSP.evaluatePolynomial(usingCoefficients: coefficients,
                                     withVariables: variables)
```

On return, `result` contains `[18.0, 39.0, 70.0]` by performing the following:

$$
\begin{aligned}
(5 \times 1^2) + (6 \times 1^1) + (7 \times 1^0) &= 5 + 6 + 7 &= 18 \\
(5 \times 2^2) + (6 \times 2^1) + (7 \times 2^0) &= 20 + 12 + 7 &= 39 \\
(5 \times 3^2) + (6 \times 3^1) + (7 \times 3^0) &= 45 + 18 + 7 &= 70
\end{aligned}
$$

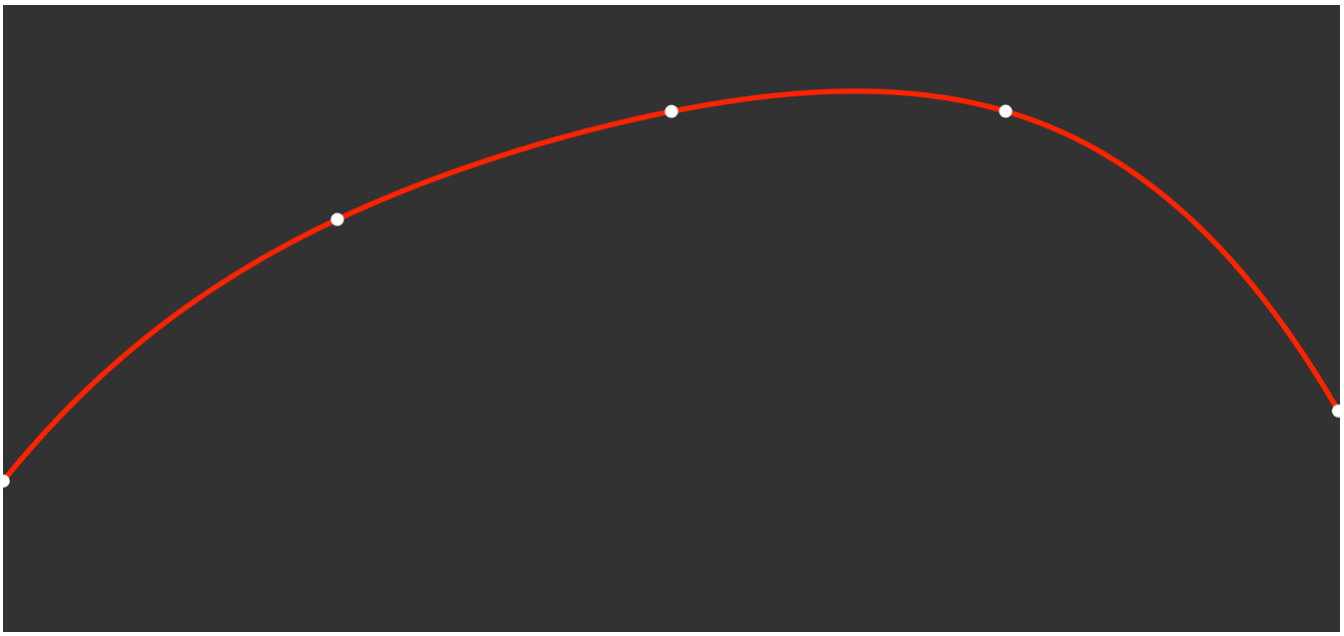
Note that the number of elements returned by the polynomial evaluation is the same as the number of elements in the `variables` array.

To create an interpolation result that contains 1024 elements, use `ramp(in:count:)` to create the variables:

```
let ramp = vDSP.ramp(withInitialValue: 0,
                     increment: Double(1),
                     count: 1024)

let polynomialResult = vDSP.evaluatePolynomial(usingCoefficients: coefficients,
                                               withVariables: ramp)
```

On return, `polynomialResult` contains 1024 elements with the indices corresponding to the x-components, and the values corresponding to the interpolated y-components:



See Also

Linear Algebra

{} Solving systems of linear equations with LAPACK

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

{} 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).