Class library `lip` for multivariate scattered data interpolation. Version 1.2.

Gleb Beliakov
School of Information Technology, Deakin University
221 Burwood Hwy, Burwood, 3125, Australia
gleb@deakin.edu.au

July 17, 2004

Abstract

This report describes a method of multivariate scattered data interpolation and the programming library `lip` which implements this method. We assume that the data set is generated by a Lipschitz continuous function f, with the Lipschitz constant M. This method will interpolate the data with a Lipschitz function g, which is the best approximation to f in the worst case scenario. It provides best uniform error bounds, and thus translates into reliable learning of f.

The class library `lip` is written in `C++` language. It provides an interface to various methods of calculating g. The choice of the method depends on the dimension of the data, the size of the data set and on how many evaluations of the interpolant are required. The usage of `lip` and its numerical performance are illustrated on several examples.

Key words: Scattered data interpolation, Lipschitz approximation, Uniform approximation, Constrained interpolation, Multivariate approximation.

Contents

1 Introduction

This report describes the programming library `lip`, which implements a method of reliable multivariate interpolation of scattered data. The underlying interpolation method assumes that the interpolated function is Lipschitz continuous, and relies on the knowledge (or estimate) of its Lipschitz constant. The Lipschitz constant is easily interpreted as the upper bound on the rate of change of the function, but the function need not be differentiable. The computed interpolant has a distinctive feature of providing the best approximation to the unknown function that generated the data in the worst case scenario.

The Lipschitz interpolant possesses a number of desirable features, such as continuous dependence on the data, preservation of Lipschitz properties and of the range of the data, uniform approximation and best error bounds. On the practical side, construction and evaluation of the interpolant is computationally stable. There are two alternative ways to compute the interpolant: fast and explicit. The fast method involves a preprocessing step after which the speed of evaluation is proportional to the logarithm of the size of the data set. For higher dimensions, the preprocessing step becomes too expensive, and limited RAM may prevent efficient storage and use of the data structures. In this case there is the second alternative to use the explicit evaluation method, which does not require preprocessing. The availability of two alternative evaluation methods is a distinctive feature of the presented approach.

The described interpolation method is highly competitive to the alternative approaches in terms of efficiency and accuracy, and works irrespectively of the dimension or distribution of data points.

Section 2 describes the scattered data interpolation problem. In section 3 we outline the theory behind the Lipschitz interpolation method. Section 4 presents distinctive features of Lipschitz interpolant, and compares it to some alternative methods. The description of the class library `lip` which implements this method is given in section 5 together with the examples of its usage. Section 6 analyses the performance of the algorithms, and their applicability. Appendices provide outline of the algorithms and some implementation details.

2 Description of the interpolation problem

Throughout this report m will denote the dimensionality of the space, and K will denote the size of the data set. We are given a data set representing values of an unknown function f

x_1	x_2	x_3	x_4	y
x_1^1	x_2^1	x_3^1	x_4^1	y^1
x_1^2	x_2^2	x_3^2	x_4^2	y^2
x_1^3	x_2^3	x_3^3	x_4^3	y^3
$\vdots$				
x_1^K	x_2^K	x_3^K	x_4^K	y^K

There is no special structure in the data set, i.e., the data are *scattered*. We assume that the data set was generated by an unknown function f which satisfies Lipschitz condition with the Lipschitz constant M:

$$|f(x) - f(z)| \leq Md(x, z),$$

for all x and z, where $d(x, z)$ is a distance function. We look for an interpolant $g \approx f$, such that

$$g(x^k) = y^k, k = 1, \ldots, K,$$

which provides the best uniform approximation to f in the worst case scenario, i.e., g minimizes the maximal possible error at any x

$$\max_f \max_{x \in X} |f(x) - g(x)|.$$

We will give an explicit formula for such an interpolant in section 3.

Besides theoretical guarantees on the accuracy of approximation, we look for a number of practical features.

- Efficient construction of the interpolant: we would like the algorithm to be numerically stable and fast. However we can sacrifice the speed of construction in favor of a faster evaluation algorithm.

- Efficient evaluation of the interpolant: ideally we would like the algorithm to perform a very limited number of operations, of order of the logarithm of the number of data points K, as this number can be very large.

- Generality: we would like to have the same generic algorithm for any dimension $m > 1$.

- Local interpolation scheme: Ideally we would like the interpolant $g(x)$ to depend only on a few data values closest to x and distributed all around x.

3 Theoretical background

3.1 The problem of reliable interpolation

We are given a data set

$$\{(x^k, y^k)\}_{k=1}^K, x^k \in R^m, y^k \in R,$$

where y^k are the values of some function $f(x^k) = y^k$, which is unknown to us and which we want to approximate with g, $g \approx f$. Thus we look for an interpolant $g : R^m \to R$, such that

$$g(x^k) = y^k, k = 1, \dots, K.$$

It is known (e.g., see [10]) that it is impossible to give finite bounds on the values $f(x)$, $x \neq x^k, k = 1, \dots, K$ in terms of the data set, if the only additional information is that f is the element of a linear space $\mathcal{V}$, no matter how restricted the space $\mathcal{V}$ is in terms of conditions of continuity, smoothness, analyticity, etc. There exist many interpolation schemata that provide $O(h^2)$, $O(h^4)$, etc. order of accuracy, where h is the distance from x to its neighbours from the data set. One should always keep in mind that big-O notation involves a factor depending on the maximum value of the second or higher derivatives of the function f, which are unknown to us, and which in principle can be infinitely large. Therefore a high order of accuracy of the interpolation scheme does not guarantee small approximation error. In fact, the errors can be infinitely large, regardless how smooth f is.

We shall consider the space of continuous functions with the supremum norm, i.e., $\mathcal{V} = C(X), X \subset R^m$. We shall assume that f is bounded and Lipschitz continuous,

$$\forall x, z \in X \subset R^m, \exists M > 0 : |f(x) - f(z)| \leq M d(x, z); \tag{1}$$

the smallest such number M is called the Lipschitz constant of f in the norm $||\centerdot||$. We denote the class of such functions $Lip(M)$.

Lipschitz condition (1) is not very restrictive, and it is easy to interpret in terms of the problem in hand, as opposed, say, to higher order derivatives. It is simply the upper bound on the rate of change of function f. No differentiability of f is required.

Our goal is to find an interpolant g which approximates f well at the points x distinct from the data, given that $f \in Lip(M)$. We are interested in reliable approximation of f, which means that we want to obtain a good approximation regardless of how inconvenient f is, even in the worst case scenario. That is, we solve the following problem.

Interpolation Problem

Given the data set as above, find the best interpolating function $g : R^m \to R$,

$$g = \arg\inf\{\max_{f \in Lip(M)} ||f - g||_{C(X)},\} \tag{2}$$

such that

$$g(x^k) = f(x^k) = y^k, k = 1, \dots, K.$$

Observation 1 *We can use any norm or any convex distance function in the Lipschitz condition (1), as all such distances are equivalent. In particular we will use simplicial distance function, defined in the Appendix A.*

3.2 Solution to the interpolation problem

The described method relies on building tight upper and lower approximations to f, denoted by H^{upper} and H^{lower}. From Lipschitz condition (1) it follows that

$$\forall x, x^k \in X : \ |f(x) - y^k| \le Md(x, x^k),$$

where $d(x, y)$ is a distance function, which yields tight bounds

$$\max_k (y^k - Md(x, x^k)) \le f(x) \le \min_k (y^k + Md(x, x^k)).$$

Denote the upper and lower bounds by

$$\begin{aligned} H^{upper}(x) &= \min_k (y^k + Md(x, x^k)), \\ H^{lower}(x) &= \max_k (y^k - Md(x, x^k)). \end{aligned} \tag{3}$$

Theorem 1 *[6] Let H^{upper} and H^{lower} be tight upper and lower approximations to f defined by (3), and let*

$$g(x) = \frac{1}{2}(H^{lower}(x) + H^{upper}(x)), \forall x \in X.$$

Then g is the solution to the Interpolation Problem (2) over the set of all continuous functions $X \to R$ that interpolate the data, i.e.,

$$g = \arg\min_{h} \max_{f \in Lip(M)} ||f - h||_{C(X)},$$

and the best error bound is

$$\max_{f \in Lip(M)} ||f - g||_{C(X)} = M \max_{x \in X} \min_{k=1,\ldots,K} d(x, x^k).$$

3.3 Evaluation of the interpolant

Equations (3) provide the way to evaluate the interpolant $g(x)$ at any x. There is no need for any preprocessing, and the number of basic arithmetic operations is proportional to the size of the data set K. We will call this approach `explicit` computation of g.

In some applications the explicit evaluation of g can be too slow. Below we describe an alternative approach, which we call `fast` evaluation. It requires a preprocessing step, whose complexity depends on the number of data points and dimensionality of the space. However, evaluation is much faster than in the explicit method, and requires of order $O(\log K)$ arithmetical operations.

We will describe this method for the lower interpolant H^{lower}, as it can be trivially generalised for the upper interpolant by inverting the sign of all data values: consider function $\bar{f} = -f$; find the lower approximation to $\bar{f}$, call it $\bar{H}^{lower}$; then set $H^{upper} = -\bar{H}^{lower}$. We will also consider a special simplicial distance $d(x, y)$ defined in the Appendix A,

$$d(x, y) = \max_{i=1,\ldots,m+1} C_i(y_i - x_i), \tag{4}$$

where positive constants C_i are related to the Lipschitz constant M ($C_i = M$), and $x_{m+1} = 1 - \sum_{i=1}^{m} x_i$ is a slack variable. We take $n = m + 1$.

In this case we write

$$\begin{aligned} H^{lower}(x) &= \max_{k=1,\dots,K} \min_{i=1,\dots,n} (y^k - C_i(x_i^k - x_i)), \\ H^{upper}(x) &= \min_{k=1,\dots,K} \max_{i=1,\dots,n} (y^k + C_i(x_i^k - x_i)). \end{aligned} \tag{5}$$

Next we will identify the local minima of H^{lower}, and organise them in a tree structure. By introducing the *support vectors* $l^k \in R^n, k = 1, \dots, K$:

$$l_i^k = \frac{y^k}{C_i} - x_i^k, \tag{6}$$

we represent H^{lower} as

$$H^{lower}(x) = \max_{k=1,\dots,K} \min_{i=1,\dots,n} C_i(l_i^k + x_i), \tag{7}$$

Now we can represent all local minimisers of H^{lower} as ordered combinations of $n = m + 1$ support vectors $L = \{l^{k_1}, l^{k_2}, \dots, l^{k_n}\}$ [3, 4, 5]. It is helpful to visualize this combination as a matrix L whose rows are the participating support vectors l^{k_i}:

$$L = \begin{pmatrix} l_1^{k_1} & l_2^{k_1} & \dots & l_n^{k_1} \\ l_1^{k_2} & l_2^{k_2} & \dots & l_n^{k_2} \\ \vdots & \vdots & \ddots & \vdots \\ l_1^{k_n} & l_2^{k_n} & \dots & l_n^{k_n} \end{pmatrix}, \tag{8}$$

so that its components are given by $L_{ij} = \frac{y^{k_i}}{C_j} - x_j^{k_i}$.

Theorem 2 *[5]. Let the support vectors $l^k, k = 1, \dots, K$ be defined using (6). Let x^* denote a local minimizer of $H^{lower}(x)$ and $d = H^K(x^*)$. Then matrix (8) corresponding to x^* enjoys the following properties:*
1) $\forall i, j \in \{1, \dots, n\}, i \neq j : l_i^{k_j} > l_i^{k_i}$;
2) $\forall r \notin \{k_1, k_2, \dots, k_n\}\ \exists i \in \{1, \dots, n\} : L_{ii} = l_i^{k_i} \geq l_i^r$;
3) $d = \frac{Trace(L)+1}{C^-}$, and
4) $x_i^* = \frac{d}{C_i} - l_i^{k_i}$,
where $C^- = \sum_{i \in I} \frac{1}{C_i}$.

Conditions 1)-2) of the Theorem 2 are easily interpreted. Condition 1) implies that the diagonal elements of matrix L are *dominated* by their respective columns, and condition 2) implies that no support vector l^r (which is not part of L) strictly dominates the diagonal of L. Thus we can simply test all combinations of n different support vectors for conditions 1) and 2) of Theorem 2, and those that do satisfy both conditions will provide us with the value and coordinates of the local minimum in 3)-4).

We have shown in [3, 4, 5] that if we start with only n data points (and the unique minimum of H^n) and then add one data point at a time, we can build a tree of local minimisers of a sequence of functions $H^n, H^{n+1}, \ldots, H^K = H^{lower}$. In this tree the nodes correspond to combinations L that satisfy conditions of Theorem 2, and parent and child nodes differ by only one support vector in the i-th position, $i = 1, \ldots, n$. Each node can have up to n children. The leaves of the tree correspond to the minimisers of H^{lower}.

When we add a data point x^k, we can run tests 1)-2) from the root of the tree using depth-first search, and cut off the branches (if any node has satisfied test 2), all the descendants will also satisfy it). This way we can identify the leaves that violate condition 2) very efficiently in $O(\log V)$ time, where V is the number of local minimisers of H^{lower}.

The reason why we determine all local minimisers of H^{lower} and organise them into a tree structure, is that we can use use this tree when we need to determine which of the functions $h^k(x) = \min_{i=1,\ldots,n} C_i(l_i^k + x_i)$ attain the largest values at x, essentially by using a modification of the test 2). Then we can use a small subset of $\{1, 2, \ldots, K\}$ when computing the maximum in (7). We can do this in $O(\log V)$ operations. The number of nodes of this tree grows as $O(K^m)$, but since $\log K^m = m \log K$, the processing of any point x takes $O(\log K)$ operations, and hence we achieve logaritmic evaluation time.

4 Features of the interpolant

4.1 The main properties

The interpolant $g(x)$ implemented in this library provides the best estimate of the unknown function f in the worst case scenario, based on the provided data and its Lipschitz constant M. An estimate of M should be provided by the user, but can also be computed from the data. In many cases an educated guess is enough.

In addition, this method has several useful features.

(1) Preservation of the range of the data: $\min_k\{y^k\} \le g(x) \le \max_k\{y^k\}$.

(2) g approximates f uniformly. The upper bound on the error of approximation is $M \max_x \min_k d(x, x^k)$, i.e., proportional to the distance between the most remote x and its nearest neighbour in the data set. This upper bound provides a guarantee on the quality of approximation regardless the distribution of data points or which particular function $f \in Lip(M)$ generated these data.

(3) For simplicial distance $d(x, x^k)$ the interpolant is a piecewise continuous linear function.

(4) The Lipschitz constant of g with respect to d is M.

(5) The interpolant g depends continuously on the data.

(6) The interpolant g provides a local approximation scheme (i.e., values of g depend only on the nearest data points).

4.2 Comparison with other methods

Multivariate data interpolation and approximation is a very common problem in many branches of science. Sometimes this problem is referred to as regression, estimation, data fitting, learning of functions and other names. There is a great number of techniques developed for various instances of this problem, such as polynomial regression, spline interpolation and smoothing, wavelets, nearest neighbour search, Sibson interpolation, MARS (multivariate adaptive regression splines), machine learning techniques (e.g., decision trees), neural networks, radial basis functions, etc. For an overview the reader is referred to [1, 8, 11].

The method of Lipschitz interpolation described in this report is based on the assumption that the data set was generated by a Lipscitz continuous function f, which is unknown to us. We want to approximate f with a function g, which would have the same Lipschitz properties as f, and would provide the best possible approximation to f at any x, regardless how inconvenient, or bad, f is. That is, we are looking for an interpolant that approximates f reliably in the worst case scenario.

The lack of reliability is a feature of many existing methods of approximation (polynomial interpolation, neural networks). The interpolant may approximate f well at the data points, but it may provide very poor approximation at any other point. Lipschitz interpolation ensures that this does not happen.

Interpolation methods for multivariate scattered data are scarce [1]. The most popular are radial basis functions (RBF), Sheppard's method, triangulation based methods, nearest neighbour and natural neigbour methods. Reference [6] provides a detailed comparison of Lipschitz interpolation and these alternatives. Here we just list their distinctive properties.

Radial basis functions interpolant is a linear combination $g(x) = \sum_k \alpha_k \phi(||x - x^k||)$. The coefficients α_k need to be found from the interpolation conditions, which involves a solution of $K \times K$ linear system of equations. For large K it requires substantial computational effort. Evaluation is performed directly using evaluation of all K functions ϕ. The value depends on all data points.

A special case is Sheppard's method, in which no linear system needs to be solved. Still evaluation is also done directly, and the scheme is global.

Nearest neighbour methods are related to RBF, and employ functions ϕ which take the value 0, except for the top k functions (nearest neighbours). This method can be used with or without solving $K \times K$ linear system. Evaluation is done by computing all K functions ϕ. Another disadvantage is that all nearest neighbours may be on the one side of x, and the scheme for this x is extrapolation rather than interpolation.

Compared to these methods, Lipschitz interpolation is faster ($O(\log K)$ operations), and is equaly fast when using the explicit evaluation, it does not require solving a large linear system, always provides interpolation (in the convex hull of the data points), and is local.

Natural Neighbour interpolation provides an interpolant whose value at x would depend on a few data points close to x at the same time distributed *all around* x. This method is based on the concept of Voronoi diagram (see Appendix). The computation of $g(x)$ requires building of a Voronoi diagram in which x is taken as one of the sites along with the data points. Efficient evaluation methods exist only for $m = 2, 3$.

Lipschitz interpolation is quite similar to the natural neighbour scheme, but it uses a more complicated weighted Voronoi diagram. It has similar computational complexity of the preprocessing step, but allows faster computation for $m > 3$. Local minima of the upper and lower approximations correspond to the vertices of the additively weighted Voronoi diagrams (with the weights taken as $\frac{y^k}{M}$ and $-\frac{y^k}{M}$ respectively). It shares the good features of the natural neigbour interpolation, and in addition allows one to use explicit evaluation of the interpolant, without preprocessing, when the complexity of the Voronoi diagram is too high.

Triangulation based methods require partitioning of the data set into triangles (simplices), and produce piecewise continuous interpolants. Delaunay triangulation is the dual of Voronoi diagram, and hence its construction is equivalent to the preprocessing step in the natural neighbour method and Lipschitz interpolation. However, evaluation step is complicated for $m > 2$, as locating the simplex containing x frequently requires exhaustive search, and the number of such simplices grows as $O(K^m)$. None of the triangulation based methods provides continuous dependence of g on the data.

5 Description of the library

5.1 Programming interface

The methods described in section 3 have been implemented in a class library `lip` in `C++` language. The main classes which provide the interface to the preprocessing and computation are called `STCInterpolant` and `SLipInt`.

```
class STCInterpolant { //saw-tooth cover interpolant
public:
 // supplies the data set to this class.
 // All other methods are called after SetData.
      void SetData(int dim, int K, double* x, double* y, int test=0);
 // Determines an estimate of the Lipschitz constant from the data
      double DetermineLipschitz();
 // Constructs the interpolant for either fast or explicit evaluation
      void Construct();
      void ConstructExplicit();
 // Computes the value of the interpolant
      double Value(int dim, double* x);
      double ValueExplicit(int dim, double* x);
// Sets the value of the Lipschitz constant (or its estimate)
      void SetConstants(double newconst);
...
}

class SLipInt { //simple Lipschitz interpolant
public:
//  Computes the interpolant value by explicit method in l-2,l-Inf norm
 double  ValueInf(int dim,int K,double* x,double* X,double* Y,double LC);
 double  Value2(int dim,int K, double* x, double* X,double* Y,double LC);
// Estimates the Lipschitz constant in l-2 or l-Inf norm
    void    ComputeLipschitzInf(int dim, int K, double* X, double* Y);
    void    ComputeLipschitz2(int dim, int K, double* X, double* Y);
...
 };
```

The class `STCInterpolant` uses simplicial distance, and by using preprocessing method described in section 3, can achieve faster evaluation time for $dim < 5$. The class `SLipInt` uses explicit evaluation when distance d is Euclidean (l_2) or Chebyshev (l_∞) norm. Recommended for higher dimension.

5.2 Members of `STCInterpolant` class.

Public:

`SetData(int dim, int K, double* x, double* y, int test=0)`

This method supplies the data set to be interpolated. The data set is supplied in the variables `x` and `y`. `x` is a two-dimensional array $K \times dim$ that contains values x_i^k in its rows, and `y` is an array of size K which contains the values y^k. K is the size of the data set and dim is the dimension. The last parameter `test` may be omitted, but if it is set to 1, then the data will be screened for repeated x (which slows down the process marginally). The construction algorithm assumes that all data points are unique. If the user is unsure, `test` must be set to 1. The original data in `x`, `y` is not needed after the call to `SetData`. `SetData` **must be called only once before any other method.**

`double DetermineLipschitz()`

This method determines an (under)estimate of the Lipschitz constant based on the data. Bear in mind that `DetermineLipschitz` is a rather slow procedure $O(mK^2)$ and should be avoided if there are other means to estimate the Lipschitz constant. Returns the computed value of the Lipschitz constant.

`SetConstants(double newconst)`

`SetConstants()`

Supplies the Lipschitz constant to the algorithm, either provided by the user, or calculated from the data. It can be called with no parameters after `DetermineLipschitz` only.

`Construct()`

Performs the preprocessing step, necessary for the subsequent evaluation by the fast method. Calls Algorithm 2 described in the Appendix. This method may require significant computational effort. **Should not be called after `ConstructExplicit()`.**

`ConstructExplicit()`

Prepares the data for the subsequent evaluation of the interpolant using explicit evaluation. No significant preprocessing. **Should not be called after `Construct()`.**

`double Value(int dim, double* x )`

Computes and returns the value of the interpolant $g(x)$ using the fast method. Should be called after preprocessing by `Construct`. If called after `ConstructExplicit`, the explicit evaluation routine `ValueExplicit`will be executed instead. Parameter x is an array of size dim. dim should be specified as the dimension of the data m (the algorithm will automatically compute the required slack variable). The user may precompute the slack variable himself (see examples in the next section), and specify dim as $m+1$, which is marginally faster.

`double ValueExplicit(int dim, double* x )`

Computes the value of the interpolant at x using explicit method. Can be called after `Construct` or `ConstructExplicit`. As in `Value`, dim should be specified as the dimension of the data m (the algorithm will automatically compute the required slack variable). The user may precompute the slack variable himself, and specify dim as $m+1$, which is marginally faster. Parameter x is an array of size dim. Returns $g(x)$.

`FreeMemory()`

Frees the memory occupied by the data structures computed in `Construct()`, which can be very large. **It destroys the interpolant, and `Value()` methods cannot be called after `FreeMemory()`.** Automatically called from the destructor. This method is useful to deallocate memory while the object `STCInterpolant` still exists.

`int LastError()`

Returns the error code of the last operation. Useful to check whether the construction or evaluation of the interpolant were successful. 0 indicates a successful operation. Other possible values defined in the `interpol.h` file are: 1 – lower interpolation failed, 2– upper interpolation failed, 3 – both failed, 10 – Lipschitz constant too small, or repeated data.

There are variations of `Value` and `ValueExplicit` methods, which use parameter x of type `dVec`. This type is defined as `TNT::Vector<double>` in `forest.h`, and it uses `TNT` library (included). `TNT` library available from (`http://math.nist.gov/tnt/`) provides a convenient way of coding vectors and matrices, and is used internally by `lip`. Since the user may prefer other ways to code vectors, generic type `double*` is used in this document. See `interpol.h` and examples for more detailed information.

5.3 Members of `SLipInt` class.

Class `SLipInt` implements explicit evaluation of Lipschitz interpolant in l_2 and l_∞-norms. It does not use any special data structure or preprocessing, and completely relies on Eg. (3). The simplicity of the evaluation process and program code is a feature of this class. It can be used for high dimension.

`double ValueInf(int dim,int K,double* x,double* X,double* Y,double LC)`
Computes and returns the value of the interpolant $g(x)$ with $d(x,z) = \max(|x_i - z_i|)$. Does not require any preprocessing. dim is the dimension, K is the number of data, x is the vector of size dim, X is the vector of data of size $K \times dim$ which contains values x_i^k in its rows, y is the vector of size K of values to be interpolated, LC is the Lipschitz constant.

`double ValueInf(int dim,int K,double* x,double* X,double* Y,double*LC)`
Variation of the above, but LC is a vector of size dim of Lipschitz constants with respect to the i-th variable.

`double ValueInf(int dim,int K,double* x,double* X,double* Y)`
Variation of the above, uses Lipschitz constants automaticlly identified from the data. **Should only be called after** `ComputeLipschitzInf()`.

`double Value2(int dim,int K, double* x, double* X,double* Y,double LC);`
Computes and returns the value of the interpolant $g(x)$ with $d(x,z)$ being Euclidean distance. The parameters are the same as in `ValueInf()`.

`double Value2(int dim,int K, double* x, double* X,double* Y);`
Variation of the above, uses Lipschitz constants automaticlly identified from the data. **Should only be called after** `ComputeLipschitz2()`.

`void ComputeLipschitzInf(int dim,int K,double* X,double* Y);`
Computes an (under)estimate of the Lipschitz constant in $l-\infty$-norm from the data. Bear in mind that `ComputeLipschitzInf` is a rather slow procedure $O(nK^2)$ and should be avoided if there are other means to estimate the Lipschitz constant. The value(s) are kept in $LipConst$ member variable.

`void ComputeLipschitz2(int dim,int K,double* X,double* Y);`
Same as above, but uses Euclidean norm.

There are versions of the above methods with `Col` suffix in their names, these are used when the calling routine is written in another language, like Fortran, which stores the data columnwise (not rowwise, like C). The only difference is that the matrix X is transposed.

5.4 Typical usage

There are several examples of the usage of `lip` provided in the distribution. The best way to use `lip` is to declare an instance of the class `STCInterpolant` and call its members directly. There are four basic steps: to supply the data, to supply the Lipschitz constant, to construct the interpolant, and to evaluate it at the desired point.

```
#include "interpol.h"
int dim=4;             // the dimension and size of the data set
int npts=1000;

void main(int argc, char *argv[]){
   double LipConst;
   double *x, *XData, *YData;
// arrays to store the data
   x=(double*)malloc(dim*sizeof(double));
   XData=(double*)malloc(dim*npts*sizeof(double));
   YData=(double*)malloc(npts*sizeof(double));

   STCInterpolant LipInt;
   for(i=0;i<npts;i++) {
     for(j=0;j<dim;j++) // generate random data in [0,3]^m
       XData[i*dim + j]=x[j]=random(3.0,0);
     YData[i]=fun(x); // some function values
    }
// suppy the data and eliminate repeated values
   LipInt.SetData(dim,npts, XData,YData,1);
   LipConst=LipInt.DetermineLipschitz();
   LipInt.SetConstants();  // supply Lipschitz constant
   LipInt.Construct();     // construct the interpolant
   free(XData); free(YData); // may now destroy the data

   double w,s,x1[10];  // reserve space for at least dim+1 components
   for(j=0;j<dim;j++) x1[j]=random(3.0,0); // some random x
   w=LipInt.Value(dim,x1); // calculate the value

// alternatively, pre-compute the slack variable
   for(s=0,j=0; j<dim; j++) s+=x1[j];
   x1[dim]= 1.0-s;
   w=LipInt.Value(dim+1,x1);         // calculate the value
   w=LipInt.ValueExplicit(dim+1,x1); // same using explicit method
   LipInt.FreeMemory();  // destroys the interpolant
}
```

```
// another example using TNT library
#include "interpol.h"

int dim=4;             // the dimension and the data set

int npts=1000; double LipConst=10;

void main(int argc, char *argv[]){
// arrays to store the data
   dVec  x(dim), YData(npts); // these types defined in forest.h
   dMat  XData(npts,dim);
   STCInterpolant LipInt;
   for(i=0;i<npts;i++) {
     for(j=0;j<dim;j++) {     // generate random data in [0,3]^dim
       XData[i][j]=x[j]=random(3.0,0);
     }
     YData[i]=fun(x.begin()); // some function values
   }
// suppy the data
   LipInt.SetData(dim,npts,&(XData[0][0]),YData.begin());
   LipInt.SetConstants(LipConst);  // supply Lipschitz constant
   LipInt.Construct();             // construct the interpolant

   double s,w;
   dVec x1(dim+1);
   for(j=0;j<dim;j++) x[j]=random(3.0,0);  // some random x
   w=LipInt.ValueSlack(x);         // calculate the value of g(x)
   w=LipInt.Value(dim,x.begin());  // the same calculation

// alternatively, pre-compute the slack variable
   for(s=0,j=0; j<dim; j++) { x1[j]=random(3.0,0); s+=x1[j];}
   x1[dim]= 1.0-s;
   w=LipInt.Value(x1);         // calculate the value
   w=LipInt.ValueExplicit(x1); // same using explicit method
}
```

```
// example of usage of SLipInt class
#include "slipint.h"
int dim=4;        // the dimension and size of the data set
int npts=1000;

void main(int argc, char *argv[]){
   double LipConst=4;
   double *x, *XData, *YData;
// arrays to store the data
   x=(double*)malloc(dim*sizeof(double));
   XData=(double*)malloc(dim*npts*sizeof(double));
   YData=(double*)malloc(npts*sizeof(double));

   SlipInt LipInt;
   for(i=0;i<npts;i++) {
     for(j=0;j<dim;j++) // generate random data in [0,3]^m
       XData[i*dim + j]=x[j]=random(3.0,0);
     YData[i]=fun(x); // some function values
    }

   for(j=0;j<dim;j++) x[j]=random(3.0,0); // some random x

// calculate the value
   w=LipInt.ValueInf(dim,npts,x,XData, YData,LipConst);
// estimate Lipschitz constant
   LipInt.ComputeLipschitzInf(dim,npts,XData, YData);
// uses the computed Lipschitz constant
   w=LipInt.ValueInf(dim,npts,x,XData, YData);

// the same in Euclidean norm
   LipInt.ComputeLipschitz2(dim,npts,XData, YData);
// calculate the value
   w=LipInt.ValueInf(dim,npts,x,XData, YData);

   // Overwrite Lipschitz constant and calculate value
   w=LipInt.ValueInf(dim,npts,x,XData, YData,LipConst);

    free(XData); free(YData); free(x);
}
```

6 Performance of the algorithms

The table below illustrates the performance of the `lip` library on test data sets. Measurements were performed on a modest workstation with Pentium III processor (750MHz) and 512 MB of RAM. The table indicates the range of applicability of the algorithms. Exhaustive evaluation does not require preprocessing, but the evaluation time grows as $O(K)$. The fast evaluation method (Algorithm 5) requires preprocessing, but the evaluation time grows as $O(\log K)$. However, for higher dimension exhaustive evaluation may prove to be more efficient. Comparison of the last two columns indicates when the exhaustive evaluation is preferable. Fig.1 presents these data graphically.

The user should be aware of the limitations of `lip` due to hardware constraints. The fast evaluation method requires enumeration of local optima of the lower and upper interpolants, and their number can grow as $O(K^m)$. For $m > 4$ and large K, the method can easily occupy all the available RAM. Therefore it is advisable to estimate memory requirements for a particular problem before calling `Construct` method. Table 1 gives an idea of what are typical memory requirements for $m = 2, \ldots, 5$. The row with the largest K corresponds to approximately 500MB of RAM requested by `lip`.

In case of a larger data set or higher dimension, we advise to use explicit evaluation by calling `ConstructExplicit`, or by using the class `SLipInt`. At the same time we would like to point out that efficient parallelisation of the presented methods for computer clusters is available. This way, combined RAM of several workstations can be used, and data sets of larger dimension can be processed. Contact the author for an implementation of `lip` which uses MPI standard.

We would like to emphasize that proper scaling of the data is important. While none of the algorithms relies on a particular range of the data, there are issues that arise in the implementation of the algorithm, such as finite precision of floating point numbers. Thus we recommend scaling the data abscissae to a reasonable range, like $[0, 100]^m$ or $[-1, 1]^m$, as well as the data values. Scaling will affect the Lipschitz constant, which must be adjusted accordingly.

m	K	preprocessing time(s) `Construct()`	evaluation time ($s \times 10^{-3}$) `Value()`	explicit evaluation ($s \times 10^{-3}$) `ValueExplicit()`
2	1000	0.03	0.10	0.32
	10000	0.48	0.16	3.2
	20000	1.19	0.18	6.3
	40000	2.78	0.22	13.0
	80000	6.09	0.25	26.1
	160000	13.8	0.29	52.1
	320000	31.3	0.40	104.4
	640000	70.6	0.53	208.9
	1280000	152.2	0.68	419.1
3	1000	0.17	0.72	0.38
	10000	2.81	1.20	3.6
	20000	6.67	1.46	7.1
	40000	15.69	1.55	14.3
	80000	35.57	1.61	27.8
	160000	81.7	1.87	56.1
	320000	184.0	2.21	119.8
4	1000	0.78	2.37	0.44
	5000	7.29	4.59	2.04
	10000	18.2	5.89	3.9
	20000	45.3	7.42	8.1
	40000	110.0	9.18	16.1
	80000	245.1	12.2	33.0
5	1000	4.66	15.0	0.48
	5000	54.08	36.8	2.7
	10000	149.7	47.0	5.3

Table 1. Performance of the algorithms from `lip` as a function of the number of data points and dimension. Explicit evaluation refers to directly using Eq.(3) in computations, whose complexity grows linearly with K. For every m, the row with the largest K is when the library has used 500 MB of RAM.

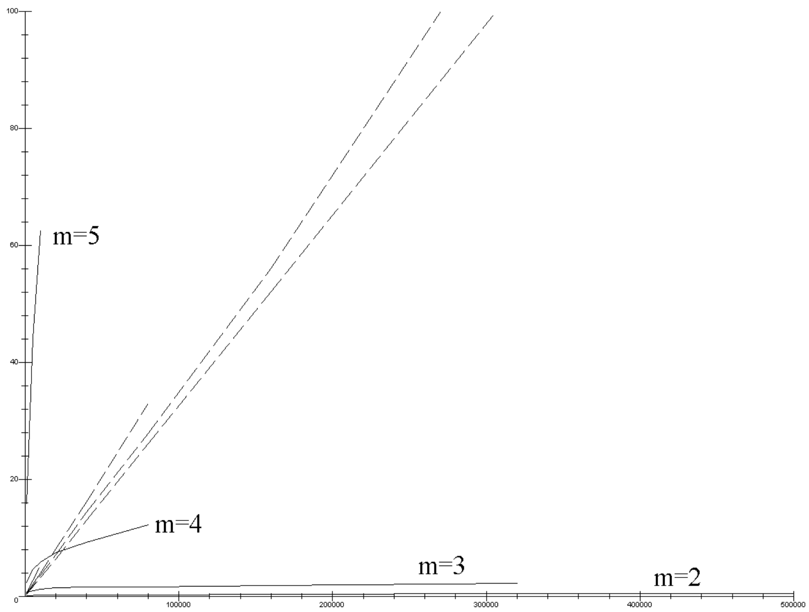

Figure 1: Performance of the evaluation algorithms. Solid lines correspond to fast evaluation, dashed lines correspond to explicit evaluation.

References

[1] P. Alfeld. Scattered data interpolation in three or more variables. In L.L. Schumaker and T. Lyche, editors, *Mathematical Methods in Computer Aided Geometric Design*, pages 1–34. Academic Press, New York, 1989.

[2] F. Aurenhammer. Voronoi diagrams - a survey of a fundamental data structure. *ACM Computing Surveys*, 23:345–405, 1991.

[3] L.M. Batten and G. Beliakov. Fast algorithm for the cutting angle method of global optimization. *Journal of Global Optimization*, 24:149–161, 2002.

[4] G. Beliakov. Geometry and combinatorics of the cutting angle method. *Optimization*, 52:379–394, 2003.

[5] G. Beliakov. Extended cutting angle method of constrained global optimization. In L. Caccetta, editor, *Global Optimization*, pages xx–xx. Kluwer, Dordrecht, 2004.

[6] G. Beliakov. Interpolation of Lipschitz functions. *submitted*, 2004.

[7] J.-D. Boissonnat, M. Sharir, B. Tagansky, and M. Yvinec. Voronoi diagrams in higher dimensions under certain polyhedral distance functions. *Discrete and Comput. Geometry*, 19:485–519, 1998.

[8] V. Cherkassky and F Mulier. *Learning from Data.* Wiley, New York, 1998.

[9] V.F Demyanov and A.M. Rubinov. *Constructive Nonsmooth Analysis.* Peter Lang, Frankfurt am Main, 1995.

[10] M. Golomb and H.F. Weinberger. Optimal approximation and error bounds. In R. E. Langer, editor, *On Numerical Approximation*, pages 117–190. The Univ. of Wisconsin Press,, Madison, 1959.

[11] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning.* Springer-Verlag, New York, Berlin, Heidelberg, 2001.

[12] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams.* John Wiley, Chichester, 2nd edition, 2000.

7 Appendix

A Polyhedral distances

Definition 1 *Polyhedral distance.*

Let P be a finite convex polyhedron in R^m defined by the intersection of r halfspaces, containing the origin in its interior (example 7.2 from [9])

$$P = \bigcap_{i=1}^{r} \{x : x \centerdot h_i \leq 1\}, \tag{9}$$

where $h_i \in R^m$ are the directional vectors. The polyhedral distance is

$$d(x, y) = \max\{(x - y) \centerdot h_i : 1 \leq i \leq r\}.$$

As a special case consider the distance defined by a simplex centered at 0.

Definition 2 *Simplicial distance.*

Let P be a simplex defined as the intersection of $m + 1$ halfspaces (9), defined by vectors

$$\begin{aligned} h_1 &= (-v_1, 0, 0, \ldots), \\ h_2 &= (0, -v_2, 0, \ldots), \\ &\vdots \\ h_{m+1} &= (v_{m+1}, \ldots, v_{m+1}), \end{aligned} \tag{10}$$

$v_i > 0$. The simplicial distance is

$$d(x, y) = \max\{\max_{i=1,\ldots,m} v_i(y_i - x_i), v_{m+1} \sum_{i=1}^{m} (x_i - y_i)\}. \tag{11}$$

Let us now for the purposes of convenience introduce a slack variable $x_{m+1} = 1 - \sum_{i=1}^m x_i$. With the help of the new coordinate, and using $\sum_{i=1}^m (x_i - y_i) = 1 - \sum_{i=1}^m y_i - (1 - \sum_{i=1}^m x_i) = y_{m+1} - x_{m+1}$, we can write (11) in a more symmetric form

$$d(x, y) = \max_{i=1,\ldots,m+1} v_i(y_i - x_i). \tag{12}$$

Observation 2 *Any simplex containing the origin can be transformed into P, which was defined by vectors h_i in (10), by using a linear change of variables. Hence we can effectively use other simplicial distances, defined by other choices of vectors h_i, and convert these distances into the form (12). Therefore, the methods described here are applicable to any simplicial distance.*

Consider a set of points $\{x^k\}_{k=1}^K, x^k \in R^m$, called *sites*.

Definition 3 *The set*

$$Vor(x^k) = \{x \in R^m : ||x - x^k|| \leq ||x - x^j||, \forall j \neq k\}$$

is called the Voronoi cell of x^k.

One can choose any norm, or in fact any distance function d_P in this definition. The collection of Voronoi cells for all sites $x^k, k = 1, \ldots, K$ is called the Voronoi diagram of the data set. Voronoi diagram is one of the most fundamental data structures of a data set with a long history [2, 12, 7].

Definition 4 *Let $\{x^k\}_{k=1}^K, x^k \in R^m$ be the set of sites, and $w \in R^K$ be the vector of weights. The set*

$$Vor(x^k, w) = \{x \in R^m : w_k + ||x - x^k|| \leq w_j + ||x - x^j||, \forall j \neq k\}.$$

is called Additively Weighted Voronoi cell. The collection of such cells is called Additively Weighted Voronoi diagram.

B Algorithms

Explicit evaluation of the interpolant using Eqs.(5) is straightforward and will not be discussed. The fast evaluation method requires building a tree representing local minimisers of H^{lower} (and a similar tree of H^{upper}). This is performed by Algorithm 2.

To construct the interpolant, Algorithm 2 is applied twice, first to represent H^{lower}, and then to represent H^{upper} using a modified data set in which all y^k have inverted sign. As the output, we obtain two trees, T^{lower} and T^{upper}.

Algorithm 2 starts with an empty tree and grows it by adding one data point at a time. Algorithm 2 relies on the recursive Algorithm 1, which tests a given node of the tree for condition 2) of Theorem 2, and when required, the node is split into $m+1$ child nodes. Thus Algorithm 2 repeatedly calls Algorithm 1, which grows the tree.

Algorithm 5 is used co calculate the value of the interpolant using $g(x) = \frac{1}{2}(H^{lower}(x) + H^{upper}(x))$. It calls Algorithm 4 twice (for H^{lower} and H^{upper}). Algorithm 4 relies on Algorithm 3, which computes the reduced subset $U \subset \{1, 2, \ldots, K\}$, such that

$$H^{lower}(x) = \max_{k=1,\ldots,K} \min_{i=1,\ldots,n} C_i(l_i^k + x_i) = \max_{k \in U} \min_{i=1,\ldots,n} C_i(l_i^k + x_i).$$

Algorithm 3 constructs a support vector using the query point x and a value of y adapted to the tested node L. It is a modification of Algorithm 1.

The recursive algorithm used to determine which nodes of the tree of local minima of H^{lower} violate condition 2) of Theorem 2. If the node is a leaf, it is branched into $m+1$ nodes, and the tree grows.

Algorithm 1 (update of the tree T^{K-1})
Purpose: Given the representation of all local minimizers of functions $H^n, H^{n+1}, \ldots, H^{K-1}$ as nodes of a tree, and a new support vector l^K, update the tree structure to include local minimizers of H^K.

Input: The tree T^{K-1} containing local minimizers of $H^n, H^{n+1}, \ldots, H^{K-1}$; the new support vector l^K; tested node L.
Output: The tree T^K.
Note: The trees T^{K-1} and T^K share the same memory and are accessed through the root. Assignment at Step 5 is symbolic.

Step 1 Test L against condition 2) of Theorem 2, with $l^r = l^K$.

Step 2 If test succeeds, go to Step 5 (cut off this branch).

Step 3 If test fails, and L is not a leaf, then
call Algorithm 1 (T^{K-1},l^K,child(L), T^K) for all children of L.
Go to Step 5

Step 4 Otherwise (test failed, and L is a leaf) add n children to L.
Each child node is a copy of L, with l^{k_i} replaced with l^K in the i-th child.
Test condition 1) of Theorem 2 for each child. If test fails, delete this child node.

Step 5 If L is V^n (root), then T^K=T^{K-1}.
Return.

The algorithm for enumerating local minima of H^{lower}. The algorithm starts with an empty tree T^K, and grows this tree by adding one support vector at a time and calling Algorithm 1.

Algorithm 2 (Construction of the tree T^K)
Purpose: Given the dataset $D = \{(x^k, y^k)\}_{k=1}^K$, $x^k \in R^m, y^k \in R$, build the tree of local minima of $H^n, H^{n+1}, \ldots, H^K = H^{lower}$.

Input: The dataset D.
Output: The tree T^K.
Note: We use the assumption that $x_k^k \to \infty, k = 1, \ldots, n$, which would involve adding n auxiliary data points to the data set D, so that the original data are re-labeled $x^{1+n}, x^{2+n}, \ldots, x^{K+n}$.

Step 0 Add auxiliary data to the dataset D.
Preprocess the data set to compute the slack variables $x_n^k, k = 1, \ldots, K$.
Form the basis support vectors $l^1, \ldots, l^n$ using (6).

Step 1 Set $root(T^K) = \{l^1, \ldots, l^n\}$.

Step 2 For $k = 1 + n, \ldots, K + n$ do:

Step 2.1 Form new support vector l^k using (6).

Step 2.2 call Algorithm $1(T^K, l^k, root(T^K), T^K)$

Step 3 Stop.

The following algorithm uses the tree T^{lower} to determine a reduced subset $U \subset \{1, 2, \ldots, K\}$, such that

$$H^{lower}(x) = \max_{k=1,\ldots,K} \min_{i=1,\ldots,n} C_i(l_i^k + x_i) = \max_{k \in U} \min_{i=1,\ldots,n} C_i(l_i^k + x_i).$$

Algorithm 3 (Determination of subset U)
Purpose: Given the tree T^{lower} produced by Algorithm 2, and a query point x, determine a subset $U \subset \{1, 2, \ldots, K\}$, such that

$$H^{lower}(x) = \max_{k \in U} \min_{i=1,\ldots,n} C_i(l_i^k + x_i).$$

Input: The tree T containing local minimizers of $H^n, H^{n+1}, \ldots, H^{lower}$; the query point x; tested node L.
Output: The set U.

Step 1 Form support vector l^r using (6), with x and the maximum of y^{k_i} as y

Step 2 Test $L = \{l^{k_1}, l^{k_2}, \ldots, l^{k_n}\}$ against condition 2) of Theorem 2.

Step 3 If test succeeds, go to Step 6 (cut off this branch).

Step 4 If test fails, and L is not a leaf, then
call Algorithm 3 $(T, x, child(L), U)$ for all children of L.
Go to Step 6

Step 5 Otherwise (test failed, and L is a leaf)
Set $U = U \cup \{k_1, k_2, \ldots, k_n\}$

Step 6 Return.

Algorithm 4 (Evaluation of $H^{lower}(x)$)
Purpose: Given the tree T^{lower} produced by Algorithm 2, and a query point x, determine $H^{lower}(x)$

Input: The tree T containing local minimizers of $H^n, H^{n+1}, \ldots, H^{lower}$; the query point x.
Output: The value $H^{lower}(x)$.

Step 1 Call Algorithm $3(T, x, root(T), U)$;

Step 2 Compute $H^{lower}(x) = \max_{k \in U} \min_{i=1,\ldots,n} C_i(l_i^k + x_i)$;

Step 3 Return.

Algorithm 5 (Evaluation of $g(x)$)
Purpose: Given the trees T^{lower} and T^{upper} produced by Algorithm 2, and a query point x, determine $g(x)$

Input: The trees T^{lower} and T^{upper}; the query point x.
Output: The value $g(x)$.

Step 1 Call Algorithm $4(T^{lower}, x, Lo)$;

Step 2 Call Algorithm $4(T^{upper}, x, Up)$;

Step 3 $g = \frac{1}{2}(Lo - Up)$.

Step 4 Return .

C Implementation notes

The algorithms used in `lip` rely on several standard routines for maintaining arrays, queues, sets, available either in the standard `C++` template library, or in the NIST library `TNT`, which implements such objects as vectors and matrices `http://math.nist.gov/tnt/`. Both libraries are essential for `lip`. However, their use in the user's own code is optional.

`lip` uses the type `real` which is defined in `forest.h` as `double`. This is done to simplify the modification of the library for other data types, such as `float`. If the user wishes to change `double` to `float`, this should be done in `forest.h`. In this case, `double` should be changed to `float` in all the examples.

The main class `STCInterpolant` has two members instances of type `Interpolant`: `m_lower` and `m_upper`. These two members contain the lower and upper interpolants H^{lower} and H^{upper}, and provide methods for building the interpolant and for calculating its value. Since the calculation of H^{upper} is identical to that of H^{lower} up to the sign of function values, the members `m_lower` and `m_upper` are instances of the same class.

The class `Interpolant` has two essential members: `SVectors` - an array of support vectors, and `HeapPossibleMin` - a tree containing combinations of support vectors that correspond to local minimisers of H^{lower}. `Interpolant` implements Algorithms 2 and 4.

`HeapPossibleMin` is an instance of the class `Forest` (we allow for several roots, which is useful for distributed memory architecture. In this implementation we only use one root). `Forest` implements the usual tree management routines, and the Algorithms 1 and 3.

The class `SVSetNode` corresponds to one node of the tree and implements tests of condition 1) and 2) of Theorem 2 and calculation of the value and position of the local minimiser using conditions 3) and 4).

Because the size of the tree can be very large (tens of millions of nodes), efficient memory management is paramount. In `lip` this is achieved by a) an efficient coding of the information contained in one node, and b) by allocating memory for the nodes of the tree in large blocks (to avoid overheads associated with memory management). The size of each node (an instance of `SVSetNode`) is 12 bytes, in which we coded the reference to child nodes, the position and the label of the support vector, which differs this node from the parent, and the maximum value of y^{k_i}, needed for Algorithm 3.

A procedural interface to `lip` is provided in the file `intwrapper.h`. This file declares several functions, that simply pass their arguments to the respective members of one instance of the class `STCInterpolant`. These functions are useful for calls to `lip` library from procedural languages, such as `Fortran`, or other packages, like `Matlab`. An example of usage of `intwrapper.h` is provided.

Since `C` and `Fortran` store matrices using different conventions, two special functions `BuildLipInterpolantColumn` and `BuildLipInterpolantExplicitColumn` are provided to accept the data stored in columns (as in `Fortran`). These two functions do not differ from `BuildLipInterpolant` and `BuildLipInterpolantExplicit`, except they process the data coordinates `x` in column format. Use these functions if you call `lip` from `Fortran` or `matlab`. Both functions call `SetDataColumn` method of `STCInterpolant`, which transposes the matrix of coordinates.

The class `SLipInt` has been added to Version 1.2 for the sake of simplicity of program code. It uses explicit evaluation of the interpolant and requires no preprocessing. It can be used independently of `interpol.h` or any other file in this package. The user can strip `slipint.h` and `slipint.cpp` from the library and use this class on its own.