

# **SSJ User's Guide**

Package `functionfit`

Function fit utilities

Version: September 29, 2015

This package provides basic facilities for curve fitting and interpolation with polynomials as, for example, least square fit and spline interpolation.

## Contents

PolInterp . . . . .	2
LeastSquares . . . . .	3
BSpline . . . . .	5
SmoothingCubicSpline . . . . .	7

# PolInterp

Represents a polynomial that interpolates through a set of points. More specifically, let  $(x_0, y_0), \dots, (x_n, y_n)$  be a set of points and  $p(x)$  the constructed polynomial of degree  $n$ . Then, for  $i = 0, \dots, n$ ,  $p(x_i) = y_i$ .

---

```
package umontreal.iro.lecuyer.functionfit;
public class PolInterp extends Polynomial implements Serializable
```

## Constructors

```
public PolInterp (double[] x, double[] y)
    Constructs a new polynomial interpolating through the given points  $(x[0], y[0]), \dots, (x[n], y[n])$ . This constructs a polynomial of degree  $n$  from  $n+1$  points.
```

## Methods

```
public static double[] getCoefficients (double[] x, double[] y)
    Computes and returns the coefficients the polynomial interpolating through the given points  $(x[0], y[0]), \dots, (x[n], y[n])$ . This polynomial has degree  $n$  and there are  $n+1$  coefficients.

public double[] getX()
    Returns the  $x$  coordinates of the interpolated points.

public double[] getY()
    Returns the  $y$  coordinates of the interpolated points.

public static String toString (double[] x, double[] y)
    Makes a string representation of a set of points.

public String toString()
    Calls toString (double[], double[]) with the associated points.
```

# LeastSquares

This class implements different *linear regression* models, using the least squares method to estimate the regression coefficients. Given input data  $x_{ij}$  and response  $y_i$ , one want to find the coefficients  $\beta_j$  that minimize the residuals of the form (using matrix notation)

$$r = \min_{\beta} \|Y - X\beta\|_2,$$

where the  $L_2$  norm is used. Particular cases are

$$r = \min_{\beta} \sum_i \left( y_i - \beta_0 - \sum_{j=1}^k \beta_j x_{ij} \right)^2.$$

for  $k$  regressor variables  $x_j$ . The well-known case of the single variable  $x$  is

$$r = \min_{\alpha, \beta} \sum_i (y_i - \alpha - \beta x_i)^2.$$

Sometimes, one wants to use a basis of general functions  $\psi_j(t)$  with a minimization of the form

$$r = \min_{\beta} \sum_i \left( y_i - \sum_{j=1}^k \beta_j \psi_j(t_i) \right)^2.$$

For example, we could have  $\psi_j(t) = e^{-\lambda_j t}$  or some other functions. In that case, one has to choose the points  $t_i$  at which to compute the basis functions, and use a method below with  $x_{ij} = \psi_j(t_i)$ .

```
package umontreal.iro.lecuyer.functionfit;
public class LeastSquares
```

## Methods

```
public static double[] calcCoefficients (double[] X, double[] Y)
```

Computes the regression coefficients using the least squares method. This is a simple linear regression with 2 regression coefficients,  $\alpha$  and  $\beta$ . The model is

$$y = \alpha + \beta x.$$

Given the  $n$  data points  $(X_i, Y_i)$ ,  $i = 0, 1, \dots, (n - 1)$ , the method computes and returns the array  $[\alpha, \beta]$ .

```
public static double[] calcCoefficients (double[] X, double[] Y, int deg)
```

Computes the regression coefficients using the least squares method. This is a linear regression with a polynomial of degree  $\text{deg} = k$  and  $k + 1$  regression coefficients  $\beta_j$ . The model is

$$y = \beta_0 + \sum_{j=1}^k \beta_j x^j.$$

Given the  $n$  data points  $(X_i, Y_i)$ ,  $i = 0, 1, \dots, (n - 1)$ , the method computes and returns the array  $[\beta_0, \beta_1, \dots, \beta_k]$ . Restriction:  $n > k$ .

```
public static double[] calcCoefficients0 (double[] [] X, double[] Y)
```

Computes the regression coefficients using the least squares method. This is a model for multiple linear regression. There are  $k+1$  regression coefficients  $\beta_j$ , and  $k$  regressors variables  $x_j$ . The model is

$$y = \beta_0 + \sum_{j=1}^k \beta_j x_j.$$

There are  $n$  data points  $Y_i, X_{ij}$ ,  $i = 0, 1, \dots, (n - 1)$ , and each  $X_i$  is a  $k$ -dimensional point. Given the response  $Y[i]$  and the regressor variables  $X[i][j]$ ,  $i = 0, 1, \dots, (n - 1)$ ,  $j = 0, 1, \dots, (k - 1)$ , the method computes and returns the array  $[\beta_0, \beta_1, \dots, \beta_k]$ . Restriction:  $n > k + 1$ .

```
public static double[] calcCoefficients (double[] [] X, double[] Y)
```

Computes the regression coefficients using the least squares method. This is a model for multiple linear regression. There are  $k$  regression coefficients  $\beta_j$ ,  $j = 0, 1, \dots, (k - 1)$  and  $k$  regressors variables  $x_j$ . The model is

$$y = \sum_{j=0}^{k-1} \beta_j x_j.$$

There are  $n$  data points  $Y_i, X_{ij}$ ,  $i = 0, 1, \dots, (n - 1)$ , and each  $X_i$  is a  $k$ -dimensional point. Given the response  $Y[i]$  and the regressor variables  $X[i][j]$ ,  $i = 0, 1, \dots, (n - 1)$ ,  $j = 0, 1, \dots, (k - 1)$ , the method computes and returns the array  $[\beta_0, \beta_1, \dots, \beta_{k-1}]$ . Restriction:  $n > k$ .

## BSpline

Represents a B-spline with control points at  $(X_i, Y_i)$ . Let  $\mathbf{P}_i = (X_i, Y_i)$ , for  $i = 0, \dots, n-1$ , be a *control point* and let  $t_j$ , for  $j = 0, \dots, m-1$  be a *knot*. A B-spline [1] of degree  $p = m - n - 1$  is a parametric curve defined as

$$\mathbf{P}(t) = \sum_{i=0}^{n-1} N_{i,p}(t) \mathbf{P}_i, \text{ for } t_p \leq t \leq t_{m-p-1}.$$

Here,

$$N_{i,p}(t) = \frac{t - t_i}{t_{i+p} - t_i} N_{i,p-1}(t) + \frac{t_{i+p+1} - t}{t_{i+p+1} - t_{i+1}} N_{i+1,p-1}(t)$$

$$N_{i,0}(t) = \begin{cases} 1 & \text{for } t_i \leq t \leq t_{i+1}, \\ 0 & \text{elsewhere.} \end{cases}$$

This class provides methods to evaluate  $\mathbf{P}(t) = (X(t), Y(t))$  at any value of  $t$ , for a B-spline of any degree  $p \geq 1$ . Note that the `evaluate` method of this class can be slow, since it uses a root finder to determine the value of  $t^*$  for which  $X(t^*) = x$  before it computes  $Y(t^*)$ .

```
package umontreal.iro.lecuyer.functionfit;
public class BSpline implements MathFunction
```

### Constructors

```
public BSpline (final double[] x, final double[] y, final int degree)
    Constructs a new uniform B-spline of degree degree with control points at  $(x[i], y[i])$ .
    The knots of the resulting B-spline are set uniformly from  $x[0]$  to  $x[n-1]$ .

public BSpline (final double[] x, final double[] y, final double[] knots)
    Constructs a new uniform B-spline with control points at  $(x[i], y[i])$ , and knot vector
    given by the array knots.
```

### Methods

```
public double[] getX()
    Returns the  $X_i$  coordinates for this spline.

public double[] getY()
    Returns the  $Y_i$  coordinates for this spline.

public double getMaxKnot()
    Returns the knot maximal value.
```

```
public double getMinKnot()
```

Returns the knot minimal value.

```
public double[] getKnots()
```

Returns an array containing the knot vector  $(t_0, t_{m-1})$ .

```
public static BSpline createInterpBSpline (double[] x, double[] y,
                                           int degree)
```

Returns a B-spline curve of degree `degree` interpolating the  $(x_i, y_i)$  points [1]. This method uses the uniformly spaced method for interpolating points with a B-spline curve, and a uniformed clamped knot vector, as described in <http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/>.

```
public static BSpline createApproxBSpline (double[] x, double[] y,
                                           int degree, int h)
```

Returns a B-spline curve of degree `degree` smoothing  $(x_i, y_i)$ , for  $i = 0, \dots, n$  points. The precision depends on the parameter  $h$ :  $1 \leq \text{degree} \leq h < n$ , which represents the number of control points used by the new B-spline curve, minimizing the quadratic error

$$L = \sum_{i=0}^n \left( \frac{Y_i - S_i(X_i)}{W_i} \right)^2.$$

This method uses the uniformly spaced method for interpolating points with a B-spline curve and a uniformed clamped knot vector, as described in <http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/>.

```
public BSpline derivativeBSpline()
```

Returns the derivative B-spline object of the current variable. Using this function and the returned object, instead of the `derivative` method, is strongly recommended if one wants to compute many derivative values.

```
public BSpline derivativeBSpline (int i)
```

Returns the  $i$ th derivative B-spline object of the current variable;  $i$  must be less than the degree of the original B-spline. Using this function and the returned object, instead of the `derivative` method, is strongly recommended if one wants to compute many derivative values.

## SmoothingCubicSpline

Represents a cubic spline with nodes at  $(x_i, y_i)$  computed with the smoothing cubic spline algorithm of Schoenberg [1, 2]. A smoothing cubic spline is made of  $n + 1$  cubic polynomials. The  $i$ th polynomial of such a spline, for  $i = 1, \dots, n - 1$ , is defined as  $S_i(x)$  while the complete spline is defined as

$$S(x) = S_i(x), \quad \text{for } x \in [x_{i-1}, x_i].$$

For  $x < x_0$  and  $x > x_{n-1}$ , the spline is not precisely defined, but this class performs extrapolation by using  $S_0$  and  $S_n$  linear polynomials. The algorithm which calculates the smoothing spline is a generalization of the algorithm for an interpolating spline.  $S_i$  is linked to  $S_{i+1}$  at  $x_{i+1}$  and keeps continuity properties for first and second derivatives at this point, therefore  $S_i(x_{i+1}) = S_{i+1}(x_{i+1})$ ,  $S'_i(x_{i+1}) = S'_{i+1}(x_{i+1})$  and  $S''_i(x_{i+1}) = S''_{i+1}(x_{i+1})$ .

The spline is computed with a smoothing parameter  $\rho \in [0, 1]$  which represents its accuracy with respect to the initial  $(x_i, y_i)$  nodes. The smoothing spline minimizes

$$L = \rho \sum_{i=0}^{n-1} w_i (y_i - S_i(x_i))^2 + (1 - \rho) \int_{x_0}^{x_{n-1}} (S''(x))^2 dx$$

In fact, by setting  $\rho = 1$ , we obtain the interpolating spline; and we obtain a linear function by setting  $\rho = 0$ . The weights  $w_i > 0$ , which default to 1, can be used to change the contribution of each point in the error term. A large value  $w_i$  will give a large weight to the  $i$ th point, so the spline will pass closer to it. Here is a small example that uses smoothing splines:

```
int n;
double[] X = new double[n];
double[] Y = new double[n];
// here, fill arrays X and Y with n data points (x_i, y_i)
// The points must be sorted with respect to x_i.

double rho = 0.1;
SmoothingCubicSpline fit = new SmoothingCubicSpline(X, Y, rho);

int m = 40;
double[] Xp = new double[m+1]; // Xp, Yp are spline points
double[] Yp = new double[m+1];
double h = (X[n-1] - X[0]) / m; // step

for (int i = 0; i <= m; i++) {
    double z = X[0] + i * h;
    Xp[i] = z;
    Yp[i] = fit.evaluate(z); // evaluate spline at z
}
```

---



```

package umontreal.iro.lecuyer.functionfit;
import umontreal.iro.lecuyer.functions.*;
import umontreal.iro.lecuyer.functions.Polynomial;

public class SmoothingCubicSpline implements MathFunction,
        MathFunctionWithFirstDerivative, MathFunctionWithDerivative,
        MathFunctionWithIntegral

```

## Constructors

```

public SmoothingCubicSpline (double[] x, double[] y, double[] w,
                             double rho)

```

Constructs a spline with nodes at  $(x_i, y_i)$ , with weights  $w_i$  and smoothing factor  $\rho = \text{rho}$ . The  $x_i$  must be sorted in increasing order.

```

public SmoothingCubicSpline (double[] x, double[] y, double rho)

```

Constructs a spline with nodes at  $(x_i, y_i)$ , with weights = 1 and smoothing factor  $\rho = \text{rho}$ . The  $x_i$  must be sorted in increasing order.

## Methods

```

public double evaluate (double z)

```

Evaluates and returns the value of the spline at  $z$ .

```

public double integral (double a, double b)

```

Evaluates and returns the value of the integral of the spline from  $a$  to  $b$ .

```

public double derivative (double z)

```

Evaluates and returns the value of the *first* derivative of the spline at  $z$ .

```

public double derivative (double z, int n)

```

Evaluates and returns the value of the  $n$ -th derivative of the spline at  $z$ .

```

public double[] getX()

```

Returns the  $x_i$  coordinates for this spline.

```

public double[] getY()

```

Returns the  $y_i$  coordinates for this spline.

```

public double[] getWeights()

```

Returns the weights of the points.

```

public double getRho()

```

Returns the smoothing factor used to construct the spline.

```

public Polynomial[] getSplinePolynomials()

```

Returns a table containing all fitting polynomials.

```
public int getFitPolynomialIndex (double x)
```

Returns the index of  $P$ , the `Polynomial` instance used to evaluate  $x$ , in an `ArrayList` table instance returned by `getSplinePolynomials()`. This index  $k$  gives also the interval in table **X** which contains the value  $x$  (i.e. such that  $x_k < x \leq x_{k+1}$ ).

## References

- [1] C. de Boor. *A Practical Guide to Splines*. Number 27 in Applied Mathematical Sciences Series. Springer-Verlag, New York, 1978.
- [2] D. S. G. Pollock. Smoothing with cubic splines. Technical report, University of London, Queen Mary and Westfield College, London, 1993.