DETAILS OF CHEBPOL

SIMEN GAURE

ABSTRACT. **chebpol** is a package for multivariate interpolation using Chebyshev-polynomials. Interpolation means that the approximating function it produces matches the original function in prespecified points, and try to fill in between the gaps. Thus, it is not smoothing like the **mgcv** package. Indeed, the package also contains some other multivariate interpolation methods. This document outlines how **chebpol** works.

1. Introduction

We consider the problem of interpolating a continuous function $f : [-1, 1] \mapsto \mathbb{R}$ based on its values in n points $\{x_i\}_{i=1..n}$ called *knots*. I.e. we want to find a reasonably behaved function P_f^n defined on $[-1, 1]$ such that $P_f^n(x_i) = f(x_i)$ for $i = 1..n$.

A classical approach is to let P_f^n be a polynomial of degree $n - 1$ and find the coefficients by solving the linear system $P_f^n(x_i) = f(x_i)$ $(i = 1..n)$. However, high-degree polynomials do not always behave *reasonably*. In particular there is the Runge phenomenon: If the points x_i are uniformly spaced in $[-1, 1]$ and one tries to interpolate the Runge function $f : x \mapsto (1 + 25x^2)^{-1}$, there will be oscillations in P_f^n near the end-points. As n grows, the amplitude of these oscillations grow without bounds.

2. Chebyshev interpolation

The classical solution to the Runge phenomenon is to use a particular set of knots, the Chebyshev knots $x_i^n = \cos(\pi(i - 0.5)/n)$ for $i = 1..n$. This will ensure that P_f^n will converge uniformly to f as $n \to \infty$, provided f is uniformly continuous. In this case one uses a special basis for the space of polynomials of degree up to $n - 1$, the Chebyshev polynomials $\{T_i(x)\}_{i=0..n-1}$, which are orthogonal w.r.t to a suitable inner product. Due to a trigonometric identity, we have $T_i(x) = \cos(i \cos^{-1}(x))$. The coefficients for these polynomials may be computed by a variant of the DCT-II transform. Thus, the method is fast.

More modern methods use splines, where the idea is to glue pieces of simple functions together, either low-degree polynomials or rational functions, subject to various constraints. Splines are in many respects superior to Chebyshev-interpolation.

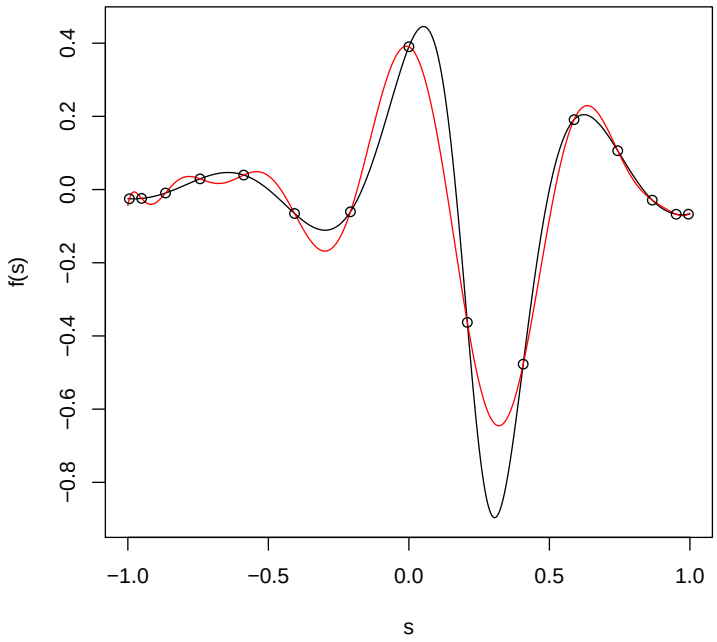
For the multivariate case, where $f : [-1, 1]^r \to \mathbb{R}$, the DCT-II transform, being a variant of the Fourier transform, factors over tensor-products, so a natural choice is to use this tensor-product transform in the multivariate case. The knots are the Cartesian product of one-dimensional knots. This is a classical way to interpolate multivariate real functions.

Date: February 25, 2013.

This procedure is available in package **chebpol** in two variants. One for the case that $f(x_i^n)$ is known for each $i = 1..n$, this is the function `chebappx`. For the case that the function f is available, not only its values in the knots, we have the function `chebappxf`. This is a short wrapper for `chebappx` which simply evaluates f in the knots.

Thus, given a function f we may compute its Chebyshev approximation and plot it:

```
> f <- function(x) cos(3*pi*x)/(1+25*(x-0.25)^2)
> ch <- chebappxf(f,15)
> s <- seq(-1,1,length.out=401)
> plot(s,f(s),type='l')
> lines(s,ch(s), col='red')
> kn <- chebknots(15)[[1]]
> points(kn,f(kn))
```



Even though there still are oscillations near the end points, their amplitude will diminish as n grows. The knots are the locations where the curves intersect.

In the multivariate case, the `dims` argument to `chebappxf` is a vector of integers, the number of knots in each dimension. The `chebappx` variant instead uses the `dim` attribute of its input to determine this.

3. Uniformly spaced grids

In some applications it is not feasible to evaluate the function f in the Chebyshev knots. Rather it may have to be evaluated on a uniformly spaced grid. To avoid the Runge phenomenon, **chebpol** transforms the domain of the function as follows.

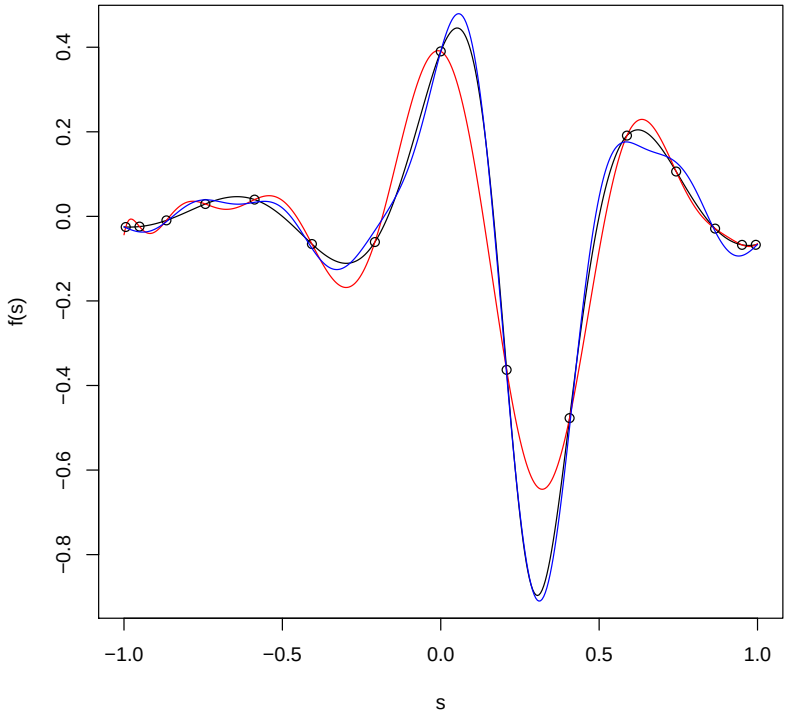
A fairly simple function exists which monotonically maps uniform grid points into Chebyshev knots. It's $g : x \mapsto \sin\left(\frac{\pi x(1-n)}{2n}\right)$. We omit the dependence on n to avoid clutter. This g has the property that for Chebyshev knots x_i^n and a uniform grid $y_i^n = -1 + 2(i-1)/(n-1)$ we have $g(y_i^n) = x_i^n$ for $i = 1..n$.

Thus, given a function f to interpolate on a uniform grid, we construct the function $h : x \mapsto f(g^{-1}(x))$. We then create the Chebyshev approximation P_h^n which requires h to be evaluated in the Chebyshev knots x_i^n, but $g^{-1}(x_i^n) = y_i^n$ are the uniform grid points, so f is evaluated there. We then use the function $Q^n : x \mapsto P_h^n(g(x))$ for interpolation.

It is readily verified that for $i = 1..n$ we have $Q^n(y_i^n) = f(y_i^n)$, thus Q^n agrees with the function f on the uniform grid. This is no longer a polynomial interpolation, so the Runge phenomenon is not necessarily present.

This procedure is available in the function `ucappx`, with a function variant in `ucappxf`. Continuing the former example, we have

```
> uc <- ucappxf(f,15)
> lines(s,uc(s),col='blue')
```



For the multivariate case, `ucappx` creates a separate map function `g` for each dimension, which may have different number of knots. One could imagine a case where some dimensions are evaluated on Chebyshev grids, whereas other dimensions are evaluated on uniform grids. Although **chebpol** has no ready-made wrapper function to do this, it is not particularly difficult to achieve.

4. Non-standard hypercubes

Often, the domain of the function is not $[-1,1]$, but rather some other interval $[a,b]$. This interval may be affinely mapped onto $[-1,1]$ by $x \mapsto (2x-(a+b))/(b-a)$. For the multidimensional case this may be done for each dimension separately. The functions in **chebpol** have an optional `intervals` argument, a list of such intervals, one for each dimension, to support such hyper-rectangles.

In principle, the same could be done with infinite intervals, with e.g. a mapping $(-\infty,\infty) \mapsto (-1,1)$ like $x \mapsto \frac{2}{\pi}\tan^{-1}(x)$, but **chebpol** does not implement it.

4.1. **A multivariate example.** Let f be the function $f : (x,y) \mapsto \log(x)\sqrt{y}/\log(x+y)$ defined on $[1,2]\times[15,20]$. Let's approximate it with 5 knots in x and 8 in y and see how it fares in a random point:

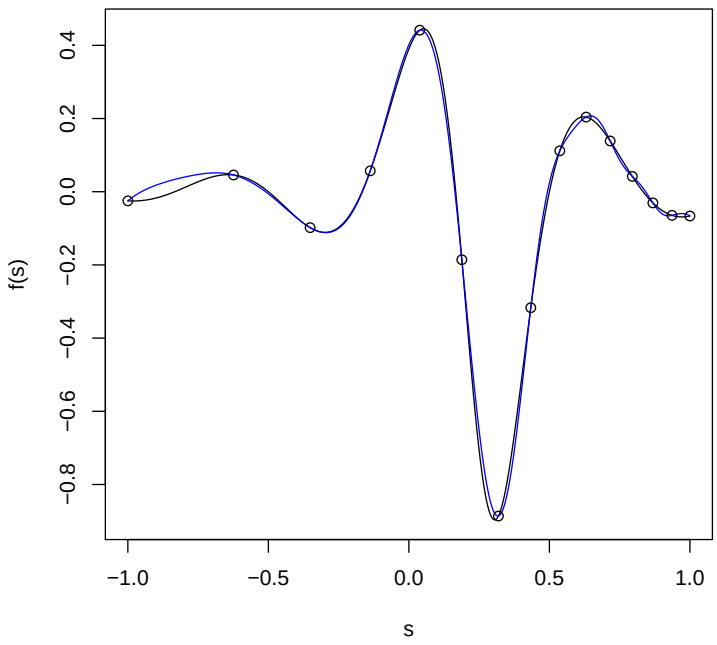
```
> library(chebpol)
> f <- function(x) log(x[[1]])*sqrt(x[[2]])/log(sum(x))
> ch <- chebappxf(f, c(5,8), list(c(1,2), c(15,20)))
> uc <- ucappxf(f, c(5,8), list(c(1,2), c(15,20)))
> tp <- c(runif(1,1,2), runif(1,15,20))
> cat('arg:',tp,'true:', f(tp), 'ch:', ch(tp), 'uc:',uc(tp),'\n')

arg: 1.627951 16.40075 true: 0.6824249 ch: 0.6823502 uc: 0.6751227
```

5. Arbitrarily spaced grids

In some applications not even uniformly spaced grids are feasible. In this case we do something similar as in the uniformly spaced case. We have grid-points $\{y_i\}_{i=1..n}$ in ascending or descending order. We then use `splinefun` in package **stats** with `method='monoH.FC'` to create a monotone function g from the grid-points $\{y_i\}_{i=1..n}$ to the Chebyshev knots $\{x_i^n\}_{i=1..n}$. Otherwise, the same method as for uniform grids are used.

The function `chebappxg` performs this procedure. With an accompanying function variant `chebappxgf`. In the multivariate case, it is assumed that the grid in each dimension is arbitrary, but the multi-dimensional grid is still a Cartesian product of these. We may test this on a non-uniform grid.

```
> f <- function(x) cos(3*pi*x)/(1+25*(x-0.25)^2)
> gr <- log(seq(exp(-1),exp(1),length=15))
> chg <- chebappxgf(f,gr)
> plot(s, f(s), col='black', type='l')
> lines(s, chg(s), col='blue')
> points(gr,f(gr))
```

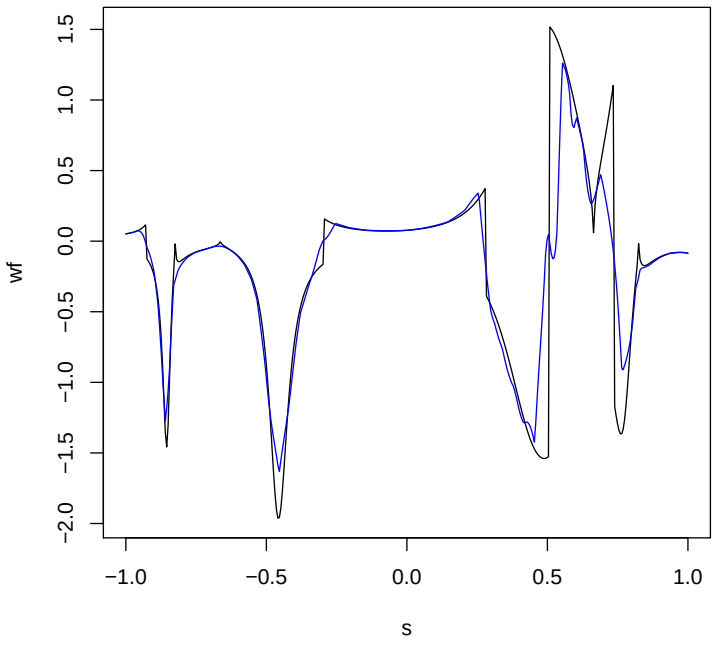


6. Multilinear interpolation

For demanding functions **chebpol** also contains a multilinear interpolation in `mlappx`. A multilinear function is a function $f(x_1, x_2, ..., x_n)$ such that for each $i = 1..n$ the function $g_i(x) = f(x_1, ..., x_{i-1}, x, x_{i+1}, ..., x_n)$ is of the form $ax + b$ for every choice of the x_is. A typical example is $f(x, y, z) = xyz + xy + 2yz - x + 2$. `mlappx` may take either a function or function values as its first argument and create a piecewise multilinear interpolation. It is a straightforward implementation which interpolates a point by a convex combination of the function values in the corners of the surrounding hypercube. We try it out on an intricate function on $[-1, 1]^4$ and evaluate the result along an intricate parametric curve. Of course, if one is interested in this particular curve, one could interpolate along it instead. This is just an example of multilinear interpolation.

```
> f <- function(x) sign(sum(x^3)-0.1)*
+                    sqrt(abs(25*prod(x)-4))/
+                    (1+25*sum(x)^2)
> grid <- replicate(4,list(seq(-1,1,length=15)))
> ml <- mlappx(f,grid)
> s <- seq(-1,1,length=400)
> curve <- function(x) c(cos(1.2*pi*x),
+                        sin(1.5*pi*x^3),
+                        x^2, -x/(1+x^2))
> wf <- sapply(s,function(x) f(curve(x)))
> wml <- sapply(s,function(x) ml(curve(x)))
```

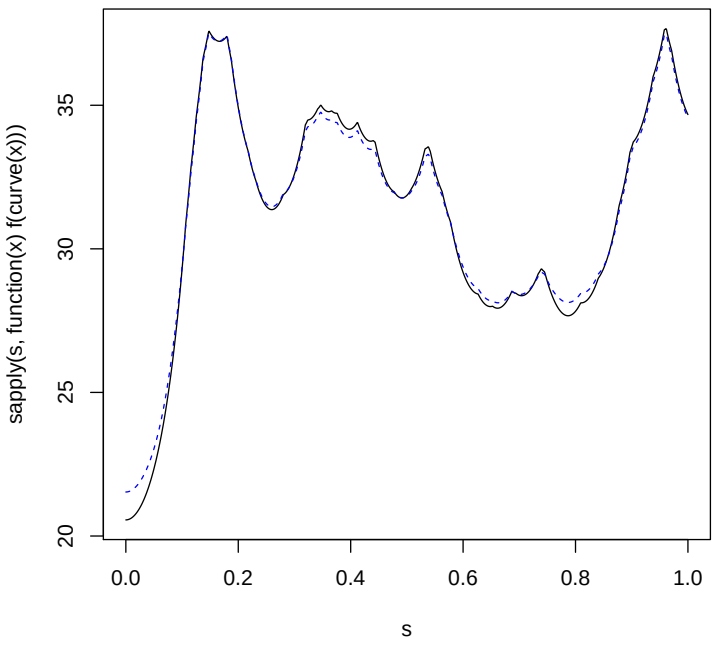
```
> plot(s,wf,typ='l')  # function
> lines(s,wml,col='blue') # multilinear interpolation
```



7. Scattered data

In some cases multidimensional data points are not organized in a grid, rather they are scattered. For this case, there is an experimental implementation of polyharmonic splines in `polyh`. It accepts a function, or function values, and a matrix of centres (knots), one centre in each column. Here is a 20-dimensional example with 3000 randomly placed knots:

```
> r <- runif(20)
> r <- r/sum(r)
> f <- function(x) 1/mean(log1p(r*x))
> knots <- matrix(runif(60000), 20)
> phs <- polyh(f, knots, 3)
> rr <- runif(20)
> curve <- function(x) abs(cos(5*pi*rr*x))
> s <- seq(0,1,length.out=1000)
> plot(s,sapply(s,function(x) f(curve(x))),typ='l')
> lines(s,sapply(s,function(x) phs(curve(x))),col='blue',lty=2)
```



`polyh(f,knots,k)` fits a function of the form $P(x) = \sum_{i=1}^{n} w_i \phi(\|x - c_i\|) + L(x) + c$ such that $P(c_i) = f(c_i)$ for each i where the $c_i \in \mathbb{R}^d$ are the knots, $\|\cdot\|$ is the Euclidean norm on $\mathbb{R}^d$, $w_i \in \mathbb{R}$ are weights, L is linear $\mathbb{R}^d \mapsto \mathbb{R}$, and $c \in \mathbb{R}$ is a constant. ϕ is the function $\mathbb{R}^+ \mapsto \mathbb{R}$

$$\phi(x) = \begin{cases} x^k & \text{when } k \in \mathbb{N} \text{ is odd} \\ x^k \log(x) & \text{when } k \in \mathbb{N} \text{ is even} \\ \exp(kx^2) & \text{when } k < 0 \end{cases}$$

Note that this differs from some other expositions, which use $2m - d$ as exponent for a natural number m.

For $k = 2$, we get the thin plate spline. Note that the fitting may fail, in particular for $k < 0$ and irregular data. In this case a least squares fit is used, and a warning is issued. The k parameter then may need to be tuned for the problem at hand. Note that `polyh` by default transforms the knots and coordinates into a unit hypercube if any of the knots are outside, see the help entry for `polyh`.

There do exist some acceleration schemes for low dimensions, but the current implementation does not use any. It is an entirely straightforward implementation which fits by solving a linear system, and evaluates the $P(x)$ directly.

Ragnar Frisch Centre for Economic Research, Oslo, Norway