

Maple toolbox LibLip for multivariate scattered
data interpolation.
Version 2.1.
User Manual

Gleb Beliakov
`gleb@deakin.edu.au`

Copyright Gleb Beliakov, 2006.

Summary

This manual describes methods of multivariate scattered data interpolation and smoothing using Lipschitz functions, including monotonicity and bounds constraints, in Maple environment by using **LibLip** toolbox.

LibLip provides many methods to interpolate scattered data (with or without preprocessing) by using only the data itself and one additional parameter – the Lipschitz constant (which is basically the upper bound on the slope of the function). The Lipschitz constant can be automatically estimated from the data. **LibLip** also provides approximation methods using locally Lipschitz functions.

If the data contains noise, it can be smoothened using special techniques which rely on linear programming. Lipschitz constant can also be estimated from noisy data by using sample splitting and cross-validation.

In addition **LibLip** also accommodates monotonicity and range constraints. It is useful for approximation of functions that are known to be monotone with respect to all or a subset of variables, as well as monotone only on parts of the domain. Range constraints accommodate non-constant bounds on the values of the data and the interpolant.

Please consult <http://www.deakin.edu.au/~gleb> for updates.

License agreement

LibLip is distributed under GNU GENERAL PUBLIC LICENSE. The terms of the license are provided in the file "copying" in the root directory of this distribution.

You can also obtain the GNU License Agreement from <http://www.gnu.org/licenses/licenses.html>

Contents

1	Introduction	5
2	Interpolation problem	7
2.1	Lipschitz interpolation	8
2.2	Construction of the interpolant	9
2.3	Features of the interpolant	9
2.3.1	The main properties	9
2.3.2	Types of approximation problems	10
2.4	Computational methods	11
2.4.1	Interpolation	11
2.4.2	Monotone Interpolation	11
2.4.3	Smoothing	12
2.4.4	Estimation of Lipschitz constant	13
2.4.5	Locally Lipschitz functions	14
2.4.6	Bounds on f	15
3	Description of the functions	17
3.1	Installation	17
3.2	LibLip functions	17
4	Examples of usage	25
4.1	Sample code	25
4.1.1	Interpolation and smoothing	25
4.1.2	Monotone interpolation	27
4.2	Tips	28
4.3	Performance of the algorithms	28
5	More information	31
5.1	Where to get help	31
5.2	References	31

Chapter 1

Introduction

This manual describes the Maple toolbox `LibLip`, which implements a method of reliable multivariate interpolation of scattered data. The underlying interpolation method assumes that the data are generated by a continuous function f (Lipschitz continuous). Given information about the Lipschitz constant, or its estimate, the interpolant is the best possible approximation to the unknown function f in the worst case scenario, also called an optimal interpolant. Thus `LibLip` delivers reliable approximation.

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 is no accumulation of errors with the size of the data set and dimension.

In addition to the Lipschitz constant, the user can provide information about other properties of f , such as monotonicity with respect to any subset of variables, upper and lower bounds (not necessarily constant bounds). If the data are given with errors, then it can be smoothened to satisfy the required properties. The Lipschitz constant, if unknown, can be estimated from the data using sample splitting and cross-validation techniques. The library also provides methods for approximation of locally Lipschitz functions.

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. It is useful for up to 4 variables, and large data sets, and only works with the simplicial distance (see later). For more variables, the preprocessing step becomes too expensive, and limited RAM may prevent efficient storage and use of the data structures.

The second alternative is to use the explicit evaluation method, which does not require any preprocessing. We recommend this method for most applications, as it provides more flexibility with smoothing and incorporating other properties of f .

The implemented interpolation method is highly competitive to the alterna-

tive approaches in terms of efficiency and accuracy, and works irrespectively of the dimension or distribution of data points.

Chapter 2 describes the scattered data interpolation problem and the basics of the theory behind Lipschitz interpolation method. Section 2.3 presents distinctive features of Lipschitz interpolant, and lists several types of interpolation/approximation problems that can be solved using **LibLip**. The description of the Maple toolbox **LibLip** is given in Chapter 3. Examples of its usage are provided in Chapter 4. Section 4.3 analyses the performance of the algorithms, and their applicability.

Note that a more detailed description of the methods implemented in **LibLip** is given in references

Beliakov, G., Interpolation of Lipschitz functions. *J. of Comp. and Applied Mathematics*, 2006. **196**: p. 20-44.

Beliakov, G., Monotonicity preserving approximation of multivariate scattered data. *BIT*, 2005. **45**: p. 653-677.

Beliakov, G., A review of applications of the Cutting Angle methods, in *Continuous Optimization*, A. Rubinov and V. Jeyakumar, Editors. 2005, Springer: New York. p. 209-248.

Please consult <http://www.deakin.edu.au/~gleb> for updates.

Chapter 2

Interpolation problem

Throughout this manual d will denote the dimensionality of the space, and N will denote the size of the data set. We are given a data set representing the 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^N	x_2^N	x_3^N	x_4^N	y^N

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

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

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

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

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 N , as this number can be very large.
- Generality: we would like to have the same generic algorithm for any dimension $d > 1$.
- 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 .
- Additional properties: We would like to incorporate other information about $f(x)$, such as monotonicity or upper and lower bounds.

The interpolation methods implemented in `LibLip` possesses the features mentioned above. In addition it possesses many other useful features, such as preservation of the Lipschitz properties of f and continuous dependence on the data.

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 neighbors 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.

2.1 Lipschitz interpolation

Lipschitz condition is easy to interpret in terms of the problem in hand. 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 is Lipschitz. 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.

Find the best interpolating function $g : R^d \rightarrow R$,

$$g = \arg \inf \left\{ \max_{f \in Lip(M)} \|f - g\|_{C(X)}, \right\} \quad (2.1)$$

such that

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

$Lip(M)$ denotes the class of functions whose Lipschitz constant is smaller or equal to M .

The method used in **LibLip** relies on building tight upper and lower approximations to f , denoted by H^{upper} and H^{lower}

$$\begin{aligned} H^{upper}(x) &= \min_k (y^k + Md(x, x^k)), \\ H^{lower}(x) &= \max_k (y^k - Md(x, x^k)). \end{aligned} \quad (2.2)$$

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 (2.1) over the set of all continuous functions $X \rightarrow R$ 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

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

2.2 Construction of the interpolant

Equations (2.2) 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 N . We will call this approach **explicit** computation of g . The distance $d()$ is either Euclidean (l_2 -norm) or Chebyshev-distance (l_∞ -norm), or any l_p -norm, $p \geq 1$.

In some applications the explicit evaluation of g can be too slow. **LibLip** also implements an alternative approach, called **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 $O(\log N)$ arithmetical operations. The distance $d()$ is the simplicial distance.

2.3 Features of the interpolant

2.3.1 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. It also deals with Locally Lipschitz functions, where Lipschitz constant depends on the location x . This method is more flexible, and is suitable for many functions that change rapidly in one part of the domain and change slowly in the other parts.

In addition, this method has several useful features.

- (1) Preservation of the range of the data: $\min_k \{y^k\} \leq g(x) \leq \max_k \{y^k\}$.
- (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 polyhedral distance $d(x, x^k)$ the interpolant is a piecewise continuous linear function.
- (4) The Lipschitz constant of g with respect to $d(\cdot)$ 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).

2.3.2 Types of approximation problems

LibLip implements various interpolation and approximation methods, which are suitable for the following problems.

- (1) Interpolation problem: Given the data set and the Lipschitz constant M (i.e., the class $Lip(M)$).
- (2) Interpolation problem: No Lipschitz constant is given. The smallest M compatible with the data is calculated.
- (3) Monotone interpolation: Given data set, Lipschitz constant M , and the information that f is monotone increasing (decreasing) with respect to the subset $\mathcal{V} \subset \{1, \dots, d\}$ of arguments.
- (4) Smoothing problem: Given noisy data set, and $Lip(M)$, compute the approximation from $Lip(M)$ which minimizes the norm of the residuals $r^k = \tilde{y}^k - y^k$.
- (5) Smoothing problem, estimation of M : Given noisy data set, estimate the best value of M by using sample splitting or cross-validation, and compute the approximation with this value of M .
- (6) Monotone approximation: Approximation of the data with or without knowledge of M subject to monotonicity with respect to the variables from \mathcal{V} .
- (7) Monotonicity on parts of the domain: Same as monotone approximation and interpolation, but monotonicity holds only for $x \preceq A$ or $x \succeq B$, i.e. in the bottom left or upper right corners of the domain.

- (8) Approximation subject to bounds: interpolation or smoothing, subject to non-constant upper and lower bounds on f .
- (9) Locally Lipschitz functions: The Lipschitz constant varies with the coordinates. All the interpolation problems above with local Lipschitz constants.

2.4 Computational methods

2.4.1 Interpolation

Given a data set $\mathcal{D} = \{(x^k, y^k)\}, k = 1, \dots, N, x^k \in R^d, y \in R$, and the Lipschitz class $Lip(M)$ of functions whose Lipschitz constant is smaller or equal to M , build the optimal interpolant

$$g(x) = \frac{1}{2}(H^{lower}(x) + H^{upper}(x)), \quad (2.3)$$

where the upper and lower bounds are given as

$$\begin{aligned} H^{upper}(x) &= \min_k \{y^k + M\|x - x^k\|\}, \\ H^{lower}(x) &= \max_k \{y^k - M\|x - x^k\|\}. \end{aligned} \quad (2.4)$$

The method of calculation is straightforward. The functions `LipIntValue*`() implement this approach.

If M is unknown, the automatic computation of the smallest Lipschitz constant compatible with the data is performed by solving

$$M = \inf\{C : |y^i - y^j| \leq Cd(x^i, x^j)\}.$$

The function `LipIntComputeLipschitz` implements this.

2.4.2 Monotone Interpolation

Given the data set \mathcal{D} , the class $Lip(M)$ and the knowledge that f is monotone increasing with respect to a subset of variables $\mathcal{V} \subset \{1, \dots, d\}$. This means that f is an increasing function of each variable from \mathcal{V} , as long as the rest of the variables is kept fixed.

Computations are performed by using (2.3), with the bounds given as

$$\begin{aligned} H^{upper}(x) &= \min_k \{y^k + M\|(x - x^k)_{\mathcal{V}+}\|\}, \\ H^{lower}(x) &= \max_k \{y^k - M\|(x^k - x)_{\mathcal{V}+}\|\}, \end{aligned} \quad (2.5)$$

where $z_{\mathcal{V}+}$ denotes the positive part of vector z with respect to subset of components \mathcal{V} : $z_{\mathcal{V}+} = (\bar{z}_1, \dots, \bar{z}_n)$, with

$$\bar{z}_i = \begin{cases} \max\{z_i, 0\}, & \text{if } i \in \mathcal{V}, \\ z_i & \text{otherwise.} \end{cases}$$

Monotone decreasing functions are dealt with in the same way, by simply exchanging the signs of the respective components of x and x^k . Functions decreasing in some arguments and increasing in the other arguments are accommodated. Note that $\|(x - x^k)_{\mathcal{V}^+}\| \neq \|(x^k - x)_{\mathcal{V}^+}\|$.

Functions `LipIntValueCons()` implement this algorithm.

Variation of this problem: the function is known to be monotone but only on the subset $x \preceq A$ or $x \succeq B$, $A, B \in R^d$ and $a \preceq b$ means $\forall i \in \mathcal{V} : a_i \leq b_i$. The bounds are calculated using (2.5), with

$$(a - b)_{\mathcal{V}^+, A} = (\bar{z}_1, \dots, \bar{z}_n),$$

where for monotone increasing functions

$$\bar{z}_i = \begin{cases} \max\{a_i - b_i, \min(0, A_i - b_i)\}, & \text{if } i \in \mathcal{V}, \\ a_i - b_i & \text{otherwise.} \end{cases}$$

and $(a - b)_{\mathcal{V}^+, B} = (\bar{z}_1, \dots, \bar{z}_n)$, with

$$\bar{z}_i = \begin{cases} \max\{a_i - b_i, \min(0, a_i - B_i)\}, & \text{if } i \in \mathcal{V}, \\ a_i - b_i & \text{otherwise.} \end{cases}$$

For monotone decreasing functions we use

$$(a - b)_{\mathcal{V}^+, A} = (\bar{z}_1, \dots, \bar{z}_n),$$

$$\bar{z}_i = \begin{cases} \max\{b_i - a_i, \min(0, A_i - a_i)\}, & \text{if } i \in \mathcal{V}, \\ a_i - b_i & \text{otherwise.} \end{cases}$$

and $(a - b)_{\mathcal{V}^+, B} = (\bar{z}_1, \dots, \bar{z}_n)$, with

$$\bar{z}_i = \begin{cases} \max\{b_i - a_i, \min(0, b_i - B_i)\}, & \text{if } i \in \mathcal{V}, \\ a_i - b_i & \text{otherwise.} \end{cases}$$

These formulae are implemented in the functions `LipIntValueConsLeftRegion()` and `LipIntValueConsRightRegion()` respectively. Some other regions can be converted to the the above cases by exchanging the sign of the respective components of x .

2.4.3 Smoothing

To ensure that the data set \mathcal{D} is compatible with the given class $Lip(M)$, we minimize the norm of the residuals $r_k = \tilde{y}^k - y^k$, (\tilde{y}^k denotes the smoothened data), subject to the Lipschitz conditions

$$\forall i, j \in \{1, \dots, N\} : y^i - y^j \leq M \|(x^i - x^j)\|. \quad (2.6)$$

For monotone functions we use $\|(x^i - x^j)_{\mathcal{V}^+}\|$.

We solve the following optimization problem

$$\begin{aligned} & \min \sum_{k=1}^N w_k |r_k|, \\ \text{s.t.} \quad & r_k - r_j \leq y^j - y^k + M \|(x^k - x^j)\|, \\ & \forall j, k \in \{1, \dots, N\}. \end{aligned} \quad (2.7)$$

It is converted to a linear programming problem by splitting $r_k = r_k^+ - r_k^-$ and using $|r_k| = r_k^+ + r_k^-$. w denotes an optional vector of weights, which reflect relative accuracy of the values y^k .

For numerical efficiency we solve the dual problem to (2.7). We use the simplex method implemented in `glpk` library.

Functions `LipIntSmoothLipschitz()` `LipIntInfSmoothLipschitz()` implement these smoothing techniques and also are used for smoothing when monotonicity is on the parts of the domain $x \preceq A$ and $x \succeq B$.

2.4.4 Estimation of Lipschitz constant

Assume the data set \mathcal{D} is noisy, and the Lipschitz constant M is unknown. We estimate the value of M using sample splitting and cross-validation.

Sample splitting

In sample splitting method, we randomly subdivide \mathcal{D} into nonintersecting subsets \mathcal{D}_1 and \mathcal{D}_2 (parameter *ratio* controls the ratio of data allocated to \mathcal{D}_1). The data set \mathcal{D}_1 is used to approximate the values from \mathcal{D}_2 using (2.3) and (2.4) ((2.5) for monotone functions). We try different values of M , in order to minimize the norm of the difference between the predicted and observed values from the set \mathcal{D}_2 . We solve a bi-level optimization problem

$$\min_{M \geq 0} \frac{1}{4} \sum_{i \in \mathcal{I}_2} \left[\max_{k \in \mathcal{I}_1} (\hat{y}^k(M) - M d_{ki}) + \min_{k \in \mathcal{I}_1} (\hat{y}^k(M) + M d_{ik}) - 2y^i \right]^2, \quad (2.8)$$

where $d_{kj} = \|(x^k - x^j)_{\mathcal{V}^+}\|$. The values $\hat{y}^k(M) = y^k + r_k(M)$ are found as a solution to problem (2.7), which now includes only the data from \mathcal{D}_1 . To minimize wrt M we use the golden section algorithm combined with Fibonacci search.

Once the optimal value of M is found, we smoothen the whole data set \mathcal{D} by solving (2.7).

Method `LipIntComputeLipschitzSplit()` implements this technique.

Cross-validation

In cross-validation, we repeatedly remove one datum from the set \mathcal{D} , and approximate it using the rest $N - 1$ data and a fixed value of M . We repeat this process for all N data, and minimize the norm of the difference between

the predicted and observed values to obtain an optimal M . We solve a bi-level optimization problem

$$\min_{M \geq 0} \frac{1}{4} \sum_{i=1}^N \left[\max_{k \neq i} (\hat{y}^{ki}(M) - Md_{ki}) + \min_{k \neq i} (\hat{y}^{ki}(M) + Md_{ik}) - 2y^i \right]^2. \quad (2.9)$$

At the inner level, for every fixed M we solve N problems for all $i \in \{1, \dots, N\}$

$$\begin{aligned} & \min \sum_{k \neq i} |r_k^i| \\ \text{s.t.} \quad & r_k^i - r_j^i \leq y^j - y^k + Md_{kj}, \\ & \forall k, j \in \{1, \dots, N\} \setminus \{i\}. \end{aligned} \quad (2.10)$$

Once the optimal value of M is found, we smoothen the whole data set \mathcal{D} by solving (2.7). For numerical efficiency, the dual to (2.10) is solved.

Method `LipIntComputeLipschitzCV()` implements this technique. Note that it is computationally expensive for large N . We advise to estimate the running time for small N first.

2.4.5 Locally Lipschitz functions

A function is called locally Lipschitz, if for any x there exist a neighborhood δ_x and a constant $M(\delta_x)$, such that for all $y, z \in \delta_x$,

$$|f(y) - f(z)| \leq M(\delta_x) \|y - z\|.$$

We denote the class of locally Lipschitz functions by $LLip(M)$, and understand that M varies with δ_x .

Using locally Lipschitz functions offers more flexibility, as one can better model the shape of the functions flat in some parts of the domain, and rapidly changing in the other parts.

We will employ the notion of local Lipschitz-constant function

$$\begin{aligned} M_f(x) &= \lim_{\delta \downarrow 0} M(\delta_x) = \inf_{\delta > 0} M(\delta_x), \quad \text{where} \\ \delta_x &= \{z : \|z - x\| < \delta\}, \\ M(\delta_x) &= \sup \left\{ \frac{|f(y) - f(z)|}{\|y - z\|} : y, z \in \delta_x, y \neq z \right\}. \end{aligned}$$

It is shown that in our finite dimensional case

$$M_f(x) = \max_{\|v\|=1} f'(x, v),$$

where $f'(x, v)$ is Clarke's derivative

$$f'(x, v) = \limsup_{\alpha \downarrow 0, y \rightarrow x} \frac{1}{\alpha} (f(y + \alpha v) - f(y)).$$

Let $M(x)$ be a local Lipschitz-constant function, and let $a \in X$ be some fixed point. Define on X the function

$$h_a(x) = \min_{\gamma(a,x)} \int_{\gamma(a,x)} M(r) dr, \quad (2.11)$$

where $\gamma(a, x) \in X$ is any rectifiable contour joining a and x . Then $h_a \in LLip(M)$, and further, the function $h_a(x)$ is the tight upper bound on any locally Lipschitz function with the local Lipschitz-constant function $M(x)$, which satisfies $h(a) = 0$. It follows that the values of any locally Lipschitz function from $LLip(M)$ interpolating $y^k = f(x^k)$ are bounded by

$$y^k - h_{x^k}(x) \leq f(x) \leq y^k + h_{x^k}(x).$$

Interpolating the whole data set \mathcal{D} , we obtain the tight bounds

$$\sigma_l(x) = \max_k \{y^k - h_{x^k}(x)\} \leq f(x) \leq \min_k \{y^k + h_{x^k}(x)\} = \sigma_u(x),$$

and the optimal interpolant is given as earlier by

$$g(x) = \frac{1}{2} \{ \max_k \{y^k - h_{x^k}(x)\} + \min_k \{y^k + h_{x^k}(x)\} \}. \quad (2.12)$$

It is also possible to incorporate monotonicity condition in the equations given above. However such equations are not computationally efficient. The method implemented in **LibLip** uses an approximation to functions h_{x^k} with radial basis functions $\tilde{h}_{x^k}(\|x^k - x\|)$. We use a linear spline to represent \tilde{h}_{x^k} , with the knots at $0, t_1, t_2, \dots, t_{N_k}$. To calculate $\tilde{h}_{x^k}(t_i)$, take all the data within the radius t_i from x^k , and ensure the interpolation conditions hold

$$\forall j \in J : |y^j - y^k| \leq \tilde{h}_{x^k}(\|x^k - x^j\|),$$

where $J = \{j : \|x^j - x^k\| \leq t_i\}$.

The functions `LipIntValueLocal()`, `LipIntValueLocalCons()`, `LipIntValueLocalConsLeftRegion()` and `LipIntValueLocalConsRightRegion()` implement this interpolation scheme. These functions should be called after `LipIntComputeLocalLipschitz()`.

2.4.6 Bounds on f

This toolbox provides a way to specify further bounds on the values of the interpolant $Lo(x) \leq g(x) \leq Up(x)$. These bounds are included into the routines for calculation of the bounds H^{upper} and H^{lower} , as well as in all optimization problems as additional constraints

$$Lo(x^k) \leq y^k + r_k \leq Up(x^k),$$

added to problems such as (2.7).

Computation of these extra bounds could involve sophisticated algorithms, but it is transparent for the `LibLip` toolbox. These bounds frequently arise when one has to ensure that the interpolant takes certain values at a subset of points, or is bounded. For example, when interpolating bivariate data and ensuring that $f(x, 0) = f(0, x) = x, x \in [0, 1]$ and $f(x, y) \geq \max(x, y)$.

The bounds are incorporated into the computations by using `LipIntSetBounds(Up, Low)` function, and cleared by using `LipIntClearBounds()`.

Chapter 3

Description of the functions

3.1 Installation

Installation of LibLip is very simple. Just copy the file `mapleliblip.dll` into your working directory (or any directory on the PATH), and you are ready to use it. The examples in the subsequent sections illustrate the use of LibLip. The example worksheet is also provided.

3.2 LibLip functions

Calling LibLip functions should be done after declaring interface to the methods in LibLip using the commands

```
DLLPATH:="mapleliblip.dll":

# if you need to provide full path, use, e.g., DLLPATH:="c:/work/maple/mapleliblip.dll":

LipIntValue := define_external('MWRAP_LipIntValue', 'MAPLE', LIB=DLLPATH):
LipIntValueAuto := define_external('MWRAP_LipIntValueAuto', 'MAPLE', LIB=DLLPATH):
LipIntValueCons := define_external('MWRAP_LipIntValueCons', 'MAPLE', LIB=DLLPATH):
LipIntValueConsLeftRegion := define_external('MWRAP_LipIntValueConsLeftRegion', 'MAPLE',
LIB=DLLPATH):
LipIntValueConsRightRegion := define_external('MWRAP_LipIntValueConsRightRegion', 'MAPLE',
LIB=DLLPATH):
LipIntValueLocal := define_external('MWRAP_LipIntValueLocal', 'MAPLE', LIB=DLLPATH):
LipIntValueLocalCons := define_external('MWRAP_LipIntValueLocalCons', 'MAPLE', LIB=DLLPATH):
LipIntValueLocalConsLeftRegion := define_external('MWRAP_LipIntValueLocalConsLeftRegion', 'MAPLE',
LIB=DLLPATH):
LipIntValueLocalConsRightRegion := define_external('MWRAP_LipIntValueLocalConsRightRegion', 'MAPLE',
LIB=DLLPATH):
LipIntComputeLipschitz := define_external('MWRAP_LipIntComputeLipschitz', 'MAPLE', LIB=DLLPATH):
LipIntComputeLocalLipschitz := define_external('MWRAP_LipIntComputeLocalLipschitz', 'MAPLE',
LIB=DLLPATH):
LipIntComputeLipschitzCV := define_external('MWRAP_LipIntComputeLipschitzCV', 'MAPLE', LIB=DLLPATH):
LipIntComputeLipschitzSplit := define_external('MWRAP_LipIntComputeLipschitzSplit', 'MAPLE',
LIB=DLLPATH):
LipIntSmoothLipschitz := define_external('MWRAP_LipIntSmoothLipschitz', 'MAPLE', LIB=DLLPATH):
```

```

LipIntGetLipConst :=define_external('MWRAP_LipIntGetLipConst','MAPLE', LIB=DLLPATH):
LipIntGetScaling :=define_external('MWRAP_LipIntGetScaling','MAPLE', LIB=DLLPATH):
LipIntComputeScaling :=define_external('MWRAP_LipIntComputeScaling','MAPLE', LIB=DLLPATH):
LipIntVerifyMonotonicity :=define_external('MWRAP_LipIntVerifyMonotonicity','MAPLE',LIB=DLLPATH):
LipIntVerifyMonotonicityLeftRegion :=define_external('MWRAP_LipIntVerifyMonotonicityLeftRegion',
'MAPLE',LIB=DLLPATH):
LipIntVerifyMonotonicityRightRegion :=define_external('MWRAP_LipIntVerifyMonotonicityRightRegion',
'MAPLE',LIB=DLLPATH):

LipIntInfValue := define_external('MWRAP_LipIntInfValue','MAPLE',LIB=DLLPATH):
LipIntInfValueAuto :=define_external('MWRAP_LipIntInfValueAuto','MAPLE', LIB=DLLPATH):
LipIntInfValueCons :=define_external('MWRAP_LipIntInfValueCons','MAPLE', LIB=DLLPATH):
LipIntInfValueConsLeftRegion :=define_external('MWRAP_LipIntInfValueConsLeftRegion','MAPLE',
LIB=DLLPATH):
LipIntInfValueConsRightRegion :=define_external('MWRAP_LipIntInfValueConsRightRegion','MAPLE',
LIB=DLLPATH):
LipIntInfValueLocal :=define_external('MWRAP_LipIntInfValueLocal','MAPLE', LIB=DLLPATH):
LipIntInfValueLocalCons :=define_external('MWRAP_LipIntInfValueLocalCons','MAPLE',LIB=DLLPATH):
LipIntInfValueLocalConsLeftRegion :=define_external('MWRAP_LipIntInfValueLocalConsLeftRegion',
'MAPLE',LIB=DLLPATH):
LipIntInfValueLocalConsRightRegion :=define_external('MWRAP_LipIntInfValueLocalConsRightRegion',
'MAPLE',LIB=DLLPATH):
LipIntInfComputeLipschitz :=define_external('MWRAP_LipIntInfComputeLipschitz','MAPLE',LIB=DLLPATH):
LipIntInfComputeLocalLipschitz :=define_external('MWRAP_LipIntInfComputeLocalLipschitz','MAPLE',
LIB=DLLPATH):
LipIntInfComputeLipschitzCV :=define_external('MWRAP_LipIntInfComputeLipschitzCV','MAPLE',
LIB=DLLPATH):
LipIntInfComputeLipschitzSplit :=define_external('MWRAP_LipIntInfComputeLipschitzSplit','MAPLE',
LIB=DLLPATH):
LipIntInfSmoothLipschitz :=define_external('MWRAP_LipIntInfSmoothLipschitz','MAPLE',LIB=DLLPATH):
LipIntInfGetLipConst :=define_external('MWRAP_LipIntInfGetLipConst','MAPLE', LIB=DLLPATH):
LipIntInfGetScaling :=define_external('MWRAP_LipIntInfGetScaling','MAPLE', LIB=DLLPATH):
LipIntInfComputeScaling :=define_external('MWRAP_LipIntInfComputeScaling','MAPLE',LIB=DLLPATH):
LipIntInfVerifyMonotonicity :=define_external('MWRAP_LipIntInfVerifyMonotonicity','MAPLE',
LIB=DLLPATH):
LipIntInfVerifyMonotonicityLeftRegion :=
define_external('MWRAP_LipIntInfVerifyMonotonicityLeftRegion','MAPLE',LIB=DLLPATH):
LipIntInfVerifyMonotonicityRightRegion :=
define_external('MWRAP_LipIntInfVerifyMonotonicityRightRegion','MAPLE',LIB=DLLPATH):
LipIntSetBounds := define_external('MWRAP_SetBoundaries','MAPLE',LIB=DLLPATH):
LipIntClearBounds :=define_external('MWRAP_ClearBoundaries','MAPLE', LIB=DLLPATH):

STCBuildLipInterpolant :=define_external('MWRAP_STCBuildLipInterpolant','MAPLE',LIB=DLLPATH):
STCBuildLipInterpolantExplicit :=define_external('MWRAP_STCBuildLipInterpolantExplicit','MAPLE',
LIB=DLLPATH):
STCSetLipschitz := define_external('MWRAP_STCSetLipschitz','MAPLE',LIB=DLLPATH):
STCValue := define_external('MWRAP_STCValue','MAPLE', LIB=DLLPATH):
STCValueExplicit :=define_external('MWRAP_STCValueExplicit','MAPLE', LIB=DLLPATH):
STCFreeMemory := define_external('MWRAP_STCFreeMemory','MAPLE',LIB=DLLPATH):

```

Note that these commands have been set in the example Maple worksheet, and can be simply copied from there. The name on the left can be changed by the user, but we describe the procedures using these names.

The main parameters that need to be supplied to `LibLip` are the arrays containing the data, their dimensions, the point x at which the value has to be

computed, and the Lipschitz constant.

There are procedures with the name starting with `LipInt`, procedures starting with `LipIntInf` and those starting with `STC`. Procedures that start with `LipInt` and `LipIntInf` are equivalent, they just use the Euclidean or Chebyshev distance in the Lipschitz condition. However the procedures starting with `STC` use the simplicial distance and contain both preprocessing and fast evaluation methods. These methods do not handle monotonicity or range constraints, nor smoothing.

Evaluation of the interpolant

Methods to compute the value of the interpolant at a given point x .

`LipIntValue(dim, N, x, X,Y , LC, index)`

`LipIntInfValue(dim, N, x, X,Y , LC, index)`

Computes and returns the value of the interpolant $g(x)$. Does not require any preprocessing. dim is the dimension, N is the number of data, x is the vector of size dim , X is the vector of data of size $N \times dim$ which contains values x_i^k in its rows, y is the vector of size N of values to be interpolated, LC is the Lipschitz constant.

Notes: The optional parameter *index* is an array of integers of size N , used to index the data in a large data set, which are used in the construction of the interpolant. For example, `index[1]:=1, index[2]:=5, ...` One can use the data for interpolation selectively, by indexing the required values. The parameter N should be the number of selected data used in the interpolation, not the size of the whole data set.

`LipIntValueAuto(dim, N, x, X,Y , index)`

`LipIntInfValueAuto(dim, N, x, X,Y , index)`

Variation of the above, uses Lipschitz constants automatically identified from the data. **Should only be called after `LipIntComputeLipschitz()`, or `LipIntComputeLipschitzCV()`, or `LipIntComputeLipschitzSplit()`.** These functions calculate the Lipschitz constant and store it internally. It can be retrieved using `LipIntGetLipConst()`.

`LipIntValueLocal(dim, N, x, X,Y)`

`LipIntInfValueLocal(dim, N, x, X,Y)`

Variation of the above for locally Lipschitz functions, uses Lipschitz constants dependent on the position x , automatically identified from the data. **Should only be called after `LipIntComputeLocalLipschitz()`.**

Variations for constrained interpolation

`LipIntValueCons(dim, N, Cons, x, X,Y , LC, index)`

`LipIntInfValueCons(dim, N, Cons, x, X,Y , LC, index)`

Same as `LipIntValue()`, but for monotone functions. *Cons* is an array of size dim specifying monotonicity constraints. Constraints are coded as follows: $Cons[i] = 1$ means the function is increasing with respect to the i -th variable, $Cons[i] = -1$ means it is decreasing, $Cons[i] = 0$ means unrestricted.

`LipIntValueConsLeftRegion(dim, N, Cons, x, X,Y , LC, Region, index)`

`LipIntValueConsRightRegion(dim, N, Cons, x, X,Y , LC, Region, index)`

`LipIntInfValueConsLeftRegion(dim, N, Cons, x, X,Y , LC, Region, index)`

`LipIntInfValueConsRightRegion(dim, N, Cons, x, X,Y , LC, Region, index)`

Same as `LipIntValueCons()`, for monotone functions in the region $x \preceq Region$ (left region) or $x \succeq Region$ (right region). *Region* is a vector of size *dim* denoting the top right corner (left region) of the region of monotonicity, or its bottom left corner (right region).

```
LipIntValueLocalCons( dim, N, Cons, x, X,Y, Region )
LipIntInfValueLocalCons( dim, N, Cons, x, X,Y, Region )
LipIntValueLocalConsLeftRegion( dim, N, Cons, x, X,Y , Region)
LipIntInfValueLocalConsLeftRegion( dim, N, Cons, x, X,Y, Region )
LipIntValueLocalConsRightRegion( dim, N, Cons, x, X,Y, Region )
LipIntInfValueLocalConsRightRegion( dim, N, Cons, x, X,Y , Region)
```

Same as the methods above, but for locally Lipschitz functions. Can only be called after `LipIntComputeLocalLipschitz()` or `LipIntInfComputeLocalLipschitz()`.

Smoothing the data

Methods for smoothing noisy data. The output is the smoothened data vector *TData*, which should be subsequently used instead of *Y* in `LipIntValue*`() and its variations. These methods use `glpk` linear programming programming library. They can be quite expensive if *N* is more than 200 or so.

```
LipIntSmoothLipschitz( dim, N,X,Y,TData, LC, fw,fc,fr,W, Cons,Region )
LipIntInfSmoothLipschitz( dim, N,X,Y,TData, LC, fw,fc,fr,W, Cons,Region )
```

Computes the vector *TData* of modified (smoothened) data values, consistent with the specified Lipschitz constant *LC*. *dim* is the dimension, *N* is the size of the data set, the abscissae of the data points are in the array *X* of size $N \times dim$, stored in rows, the data values are supplied in *Y* of size *N*. The memory for the array *TData* should be provided in the calling worksheet, see examples.

The parameters *fw*, *fc*, *fr* are integers (flags) which identify whether the arrays *W*, *Cons*, *Region* will be used, 0 if no, yes, otherwise.

The array *W* is the vector of size *N* of non-negative weights, which reflect the relative confidence in the accuracy of data values. Data with high weights are not modified. Weights do not have to be normalized to one. Needed only if *fw* = 1, otherwise the weights are assumed to be equal (default).

The vector *Cons* of size *dim* contains information about monotonicity constraints (only of *fc* = 1). Constraints are coded as follows: *Cons*[*i*] = 1 means the function is increasing with respect to the *i*-th variable, *Cons*[*i*] = -1 means it is decreasing, *Cons*[*i*] = 0 means unrestricted.

The vector *Region* of size *dim* contains information about the region where monotonicity constraints are imposed. Namely, $x \preceq Region$ (left region) or $x \succeq Region$ (right region). *fr* be specified as 1 if Left Region, -1 of Right Region, 0 if no region. *Region* is a vector of size *dim* denoting the top right corner (left region) of the region of monotonicity, or its bottom left corner (right region).

Estimation of the Lipschitz constant

Various methods of estimating the Lipschitz constant from the data. If the data is noisy, we use sample splitting or cross-validation methods, and then smoothen the data with the computed Lipschitz constant.

```
LipIntComputeLipschitz( dim, N, X,Y )
```

LipIntInfComputeLipschitz(dim, N, X,Y)

Computes the smallest Lipschitz constant consistent with the data. Bear in mind that **ComputeLipschitz** requires $O(dN^2)$ operations and should be avoided if there are other means to estimate the Lipschitz constant. The value is retrieved by using **LipIntGetLipConst** () or **LipIntInfGetLipConst** ().

LipIntComputeLocalLipschitz(dim, N, X,Y)

LipIntInfComputeLocalLipschitz(dim, N, X,Y)

Computes various arrays which contain information about the local Lipschitz constants estimated from the data. After this method, the value of the interpolant can be obtained by using **LipIntValueLocal**() or **LipIntInfValueLocal**() or their variations.

LipIntComputeLipschitzSplit(dim, N, X,Y, ,TData, Ratio, type, Cons, Region, W)

LipIntInfComputeLipschitzSplit(dim, N, X,Y, TData, Ratio,type, Cons, Region, W)

Computes an estimate of the Lipschitz constant from noisy data using sample splitting (with the ratio *ratio*), and then smoothens the data using the computed Lipschitz constant. The smoothened data are returned in *TData*, and the computed Lipschitz constant is retrieved by using **LipIntGetLipConst**() or **LipIntInfGetLipConst**(). The data is split randomly into subsets \mathcal{D}_1 and \mathcal{D}_2 , the first one is used to predict the values in the second. *ratio* is the probability that a datum is allocated to subset \mathcal{D}_1 . The parameters *Cons*, *Region* and *W* are optional.

Parameter *type* can have four values. *type* = 0 means normal Lipschitz approximation, *type* = 1 means monotone approximation, in which case vector *Cons* denotes the monotonicity constraints as in **LipIntValueCons**(), and should be set by the user. *type* = 2 means monotone in the left region, and *type* = 4 means monotone in the right region, in which cases the parameter *Region* must also be set (as in **LipIntSmoothLipschitz**(). If the accuracy of the data is not the same, the vector of non-negative weights should be provided in the optional vector *W* (as in **LipIntSmoothLipschitz**().

LipIntComputeLipschitzCV(dim, N, X,Y, ,TData, type, Cons, Region, W)

LipIntInfComputeLipschitzCV(dim, N, X,Y, TData, type, Cons, Region, W)

Computes an estimate of the Lipschitz constant from noisy data using Cross-Validation, and then smoothens the data using the computed Lipschitz constant. The smoothened data are returned in *TData*, and the computed Lipschitz constant is retrieved by using **LipIntGetLipConst**() or **LipIntInfGetLipConst**().

The parameters have the same meaning as in **LipIntComputeLipschitzSplit**(), but no *Ratio* is required. This method uses *N*-fold cross-validation technique, in which each datum is removed from the data set and its value is predicted using the rest of the data and an estimate of the Lipschitz constant. Thus it involves solving *N* smoothing problems, i.e., quite an expensive procedure. Avoid it when *N* is large. For small data set it is preferable to sample splitting, as the data sets in the latter method may be too small.

Auxiliary methods

This is a collection of methods useful for verifying monotonicity of the data or its scaling.

LipIntComputeScaling(dim, N,X, Y, Scaling)

LipIntInfComputeScaling(dim, N, X, Y,Scaling)

Computes the array of scaling factors, returned in the array of size *dim* *Scaling*, which is needed to standardize the data to have standard deviation 1 in respect to all variables. *Scaling* contains 1/the standard deviations. After calling this method,

one can standardize the data by using $X[i,j] := X[i,j] * \text{Scaling}[j]$ for all i and $j = 1, \dots, \text{dim}$ and $i = 1, \dots, N$.

LipIntGetScaling(Scaling)

LipIntInfGetScaling(Scaling)

Returns the array *Scaling*, computed by using **LipIntComputeScaling()** or **LipIntInfComputeScaling()**.

LipIntVerifyMonotonicity(dim, N,Cons, X, Y, LC, eps)

LipIntInfVerifyMonotonicity(dim, N, Cons, X, Y,LC, eps)

LipIntVerifyMonotonicityLeftRegion(dim, N,Cons, X, Y, Region, LC, eps)

LipIntInfVerifyMonotonicityLeftRegion(dim, N, Cons, X, Y,Region, LC, eps)

LipIntVerifyMonotonicityRightRegion(dim, N,Cons, X, Y, Region, LC, eps)

LipIntInfVerifyMonotonicityRightRegion(dim, N, Cons, X, Y,Region, LC, eps)

Return 1 if the data set is compatible with the given monotonicity and Lipschitz conditions. That is, the data set should be compatible with the class *Lip(LC)*, and also $x^k \succeq x^i$ should imply $y^k \geq y^i$. The direction of the inequality changes for monotone decreasing functions. Functions can be increasing in some variables and decreasing in the others. This case is reduced to functions increasing in all variables by changing the sign of some components of x . The method accommodates all these cases (coded in *Cons*, $\text{Cons}[j] = 1$ means the function is increasing wrt j -th variable, $\text{Cons}[j] = -1$ means decreasing). This method is meaningful when *Cons* does not have zero components (i.e., functions unrestricted in some variables). *eps* is the tolerance parameter (i.e., we require $y^k - y^i \geq \text{eps}$).

The last for procedures are variations of **LipIntVerifyMonotonicity()**, where monotonicity condition is checked in the regions $x \preceq \text{LeftRegion}$, or $x \succeq \text{RightRegion}$.

Parameters are the same as in **LipIntValueCons()**, and *eps* is a given tolerance.

Extra bounds

When extra bounds are required, they must be implemented as Maple procedures that take exactly $\text{dim} + 1$ parameters, the first dim parameters is the point x at which the bounds are computed, and the remaining parameter is the current value of the Lipschitz constant (which may or may not be used by the procedures).

The address of the procedures to calculate the bounds is supplied in

LipIntSetBounds(Upper, Lower)

before calling any smoothing or calculation methods. The bounds can be cleared by calling

LipIntClearBounds()

Example:

```
dim:=2;
low := proc( x, y,p)  max(0,1-p*((1-x)+(1-y)));  end proc;
up:=proc(x,y,p) min(x,y); end proc;

LipIntSetBounds(up,low);
...
LipIntSmoothLipschitz(dim, Ndata,XD,YD,TD,LC,0,0,0);
```

Fast evaluation procedures

STCBuildLipInterpolant(dim, N,X, Y, LC)

Preprocesses the data supplied in X, Y , using Lipschitz constant LC . If LC is not supplied, it will be computed from the data automatically. This procedure should be called before `STCValue`.

STCValue(x)

Calculates the value of the interpolant at x using **fast** evaluation method. Should be called after `STCBuildLipInterpolant()`.

STCBuildLipInterpolantExplicit(dim, N, X, Y, LC)

Preprocesses the data supplied in X, Y , using Lipschitz constant LC , in order to be used by the explicit evaluation method. This procedure just makes a copy of the supplied data, no time consuming preprocessing is done. If LC is not supplied, it will be computed from the data automatically. This procedure should be called before `STCValueExplicit`.

STCValueExplicit(x)

Calculates the value of the interpolant at x using **explicit** evaluation method. Should be called after `STCBuildLipInterpolantExplicit()` or `STCBuildLipInterpolant()`.

STCFreeMemory(x)

Frees the memory occupied by the data structures computed in `STCBuildLipInterpolant()`, which can be very large. **It destroys the interpolant, and `STCValue*()` methods cannot be called after `STCFreeMemory()`.**

Warning: do not use `STCBuildLipInterpolant()` after `STCBuildLipInterpolantExplicit()` or vice versa.

Chapter 4

Examples of usage

4.1 Sample code

There are several examples of the usage of `LibLip` toolbox provided in the example worksheet. The best way to use `LibLip` is to follow these exammples.

4.1.1 Interpolation and smoothing

When using the toolbox `LipInt`, there is no need for preprocessing. The user just calls various evaluation routines and supplies the data set as well as the point x . The Lipschitz constant can be estimated from the data set automatically. If the user desires to use local Lipschitz interpolation, then the array of Lipschitz constants must be computed automatically from the data by calling `LipIntComputeLocalLipschitz`.

When using the procedures `STCValue*()`, there is a need for preprocessing.

```

# first we generate 2d sample data using some function f
Lo:=Vector(1..2,datatype=float[8],[-2,-1]):
Up:=Vector(1..2,datatype=float[8],[2,2]):
dim:=2:
Ndata:=600:
Xr:=[stats[random, uniform]((Ndata)*dim)]: # uniformly distributed numbers
XD:=Matrix(1..Ndata, 1..dim, datatype=float[8], order=C_order):
#will hold the data abscissae
YD:=Vector(1..Ndata,datatype=float[8], order=C_order):
# will hold the function values

X:=Vector(1..dim,datatype=float[8]):
scale:=(x,a,b)->x*(b-a)+a: # just a scaling function
for i from 1 to Ndata do
  for j from 1 to dim do
    X[j]:=scale(Xr[i*j],Lo[j],Up[j]):
  od:
  YD[i]:= f(X[1],X[2]):          # record function values
  for j from 1 to dim do
    XD[i,j]:=X[j]:              # record data abscissae
  od:
od:

LC:=10: X[1]:=0: X[2]:=0.5: #just a sample point
fint:=LipIntValue(dim,Ndata,X,XD,YD,LC):

LipIntComputeLipschitz(dim, Ndata, XD,YD);
LC := LipIntGetLipConst();
fint:=LipIntValue(dim,Ndata,X,XD,YD,LC):

LC:=1.5: # this is a desired value of Lipschitz constant
TD:=Vector(1..Ndata,datatype=float[8], order=C_order):
# will hold the smoothened values
LipIntSmoothLipschitz(dim, Ndata,XD,YD,TD,LC,0,0,0);

fint:=LipIntValue(dim,Ndata,X,XD,TD,LC):

LipIntComputeLocalLipschitz(dim, Ndata, XD,YD);
fint:=LipIntValueLocal(dim,Ndata,X,XD,YD):

STCBuildLipInterpolant(dim,Ndata,XD,YD);
STCValue(X);
STCValueExplicite(X);
STCFreeMemory();

```

4.1.2 Monotone interpolation

...

```

# data generated as before
Cons:=Vector(1..2,datatype=integer[4],[1,-1]):
Region:=Vector(1..2,datatype=float[8],[0.5,0.5]):

LipIntValueCons(dim, Ndata, Cons, X, XD, YD, LC):
LipIntValueConsLeftRegion(dim, Ndata, Cons, X, XD, YD, LC, Region):

LipIntComputeLocalLipschitz(dim, Ndata, XD,YD);
fint:=LipIntValueLocalCons(dim,Ndata,Cons,X,XD,YD):

#no constraints
LipIntComputeLipschitzSplit(dim,Ndata,XD,YD,TD,0.2,0);
LC := LipIntGetLipConst();
fint:=LipIntValue(dim,Ndata,X,XD,TD,LC):

# Use monotonicity constraints
LipIntComputeLipschitzSplit(dim,Ndata,XD,YD,TD,0.2,1,Cons);
LC := LipIntGetLipConst();
fint:=LipIntValueCOns(dim,Ndata,Cons,X,XD,TD,LC):

# smooth using a desired LC
LC:=1:
LipIntSmoothLipschitz(dim,Ndata,XD,YD,TD,LC,0,1,0,Cons);
fint:=LipIntValueCons(dim,Ndata,Cons,X,XD,TD,LC):

#plot
f_aux:=(t1,t2)->LipIntValue(dim,Ndata, convert([t1,t2],Vector,datatype=float[8]),
XD,TD,LC):
R:=20.0: #plot resolution
sdata :=[seq([ seq([i/R,j/R,f_aux2(i/R,j/R)], i=R*Lo[1]..R*Up[1]),j=R*Lo[2]..R*Up[2]])]:
S1:=surfdata(sdata,axes=normal,grid=[100,100],style=HIDDEN,axes=BOXED,orientation=[25,50]):
S2:=pointplot3d({seq([XD[k,1],XD[k,2],YD[k]],k=1..Ndata)},axes=BOXED,color=BLACK):
display({S1,S2});

#use extra bounds
low := proc( x, y,p) max(0,1-p*((1-x)+(1-y))); end proc;
up:=proc(x,y,p) min(x,y); end proc;
LipIntSetBounds(up,low);

LC:=1.0:
LipIntSmoothLipschitz(dim, Ndata,XD,YD,TD,LC,0,0,0);
fint:=LipIntValueCons(dim,Ndata,Cons,X,XD,TD,LC);

LipIntClearBounds();

```

4.2 Tips

It is a common problem when the user incorrectly sets the Lipschitz constant for his/her particular case. The value `LipConst=1` in the examples is for the sake of example only. The user must use their own values, which depend on the data and problem at hand.

The procedure `STCBuildInterpolant` will not tolerate low values of the Lipschitz constant. The algorithm may fail to build the internal data structures, and to properly compute the value of the interpolant.

If the value of the Lipschitz constant is unknown, the user is advised to compute it from the data using `LipIntComputeLipschitz()`, or `LipIntComputeLipschitzSplit()`, or `LipIntComputeLipschitzCV()`.

The smoothing methods become computationally expensive for $N > 500$, and the user is advised to estimate the running time on some examples.

To obtain a smoother interpolant, we advise to use `LipIntComputeLocalLipschitz..()` and `LipIntValueLocal..()` methods. In this case the Lipschitz constant will be computed locally (i.e., it will depend on the position x). Some functions may have large gradients at some points, and small gradients on the rest of the domain. Using one (large) Lipschitz constant may be inappropriate in these cases.

We also advise to ensure that the data are not repeated, as this may upset the preprocessing algorithm.

4.3 Performance of the algorithms

The table below illustrates the performance of the `LibLip` library on test data sets. Measurements were performed on a modest workstation with Pentium VI processor (1.2GHz) 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 $O(N)$. The fast evaluation method requires preprocessing, but the evaluation time grows as $O(\log N)$. 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. Note that procedures named `LipInt*` and `LipIntInf*` perform exhaustive evaluation only, but are more flexible in accommodating constraints.

The user should be aware of the limitations of `LibLip` 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 $O(N^d)$. For $d > 4$ and large N , 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 $d = 2, \dots, 5$. The row with the largest N corresponds to approximately 500MB of RAM requested by `LibLip`.

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 $[0, 100]^d$ or $[-1, 1]^d$, as well as the data values. Scaling will affect the Lipschitz constant, which must be adjusted accordingly.

d	N	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 **LibLip** as a function of the number of data points and dimension. Explicit evaluation refers to directly using Eq.(2.2) in computations, whose complexity grows linearly with N . For every d , the row with the largest N is when the library has used 500 MB of RAM.

Chapter 5

More information

5.1 Where to get help

The software library `LibLip` and its components and toolboxes, are distributed by G.Beliakov AS IS, with no warranty, explicit or implied, of merchantability or fitness for a particular purpose. G.Beliakov, at his sole discretion, may provide advice to registered users on the proper use of `LibLip` and its components.

Any queries regarding technical information, sales and licensing should be directed to `gleb@deakin.edu.au`.

5.2 References

G. Beliakov, Interpolation of Lipschitz functions, *Journal of Computational and Applied Mathematics*, 2006, 196, pp. 20-44.

G. Beliakov, Monotonicity preserving approximation of multivariate scattered data, *BIT*, 2005, 45 pp. 653-677.