Programming library `liblip` for multivariate scattered data interpolation. Version 2.0. User Manual

Gleb Beliakov
gleb@deakin.edu.au

License agreement

`LibLip` is distributed under GNU LESSER 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 . . . 10
  - 2.3.1 The main properties . . . 10
  - 2.3.2 Types of approximation problems . . . 11
- 2.4 Computational methods . . . 12
  - 2.4.1 Interpolation . . . 12
  - 2.4.2 Monotone Interpolation . . . 12
  - 2.4.3 Smoothing . . . 14
  - 2.4.4 Estimation of Lipschitz constant . . . 14
  - 2.4.5 Locally Lipschitz functions . . . 16
  - 2.4.6 Bounds on f . . . 17

3 Description of the library **19**
- 3.1 Installation . . . 19
- 3.2 Programming interface . . . 20
- 3.3 Members of `SLipInt` class. . . . 22
- 3.4 Members of `SLipIntInf` class. . . . 27
- 3.5 Members of `SLipIntLp` class. . . . 28
- 3.6 Members of `STCInterpolant` class. . . . 29
- 3.7 Procedural interface . . . 31

4 Examples of usage **37**
- 4.1 Sample code . . . 37
  - 4.1.1 Interpolation . . . 37

4.1.2 Smoothing 43
4.2 Linking 46
4.3 Fortran interface 46
4.4 Tips 47
4.5 Performance of the algorithms 47

5 Where to get help **51**

Chapter 1

Introduction

This manual describes the programming library `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 alternative 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 programming library `LibLip` is given in Chapter 3. Examples of its usage are provided in Chapter 4. Section 4.5 analyses the performance of the algorithms, and their applicability.

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

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

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 $g : R^d \rightarrow R$,

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

such that

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

$Lip(M)$ 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}

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

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

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

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

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

where the upper and lower bounds are given as

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

The method of calculation is straightforward. The method `Value()` of the class `SLipInt` implements 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 method `ComputeLipschitz` 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 $\mathcal{V} \subset \{1, \ldots, d\}$. 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

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

where $z_{\mathcal{V}+}$ denotes the positive part of vector z with respect to subset of components $\mathcal{V}$: $z_{\mathcal{V}+} = (\bar{z}_1, \ldots, \bar{z}_n)$, 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 $||(x - x^k)_{\mathcal{V}+}|| \neq ||(x^k - x)_{\mathcal{V}+}||$.

Method `ValueCons()` implements 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

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

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

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

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

and $(a - b)_{\mathcal{V}+,B} = (\bar{z}_1, \ldots, \bar{z}_n)$, 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 methods `ValueConsLeftRegion()` and `ValueConsRightRegion()` 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

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

For monotone functions we use $||(x^i - x^j)_{\mathcal{V}_+}||$.
We solve the following optimization problem

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

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.

Methods `SmoothLipschitz()`, `SmoothLipschitzW()`, `SmoothLipschitzCons()`, `SmoothLipschitzWCons()` implement these smoothing techniques. Methods `SmoothLipschitzConsLeftRegion()`, `SmoothLipschitzConsRightRegion()`, `SmoothLipschitzWConsLeftRegion()`, `SmoothLipschitzWConsRightRegion()` 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 form $\mathcal{D}_2$ 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

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

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 `ComputeLipschitzSplit()` 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 $i \in \{1,\dots,N\}$

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

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 `ComputeLipschitzCV()` implements this technique. Note that is 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

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

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

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

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

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

It is also possible in 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 $0, t_1, t_2, \ldots, t_{N_k}$. 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 methods `ValueLocal()`, `ValueLocalCons()`, `ValueLocalConsLeftRegion()` and `ValueLocalConsRightRegion()` implement this interpolation scheme. These methods should be called after `ComputeLocalLipschitz()`.

2.4.6 Bounds on f

This library 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` library. 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 member variable `UseOtherBounds` must be set to 1, and the virtual functions `ExtraLowerBound()` and `ExtraUpperBound()` should be implemented in a class derived from `SLipInt` (see the description of the library).

Chapter 3

Description of the library

3.1 Installation

Installation of `LibLip` package is simple: the user needs to unpack the distribution files and run `lipinstall` script (on unix), or simply copy the relevant source and .lib files into the desired directories (Windows). For instance, the user can place them into his project directory. Unix distribution uses `libtool` software to generate the library files by compiling the source code. If the user does not have root access on her unix workstation, execute `lipinstall installation_directory` command, which installs `LibLip` into the specified directory, rather than the default library directories.

The package contains a number of header files, and binary static library files (we also provide .dll files for windows developers). To use `LibLip` add

```
#include "LibLip.h"
```

line to your code, and link against `liblip.a` library (using `-llip` option in the make file).

Some methods in `LibLip` rely on external packages (also distributed under GNU licenses), namely `GLPK` for solving linear programming problems. The current versions of this package can be downloaded from `http://www.gnu.org/software/glpk/glpk.html`, or from the author's web site. Windows distribution includes the precompiled lib and dll files.

If the user wants to use smoothing features of `LibLip`, he should link against `glpk` library as well, by using `-lglpk -llip` in the makefile

There is a number of sample programs included with this distribution,

which illustrate the major features of the library.

3.2 Programming interface

The method of Lipschitz interpolation is implemented in the programming library `LibLip` in `C++` language. The interpolation methods can be accessed via two classes or via a number of procedures.

There are four main classes which provide the interface to the preprocessing and computation and are called `SLipInt`,`SLipIntInf`, `SLipIntLp` and `STCInterpolant`. The class `SLipInt` should be used in the majority of cases, as it includes most features. Class `SLipIntInf` is analogous to `SLipInt`, but uses l_∞ norm instead of Euclidean distance. In lower dimensions it may offer some computational advantages when smoothing the data. Class `SLipIntLp` uses an arbitrary l_p-norm in Eq. (2.4).

Note in the source code, both `SLipInt` and `SLipIntInf` are derived from a basic class `SLipIntBasic`, and many of the methods described below are actually declared in `SLipIntBasic`, and are inherited by `SLipInt` and `SLipIntInf`. `SLipIntBasic` contains pure virtual functions and no instances of this class should be used. The class `SLipIntLp` is derived from `SLipInt` and differs only by using an arbitrary $p \geq 1$. The value of p must be of course specified by the user.

```
class SLipInt { //simple Lipschitz interpolant
public:
//  Computes the interpolant value using the Eulidean norm
 double Value(int dim,int N,double* x,double* X,double* Y,double LC);
// Same subject to monotonicity constraints
 double ValueCons(int dim,int N, int* Cons, double* x, double* X,
                  double* Y,double LC);
// Estimates the Lipschitz constant compatible with the data
    void ComputeLipschitz(int dim, int N, double* X, double* Y);
// Smoothens the data to match given Lipschitz constant
    void SmoothLipschitz(int dim, int N, double* X, double* Y,
                  double* YT, double LC);
// Computes the Lip. constant using sample splitting and smoothens the data
    void ComputeLipschitzSplit(int dim, int N, double* X, double* Y,
                  double* YT, double ratio);
...
 };
```

```
class STCInterpolant {
//Interpolant which uses simplicial distance
//and fast evaluation method with preprocessing
public:
 // supplies the data set to this class.
 // All other methods are called after SetData.
      void SetData(int dim, int N, 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);
...
}
```

The class `STCInterpolant` uses simplicial distance, and by using preprocessing method, can achieve faster evaluation time for $dim < 5$. The class `SLipInt` uses explicit evaluation when distance $d()$ is Euclidean (l_2), and `SLipIntInf` uses Chebyshev (l_∞) norm. Recommended for higher dimension. Class `SLipIntLp` uses an arbitrary l_p-norm.

3.3 Members of `SLipInt` class.

Class `SLipInt` implements explicit evaluation of Lipschitz interpolant and computation of the Lipschitz constant. It does not use any special data structure or preprocessing, except for smoothing, and relies on Eq. (2.2). The simplicity of the evaluation process and program code is a feature of this class. It can be used for high dimension. It can be used for Locally Lipschitz functions, where the Lipschitz constant depends on the position x, and is estimated automatically from the data.

Evaluation of the interpolant

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

`double Value(int dim,int N,double* x,double* X,double* Y,double LC, int* index=NULL)`
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 size N, used to index the data in a large data set, which are used in the construction of the interpolant. For example, index[0]=1, index[1]=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.

`double Value(int dim,int N, double* x, double* X,double* Y, int* index=NULL);`
Variation of the above, uses Lipschitz constants automaticlly identified from the data. **Should only be called after `ComputeLipschitz()`, or `ComputeLipschitzCV()`, or `ComputeLipschitzSplit()`, or setting the member variable `MaxLipConst`.**

`double ValueLocal(int dim, int N, double* x, double* X,double* Y);`
Variation of the above for locally Lipschitz functions, uses Lipschitz constants dependent on the position x, automaticlly identified from the data. **Should only be called after `ComputeLocalLipschitz()`.**

Variations for constrained interpolation

`double ValueCons(int dim,int N, int* Cons, double* x, double* X,double* Y, double LC, int* index=NULL);`

Same as `Value()`, 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.

`double ValueConsLeftRegion(int dim,int N, int* Cons, double* x, double* X,double* Y, double LC, double* LeftRegion, int* index=NULL);`

Same as `ValueCons()`, for monotone functions in the region $x \preceq LeftRegion$. $LeftRegion$ is a vector of size *dim* denoting the top right corner of the region of monotonicity.

`double ValueConsRightRegion(int dim,int N, int* Cons, double* x, double* X,double* Y, double LC, double* RightRegion, int* index=NULL);`

Same as `ValueConsLeftRegion()`, for monotone functions in the region $x \succeq RightRegion$. $RightRegion$ is a vector of size *dim* denoting the bottom left corner of the region of monotonicity.

`double ValueLocalCons(int dim, int N, int* Cons, double* x, double* X,double* Y);`

Same as `ValueLocal()`, but for monotone functions. *Cons* is an array of size *dim* specifying monotonicity constraints, as in `ValueCons()`.

`double ValueLocalConsLeftRegion(int dim, int N, int* Cons, double* x, double* X,double* Y, double* Region);`

Same as `ValueLocalCons()`, for monotone functions in the region $x \preceq LeftRegion$. $LeftRegion$ is a vector of size *dim* denoting the top right corner of the region of monotonicity.

`double ValueLocalConsRightRegion(int dim,int N, int* Cons, double* x, double* X,double* Y, double* RightRegion);`

Same as `ValueLocalConsLeftRegion()`, for monotone functions in the region $x \succeq RightRegion$. $RightRegion$ is a vector of size *dim* denoting the bottom left corner of the region of monotonicity.

Smoothing the data

Methods for smoothing noisy data. The output is the smoothened data vector $TData$, which can subsequently be used as Y in `Value()` and its variations. These methods use `glpk` programming library.

`void SmoothLipschitz(int dim, int N, double* X, double* Y, double* TData, double LC);`

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 routine.

`void SmoothLipschitzW(int dim, int N, double* X, double* Y, double* TData, double LC, double* W)`

Variation of the above, requires an array of non-negative weights of size N. Weights 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.

`void SmoothLipschitzCons(int dim, int N, int* Cons, double* X, double* Y, double* TData, double LC)`

Same as `SmoothLipschitz`, subject to monotonicity constraints. The vector $Cons$ of size dim contains information about 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.

`void SmoothLipschitzConsLeftRegion(int dim, int N, int* Cons, double* X, double* Y, double* TData, double LC, double* LeftRegion)`

Same as `SmoothLipschitzCons`, subject to monotonicity constraints in the region $x \preceq LeftRegion$.

`void SmoothLipschitzConsRightRegion(int dim, int N, int* Cons, double* X, double* Y, double* TData, double LC, double* RightRegion)`

Similar to `SmoothLipschitzConsLeftRegion`, subject to monotonicity constraints in the region $x \succeq RightRegion$.

`void SmoothLipschitzWCons(int dim, int N, int* Cons, double* X, double* Y, double* TData, double LC, double* W)`

Same as `SmoothLipschitzCons`, subject to weighting vector W.

`void SmoothLipschitzWConsLeftRegion(int dim, int N, int* Cons, double* X, double* Y, double* TData, double LC, double *W, double* LeftRegion)`

`void SmoothLipschitzWConsRightRegion(int dim, int N, int* Cons, double* X, double* Y, double* TData, double LC, double *W, double* RightRegion)`

Same as `SmoothLipschitzConsLeftRegion`, and `SmoothLipschitzConsRightRegion` subject to weighting vector W.

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.

`void ComputeLipschitz(int dim,int N,double* X,double* 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 kept in $MaxLipConst$ member variable.

`void ComputeLocalLipschitz(int dim,int N,double* X,double* Y);`
Computes various arrays which contain information about the local Lipschitz constants estimated from the data. The values are kept in $GridLim, GridR, GridVal$, member variables. After this method, the value of the interpolant can be obtained by using `ValueLocal()`.

`void ComputeLipschitzSplit(int dim,int N,double* X,double* Y, double* TData, double ratio=0.5,int type=0, int* Cons=NULL, double* Region=NULL, double *W=NULL);`
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 kept in the member variable `MaxLipConst`. 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 $\mathcal{D}_1$.
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 `ValueCons()`, 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 be set (as in `ValueConsLeftRegion()`. If the accuracy of the data is not the same, the vector of non-negative weights should be provided, as in `SmoothLipschitzW()`.

`void ComputeLipschitzCV(int dim,int N,double* X,double* Y, double* TData, int type=0, int* Cons=NULL, double* Region=NULL, double *W=NULL);`
Computes an estimate of the Lipschitz constant from noisy data using Cross-Validation, and then smoothens the data using the computed Lipschitz constant. The smooehtned data are returned in $TData$, and the computed Lipschitz constant is kept in the member variable `MaxLipConst`. The parameters have the same meaning as in `ComputeLipschitzSplit()`. 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 transforming the data and experimenting with various features of the library.

`void ComputeScaling(int dim,int N,double* X,double* Y)`

Computes the array of scaling factors (member variable `Scaling`, an array of size dim), 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*dim+j] *= Scaling[j]` for all i and $j = 1, \ldots, d$.

`void ConvertXData(int dim,int N,double* X);`

Transposes the matrix X. Useful when the `LibLip` is called from other program/languages which store the matrices columnwise (as in Fortran, Matlab, etc.). `LibLip` uses C convention and stores data in rows. The transposed matrix is returned in X.

`void ConvertXData(int dim,int N,double* X, double* Aux);`

Transposes the matrix X. Same as the previous method, but the transposed X is returned in the array Aux. The memory for Aux should be allocated in the calling routine ($dim \times N$).

`void Dominates(int dim, double* x, double* y, int* Cons);`

Returns 1 if $x \succeq y$, with respect to the given array of indices. That is, returns 1 if $\forall i \in \{1, \ldots, d\}, Cons[i] \neq 0 : x_i \geq y_i$. Useful when dealing with monotonicity condition.

`int VerifyMonotonicity(int dim, int N, int* Cons, double* X, double* Y, double LC, double eps);`

Returns 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 $Cons$, $Cons[j] = 1$ means the function is increasing wrt j-th variable,$Cons[j] = -1$ 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 $y^k - y^i \geq eps$).

`int VerifyMonotonicityRegionLeft(int dim, int N, int* Cons, double* X, double* Y, double* LeftRegion, double LC, double eps);`

`int VerifyMonotonicityRegionRight(int dim, int N, int* Cons, double* X, double* Y, double* RightRegion, double LC, double eps);`

Variations of `VerifyMonotonicity()`, where monotonicity condition is checked in the regions $x \preceq LeftRegion$, or $x \succeq RightRegion$.

Extra bounds

When extra bounds are required, they must be implemented in a class derived from `SLipInt` or `SLipIntLp` or `SLipIntInf` classes.

`double ExtraUpperBound(int dim, double* x, double * param)`

`double ExtraLowerBound(int dim, double* x, double * param)`

The derived class calculates these extra bounds and returns the computed values. x is the point at which the bounds are calculated and *param* contains the Lipschitz constant (its current value, when these methods are called when estimating the best value of the Lipschitz constant). See example in a later section. Note that the member variable $UseOtherBounds$ must be set to 1.

Useful member variables

Most member variables are declared as public, for easy inheritance, and access to these variables from the derived classes.

`double MaxLipConst`

The Lipschitz constant compatible with the data set.

`double g1,g2`

After calculating the value of the interpolant, *g1,g2* contain the lower and upper bounds respectively. The value of the interpolant is $\frac{g1+g2}{2}$

`int UseOtherBounds`

Flag indicating whether the range of $f(x)$ should be restricted by the additional upper and lower bounds. The user should set this variable to 1 and implement the virtual member functions `ExtraUpperBound()` and `ExtraLowerBound()` in the derived class. The bounds need not be constants, and the syntaxis is `ExtraUpperBound(int dim, double* x, double * param)`, where x is the point at which the bounds are required, and *param* is an optional parameter passed to this function (currently the current value of the Lipschitz constant is passed). See section 2.4.6 on additional bounds.

3.4 Members of `SLipIntInf` class.

This class is derived from `SLipIntBasic`, and shares most methods with `SLipInt`. The difference between the two classes is that `SLipIntInf` uses l_∞-norm and not Euclidean distance in the Lipschitz condition. For low dimension ($d < 5$) this may bring a computational advantage when smoothing the data, as the number of constraints in the LP problem can be reduced. This class also implements a smoothing method in the simplicial distance, which is useful in conjunction with the evaluation methods of `STCInterpolant` class.

This section only details the methods which are different from those of `SLipInt`. Refer to the description of `SLipInt` class for information on the other methods.

`void SmoothLipschitzSimp(int dim, int N, double* X, double* Y, double* T, double LC);`

Computes the vector T of the modified (smoothened) data values, consistent with the specified Lipschitz constant LC. The Lipschitz condition is understood in the simplicial distance (i.e., when $d()$ is simplicial distance in (2.2)). 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 T should be provided in the calling routine.

`void SmoothLipschitzSimpW(int dim, int N, double* X, double* Y, double* T, double LC, double* W);`

Variation of the previous procedure, with the vector of weights W reflecting relative accuracy of the data.

3.5 Members of `SLipIntLp` class.

This class is derived from `SLipInt`, and allows one to execute all the methods described above. The only difference is that it requires specification of the parameter p of the l_p-norm, which is done in the methods

`void SetP( double p)`	Sets the value of the parameter p.
`double GetP( double p)`	Returns the value of the parameter p.

3.6 Members of STCInterpolant class.

This class implements explicit and fast evaluation of the interpolant in the simplicial distance. This method requires a preprocessing step. The first call should be `SetData` and then the preprocessing routine should be called in `Construct` or `ConstructExplicit`. Do not use `Construct` for dimension $dim > 5$, as it becomes computationally very expensive.

`void SetData(int dim, int N, 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 $N \times dim$ that contains values x_i^k in its rows, and `y` is an array of size N which contains the values y^k. N 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.**

`void SetDataColumn(int dim, int N, double* x, double* y, int test=0)`
The same as above, but assumes the data are stored in columns (like in Fortran). Should be called instead of `SetData`.

`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 $O(dN^2)$ and should be avoided if there are other means to estimate the Lipschitz constant. Returns the computed value of the Lipschitz constant.

`void SetConstants(double newconst)`
`void 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.

`void Construct()`
Performs the preprocessing step, necessary for the subsequent evaluation by the fast method. This method may require significant computational effort. **Should not be called after** `ConstructExplicit()`**.**

`void 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 $d + 1$, 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 d (the algorithm will automatically compute the required slack variable). The user may precompute the slack variable himself, and specify dim as $d + 1$, which is marginally faster. Parameter x is an array of size dim. Returns $g(x)$.

`void 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 `mviInterpolant` 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 are: 1 – lower interpolation failed, 2– upper interpolation failed, 3 – both failed, 10 – Lipschitz constant too small, or repeated data.

3.7 Procedural interface

The following procedures provide procedural interface to the members of classes `SLipInt` `SLipIntInf` and `STCInterpolant`. It is useful when calling `LibLip` from procedural languages (like Fortran) or other packages (like Matlab, Mathematica). Note that all parameters are passed by reference, as this may be required by such languages.
Not all the methods of `SLipInt` and `SLipIntInf` classes have procedural interface. There is no interface for specifying additional bounds (`ExtraUpperBound`, `ExtraLowerBound` methods), nor interface to the members of `SLipIntLp` class.

Interface to the members of SLipInt class

`double LipIntValue(int* Dim, int* N, double* x, double* X, double* Y, double* LC, int* Index=NULL);`
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 size N, used to index the data in a large data set, which are used in the construction of the interpolant. For example, index[0]=1, index[1]=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.

`double LipIntValueAuto(int* Dim, int* N, double* x, double* X, double* Y, int* Index=NULL);`
Variation of the above, uses Lipschitz constants automaticlly identified from the data. **Should only be called after** `LipIntComputeLipschitz()`, **or** `LipIntComputeLipschitzCV()`, **or** `LipIntComputeLipschitzSplit()`.

`double LipIntValueCons(int* Dim, int* N, int* Cons, double* x, double* X, double* Y, double* LC, int* Index=NULL);`
Same as `LipIntValue()`, 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.

`double LipIntValueConsLeftRegion(int* Dim, int* N, int* Cons, double* x, double* X, double* Y, double* LC, double* Region, int* Index=NULL);`

Same as `LipIntValueCons()`, for monotone functions in the region $x \preceq LeftRegion$. $LeftRegion$ is a vector of size dim denoting the top right corner of the region of monotonicity.

`double LipIntValueConsRightRegion(int* Dim, int* N, int* Cons, double* x, double* X, double* Y, double *LC, double* Region, int* Index=NULL);`
Same as `LipIntValueConsLeftRegion()`, for monotone functions in the region $x \succeq RightRegion$. $RightRegion$ is a vector of size dim denoting the bottom left corner of the region of monotonicity.

`double LipIntValueLocal(int* Dim, int* N, double* x, double* X, double* Y);`
Variation of the above for locally Lipschitz functions, uses Lipschitz constants dependent on the position x, automaticlly identified from the data. **Should only be called after** `LipIntComputeLocalLipschitz()`.

`double LipIntValueLocalCons(int* Dim, int* N,int* Cons, double* x, double* X, double* Y);`
Same as `LipIntValueLocal()`, but for monotone functions. $Cons$ is an array of size dim specifying monotonicity constraints, as in `LipIntValueCons()`.

`double LipIntValueLocalConsLeftRegion(int* Dim, int* N, int* Cons, double* x, double* X, double* Y, double* Region);`
Same as `LipIntValueLocalCons()`, for monotone functions in the region $x \preceq LeftRegion$. $LeftRegion$ is a vector of size dim denoting the top right corner of the region of monotonicity.

`double LipIntValueLocalConsRightRegion(int* Dim, int* N,int* Cons, double* x, double* X, double* Y, double* Region);`
Same as `LipIntValueLocalConsLeftRegion()`, for monotone functions in the region $x \succeq RightRegion$. $RightRegion$ is a vector of size dim denoting the bottom left corner of the region of monotonicity.

`void LipIntComputeLipschitz(int* Dim, int* N, double* X, double* 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 using `LipIntGetLipConst()` .

`void LipIntComputeLipschitzSplit(int* Dim, int* N, double* X, double* Y, double* T, double* ratio, int* type, int* Cons=NULL, double* Region=NULL, double *W=NULL);`
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 using `LipIntGetLipConst()` . 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 $\mathcal{D}_1$.

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 be set (as in `LipIntValueConsLeftRegion()`. If the accuracy of the data is not the same, the vector of non-negative weights should be provided, as in `LipIntSmoothLipschitz()`.

`void LipIntComputeLipschitzCV(int* Dim, int* N, double* X, double* Y, double* T, int* type, int* Cons=NULL, double* Region=NULL, double *W=NULL);`

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 using `LipIntGetLipConst()`. The parameters have the same meaning as in `LipIntComputeLipschitzSplit()`. 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.

`void LipIntSmoothLipschitz(int* Dim, int* N, double* Xd, double* Y, double* T, double* LC, int *fW, int *fC, int* fR, double* W=NULL, int* Cons=NULL, double* Region=NULL);`

Computes the vector T 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 T should be provided in the calling routine.

Optional parameters: flags fW, fC and fR indicate that the optional parameters must be set. if $fW = 1$, W should be a vector of size N of non-negative weights. Weights 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.

$fC = 1$ indicates monotone approximation. The vector $Cons$ of size dim contains information about 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.

$fR = 1$ indicates monotonicity in the region $x \preceq Region$. $fR = 2$ indicates monotonicity in the region $x \succeq Region$. In both cases $Region$ should be a vector of size dim.

`double LipIntGetLipConst() ;`

Returns the computed Lipschitz constant (after calling `LipIntComputeLipschitz()` type procedures.

`int LipIntComputeScaling(int* Dim, int* N, double* XData, double* YData);`
Computes the scaling factors necessary to normalize the data to have standard deviation one in each variable.

`void LipIntGetScaling(double *S) ;`
Returns in S the scaling factors necessary to normalize the data to have standard deviation one in each variable. Should be called after `LipIntComputeScaling()`.

`void ConvertXData(int* dim, int* N, double* X);`
Transposes the matrix X. Useful when the `LibLip` is called from other program/languages which store the matrices columnwise (as in Fortran, Matlab, etc.). `LibLip` uses C convention and stores data in rows. The transposed matrix is returned in X.

`void ConvertXData(int* dim, int* N, double* X, double* Aux);`
Transposes the matrix X. Same as the previous procedure, but the transposed X is returned in the array Aux. The memory for Aux should be allocated in the calling routine ($dim \times N$).

`int LipIntVerifyMonotonicity(int* dim, int* N, int* Cons, double* X, double* Y, double* LC, double* eps);`
Returns 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 $Cons$, $Cons[j] = 1$ means the function is increasing wrt j-th variable,$Cons[j] = -1$ 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 $y^k - y^i \geq eps$).

`int LipIntVerifyMonotonicityLeftRegion(int* dim, int* N, int* Cons, double* X, double* Y, double* Region, double* LC, double* eps);`

`int LipIntVerifyMonotonicityRightRegion(int* dim, int* N, int* Cons, double* X, double* Y, double* Region, double* LC, double* eps);`
Variations of `LipIntVerifyMonotonicity()`, where monotonicity condition is checked in the regions $x \preceq Region$, or $x \succeq Region$.

Interface to the members of SLipIntInf class

The procedures are exactly the same as those that provide interface to `SLipInt` class. The naming convention: use the prefix `LipIntInf` instead of `LipInt`, for

example `LipIntInfValue()`, etc.

There are two new procedures which perform smoothing in the simplicial distance

`void LipIntInfSmoothLipschitzSimp(int* dim, int* N, double* X, double* Y, double* T, double* LC);`

Computes the vector T of the modified (smoothened) data values, consistent with the specified Lipschitz constant LC. The Lipschitz condition is understood in the simplicial distance (i.e., when $d()$ is simplicial distance in (2.2)). 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 T should be provided in the calling routine.

`void LipIntInfSmoothLipschitzSimpW(int* dim, int* N, double* X, double* Y, double* T, double* LC, double* W);`

Variation of the previous procedure, with the vector of weights W reflecting relative accuracy of the data.

Interface to the members of STCInterpolant class

`void STCSetLipschitz(double* LC);`

Sets the Lipschitz constant. Must be called before `BuildLipInterpolant()` procedure.

`int STCBuildLipInterpolant(int* Dim, int* Ndata, double* x, double* y);`

Builds Lipschitz interpolant using the simplicial distance for subsequent fast evaluation. Parameters Dim - dimension of the data set $NData$ is the size of the data set, x contains the abscissae of the data, stored rowwise, y contains the values to be interpolated.

`int STCBuildLipInterpolantExplicit(int* Dim, int* Ndata, double* x, double* y);`

As above, but for explicit evaluation of the interpolant, uses very little preprocessing.

`int STCBuildLipInterpolantColumn(int* Dim, int* Ndata, double* x, double* y);`

As above, but the data in x are stored columnwise, like in Fortran.

`int STCBuildLipInterpolantExplicitColumn(int* Dim, int* Ndata, double* x, double* y);`

As above, but for explicit evaluation of the interpolant, uses very little preprocessing.

`double STCValue( double* x );`

Computes the value of the interpolant at any given point x, using fast method. Must be called after `STCBuildLipInterpolant()` procedure.

`double STCValueExplicit( double* x );`

Same but using explicit evaluation with little preprocessing. Must be called after `STCBuildLipInterpolantExplicit()` procedure.

`void STCFreeMemory();`

Frees memory structures occupied by the interpolant. No evaluation methods are to be called after `STCFreeMemory`. However, building of a new interpolant using `STCBuildLipInterpolant` is allowed.

Chapter 4

Examples of usage

4.1 Sample code

There are several examples of the usage of `LibLip` provided in the distribution. The best way to use `LibLip` is to declare an instance of the class `SLipInt`, `SLipIntLp` or `STCInterpolant` and call its members directly.

These classes are declared in `LibLipc.h`.

4.1.1 Interpolation

When using the class `SLipInt`, 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 `ComputeLocalLipschitz`.

When using the class `STCInterpolant`, there is a need for preprocessing. 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.

```
// example of usage of SLipInt class
#include "LibLipc.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;
   // can also use SLipIntInf 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.Value(dim,npts,x,XData, YData,LipConst);
// estimate Lipschitz constant
   LipInt.ComputeLipschitz(dim,npts,XData, YData);
// uses the computed Lipschitz constant
   w=LipInt.Value(dim,npts,x,XData, YData);

// the same using local Lipschitz constants
   LipInt.ComputeLocalLipschitz(dim,npts,XData, YData);
// calculate the value
   w=LipInt.ValueLocal(dim,npts,x,XData, YData);


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

```
// example of usage of SLipInt class for monotone interpolation
#include "LibLipc.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));
   int* Cons=(int*)malloc(dim*sizeof(int));
   double* Region=(double*)malloc(dim*sizeof(double));

   SLipInt LipInt;
 // can also use SLipIntInf LipInt; or SLipIntLp 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
    }

    Cons[0]=1; // function monotone incr. wrt first variable
    Cons[1]=-1; // function monotone decr. wrt first variable
    Cons[2]=0; // unrestricted wrt other variables
    Cons[3]=0; //
    for(j=0;j<dim;j++) Region[j]=1.5;
    for(j=0;j<dim;j++) x[j]=random(3.0,0); // some random x
// calculate the value
   w=LipInt.ValueCons(dim,npts,Cons,x,XData, YData,LipConst);
// assume monotonicity for x<<Region only
   w=LipInt.ValueConsLeftRegion(dim,npts,Cons,x,XData, YData,
        LipConst,Region);

   LipInt.ComputeLocalLipschitz(dim,npts,XData, YData);
   w=LipInt.ValueLocalCons(dim,npts,Cons,x,XData, YData);
   w=LipInt.ValueLocalConsLeftRegion(dim,npts,Cons,x,XData, YData,Region);
    free(XData); free(YData); free(x); free(Cons); free(Region);
}
```

```
// example of usage of SLipInt class with extra bounds
#include "LibLipc.h"
int dim=4;        // the dimension and size of the data set
int npts=1000;
 class MySLipInt: public SLipInt { // derived class
 public:
   virtual double  ExtraUpperBound(int dim, double* x, double * param)
    { double B;
     // compute some bound at x with the Lip. constant param
     B=*param * max(x[0],x[1]);
    return B;};
    virtual double  ExtraLowerBound(int dim, double* x, double * param)
    { double B  = *param * min(x[0],x[1]);
     // compute some other bound at x with the Lip. constant param
    return B;};
 }
 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));
   MySLipInt LipInt;
   LipInt.UseOtherBounds=1;
   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.Value(dim,npts,x,XData, YData,LipConst);
// estimate Lipschitz constant
   LipInt.ComputeLipschitz(dim,npts,XData, YData);
// uses the computed Lipschitz constant
   w=LipInt.Value(dim,npts,x,XData, YData);

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

```
// example of usage of STCInterpolant class
#include "LibLipc.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
    }
// supply 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
}
```

```
// an example using procedural interface
#include "LibLip.h"

int dim=4;              // the dimension and the data set
int npts=1000;

void main(int argc, char *argv[]){
// arrays to store the data
  double LipConst=10;
   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));

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

// compute the Lipschitz constant in max-norm
   LipIntInfComputeLipschitz(&dim,&npts, XData, YData);
// calculate the value
   w=LipIntInfValue(&dim,&npts,x,XData, YData);
// the same in Euclidean norm, but using local Lipschitz values
   LipIntComputeLocalLipschitz(&dim,&npts, XData, YData);
// calculate the value
   w=LipIntValueLocal(&dim,&npts,x,XData, YData);

// now using fast method and simplicial distance
   STCSetLipschitz(&LipConst);  // supply Lipschitz constant
// suppy the data
   STCBuildLipInterpolant(&dim,&npts,XData,YData);
   w=STCValue(x); // calculate the value
}
```

4.1.2 Smoothing

LibLip implements smoothing of the data, presumably given with some noise, subject to the required Lipschitz condition. It makes the data consistent with a given Lipschitz constant, by adjusting the data values. The l_1-norm of the changes to the data is minimized using linear programming.

As a result, the user obtains a modified data set, consistent with the required Lipschitz condition (as well as specified monotonicity constraints, if any). These data can be subsequently interpolated.

```
#include "LibLipc.h"
int dim=4;        // the dimension and size of the data set
int npts=200;

void main(int argc, char *argv[]){
   double LipConst=4;
   double *x, *XData, *YData, *TData;
// arrays to store the data
   x=(double*)malloc(dim*sizeof(double));
   XData=(double*)malloc(dim*npts*sizeof(double));
   YData=(double*)malloc(npts*sizeof(double));
   TData=(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)+ 0.1*random(-1.0,1.0); // noisy function values
    }

  LipInt.SmoothLipschitz(dim, npts,XData,YData,TData,LipConst);
// other possibilities:
//  LipInt.SmoothLipschitzCons(dim, npts,Cons, XData,YData,TData,LipConst);
//  LipInt.SmoothLipschitzW(dim, npts,XData,YData,TData,LipConst,W);
//  LipInt.ComputeLipschitzCV(dim, npts,XData,YData,TData);
//  etc...

   for(j=0;j<dim;j++) x[j]=random(3.0,0); // some random x
// calculate the approximation at x
```

```
   w=LipInt.Value(dim,npts,x,XData, TData, LipConst);

// prepare data for the fast method using simplicial distance
   SLipIntInf LipIntInf;
   STCInterpolant STCLipInt;
   LipIntInf.SmoothLipschitzSimp(dim, npts,XData,YData,TData,LipConst);

   STCLipInt.SetData(dim,npts, XData,TData);
   STCLipInt.SetConstants(LipConst);  // supply Lipschitz constant
   STCLipInt.Construct();     // construct the interpolant

   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=STCLipInt.Value(dim,x1); // calculate the value

 }
```

```
// example of usage of STCInterpolant class and smoothened data
#include "LibLipc.h"
int dim=3;              // the dimension and size of the data set
int npts=1000;

void main(int argc, char *argv[]){
   double LipConst=2.5;
   double *x, *XData, *YData, *TData;
// arrays to store the data
   x=(double*)malloc(dim*sizeof(double));
   XData=(double*)malloc(dim*npts*sizeof(double));
   YData=(double*)malloc(npts*sizeof(double));
   TData=(double*)malloc(npts*sizeof(double));

   STCInterpolant LipInt;
   SLipIntInf LipIntInf;
   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
    }
// smoothen the data
    LipIntInf.SmoothLipschitzSimp(dim,npts,XData,YData,TData,LipConst);
// supply the smoothened data (TData, not YData)
   LipInt.SetData(dim,npts, XData,TData,0);
   LipInt.SetConstants(LipConst);  // supply Lipschitz constant
   LipInt.Construct();          // construct the interpolant
   free(XData); free(YData); free(TData);// 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
   LipInt.FreeMemory();  // destroys the interpolant
}
```

4.2 Linking

To link against `LibLip` library use options `-llip -glpk` . It is assumed you have installed the additional package `glpk`, version 4.8. See example makefiles.

If you downloaded the precompiled version of `LibLip`, this package will contain precompiled `glpk` library.

Note for Windows users: On Windows platform, the binaries (the *.lib* and *.dll* files) incorporate `glpk`. `LibLip` is distributed as a DLL file and the corresponding LIB file called `liblipdll.dll` and `liblipdll.lib`. In your project settings, choose LINK option and add `liblipdll.lib`. Ensure that the `liblipdll.dll` file is in the same directory as your main program, or on the path. See the `readme` file coming with your distribution.

`liblipdll.dll` is compiled using `__stdcall` option. The examples and user's calling programs should be compiled with this option as well. In Visual C++ compilers `__stdcall` **is not** the default option. The user needs to manually change the project settings (in `Project->Settings->C/C++->Code generation` to from Microsoft-specific `__cdecl` to standard `__stdcall` calling convention. The same can be done by using flag `\Gz` in the make file.

4.3 Fortran interface

It is possible to call subroutines from `LibLip` library from FORTRAN. There are some programming tricks however, necessary for calling C/C++ functions from Fortran.

Firstly, by default Fortran compilers automatically add an underscore "_" at the end of function names, this can be disabled with a compilation switch (e.g.,`-fno-underscoring` in g77) compilation flag in your make file, see examples.

Secondly, Fortran passes all parameters by reference, not by value. Thirdly, linking a fortran program with C++ library sometimes requires a wrapper, which makes the main program being declared in C and not in Fortran code.

4.4 Tips

It is a common problem when the user incorrectly sets the Lipschitz constant for his/her particular case. The value `LipConst=10` 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 class `SLipInt` will accept a lower value of the Lipschitz constant, but in this case the data will not be interpolated. This may help to smooth the data (to filter out some outliers), but for proper smoothing, the relevant smoothing methods should be used (see the description of the class and examples).

The class `STCInterpolant` 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.

The user is advised to check the value of `LastError()` member function after calling `Construct()`. Nonzero error code will indicate too low value of the Lipschitz constant, incompatible with the data.

If the value of the Lipschitz constant is unknown, the user is advised to compute it from the data using `ComputeLipschitz()`, or `ComputeLipschitzSplit()`, or `ComputeLipschitzCV()`.

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 `ComputeLocalLipschitz..()` and `ValueLocal..()` 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. An option to remove repeated data could be used in the `SetData()` method.

4.5 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 classes `SLipInt` and `SLipIntInf` 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 $d = 2, \ldots, 5$. The row with the largest N corresponds to approximately 500MB of RAM requested by `LibLip`.

In case of a larger data set or higher dimension, we advise to use the class `SLibInt`, or perform explicit evaluation by calling `STCInterpolant::ConstructExplicit`.

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

Where to get help

The software library `LibLip` and its components, are distributed by G.Beliakov AS IS, with no warranty, explicit or implied, of merchantability or fitness for a particular purpose. G.Beliakov will provide limited technical support for a period of 60 days after purchase, by electronic media. G.Beliakov, at its 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`.