# MATLAB Programming for Kernel–Based Methods

Maryam Mohammadi[a,*], Robert Schaback[b]

[a]Faculty of Mathematical Sciences and Computer, Kharazmi University, Tehran, Iran
[b]Institut für Numerische und Angewandte Mathematik, Georg-August-Universität Göttingen, Germany

**Abstract**

This paper presents an implementation of kernel–based methods using MATLAB, a powerful tool for numerical computation and data analysis. Kernel methods are pivotal in various fields of Numerical Analysis, including approximation, interpolation, meshless methods for solving partial differential equations (PDEs), neural networks, and Machine Learning. Due to the connection of kernels to Hilbert spaces of functions, kernel-based methods often have optimality properties. The package allows to switch between different kernels and different scales without changing programs, just by changing parameters. In addition, it enables applications to partial differential equations to handle derivatives of kernels efficiently, without re-programming the derivatives if the kernel is changed. Special emphasis is placed on practical implementation, showcasing MATLAB code snippets and functions that facilitate the application of these algorithms to function approximation and numerical solution of PDEs.

*Keywords:* Kernel, MATLAB programming, Function approximation

## 1. Introduction

This paper addresses researchers and practitioners seeking to leverage kernel techniques in their computational tasks. The focus is on *radial* kernels (*radial basis functions, RBFs*)

$$K(x,y) = \phi(\|x-y\|), x,y \in \mathbb{R}^d,$$

where $\|\cdot\|$ is the Euclidean norm. The kernels covered are

- Gaussian $\phi(r) = \exp(-r^2)$
- Multiquadrics $\phi_\beta(r) = (1+r^2)^\beta$ with $\beta \in \mathbb{R} \setminus \mathbb{N}_0$
- Powers $\phi_\beta = r^\beta$, $0 < \beta \notin 2\mathbb{N}$
- Thin-plate splines $\phi_n(r) = r^{2n} \log(r)$, $n \in \mathbb{N}$ [12]
- Matérn kernels $\phi_\nu(r) = r^\nu K_\nu(r)$, $\nu > d/2$ [39]
- Wendland $C^{2m}$ kernels $\phi_{3,m}(r)$, $m \geq 1$ [55]
- Hyperbolic secant $\phi(r) = \text{sech}(r)$ [9, 16]

*Corresponding author

*Email addresses:* `m.mohammadi@khu.ac.ir` (Maryam Mohammadi), `schaback@math.uni-goettingen.de` (Robert Schaback)

- “rtanh” kernel $\phi(r) = r \cdot \tanh(r/\beta)$ [19, 20]

Its basic idea comes from a rather informal technical note [45] dating back to 2009 and intended for the second author’s students. For a wider audience, this paper provides a thorough revision and completion. It also was the foundation of the package `mFDlab` on meshless finite differences by Oleg Davydov [4], and this paper will serve to explain its basics as well.

A standard amount of MATLAB knowledge is assumed everywhere, but there is no attempt to use advanced concepts or to combine the routines into a larger system with a GUI, in order to make the routines usable in a context-free way. Also, we do not even try to reach Nick Trefethen’s mastership of MATLAB as in his `chebfun` system (see `http://www.chebfun.org/`). Note that the books [13, 14] by Greg Fasshauer and Michael McCourt contain a very good and competitive collection of MATLAB programs. Mathematical background information on kernels can be obtained from books [2, 13, 14, 57] and from lecture notes like [43]. Applications are surveyed in [50], and many useful references can be obtained from the authors’ research papers at
`https://num.math.uni-goettingen.de/schaback/research/group.html`
and
`https://khu.ac.ir/cv/222/Maryam-Mohammadi`
with many references cited therein.

Technically, the MATLAB m-files are in a zipfile obtainable from the URL
`https://num.math.uni-goettingen.de/schaback/research/MPfKBM.zip`.
An included routine `dothemall.m` generates the figures of this text right from running the programs.

The paper is organized as follows. Section 2 first treats point sets and distance matrices, before kernels appear in Section 3. The basic kernel routine `frbf.m` is described in Section 3.3, but it needs to write radial kernels in $f$-form, described in Section 3.2 to facilitate taking derivatives of all possible orders using just a single routine. The mathematics behind this is treated in detail in Section 3.5. Then Section 4 provides first examples, specialized to interpolation and approximation. It includes Lagrange and Newton bases and uses the latter for adaptive matrix-free implementations of $P$-greedy and $f$-greedy methods. So far, this does not take derivatives. Section 5 explains various prefabricated routines that apply differential operators to kernels, acting directly on “$S$-matrices” that implement the $f$-form from Section 3.2. Finally, Section 6 demonstrates how to use these routines for solving partial differential equations. It covers collocation in symmetric and unsymmetric form, the Method of Lines, and shows how to calculate finite-difference approximations for use in RBF-FD and similar methods.

## 2. Point Sets and Distance Matrices

Before we come to kernels, we describe how the package handles geometry.

### *2.1. Point Sets*

Sets of $M$ points from $\mathbb{R}^d$ are stored in MATLAB/FORTRAN–style $M$ as rows of matrices with $d$ columns, e.g. $x_1, \ldots, x_M \in \mathbb{R}^d$ as rows of a matrix $X \in \mathbb{R}^{M \times d}$. Note that MATLAB runs through arrays in a columnwise (“column-major”, “first down, then across”) way, like FORTRAN. Thus dramatically unsymmetric arrays should always be stored such that there are more rows than columns, to support localized memory access. For univariate cases, note that the MATLAB command `t=-1:0.01:1` generates a row, not a column.

Uniformly distributed random sets of $M$ points in $d$ dimensions within $[0,1]^d$ are generated via `X=rand(M,d)`. To produce uniformly distributed random points in $[a,b]^d$, use `X=a+(b-a)*rand(M,d)`. The result is a matrix $X \in \mathbb{R}^{M\times d}$.

Regular grids are generated by the `meshgrid` command. The standard 2D case is

```
    [X Y]=meshgrid(a:h:b,a:h:b);
```

for generating points in $[a,b]^2$ with spacing $h > 0$. To let the spacing precisely fit, one should use something like `h=(b-a)/(n-1)` to get an $n\times n$ grid. But these arrays $X$ and $Y$ are not points in our matrix convention. Both `X` and `Y` are matrices of the same shape, with identical columns or rows. If the grid has a total of $n^2$ points, both `X` and `Y` are $n\times n$ matrices, storing the $x$ and $y$ coordinates, respectively. Use

```
P=[X(:) Y(:)];
```

to get a point matrix `P` of $n^2$ points and rows with two columns. The inverse operation is

```
  X=reshape(P(:,1),size(X));
  Y=reshape(P(:,2),size(Y));
```

to bring the point coordinates back into the correct order. In particular, if `Z` is a column vector of values at the points `P`, one often needs to reshape it by

```
  ZR=reshape(Z,size(X));
```

to the shape of `X` or `Y`.

## 2.2. *Distance Matrices*

Kernels are often applied to point distances arranged into *distance matrices*. In standard MATLAB, it is a crime to use avoidable loops, and thus we aim at kernel evaluation on distance matrices, not on single point distances.

If there are $M$ points for the $x$ argument and $N$ points for the $y$ argument of a radial kernel $K(x,y) = \phi(\|x-y\|)$, we have two input point matrices $X \in \mathbb{R}^{M\times d}$ and $Y \in \mathbb{R}^{N\times d}$ and we want to calculate the $M\times N$ *kernel matrix* $A$ with entries

$$A_{ij} = \phi\left(\|x_i - y_j\|\right),\ 1 \le i \le M,\ 1 \le j \le N,$$

based on the distance matrix $D$ with entries

$$D_{ij} := \|x_i - y_j\|,\ 1 \le i \le M,\ 1 \le j \le N.$$

MATLAB offers the command `dmat=pdist2(X,Y)` to calculate a distance matrix. For good reasons to be explained below, we shall mostly work with matrices consisting of

$$S_{ij} := \|x_i - y_j\|^2/2,\ 1 \le i \le M,\ 1 \le j \le N$$

called *S-matrices* from now on. Later routines will nearly always act on $S$-matrices. They can be calculated via `smat=(dmat.^2)/2`. But distance matrices contain a square root, and therefore going to an $S$-matrix via these commands takes a square root and a square. To avoid this, one can use the formula

$$\|x_i - y_j\|^2/2 = \|x_i\|^2/2 + \|y_j\|^2/2 - x_i^T y_j,\ 1 \le i \le M,\ 1 \le j \le N.$$

To work directly on the point matrices $X \in \mathbb{R}^{M\times d}$ and $Y \in \mathbb{R}^{N\times d}$, we see that the third summand is the matrix $-XY^T$, and the first two summands are just vectors that we have to repeat either columnwise or row-wise to get a matrix.

Here is an m-file `distsqh.m` (memo for distance squared and halved) implementing the above formula, without any loops and with complexity $\mathcal{O}(MNd)$:

```
function smat=distsqh(X, Y)
% calculates an M*N S-matrix smat
% of halved squares of point distances
% between two point matrices  X and Y of M and N points each.
[M Xdim]=size(X);
[N Ydim]=size(Y);
if Xdim~=Ydim
    error('Point sets in distsqh are of unequal dimension')
end
% the one-dimensional case gets a simpler treatment
if Xdim==1
    smat=(repmat(X,1,N)-repmat(Y',M,1)).^2/2;
else
    smat=X*Y';
    cX=sum((X.*X)')/2; % squared norms of X points, row
    cY=sum((Y.*Y)')/2; % squared norms of Y points, row
    smat=repmat(cX',1,N)+repmat(cY,M,1)-smat;
end
% Finally, protect against numerical violation
% of the Cauchy Schwarz inequality,
% because later routines may take square roots again:
smat=max(smat,zeros(size(smat)));
```

## 3. Kernels

After dealing with points and matrices, we now turn to kernels.

### 3.1. *Kernel Scaling*

*Scaled* radial kernels are of the form

$$K_c(x,y) = \phi\left(\|x-y\|/c\right), \text{ for all } x, y \in \mathbb{R}^d,$$

with a constant $c > 0$. This lets $c$ act like a support radius, in particular when we generate compactly supported radial kernels $K_c$ from scalar functions $\phi$ with support in $[0,1]$. Then $c$ is the support radius of $K_c$. Often, $c$ is called a *shape parameter*, and there is a huge literature concerning good choices of $c$. Mathematical results on scaling are in [25], while practical guidelines are in [50].

### 3.2. *Kernels in f–form*

For good reasons to be explained below, we write unscaled radial kernels $\phi$ in *f-form* via the transformation

$$\begin{aligned} f(s) &= \phi(\sqrt{2s}), \\ \phi(r) &= f(r^2/2). \end{aligned}$$

The biggest advantage of the $f$-form will show up below in section 3.5. A simple case is the Gaussian kernel $\phi(r) = \exp(-r^2)$ with $f(s) = -\exp(2s)$. The derivatives of $\phi$ are much more complicated than those of $f$. In addition, Cartesian derivatives of radial functions tend to have singularities that may cancel or not. For $\phi(r) = f(r^2/2)$ and $r = \|x-y\|$ for $x, y \in \mathbb{R}^d$, consider

$$\begin{array}{rcccl} \frac{\partial}{\partial x_i}\phi(r) & = & \phi'(r)\frac{\partial r}{\partial x_i} & = & \phi'(r)\frac{x_i}{r}, \\ \frac{\partial}{\partial x_i}f(s) & = & f'(s)\frac{\partial s}{\partial x_i} & = & f'(s)x_i. \end{array}$$

The second line looks nice at first sight, much better than the first, but we shall see that the $f$-derivatives themselves may contain singularities, because they take $s$-derivatives of functions of $\sqrt{2s}$. However, in the last line we also see that these derivatives get factors that vanish at zero, hopefully cancelling the singularities of $f'(s)$. We shall have a closer look at this phenomenon later, but users must keep in mind that such cancellations must be cared for.

Scaling enters into the $f$-form via

$$K_c(x,y) := f(\|x-y\|^2/(2c^2)) = K(x/c, y/c), \tag{1}$$

such that $\phi_c(r) = \phi(r/c) = f_c(s) := f\left(r^2/(2c^2)\right)$.

### 3.3. The Basic Kernel Routine

Our standard way to calculate with kernels is to use them in unscaled $f$–form and to call a MATLAB m-file `frbf.m` as

```
resmat=frbf(smat,k)
```

which evaluates the $k$–th ($k \in \mathbb{N}_0$) derivative of the kernel $f(s)$ with respect to $s$, using a scaled $S$-matrix `smat` with elements $\|x_i - y_j\|^2/(2c^2)$ in what follows. All other routines, e.g. for evaluating Laplacians of kernels on kernel matrices, are based exclusively on `frbf.m`. The kernel scaling is done within the input $S$-matrix. Users are strongly advised not to use these routines for single points. They are tailored for middle-size matrices. Furthermore, there are no precautions so far against evaluation of kernels for illegal parameter choices.

The use of a simple interface like `frbf.m` makes all routines independent of the chosen kernel, together with all existing derivatives. But it requires to move the control of kernel parameters into global variables. These currently are

```
RBFtype

RBFpar

RBFscale
```

to be explained in what follows. The scale $c$ of the kernel in the sense of (1) is `RBFscale`. The type of kernel is selected by setting `RBFtype` to a MATLAB string like `'g'` for the Gaussian. An additional real-valued parameter is defined by `RBFpar` depending on the type of kernel.

The list of current options for `RBFtype` is

- `g` Gaussian $\phi(r) = \exp\left(-r^2\right)$ with $f(s) = \exp(-2s)$, no `RBFpar`.
- `mq` Multiquadric $\phi(r) = (1+r^2)^\beta$ with $f(s) = (1+2s)^\beta$, with `RBFpar`$=\beta \in \mathbb{R}\backslash\mathbb{N}_0$ to avoid a polynomial kernel. Inverse multiquadrics can be treated by choosing `RBFpar`$=\beta$ negative. For $\beta > 0$, the sign $(-1)^{\lceil\beta\rceil}$ guaranteeing conditional positive definiteness of order $\lceil\beta\rceil$ is applied.
- `p` Powers $\phi(r) = r^\beta$ with $f(s) = (\sqrt{2s})^\beta$, with `RBFpar`$=\beta$, $0 < \beta \notin 2\mathbb{N}$. The correct sign for conditional positive definiteness of order $\lceil\frac{\beta}{2}\rceil$ is $(-1)^{\lceil\frac{\beta}{2}\rceil}$.
- `ms` Matern/Sobolev [39] $\phi(r) = r^\nu K_\nu(r)$, with $f(s) = (\sqrt{2s})^\nu K_\nu\left(\sqrt{2s}\right)$, with `RBFpar`$=\nu > 0$.
- `w` all $C^{2m}$ Wendland functions $\phi_{3,m}$ for $m =$`RBFpar`$\geq 0$, see [55]. These have support in $[0,1]$, are positive definite kernels in dimensions $d \leq 3$ and radial polynomials of minimal possible degree.
- `tp` Thin–plate splines [12] $\phi(r) = r^{2m}\log r$ with $f(s) = (2s)^m\frac{1}{2}\log(2s)$, for `RBFpar`$=m \in \mathbb{N}$, with sign $(-1)^{m+1}$ for conditional positive definiteness of order $m+1$.
- `sech` Hyperbolic secant [9, 16] $\phi(r) = \operatorname{sech} r$, with $f(s) = \operatorname{sech}(\sqrt{2s})$, no `RBFpar`.
- `rth` RTH kernel [19, 20] $\phi(r) = r\tanh(r/\beta)$, with $f(s) = \sqrt{2s}\tanh(\sqrt{2s}/\beta)$, for `RBFpar`$=\beta > 0$.

The newborn kernel RTH is a new transcendental RBF of the form $\phi(r) = r\tanh(r/\beta)$ introduced for the first time by Heidari et al. [19, 20]. It is a smooth approximant to $r$ by considering $\beta \to 0^+$, with higher accuracy and better convergence properties than the multiquadric with $\beta = 1/2$. It can serve to get shape-preserving univariate approximations [19].

Note that `frbf.m` does not care for addition of polynomials in case of conditional positive definiteness of positive order. Section 3.6 will deal with multivariate polynomials, and Section 4.1 with interpolation in the conditionally positive definite case. Furthermore, the Wendland function class is currently restricted to kernels working in dimensions up to 3.

### *3.4. Full Listing*

To demonstrate how the $f$-form facilitates taking derivatives of kernels, and to show how the mathematics in Section 5 is implemented, we provide a full program listing here.

```
function y=frbf(s,k)
% k-th derivative of scaled RBF kernel in f form, i.e.
% as a function of s=(r^2)/2.  There is NO scaling here.
global RBFtype;
global RBFpar;
switch lower(RBFtype)
    case ('g')   % 'g'=Gaussian, exp(-r^2)=exp(-2s)
        y=(-2)^k*exp(-2*s);
    case ('mq') % 'mq' = multiquadric, inverse or not...
    % (1+r^2)^beta=(1+2*s)^beta
        y=2^k*prod(RBFpar-k+1:RBFpar)*(1+2*s).^(RBFpar-k);
         if RBFpar>0
            y=y*(-1)^(ceil(RBFpar));
        end
    case ('p')  % 'p'=powers, r^beta=(2s)^(beta/2)
```

```
        y=2^k*prod((RBFpar/2)-k+1:RBFpar/2)*(2*s+eps).^((RBFpar-2*k)/2);
        if RBFpar>0
            y=y*(-1)^(ceil(RBFpar/2));
        end
    case ('tp')  % 'tp'=thin-plate splines,
    % r^(2n).ln(r)=(1/2).(2s)^n.ln(2s)
        ord=1;
        su=0;
        while ord<=k
            su=su*(RBFpar-ord+1)+prod(RBFpar-ord+2:RBFpar);
            ord=ord+1;
        end
        su=2^(k-1)*su.*(2*s+eps).^(RBFpar-k);
        y=2^k*prod(RBFpar-k+1:RBFpar)*(2*s+eps).^(RBFpar-k).*log(2*s+eps)/2+su;
        y=y*(-1)^(RBFpar+1);
    case ('ms')  % 'ms'=Matern/Sobolev,
    % r^\nu.K_\nu(r)=(\sqrt(2s))^\nu.K_\nu(\sqrt(2s))
        y=(-1)^k*besselk(RBFpar-k,sqrt(2*s+eps)).*(sqrt(2*s+eps)).^(RBFpar-k);
    case ('sech') % Sech, sech(r)=sech(sqrt(2s))
        u=cellfun(@(j) gsech(s,j,k),num2cell(0:k),'UniformOutput',false);
        totalSum=sum(cat(3,u{:}),3);
        y=2^k*totalSum;
    case ('rth') % 'rth'=rtanh, rtanh(r/beta)=sqrt(2s).tanh(sqrt(2s)/beta)
        u=cellfun(@(j) grth(s,j,k,RBFpar),num2cell(0:k),'UniformOutput',false);
        totalSum=sum(cat(3,u{:}),3);
        y=totalSum;
    case ('w') % 'w'=Wendland functions.
        % we use only those which are pos. def.
        % in dimension at most d=3.
        r=sqrt(2*s);
        u=zeros(size(s));
        if k<=RBFpar
            [coeff,expon]=wendcoeff(3+2*k,RBFpar-k);
            flag=max(zeros(size(r)),1-r).^expon;
            for i=1:length(coeff)
                R=r;
                R(r<=1)=(r(r<=1)).^(i-1);
                u=u+coeff(i)*R;
            end
            u=u.*flag;
            y=(-1)^k*u;
            return
        end
        % this new part cares for high-order derivatives
        if k<=2*RBFpar
            [coeff,expon]=qcoeff(k-RBFpar,RBFpar+2);
            p=RBFpar+2;
            for i=1:p
                R=r;
                R(r<=1)=(r(r<=1)).^(i-1);
                u=u+coeff(i)*R;
            end
            y=(-1)^(RBFpar)*u./(r+eps).^expon;
            return
        end
        error('Unimplemented s-derivative of Wendland function')
    otherwise
        error('RBF type not implemented')
    end
end
```

Here, we point out some technical features of the code.

*3.4.1. Avoiding Singularities*

The polyharmonic and certain kernels involving Bessel functions have singularities at the origin, as well as the derivatives of functions in $f$–form. A fast and often also sufficient trick is to add a small positive constant like the MATLAB `eps` to the endangered argument. A more sophisticated approach would be to calculate the local Taylor polynomial around zero and implement it locally, which is not considered here.

*3.4.2. Truncated Powers*

For compactly supported radial kernels, one often needs truncated powers

$$s_+^k := \left\{ \begin{array}{cc} s^k & s>0 \\ 0 & s\leq 0 \end{array} \right\}$$

elementwise on matrices. The standard trick in MATLAB is to use

```
max(zeros(size(s)),s).^k
```

in order to avoid pitfalls and loops.

*3.4.3. Factorials*

The falling factorial

$$x^{(n)} = x(x-1)\dots(x-n+1),$$

and the Pochhammer symbol (rising factorial)

$$(x)_n = x(x+1)\dots(x+n-1),$$

can be computed by `prod(x-n+1:n)` and `pochhammer(x,n)`, respectively.

*3.4.4. Finite sum of a function of the S-matrix*

In order to evaluate

$$\sum_{j=0}^{k} g(s,j,k), \tag{2}$$

we call the following MATLAB commands

```
u=cellfun(@(j) g(s,j,k),num2cell(0:k),'UniformOutput',false);
out=sum(cat(3,u{:}),3);
```

### 3.5. Recursive Scalar Radial Derivatives

This section describes the mathematical background of how the derivatives of scaled radial kernels in $f$ form can be calculated. Thus it is an explanation of how `frbf(s,k)` works for positive derivative orders `k`. If interested in programs only, readers can skip this section.

#### 3.5.1. Gaussian

The Gaussian $\phi(r) = \exp(-r^2)$ uses $f(s) = \exp(-2s)$ with simplest possible derivatives $f^{(k)}(s) = (-2)^k \exp(-2s)$.

#### 3.5.2. Multiquadrics

The multiquadric $\phi_\beta(r) = (1+r^2)^\beta$ with $\beta \in \mathbb{R}\setminus\mathbb{N}_0$ leads to the easy recursion

$$\begin{array}{rcl} f_\beta(s) &=& (1+2s)^\beta \\ f'_\beta(s) &=& 2\beta(1+2s)^{\beta-1} = 2\beta f_{\beta-1}(s) \\ f^{(k)}_\beta(s) &=& 2^k\beta(\beta-1)\dots(\beta-k+1)(1+2s)^{\beta-k} = 2^k\beta^{(k)} f_{\beta-k}(s). \end{array}$$

Note that this includes inverse multiquadrics.

#### 3.5.3. Power Kernels

For the powers $\phi_\beta(r) = r^\beta$ with $0 < \beta \notin 2\mathbb{N}$ we get

$$\begin{array}{rcl} f_\beta(s) &=& (2s)^{\beta/2} \\ f'_\beta(s) &=& 2(\beta/2)(2s)^{\beta/2-1} = 2(\beta/2) f_{\beta-2}(s) \\ f^{(k)}_\beta(s) &=& 2^k(\beta/2)^{(k)} f_{\beta-2k}(s). \end{array}$$

#### 3.5.4. Thin-Plate Splines

Also for the thin–plate splines $\phi_n(r) = r^{2n}\log(r/c)$ with $n \in \mathbb{N}$ we find

$$\begin{array}{rcl} f_n(s) &=& (2s)^n\frac{1}{2}\log(2s) \\ f'_n(s) &=& n(2s)^{n-1}\log(2s) + (2s)^{n-1} = 2nf_{n-1}(s) + (2s)^{n-1} = 2nf_{n-1}(s) + A_1(2s)^{n-1} \\ f''_n(s) &=& 2^2n(n-1)f_{n-2}(s) + 2((n-1)+n)(2s)^{n-2} = 2^2n(n-1)f_{n-2}(s) + 2A_2(2s)^{n-2} \\ f^{(3)}_n(s) &=& 2^3n(n-1)(n-2)f_{n-3}(s) + 2^2(((n-1)+n)(n-2)+n(n-1))(2s)^{n-3} \\ &=& 2^3n(n-1)(n-2)f_{n-3}(s) + 2^2A_3(2s)^{n-3} \\ f^{(k)}_n(s) &=& 2^kn^{(k)}f_{n-k}(s) + 2^{k-1}A_k(2s)^{n-k}, \end{array}$$

where $A_0 = 0$ and $A_j = A_{j-1}(n-j+1) + n^{(j-1)}$. Note that the second term is a polynomial in $s$.

#### 3.5.5. Matern/Sobolev kernels

These are the most important case in Spatial Statistics [39]. The kernels are

$$\phi_\nu(r) = r^\nu K_\nu(r),$$

with the modified Bessel function $K_\nu$ of second kind. It has the property

$$K'_\nu(z) = -K_{\nu+1}(z) + \frac{\nu}{z}K_\nu(z) = -K_{\nu-1}(z) - \frac{\nu}{z}K_\nu(z)$$

and we need

$$f_\nu(s) = (\sqrt{2s})^\nu K_\nu(\sqrt{2s}).$$

Then

$$\begin{aligned}
f_\nu'(s) &= \nu(\sqrt{2s})^{\nu-1}\frac{1}{\sqrt{2s}}K_\nu(\sqrt{2s}) + (\sqrt{2s})^\nu \frac{1}{\sqrt{2s}}K_\nu'(\sqrt{2s})\\
&= \nu K_\nu(\sqrt{2s})(\sqrt{2s})^{\nu-2} + (\sqrt{2s})^{\nu-1}K_\nu'(\sqrt{2s})\\
&= \nu K_\nu(\sqrt{2s})(\sqrt{2s})^{\nu-2} + (\sqrt{2s})^{\nu-1}\left(-K_{\nu-1}(\sqrt{2s}) - \frac{\nu}{\sqrt{2s}}K_\nu(\sqrt{2s})\right)\\
&= -(\sqrt{2s})^{\nu-1}K_{\nu-1}(\sqrt{2s})\\
&= -f_{\nu-1}(s)\\
f_\nu^{(k)}(s) &= (-1)^k f_{\nu-k}(s).
\end{aligned}$$

*3.5.6. Hyperbolic secant*

For the Hyperbolic secant $\phi(r) = \operatorname{sech}(r)$ we get $f(s) = \operatorname{sech}(\sqrt{2s})$. Then according to [54] we have

$$f^{(k)}(s) = 2^k \sum_{j=0}^{k} \frac{(-1)^{k-j}(j)_{2(k-j)}}{(k-j)!\left(2\sqrt{2s}\right)^{2k-j}} \operatorname{sech}^{(j)}(\sqrt{2s}), \tag{3}$$

where the derivatives of the sech function are computed by the following relations [59]

$$\begin{aligned}
\operatorname{sech}^{(2j)}(t) &= \sum_{i=0}^{j} (-1)^{j-i} w_{2(j-i)+1,i}\,(2(j-i))!\,(\operatorname{sech}(t))^{2(j-i)+1},\\
\operatorname{sech}^{(2j+1)}(t) &= -\tanh(t)\sum_{i=0}^{j} (-1)^{j-i} w_{2(j-i)+1,i}\,(2(j-i)+1)!\,(\operatorname{sech}(t))^{2(j-i)+1},
\end{aligned}$$

where

$$\begin{aligned}
w_{2j+1,0} &= 1,\\
w_{2j+1,1} &= \sum_{m=0}^{j} (2m+1)^2,\\
w_{1,i} &= 1,\\
w_{2j+1,i} &= (2j+1)^2 w_{2j+1,i-1} + w_{2j-1,i}.
\end{aligned}$$

In order to compute (3), we use (2) with the following routine

```
function out=gsech(s,j,k)
out=diffsech(sqrt(2*s),j).*(((-1)^(k-j)*pochhammer(j,2*(k-j)))...
    ./(factorial(k-j)*((2*sqrt(2*s)+eps).^(2*k-j))));
    function out=diffsech(s,j)
        u=cellfun(@(i) hsech(s,j,i),num2cell(0:floor(j/2)),'UniformOutput',false);
        totalSum=sum(cat(3,u{:}),3);
        if mod(j,2)
            out=-tanh(s).*totalSum;
        else
            out=totalSum;
        end
    end
    function out=hsech(s,j,i)
        p=floor(j/2);
        out=((-1)^(p-i)*Wsec(2*(p-i)+1,i)*factorial(2*(p-i)+mod(j,2)))...
            .*(sech(s).^(2*(p-i)+1));
    end
    function out=Wsec(j,i)
        p=(j-1)/2;
        if i==0
            out=1;
        elseif i==1
            out=sum((2*(0:p)+1).^2);
        elseif j==1
            out=1;
        else
            out=j^2*Wsec(j,i-1)+Wsec(2*p-1,i);
        end
    end
end
```

*3.5.7. RTH RBF*

For the RTH RBF $\phi(r) = r\tanh(r/\beta)$ we get $f(s) = \sqrt{2s}\tanh(\sqrt{2s}/\beta)$. Then according to [32] we have

$$f^{(k)}(s) = \sum_{j=0}^{k} b_{kj}\left(\sqrt{2s}\right)^{-(k+j-1)}\tanh^{(k-j)}(\sqrt{2s}/\beta), \tag{4}$$

where the coefficients $b_{kj}$ are given by the recursion

$$\begin{cases} b_{k0} = 1/\beta^k, \quad k = 0,1,\dots \\ b_{11} = 1, \\ b_{kj} = (-k-j+3)b_{(k-1)(j-1)} + (1/\beta)b_{(k-1)j}, \quad j = 1,\dots,k-1,\ k \geq 2, \\ b_{kk} = (-2k+3)b_{(k-1)(k-1)}, \end{cases}$$

and the derivatives of tanh function is computed by the following relations [59]

$$\begin{aligned} \tanh^{(2j)}(t) &= \tanh(t)\sum_{i=0}^{j}(-1)^{j-i}w_{2(j-i),i}\,(2(j-i))!\,(\operatorname{sech}(t))^{2(j-i)}, \\ \tanh^{(2j+1)}(t) &= \sum_{i=0}^{j}(-1)^{j-i}w_{2(j-i+1),i}\,(2(j-i)+1)!\,(\operatorname{sech}(t))^{2(j-i+1)}, \end{aligned}$$

where

$$\begin{aligned} w_{2j,0} &= 1, \\ w_{2j,1} &= \sum_{m=0}^{j}(2m)^2, \\ w_{0,i} &= 0, \\ w_{2,i} &= = 2^{2i}, \\ w_{4,i} &= \sum_{m=0}^{i}2^{2(i+m)}, \\ w_{2j,i} &= (2j)^2 w_{2j,i-1} + w_{2(j-1),i}. \end{aligned}$$

In order to compute (4), we use (2) with the following routine

```
function out=grth(s,j,k,RBFpar)
out=diffrth(sqrt(2*s)/RBFpar,k-j).*((b(k,j,RBFpar)*(sqrt(2*s)+eps).^(-(k+j-1))));
    function out=b(k,j,RBFpar)
        if j==0
            out=1/RBFpar^k;
        elseif k==1 && j==1
            out=1;
        elseif ismember(j,1:k-1) && k>=2
            out=(-k-j+3)*b(k-1,j-1,RBFpar)+(1/RBFpar)*(b(k-1,j,RBFpar));
        else
            out=(-2*k+3)*b(k-1,k-1,RBFpar);
        end
    end
    function out=diffrth(s,j)
        u=cellfun(@(i) hrth(s,j,i),num2cell(0:floor(j/2)),'UniformOutput',false);
        totalSum=sum(cat(3,u{:}),3);
        if mod(j,2)
            out=totalSum;
        else
            out=tanh(s).*totalSum;
        end
    end
    function out=hrth(s,j,i)
        p=floor(j/2);
        out=((-1)^(p-i)*Wrth(2*(p-i+mod(j,2)),i)*factorial(2*(p-i)+mod(j,2)))...
            .*(sech(s).^(2*(p-i+mod(j,2))));
    end
    function out=Wrth(j,i)
        p=j/2;
        if i==0
            out=1;
        elseif i==1
            out=sum((2*(0:p)).^2);
        elseif j==0
            out=0;
        elseif j==2
            out=2^(2*i);
        elseif j==4
            out=sum(2.^(2*(i+0:i)));
        else
            out=j^2*Wrth(j,i-1)+Wrth(2*(p-1),i);
        end
    end
end
```

*3.5.8. Wendland functions*

We now handle the Wendland [55, 57] kernels, but note that we now have to be careful with constant factors. First, we rewrite the dimension walk operator [57]

$$\begin{aligned} I(\phi)(r) &= \int_r^\infty t\phi(t)dt \qquad (5)\\ &= \int_{r^2/2}^\infty \underbrace{\phi(\sqrt{2s})}_{=f(s)} ds\\ &= \tilde{I}(f)(t),\ t = r^2/2,\\ I(\phi)(\sqrt{2t}) &= \tilde{I}(f)(t) = \int_t^\infty f(s)ds\\ \tilde{I}' &= -Id, \end{aligned}$$

where $Id$ is the identity operator. The Wendland functions are defined via

$$\phi_{d,m} = I^m \phi_{\lfloor d/2 \rfloor + m+1},\ \phi_\ell(r) = (1-r)_+^\ell$$

for $m \geq 0$ and $\ell \geq 1$, and we rewrite them in the form

$$\tilde{\phi}_{d,m}(s) = \phi_{d,m}(\sqrt{2s}),\ \tilde{\phi}_\ell(s) = \phi_\ell(\sqrt{2s}) = (1-\sqrt{2s})_+^\ell.$$

This definition also works for negative $m$ because of $I^{-1}\phi(r) = -\phi'(r)/r$. Then

$$\begin{aligned} \tilde{\phi}_{d,m}(s) &= \phi_{d,m}(\sqrt{2s})\\ &= (I^m \phi_{\lfloor d/2 \rfloor + m+1})(\sqrt{2s})\\ &= \tilde{I}^m \tilde{\phi}_{\lfloor d/2 \rfloor + m+1}(s)\\ \tilde{\phi}'_{d,m}(s) &= -\tilde{I}^{m-1} \tilde{\phi}_{\lfloor d/2 \rfloor + m+1}(s)\\ &= -\tilde{I}^{m-1} \tilde{\phi}_{\lfloor (d+2)/2 \rfloor + m-1+1}(s)\\ &= -\tilde{\phi}_{d+2,m-1}(s)\\ \tilde{\phi}^{(k)}_{d,m}(s) &= (-1)^k \tilde{\phi}_{d+2k,m-k}(s). \end{aligned}$$

is a derivative recursion which is easy to implement if the standard basis functions $\phi_{d,m}$ are available. We shall describe below how those can be calculated in general.

Note that the scalar factors are different from what is usually seen in tables of Wendland functions. Here is an example with correct factors when starting with the topmost one:

$$\begin{aligned} \phi_{3,3}(r) &= (1-r)_+^8(32r^3+25r^2+8r+1)\\ \phi_{5,2}(r) &= 22(1-r)_+^7(16r^2+7r+1)\\ \phi_{7,1}(r) &= 528(1-r)_+^6(6r+1)\\ \phi_{9,0}(r) &= 22176(1-r)_+^5 \end{aligned}$$

An easy way to get high–degree Wendland functions $\phi_{d,m}$ with correct constants for derivatives is to start with some polynomial $w_\ell(r) := (1-r)^\ell,\ \ell = \lfloor d/2 \rfloor + m + 1$ and then to apply $I^m$ for integer $m$. For $m > 0$, the result is $C^{2m}$ and positive definite in dimensions up to $d$. The following MAPLE procedure generates all Wendland functions $\phi_{d,k}$ for integer $k$:

```
phidk:=proc(d,k)
> local L, wL, j;
> L:=floor(d/2)+k+1;
> wL:=(1-r)^L;
> if k>=0
> then
> for j from 1 to k do
        wL:=-int(r*wL,r);wL:=factor(wL-subs(r=1,wL)); end do;
> return(simplify(wL));
> else
> for j from 1 to -k do wL:=-diff(wL,r)/r; end do;
> return(simplify(wL));
> end if;
> end proc:
```

Here are a few examples for $\phi_{d,-m}(r)$ calculated via MAPLE, up to the cutoff at $r=1$:

$$\begin{array}{rl} \frac{1}{r}, & d=3, \quad m=1, \\ \frac{2(1-r)}{r}, & d=5, \quad m=1, \\ \frac{1}{r^3}, & d=5, \quad m=2, \\ \frac{3(1-r)^2}{r}, & d=7, \quad m=1, \\ \frac{2}{r^3}, & d=7, \quad m=2, \\ \frac{3}{r^5}, & d=7, \quad m=3 \end{array}$$

and up to scalar factors like in the standard case. These functions get singular at the origin, but the singularity cancels out later, see the discussion at the end of Section 3.2.

We now have to derive recursive formulae for the coefficients of Wendland functions as polynomials in $r$ on $[0,1]$. If we start from $w_\ell(r) := (1-r)^\ell$ and let the operator $I$ act $m$ times, one can see the above process as starting from the polynomial $p_0 = 1$ and proceeding inductively via

$$\int_r^1 t(1-t)^{\ell+n} p_n(t)dt = (1-r)^{\ell+n+1} p_{n+1}(r) \tag{6}$$

for $n = 0, \dots, m-1$. Then

$$\phi_{d,m}(r) = (1-r)_+^{\ell+m} p_m(r). \tag{7}$$

Moreover, (6) yields

$$\frac{d}{dr}\Big((1-r)^{\ell+n+1} p_{n+1}(r)\Big) \quad = \quad -r(1-r)^{\ell+n} p_n(r)$$

and if we set

$$p_n(r) = \sum_{j=0}^{n} a_{j,n} r^j$$

there is a simple recursion for the coefficients via

$$\begin{array}{rcl}
-r(1-r)^{\ell+n}p_n(r) & = & -(\ell+n+1)(1-r)^{\ell+n}p_{n+1}(r)+(1-r)^{\ell+n+1}p'_{n+1}(r)\\
-rp_n(r) & = & -(\ell+n+1)p_{n+1}(r)+(1-r)p'_{n+1}(r)\\
-r\sum_{j=0}^{n} a_{j,n}r^j & = & -(\ell+n+1)\sum_{j=0}^{n+1} a_{j,n+1}r^j+(1-r)\sum_{j=1}^{n} ja_{j,n+1}r^{j-1}\\
-a_{j-1,n} & = & -(\ell+n+1)a_{j,n+1}+(j+1)a_{j+1,n+1}-ja_{j,n+1}\\
-a_{j-1,n} & = & -(\ell+n+1+j)a_{j,n+1}+(j+1)a_{j+1,n+1}\\
a_{j,n+1} & = & \dfrac{1}{\ell+n+1+j}((j+1)a_{j+1,n+1}+a_{j-1,n})
\end{array}$$

backwards for $j=n+1,\dots,0$. If the basis functions are evaluated on large matrices, the above $\mathcal{O}(m^2)$ snippet does not contribute significantly to the program complexity. The following MATLAB code generates the necessary polynomial coefficients in ascending order by applying the above recursion.

```
function [coeff,expon]=wendcoeff(d,m)
% calculate coefficients and exponent
% for  polynomial part p_{m} of Wendland's functions.
% phi_{d,m}(r)=p_{m}(r)*(1-r)^expon
% The array coeff contains the values a_{j,n} of the text.
% Since n increases and j decreases, no matrix is needed.
L=floor(d/2)+m+1;
expon=L+m;
coeff=zeros(m+1,1);
coeff(1)=1; % p_0=1
for n=0:m-1
    coeff(n+2)=coeff(n+1)/(L+n+1+n+1);
    for j=n+1:-1:2
        coeff(j)=(j*coeff(j+1)+coeff(j-1))/(L+n+j);
    end
    coeff(1)=coeff(2)/(L+n+1);
end
```

The final evaluation is a part in `frbf.m` when called for the $k$-th derivative and on a matrix in $s$ form (see the above listing of `frbf.m`). In general, the routine `wendcoeff(d,m)` calculates the coefficients of the polynomial $p_m$ for the purely polynomial Wendland function (7), and therefore according to (6) `frbf(s,k)` calls `wendcoeff(3+2*k,RBFpar-k)` which is correct for derivatives of order $k \le m =$`RBFpar`. Note that the calculation of Wendland kernels acts only on points in the support, once they are found.

But there is a problem hidden here. Recall that we have the restriction $m \ge 0$ for all of this, and we know that $\phi_{d,m}$ is in $C^{2m}$. Then it is tempting to take $2m$ derivatives of $\phi_{d,m}$ this way, leading formally to

$$\tilde{\phi}_{d,m}^{(2m)} = \tilde{\phi}_{d+4m,-m},$$

if the standard recursion and the right-hand side would be valid also for negative second indices. But this does not fall into the range of coefficients we considered so far, and standard application of the `wendcoeff` routine will fail.

Let us now look at Wendland functions for negative second indices. If $D$ is the $s$-derivative operator on functions in $f$–form, then $(\tilde{I})^{-1} = -D$, and for all $m \geq 0$ we get

$$\tilde{\phi}_{d,-m} = (-1)^m D^m \tilde{\phi}_{d-2m,0} = (-1)^m D^m \tilde{\phi}_{\lfloor \frac{d}{2} \rfloor - m+1} = (-1)^m D^m (1-\sqrt{2s})_+^{\lfloor \frac{d}{2} \rfloor - m+1}$$

if we stick to the dimension walk, and for $0 \leq m \leq \lfloor \frac{d}{2} \rfloor$. Derivatives of $\phi_{d,m}$ up to order $m$ can still be handled by the `wendcoeff` routine, but for higher derivatives we get

$$\begin{array}{rcl}
\tilde{\phi}_{d,m}^{(m+n)} & = & (-1)^{m+n}\tilde{\phi}_{d+2m+2n,-n} \\
 & = & (-1)^{m+n}(-1)^n D^n (1-\sqrt{2s})_+^{\lfloor \frac{d}{2} \rfloor + m+n+1} \\
 & = & (-1)^m D^n (1-\sqrt{2s})_+^{\lfloor \frac{d}{2} \rfloor + m+1} \\
 & = & D^n \tilde{\phi}_{d,m}^{(m)}.
\end{array}$$

To get the recursion behind this, we define

$$g_{n,p}(\sqrt{2s}) := D^n (1-\sqrt{2s})^p$$

for integers $n \geq 0$ and $p > 0$ to get

$$\begin{array}{rcl}
g_{n+1,p}(\sqrt{2s}) & = & D(g_{n,p}(\sqrt{2s})) \\
 & = & \dfrac{g'_{n,p}(\sqrt{2s})}{\sqrt{2s}} \\
t g_{n+1,p}(t) & = & g'_{n,p}(t) \\
g_{0,p}(t) & = & (1-t)^p
\end{array}$$

proving that

$$g_{n,p}(t) = \frac{q_{n,p}(t)}{t^{2n-1}}$$

holds for $n \geq 1$ with a polynomial $q_{n,p}$ of degree at most $p-1$ that has a recursion

$$q_{n+1,p} = t q'_{n,p}(t) + (-2n+1) q_{n,p}(t).$$

for $n \geq 1$ starting with $q_{1,p}(t) = -p(1-t)^{p-1}$. The argument proceeds via

$$\begin{array}{rcl}
g'_{n,p}(t) & = & \dfrac{q'_{n,p}(t)}{t^{2n-1}} + (-2n+1)\dfrac{q_{n,p}(t)}{t^{2n}} \\
g_{n+1,p}(t) = \dfrac{g'_{n,p}(t)}{t} & = & \dfrac{t q'_{n,p}(t) + (-2n+1) q_{n,p}(t)}{t^{2n+1}}.
\end{array}$$

To apply this to $\phi_{d,-m}$, we set $p = \lfloor \frac{d}{2} \rfloor - m + 1$ and $n = m$ and arrive at

$$\phi_{d,-m}(r) = (-1)^m \frac{q_{m,\lfloor \frac{d}{2} \rfloor - m+1}(r)}{r^{2m-1}}$$

for $m \geq 1$ and a numerator of degree at most $p - 1 = \lfloor \frac{d}{2} \rfloor - m \geq 0$. In the monomial representations

$$q_{n,p}(t) = \sum_{j=0}^{p-1} a_j^{(n,p)} t^j$$

we get the recursion

$$a_j^{(n+1,p)} = (1-2n+j)a_j^{(n,p)},\ 0 \le j \le p-1, n \ge 1, p \ge 1$$

with the starting values

$$a_j^{(1,p)} = -p(-1)^j \binom{p-1}{j},\ \ 0 \le j \le p-1, p \ge 1.$$

The coefficients with odd indices simplify to

$$a_{2m-1}^{(n,p)} = 0 \text{ for all } n > m, p \ge 1,$$

because for fixed odd $j$ the recursion for $a_j^{(n,p)}$ runs into zero beyond $2n-1 = j$ when increasing $n$ and stays at zero.

Therefore, for $m < k \le 2m$ we have to go to $(-1)^k \phi_{3+2k,m-k}$ as well, but now we have the rational form

$$\phi_{3+2k,m-k}(r) = (-1)^{k-m} \frac{q_{k-m,2+m}(r)}{r^{2k-2m-1}}$$

and get the coefficients by a call to `qcoeff(k-RBFpar,2+RBFpar)` while the routine `qcoeff(n,p)` proceeds like above, providing the coefficients of $q_{n,p}$. We add the code:

```
function [coeff, expon]=qcoeff(n,p)
% gets q_{n,p} coefficients in increasing sequence
expon=2*n-1;
for j=1:p
    coeff(j)=-p*(-1)^(j-1)*nchoosek(p-1,j-1);
end
for k=2:n
    for j=1:p
        coeff(j)=coeff(j)*(1-2*(k-1)+(j-1));
    end
end
```

*3.5.9. Remark*

It seems to be a strange fact that many classes of radial kernels are closed unter integration and differentiation, provided that they are written in $f$ form. But this is no miracle. The basic reason is that these classes are closed under radial Fourier transforms in $f$ form taken in different dimensions, and these radial Fourier transforms commute with differentiation and integration of the $f$ form. Details are in [51], but it turned out later that this goes back to Matheron [28].

## *3.6. Multivariate Polynomials*

In various cases, in particular for dealing with conditionally positive definite kernels, we need the $M \times Q$ matrix `polvalues` of values of multivariate polynomials of some order `order` evaluated on a $M \times d$ matrix `points` of $M$ points in $d$ dimensions. Note that *order* means *degree* $+1$ here, and the dimension $Q$ of the polynomials is dependent on the order $m$ and the space dimension $d$ via $Q = \binom{m+d-1}{d}$. Our simplest implementation is via unscaled monomials, i.e. we form the matrix of values $x_j^\alpha$ for all $j$, $1 \le j \le M$ and all multiindices $\alpha \in \mathbb{Z}_0^d$ with $0 \le |\alpha| := \|\alpha\|_1 < m$ where $m$ is the order. Again, the row index will run over points, i.e. from 1 to $M = |\mathtt{X}|$. Users should work with the following routine only near the origin, and they should possibly apply some scaling.

```
function polvalues=polynomials(points,order)
% generates all polynomials on points up to order
% The order MUST be increasing from left to right.
[numpoints,dim]=size(points); % handle trivial cases first
if order==0
    polvalues=[];
    return
end
if order==1
    polvalues=ones(numpoints,1);
    return
end
if order==2
    polvalues=[ones(numpoints,1) points];
    return
end
if dim==1
  polvalues=zeros(numpoints,order);
  polvalues(:,1)=ones(numpoints,1);
  polvalues(:,2)=points(:,1);
  for i=3:order
      polvalues(:,i)=polvalues(:,i-1).*points(:,1);
  end
  return
end           % general case done recursively
polvalues=[polynomials(points,order-1)...
        polynomials_exact_order(points,order)];
```

The recursion in the above program uses

```
function polvalues=polynomials_exact_order(points,order)
% generates all polynomials of exact order on points
[numpoints,dim]=size(points); % first some trivial cases
if order==1
    polvalues=ones(numpoints,1);
    return
end
if order==2
    polvalues=points;
    return
end
if dim==1
    polvalues=points(:,1).^(order-1);
    return
end
% What follows is a crude recursive scheme over the DIMENSION.
% Somebody MUST write a better one....
% We start with the multiindices that focus on the last component
polvalues= points(:,dim).^(order-1); % the x_{i,d}^{p-1}
for k=1:order-2
    pe=polynomials_exact_order(points(:,1:dim-1),order-k);
    % this contains all values (x_{i,1:d-1}.^{\alpha(:,1:d-1)} )_{i,\alpha}
    pp=points(:,dim).^k;
    [rpes cpes]=size(pe);
    % pps=size(points(:,dim).^(order-i));
    for j=1:cpes
        polvalues=[polvalues pe(:,j).*pp];
    end
end
polvalues=[polvalues ...
      polynomials_exact_order(points(:,1:dim-1),order)];
```

## 4. Interpolation and Evaluation

A standard square kernel–based interpolation system

$$\sum_{j=1}^{M} a_j\phi(\|x_k-x_j\|)=y_k,\ 1\leq k\leq M$$

for unconditionally positive definite radial kernels needs the $M\times M$ kernel matrix with entries $\phi(\|x_k-x_j\|,\ 1\leq k,j\leq M$. We provide a standard MATLAB routine `kermat.m` for this, hiding the scaling and the formation of the scaled `S` matrix, and allowing two point sets $X$ and $Y$.

```
function mat=kermat(X, Y)
% creates kernel matrix
% for two point sets X and Y
global RBFscale
mat=frbf(distsqh(X,Y)/RBFscale^2,0);
```

Then the kernel matrix for interpolation is

`intmat=kermat(X,X).`

The solution vector a follows from the data vector `y` via

`a=intmat\y;`

Note that these two commands work for all kernels and scalings. It is always a good idea to check the condition by preceding this with

`condition=condest(intmat)`

to avoid problems.

After finding the coefficient vector, one would usually want to evaluate the solution, e.g. for subsequent plotting. This will often need a much finer point set than `X`, and we assume that it is called `Y` here. The resulting values at these points are obtained from

```
evalmat=kermat(Y,X);
values=evalmat*a;
```

For fine–grained evaluation, this will take longer than the actual solution of the linear system, because a large unsymmetric kernel matrix has to be formed. Note that the resulting values have to be reshaped, if the points in `Y` are derived from a `meshgrid` command. The standard evaluation sequence in 2D thus is something like

```
[xe ye]=meshgrid(  .... );
Y=[xe(:) ye(:)];
evalmat=kermat(Y,X);
values=evalmat*a;
figure
surfc(xe,ye,reshape(values,size(xe)));
```

Figure 1 shows an example for interpolation of the MATLAB `peaks` function in $[-3,3]^2$ on $21\times 21 = 441$ random points, using the Hyperbolic secant kernel with `RBFscale=1`. The corresponding m-file in the package is `testint.m`, using the above snippet.

## *4.1. Interpolation with Conditionally Positive Definite Kernels*

For conditionally positive definite kernels of order $m$, a set $X=\{x_1,\dots,x_M\}$ of $M$ interpolation points must be $\mathbb{P}^d_{m-1}$–unisolvent. This means that only the zero polynomial in $\mathbb{P}^d_{m-1}$ vanishes on $X$. Then one needs an extended $(M+Q)\times(M+Q)$ matrix

$$\begin{pmatrix} A_{X,X} & P_X \\ P_X^T & 0_{Q\times Q}\end{pmatrix}$$

with matrices

$$\begin{array}{rcll} A_{X,X} & = & (K(x_j,x_k)), & 1\le j,k\le M \\ P_X & = & (p_i(x_j)), & 1\le j\le M,\ 1\le i\le Q \end{array}$$

and $Q$ being the dimension of the polynomials on $\mathbb{R}^d$ up to order $m$, using a basis $p_1,\dots,p_Q$. The interpolation data $y_1,\dots,y_M$ have to be extended to a vector $(y^T,0_Q)^T$ forming the right–hand side for the above system. The coefficients are then a vector $(a^T,b^T)^T\in\mathbb{R}^{M+Q}$, and evaluation on a fine point set $Y$ needs the matrix–vector product

$$(A_{Y,X}\ \ P_Y)\begin{pmatrix} a \\ b\end{pmatrix}$$

to generate the interpolant's values on $Y$. A corresponding program is `testintCPD.m` in the program package, with the result in Figure 2. It was executed on the same 441 points as Figure 1 using thin-plate splines of order 2.

The basic code snippet for `npp` points in a matrix `X` and conditional positive definiteness of `order` is

```
AXX=kermat(X,X); % kernel matrix on interpolation points
PX=polynomials(X,order); % polynomial values
[npp q]=size(PX) % q = dimension of polynomial space
intmat=[AXX PX; PX zeros(q,q)]; % full CPD interpolation matrix
condint=condest(intmat) % condition of full matrix
```

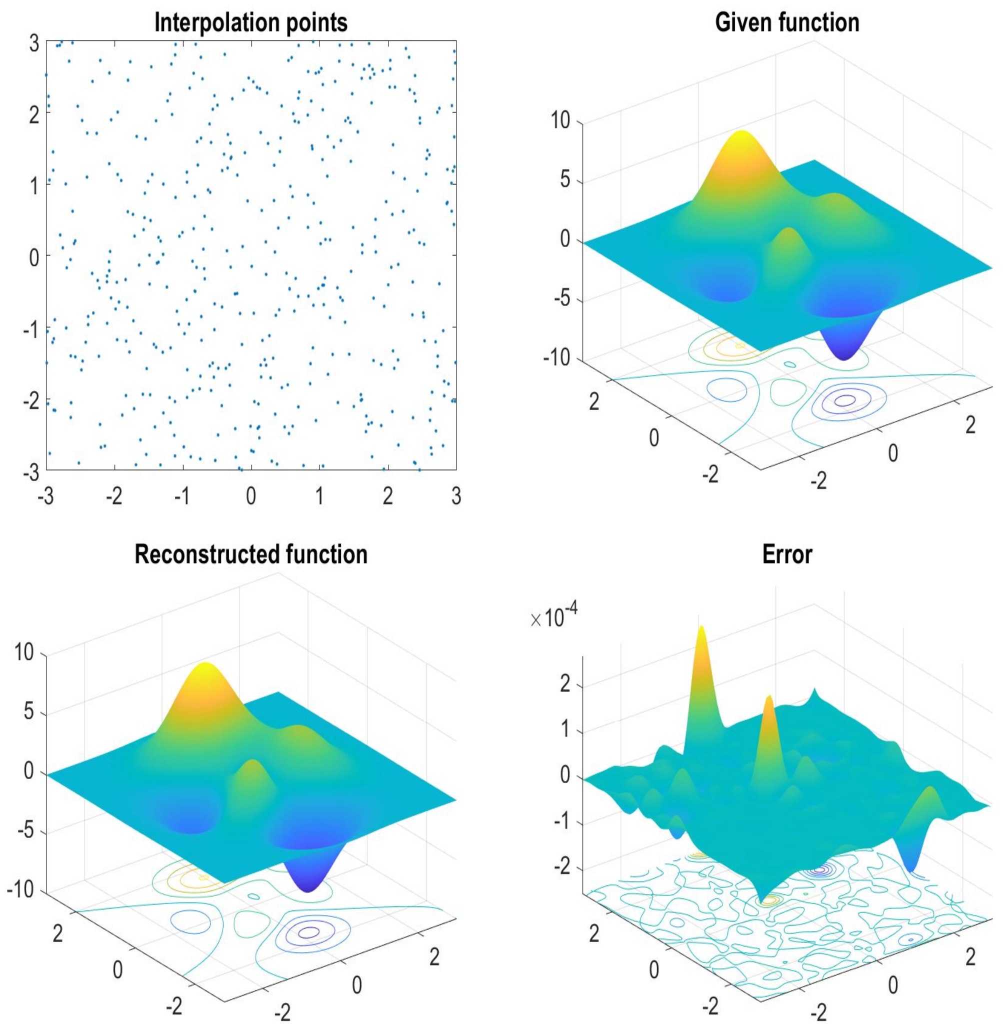

Figure 1: Numerical results for interpolation of the MATLAB peaks function with `testint.m` using the Hyperbolic secant kernel with `RBFscale=1`.

```
ZXe=[ZX; zeros(q,1)]; % right-hand side for CPD case
a=intmat\ZXe; % coefficients
evalkermat=kermat(Y,X); % evaluation kernel matrix
evalpolmat=polynomials(Y,order); % evaluation polynomial matrix
values=[evalkermat evalpolmat]*a; % resulting values
```

See how the snippet for the unconditional case was slightly modified to work in the conditionally positive definite case.

## *4.2. Lagrange Bases*

For a positive definite kernel $K$ and a point matrix `X` of $M$ points, the Lagrange basis functions $u_1(x),\dots,u_M(x)$ solve the system

$$\sum_{j=1}^{M} K(x_k,x_j)u_j(x) = K(x,x_k),\ 1 \le k \le M.$$

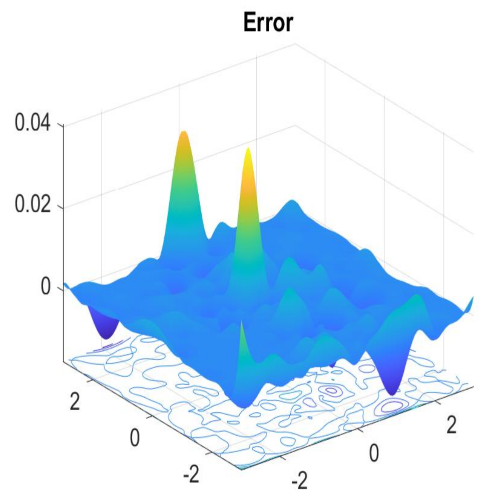

Figure 2: Error distribution for interpolation of the MATLAB peaks function with `testintCPD.m` using thin-plate splines of order 2.

To visualize these functions on a fine set of $N$ points $y_i$ given by a matrix $Y$, one should look at

$$\sum_{j=1}^{M} u_j(y_i)K(x_k,x_j) = K(y_i,x_k),\ 1 \le k \le M.$$

This is a matrix multiplication of the form $U * A = B$, and thus one gets the matrix

$$U = (u_j(y_i))_{1\le i\le N,\, 1\le j\le M} = B * A^{-1}$$

by solving $A^T U^T = AU^T = B^T$ via

```
umat=(intmat\evalmat')';
```

if the matrices `intmat` and `evalmat` are already calculated and stored as above. They are needed anyway for interpolation and evaluation, as we saw in the previous section. Now the columns of `umat` yield the values of the Lagrange basis functions. For 2D applications and `surf` plotting on `meshgrid` data, they must be `reshape`d. An example follows below.

Lagrange bases are special cases of *data–dependent bases* in [31, 38], like the Newton basis [36] described in Figure 3 below.

## *4.3. Power Functions*

Once the Lagrange basis is at hand, one can calculate the square of the optimal Power Function [57] `(eqpowSPD)`

$$P^2(x) = K(x,x) - \sum_{j=1}^{M} u_j(x)K(x_j,x) \tag{8}$$

at the points $y_i$ via

$$\begin{aligned} P^2(y_i) &= K(y_i,y_i) - \sum_{j=1}^{M} u_j(y_i)K(x_j,y_i) \\ &= f(0) - \sum_{j=1}^{M} u_j(y_i)K(x_j,y_i). \end{aligned}$$

This function is a crucial ingredient of error bounds [43], and in the stochastic setting [52] it describes the variance of the prediction error at $x$ from a Kriging predictor (i.e. the kernel interpolant) using the available data at the $x_j$.

With the matrices derived above, one can form `umat.*evalmat` to get the $N \times M$ matrix of all products $u_j(y_i)K(x_j,y_i)$. We need the sum over rows, but the `sum` operator of MATLAB sums over columns and generates a row. Thus

```
powval=frbf(0,0)-sum((umat.*evalmat)')';
```

yields the column vector of values of the optimal Power Function at the evaluation points. An equivalent command is

```
powval=frbf(0,0)-dot(umat',evalmat')';
```

For 2D applications and `surf` plotting on `meshgrid` data, they must be `reshape`d. The evaluation of the Power Function is essential for certain *greedy methods*, see e.g. [11] and Section 4.5 below.

The package contains a program `testlag.m` for Lagrange bases and Power Functions in the unconditionally positive definite case, and its output is in Figure 3. The Lagrange function for the central point is displayed, together with the squared Power Function. The kernel is the Gaussian at scale 0.7, and there are $121 = 11 \times 11$ regular interpolation points. Note that if the functions $u_j$ are not the standard Lagrange

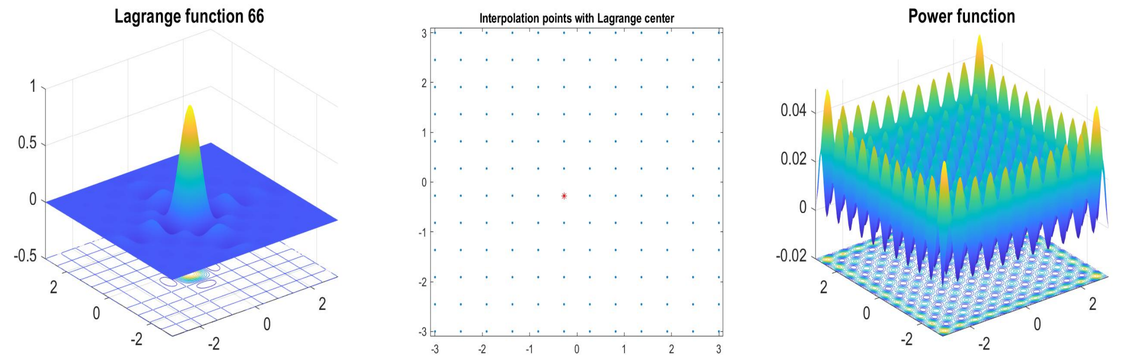

Figure 3: The Lagrange function for the central point and the squared Power Function with `testlag.m` using the Gaussian kernel with `RBFscale=0.7`.

basis using $K$, one has to use the formula `(eqpowgen)`

$$\begin{aligned} P^2(x) &= K(x,x) - 2\sum_{j=1}^{M} u_j(x)K(x_j,x) \\ &\quad + \sum_{j,k=1}^{M} u_j(x)u_k(x)K(x_j,x_k) \end{aligned} \tag{9}$$

for the non–optimal Power Function. This is useful for evaluation effects of badly chosen kernels, e.g. if a Lagrange basis $u_j$ coming from a different kernel is inserted. These programs were used to prepare examples in [10].

In the conditionally positive definite case, the Lagrange basis $u_1,\dots,u_M$ has to be extended by additional functions $v_1,\dots,v_Q$ and is to be solved for by the system

$$\begin{pmatrix} A_{X,X} & P_X \\ P_X^T & 0_{Q\times Q} \end{pmatrix}\begin{pmatrix} u(x) \\ v(x) \end{pmatrix} = \begin{pmatrix} K_X(x) \\ p(x) \end{pmatrix}$$

with

$$\begin{array}{rcl} K_X(x) & = & (K(x_1,x),\dots,K(x_M,x))^T, \\ p(x) & = & (p_1(x),\dots,p_Q(x))^T. \end{array}$$

On an evaluation set $Y$, we get

$$\begin{pmatrix} A_{X,X} & P_X \\ P_X^T & 0_{Q\times Q} \end{pmatrix}\begin{pmatrix} U_Y \\ V_Y \end{pmatrix} = \begin{pmatrix} A_{X,Y} \\ P_Y^T \end{pmatrix}$$

and the rows of $U_Y$ are now the Lagrange basis functions evaluated on $Y$. The square of the Power Function is 9. In the unconditionally positive definite case, the quadratic term cancels with one of the linear terms, thus simplifying to 8. In matrix form,

$$\begin{array}{rcl} P_X^2(x) & = & f(0) - 2u^T(x)K_X(x) - u^T(x)A_{X,X}u(x) \\ & = & f(0) - u^T(x)K_X(x) - u^T(x)P_Xv(x), \end{array}$$

to avoid work on $M\times M$ matrices. If we use $N$ points for evaluation on a set $Y$ and prepare matrices

$$\begin{array}{rclcll} \mathtt{U} & = & (u_j(y_k)) & = & U_Y^T, & 1\le k\le N,\ 1\le j\le M \\ \mathtt{V} & = & (v_i(y_k)) & = & V_Y^T, & 1\le k\le N,\ 1\le i\le Q \\ \mathtt{AXY} & = & (K(y_k,x_j)) & = & A_{Y,X}, & 1\le k\le N,\ 1\le j\le M \end{array}$$

with the row index mentioned first, then the column vector of values $P_X^2(Y)$ can be calculated in MATLAB notation via

$$f(0) - u^T(x)K_X(x) - u^T(x)P_Xv(x)$$
$$= \mathtt{frbf(0,0)\text{-}sum((U.*(AXY+V*PX)));}$$

The package contains a program `testlagCPD.m` for the conditionally positive definite case, and its output is in Figure 4. Like for Figure 2, we used the thin-plate spline, and the data are like for Figure 3. Note that the square of the Power Function is one decimal smaller now, and the Lagrangian has less additional wiggles.

### *4.4. Newton Basis Functions*

For points $x_1,x_2,\dots$, the Newton basis recursion [36, 38] is (eqNNNKKK)

$$\begin{array}{rcl} N_j(x_j)^2 & = & K(x_j,x_j) - \sum\limits_{m=1}^{j-1} N_m(x_j)^2,\ j\ge 1, \\ N_j(x)N_j(x_j) & = & K(x,x_j) - \sum\limits_{m=1}^{j-1} N_m(x)N_m(x_j),\ j\ge 1,\ x\in\Omega. \end{array} \qquad (10)$$

This generates functions $N_j$ on $\Omega$ with properties

$$\begin{array}{rcl} N_j(x) & \in & \operatorname{span}\{K(x,x_1),\dots,K(x,x_j)\} \\ N_j(x_k) & = & 0,\ 1\le k<j \\ P^2_{\{x_1,\dots,x_j\}}(x) & = & K(x,x) - \sum\limits_{m=1}^{j} N_m(x)^2,\ j\ge 1,\ x\in\Omega \end{array}$$

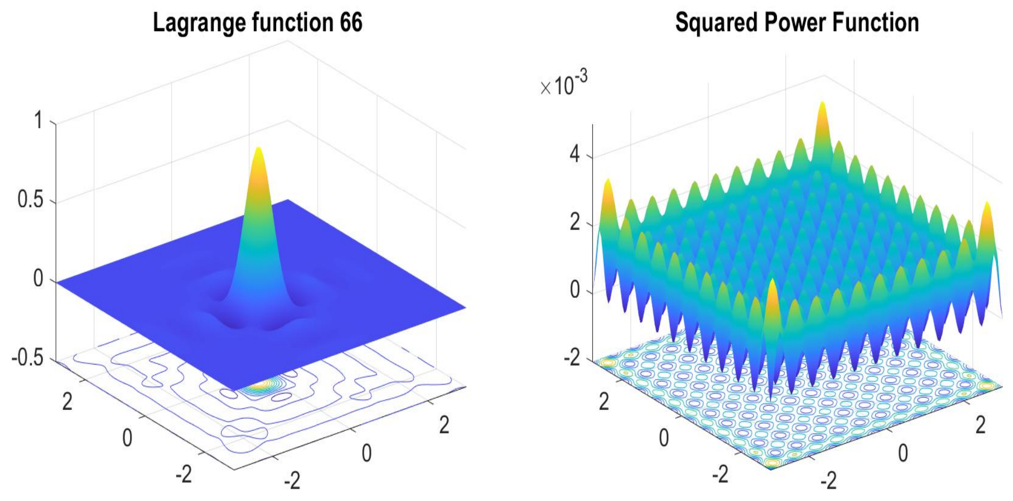

Figure 4: The Lagrange function for the central point and the squared Power Function with `testlagCPD.m` using thin-plate splines of order 2.

that explain their names when compared to the Newton basis for polynomials. The recursive algorithm implemented below works on a large point set $X_N$ of $N$ different points and stores up to step $j$ only the $j$ vectors with $N$ entries $N_i(x_k),\ 1 \leq i \leq j,\ 1 \leq k \leq N$. If a new point $x_{j+1}$ is chosen, a call of the form `kermat(X,`$x_{j+1}$`)` provides new kernel values without using a full kernel matrix. This basis has various stability advantages, in particular

$$\sum_{m=1}^{j} N_m^2(x) = K(x,x) - P^2_{\{x_1,\dots,x_j\}}(x) \leq K(x,x),\ x \in \Omega.$$

For choosing new points, a vector of $N$ values $K(x_k,x_k),\ 1 \leq k \leq N$ is allocated at startup and updated at step $j$ by subtraction of $N_j^2(x_k),\ 1 \leq k \leq N$. Then $x_{j+1}$ is picked as the $x_k$ where this vector attains its maximum, and by the above formulas it is the point where the Power Function $P^2_{\{x_1,\dots,x_j\}}(x)$ attains its maximum on $X_N$. If given $N$ points and performing $j$ steps, the storage is $\mathscr{O}(Nj)$ and the computational complexity is $\mathscr{O}(Nj^2)$. The algorithm can be viewed as a matrix-free updating version of a Cholesky decomposition or a Hilbert-Schmidt orthogonalization. When run on all $N$ points, the recursion (10) yields the Cholesky decomposition

$$K(x_j,x_k) = \sum_{m=1}^{N} N_m(x_j)N_m(x_k),\ 1 \leq j,k \leq N$$

of the full kernel matrix.

## *4.5. Adaptive Matrix–Free Routines*

The package contains a function `PFAdapCalcNewtonBasis` with the interface

```
function [ind, NBasis]=PFAdapCalcNewtonBasis(points, npmax, tol)
```

that implements the above method based on the Newton basis, avoiding to form a full kernel matrix. The input point matrix `points` should contain a large number of points, of which up to `npmax` are chosen.

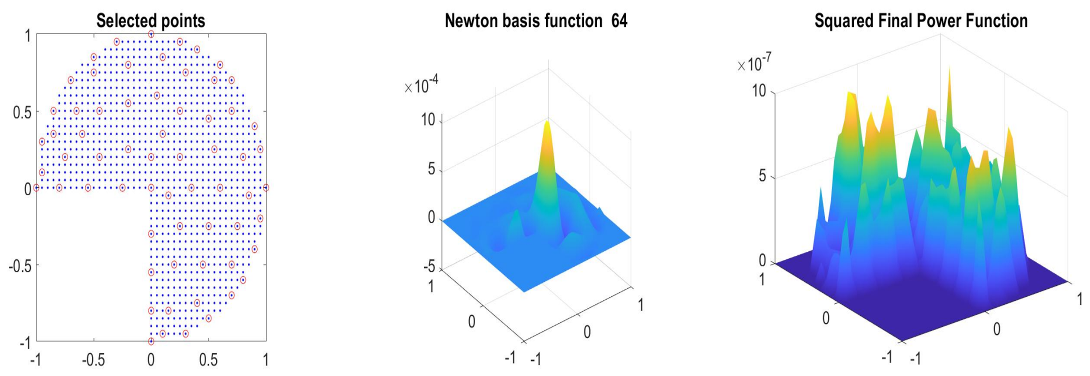

Figure 5: Selected points, the last Newton basis function, and the final squared Power Function with `testPFAdapCalcNewtonBasis.m` using the Matèrn kernel with `RBFpar`$= 3.5$, `RBFscale=1`.

The indices in `points` of the selected points are in `ind`, and `Nbasis` is an array holding the Newton basis on all points, columnwise. The final output of the calling routine `testPFAdapCalcNewtonBasis.m` is in Figure 5. When called on 963 points using the Matèrn kernel with `RBFpar`$= 3.5$, `RBFscale=1`, the routine picks 64 points at a tolerance of `tol`$= 10^{-6}$. The figure shows the selected points, the last Newton basis function, and the final squared Power Function.

The adaptivity can also be made dependent of the values of a function to be interpolated. The update rule does not use the Power Function. Instead, the new point $x_{j+1}$ is picked where the interpolant to $f$ using only $x_1,\dots,x_j$ has maximal error. It is called $f$-greedy in contrast to $P$-greedy in [35]. A convergence theory is in [58]. The $P$-greedy method dates back to [11] and has surprising optimality features [58].

The package implements the $f$-greedy method as

```
    function [ind,errfct,NBasis]=...
    AdapCalcNewtonBasisForFunction(X, fval, maxN, FunErrTol)
```

for a large point matrix `X` and function values `fval` there. Up to `maxN` points are picked, and the resulting point indices are in `ind`. The Newton basis values on `X` are in `NBasis`, and `errfct` contains the resulting error on `X`. The driver `testAdapCalcNewtonBasisForFunction.m` in the package is applied to the function $f(x,y) = \exp(|x-y|) - 1$ which has a derivative singularity for $x = y$. The output is in Figure 6, where the selected 120 interpolation points nicely gather along the singularity line. The kernel was Matèrn with `RBFpar`$= 1.5$ and `RBFscale=1`.
The basic routine is as follows:

```
function [ind, errfct, NBasis]=...
    AdapCalcNewtonBasisForFunction(X, fval, maxN, FunErrTol)
% The greedy f-adaptive Newton-based method,
% close to the 2011 paper Pazouki/Schaback.
% X : points to work on
% fval: function values on X
% maxN : maximal number of points of X to be used
% FunErrTol: stop if L_infty norm of residual is smaller than this
% ind: indices of selected points pf X
% errfct : final error on X
% NBasis: values of Newton basis on X
sk=zeros(maxN,1); % vector of coefficients in the Newton basis
[N pd]=size(X); % get info on points
V=zeros(N,maxN); % Newton basis columnwise
indselect=zeros(maxN,1); % selected poin t indices
P2=frbf(0.0,0)*ones(N,1); % squared Power Function on all points
% fval is now interpreted as the error of the actual approximation
for iter=1:maxN % main loop
    [fmax indfmax]=max(abs(fval)); % The maximum of the actual approsimation
    if fmax<FunErrTol % stopping criterion
        iter=iter-1; % reset step counter
        break;
    end
    % kernel values on the chosen point
    V(:,iter)=kermat(X,X(indfmax,:));
    for i=1:iter-1  % recursion
        V(:,iter)=V(:,iter)-V(:,i)*V(indfmax,i);
    end
    Pval=sqrt(P2(indfmax));
    V(:,iter)=V(:,iter)/Pval; % renormalization
    indselect(iter,1)=indfmax; % store new point index
    sk(iter)=fval(indfmax)/Pval; % the new cefficient in Newton basis
    for i=1:N
       fval(i)=fval(i)-V(i,iter)*sk(iter);
    end
    P2=max(0, P2-V(:,iter).^2); % update Power Function, keep nonnegative
    % and store our new point index:
    indselect(iter,1)=indfmax;
end
ind=indselect(1:iter);
errfct=fval;
NBasis=V(:,1:iter);
```

## 5. Routines for Multivariate Derivatives

This section is of quite some importance when radial basis functions are used for solving partial differential equations. One needs various derivatives of radial kernels, and one has to care for scaling and geometry. But like in `kermat(X,Y)` we just work on point sets and hide `S`-matrices and scaling. The routines are of the form `***kermat(X,Y)` and just differ by the differential operators applied to `X` or `Y`. The naming scheme is as follows:

If the operator $L$ acts on $X$, and $M$ acts on $Y$, then the routine is `LXMYkermat.m`.

Internally, they all boil down to calls of `frbf(S,k)` for various matrices `S` and derivative orders `k`. Like in `kermat(X,Y)`, the `S` matrices will contain values $\phi(\|x_i - y_j\|^2/(2c^2))$, $1 \leq i \leq m$, $1 \leq j \leq n$. If several of the following routines are used in the same project, there may be multiple calls for exactly the same

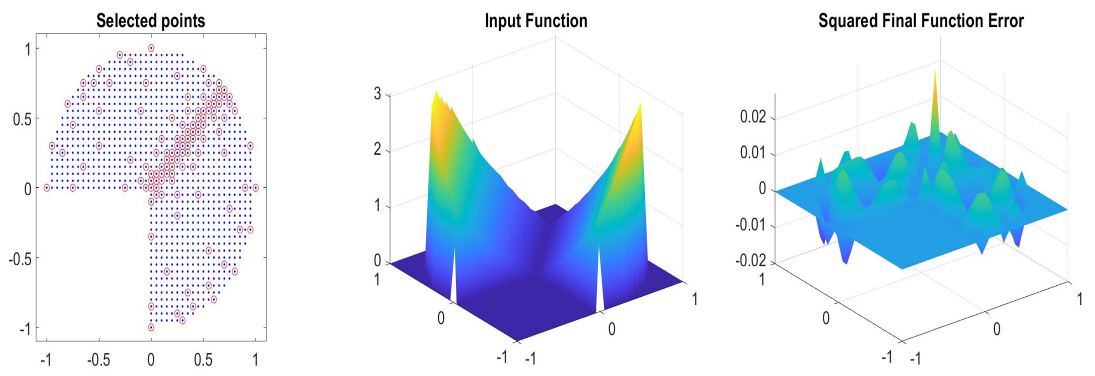

Figure 6: Selected points, input function, and the final function error with `testAdapCalcNewtonBasisForFunction.m` using the Matèrn kernel with `RBFpar`$= 1.5$, `RBFscale=1`.

`S` and `k` in different routines. When optimizing for speed, users should check this, execute the required calls outside of the routines, store the results into global variables, and avoid all recalculations.

## 5.1. *First Derivatives*

The $s$ derivatives in scalar form are

$$\begin{aligned} \frac{\partial s}{\partial x_i} &= +\frac{x_i - y_i}{c^2} \\ \frac{\partial s}{\partial y_i} &= -\frac{x_i - y_i}{c^2} \end{aligned}$$

and they occur all over again, e.g. in

$$\begin{aligned} \frac{\partial}{\partial x_j} K_c(x,y) &= f'(s)\frac{\partial s}{\partial x_j} \\ &= \frac{f'(s)}{c^2}(x_j - y_j) \end{aligned}$$

$$\begin{aligned} \frac{\partial}{\partial y_k} K_c(x,y) &= f'(s)\frac{\partial s}{\partial y_k} \\ &= -\frac{f'(s)}{c^2}(x_k - y_k). \end{aligned}$$

In matrix form for $1 \leq i \leq M,\ 1 \leq j \leq N,\ 1 \leq k \leq d$ this is

$$\begin{aligned} \left(\frac{\partial}{\partial x_k} K_c(x,y)\right)_{ij} &= \frac{f'(S_{ij})}{c^2}(X_{ik} - Y_{jk}) \\ \left(\frac{\partial}{\partial y_k} K_c(x,y)\right)_{ij} &= -\frac{f'(S_{ij})}{c^2}(X_{ik} - Y_{jk}) \end{aligned}$$

We provide a standard MATLAB routine for implementing the first formula:

```
function mat=gradXkermat(X, Y)
% creates kernel matrices
% for two point sets X and Y  of dimnsions dx=dy
% corresponding to the full gradient wrt. the X variable.
% The result is a 3-dimensional matrix of size  nx times ny times dx=dy,
global RBFscale
[nx dx]=size(X);
[ny dy]=size(Y);
fmat=frbf(distsqh(X,Y)/RBFscale^2,1)/RBFscale^2;
mat=zeros(nx,ny,dx);
for dim=1:dx
    mat(:,:,dim)=fmat.*(repmat(X(:,dim),1,ny)-repmat(Y(:,dim)',nx,1));
end
```

The second formula then is implemented by `function mat=gradYkermat(X, Y)` via `mat=-gradXkermat(X,Y)`, but in many applications one would prefer to call only one of these routines.

There also is a `function mat=gradXgradYkermat(X, Y)` that implements taking gradients with respect to both `X` and `Y`.

From here on, we omit the standard preamble in such routines and just show the implementation of the derivative formulae.

*5.2. Normals*

(`SecSubNormal`) Scalar normal or directional derivatives are prescribed via an additional matrix $N$ of normals or directions as rows, with $d$ columns. The pointwise case for the $j$-th component of the normal is

$$\begin{aligned}
\frac{\partial}{\partial \nu_j} &:= \sum_{k=1}^{d} N_{jk}\frac{\partial}{\partial x_k}\\
\frac{\partial}{\partial \nu_j}^{x} K_c(x,y) &= \sum_{k=1}^{d} N_{jk}\frac{\partial}{\partial x_k}K_c(x,y)\\
&= \frac{f'(s)}{c^2}\sum_{k=1}^{d} N_{jk}(x_k-y_k)\\
\frac{\partial}{\partial \nu_j}^{y} K_c(x,y) &= \sum_{k=1}^{d} N_{jk}\frac{\partial}{\partial y_k}K_c(x,y)\\
&= -\frac{f'(s)}{c^2}\sum_{k=1}^{d} N_{jk}(x_k-y_k)
\end{aligned}$$

and now for full matrices with $1 \le i \le M,\ 1 \le j \le N$:

$$\begin{aligned}
\left(\frac{\partial}{\partial \nu}^{x} K_c(x,y)\right)_{ij} &= \frac{f'(S_{ij})}{c^2}\sum_{k=1}^{d} N_{ik}(X_{ik}-Y_{jk})\\
&= \frac{f'(S_{ij})}{c^2}((NX^T)_{ii}-(NY^T)_{ij})
\end{aligned}$$

$$\begin{aligned}\left(\frac{\partial}{\partial \nu}^{y} K_c(x,y)\right)_{ij} &= -\frac{f'(S_{ij})}{c^2}\sum_{k=1}^{d} N_{jk}(X_{ik}-Y_{jk})\\ &= -\frac{f'(S_{ij})}{c^2}((XN^T)_{ij}-(NY^T)_{jj})\end{aligned}$$

The MATLAB routine implementing the first formula is

```
mat=normalXkermat(X, NX, Y)
```

for the normals wrt. the `X` variable. The basic statements there are

```
fmat=frbf(distsqh(X,Y)/RBFscale^2,1)/RBFscale^2;
mat=fmat.*(repmat(diag(NX*X),1,ny)-(NX*Y) );
```

The second formula then is implemented by `function mat=-normalYkermat(X, Y, NY)` via `mat=normalXkermat(Y, NY, X)`. There also is a
`function mat=normalXnormalYkermat(X, NX, Y, NY)`
that calculates normals for both variables. See Section 5.5.

*5.3. Second derivatives*

We start with the necessary full-length calculations:

$$\begin{aligned}\frac{\partial}{\partial x_r}\frac{\partial}{\partial x_p}K_c(x,y) &= \frac{\partial}{\partial x_r}\left(\frac{f'(s)}{c^2}(x_p-y_p)\right)\\ &= \frac{f'(s)}{c^2}\frac{\partial}{\partial x_r}(x_p-y_p)+(x_p-y_p)\frac{\partial}{\partial x_r}\left(\frac{f'(s)}{c^2}\right)\\ &= \frac{f'(s)}{c^2}\delta_{rp}+\frac{x_p-y_p}{c^2}f''(s)\frac{\partial s}{\partial x_r}\\ &= \frac{f'(s)}{c^2}\delta_{rp}+f''(s)\frac{x_r-y_r}{c^2}\frac{x_p-y_p}{c^2}\\ \left(\frac{\partial}{\partial x_r}\frac{\partial}{\partial x_p}K_c(x,y)\right)_{ij} &= \frac{f'(S_{ij})}{c^2}\delta_{rp}+\frac{f''(S_{ij})}{c^4}(X_{ir}-Y_{jr})(X_{ip}-Y_{jp})\\ \frac{\partial}{\partial y_s}\frac{\partial}{\partial y_k}K_c(x,y) &= \frac{\partial}{\partial y_s}\left(-\frac{f'(s)}{c^2}(x_k-y_k)\right)\\ &= -\frac{f'(s)}{c^2}\frac{\partial}{\partial y_s}(x_k-y_k)-(x_k-y_k)\frac{\partial}{\partial y_s}\left(\frac{f'(s)}{c^2}\right)\\ &= \frac{f'(s)}{c^2}\delta_{sk}-\frac{x_k-y_k}{c^2}f''(s)\frac{\partial s}{\partial y_s}\\ &= \frac{f'(s)}{c^2}\delta_{sk}+f''(s)\frac{x_s-y_s}{c^2}\frac{x_k-y_k}{c^2}\\ \left(\frac{\partial}{\partial y_s}\frac{\partial}{\partial y_k}K_c(x,y)\right)_{ij} &= \frac{f'(S_{ij})}{c^2}\delta_{sk}+\frac{f''(S_{ij})}{c^4}(X_{is}-Y_{js})(X_{ik}-Y_{jk})\\ \frac{\partial}{\partial y_k}\frac{\partial}{\partial x_p}K_c(x,y) &= \frac{\partial}{\partial y_k}\left(\frac{f'(s)}{c^2}(x_p-y_p)\right)\\ &= \frac{f'(s)}{c^2}\frac{\partial}{\partial y_k}(x_p-y_p)+(x_p-y_p)\frac{\partial}{\partial y_k}\left(\frac{f'(s)}{c^2}\right)\\ &= -\frac{f'(s)}{c^2}\delta_{kp}+\frac{x_p-y_p}{c^2}f''(s)\frac{\partial s}{\partial y_k}\\ &= -\frac{f'(s)}{c^2}\delta_{kp}-f''(s)\frac{x_p-y_p}{c^2}\frac{x_k-y_k}{c^2}\end{aligned}$$

$$\left(\frac{\partial}{\partial y_k}\frac{\partial}{\partial x_p}K_c(x,y)\right)_{ij} = -\frac{f'(S_{ij})}{c^2}\delta_{kp} - \frac{f''(S_{ij})}{c^4}(X_{ip}-Y_{jp})(X_{ik}-Y_{jk})$$

These formulas are not yet implemented in full generality in the package, since there was no application for them, so far. Instead, we shall focus on special cases below.

## *5.4. Laplace operators*

(`SecSubLaplacian`) From the previous section, we get

$$\begin{aligned}
\Delta^x K_c(x,y) &= \frac{df'(s)}{c^2}+\frac{f''(s)}{c^4}\|x-y\|^2\\
&= \frac{df'(s)}{c^2}+\frac{2sf''(s)}{c^2}\\
\Delta^y K_c(x,y) &= \frac{df'(s)}{c^2}+\frac{f''(s)}{c^4}\|x-y\|^2\\
&= \frac{df'(s)}{c^2}+\frac{2sf''(s)}{c^2}\\
&=: g(s),
\end{aligned}$$

and $g(s)$ can be considered like a new kernel generated by $g$ instead of $f$.

In matrix form:

$$(\Delta^x K_c(x,y))_{ij} = g(S_{ij}) = (\Delta^x y K_c(x,y))_{ij}$$

$$g(S_{ij}) = \frac{df'(S_{ij})}{c^2}+\frac{2S_{ij}f''(S_{ij})}{c^2}$$

with a rather trivial implementation `mat=(dx*frbf(s,1)+2*s.*frbf(s,2))/RBFscale^2+` in

`function mat=laplacekermat(X, Y).`

To keep the naming conventions, there are also routines `laplaceXkermat.m` and `laplaceYkermat.m`. For later use, we collect derivatives of $g$:

$$\begin{aligned}
g'(s) &= \frac{df''(s)}{c^2}+\frac{2(sf''(s))'}{c^2}\\
&= \frac{df''(s)}{c^2}+\frac{2f''(s)}{c^2}+\frac{2sf'''(s)}{c^2}\\
&= \frac{(d+2)f''(s)}{c^2}+\frac{2sf'''(s)}{c^2},
\end{aligned}$$

$$\begin{aligned}
g''(s) &= \frac{(d+2)f'''(s)}{c^2}+\frac{2(sf'''(s))'}{c^2}\\
&= \frac{(d+2)f'''(s)}{c^2}+\frac{2f'''(s)}{c^2}+\frac{2sf^{(4)}(s)}{c^2}\\
&= \frac{(d+4)f'''(s)}{c^2}+\frac{2sf^{(4)}(s)}{c^2}.
\end{aligned}$$

## *5.5. Mixed Derivatives*

Here, we look at cases where different operators act on the $x$ and $y$ arguments of a kernel.

### *5.5.1. Mixed Normals or Directional Derivatives*

(`SecSubMixedNormals`) For mixed normal or directional derivatives we assume two matrices $N^X$ and $N^Y$ of normals or directions wrt. the points in $X$ and $Y$. The pointwise case is

$$\begin{aligned}
\frac{\partial}{\partial v_i}^x \frac{\partial}{\partial v_p}^y K_c(x,y) &= \sum_{j=1}^d N_{ij}^X \frac{\partial}{\partial x_j}\left(\sum_{k=1}^d N_{pk}^Y \frac{\partial}{\partial y_k} K_c(x,y)\right)\\
&= \sum_{j=1}^d N_{ij}^X \sum_{k=1}^d N_{pk}^Y \frac{\partial}{\partial x_j}\frac{\partial}{\partial y_k} K_c(x,y)\\
&= \sum_{j=1}^d N_{ij}^X \sum_{k=1}^d N_{pk}^Y \left(-\frac{f'(s)}{c^2}\delta_{kj} - f''(s)\frac{x_j-y_j}{c^2}\frac{x_k-y_k}{c^2}\right)\\
&= -\frac{f'(s)}{c^2}\sum_{j=1}^d N_{ij}^X N_{pj}^Y\\
&\quad -\frac{f''(s)}{c^4}\left(\sum_{j=1}^d N_{ij}^X (x_j-y_j)\right)\left(\sum_{k=1}^d N_{pk}^Y (x_k-y_k)\right)
\end{aligned}$$

and for matrices we get

$$\begin{aligned}
&\left(\frac{\partial}{\partial v}^x \frac{\partial}{\partial v}^y K_c(x,y)\right)_{ij}\\
= \;& -\frac{f'(S_{ij})}{c^2}\sum_{k=1}^d N_{ik}^X N_{jk}^Y\\
& -\frac{f''(S_{ij})}{c^4}\left(\sum_{k=1}^d N_{ik}^X (X_{ik}-Y_{jk})\right)\left(\sum_{k=1}^d N_{jk}^Y (X_{ik}-Y_{jk})\right)\\
= \;& -\frac{f'(S_{ij})}{c^2}(N^X(N^Y)^T)_{ij}\\
& -\frac{f''(S_{ij})}{c^4}\left((N^X X^T)_{ii} - (N^X Y^T)_{ij}\right)\left((X(N^Y)^T)_{ij} - (N^Y Y^T)_{jj}\right)
\end{aligned}$$

Our MATLAB program

```
function mat=normalXnormalYkermat(X, NX, Y, NY)
```

implements this as

```
mat=-frbf(s,1).*(NX*NY)/RBFscale^2-...
frbf(s,2)/RBFscale^4.*...
(repmat(diag(NX*X),1,ny)-NX*Y).*(X*NY-repmat(diag(NY*Y),nx,1));
```

### *5.5.2. Mixed Laplacians*

Mixed scalar Laplacian values are

$$\begin{aligned}
&\Delta^x\Delta^y K_c(x,y)\\
= \;& \frac{d}{c^2}g'(s) + \frac{2s}{c^2}g''(s)\\
= \;& \frac{d}{c^2}\left(\frac{(d+2)f''(s)}{c^2} + \frac{2sf'''(s)}{c^2}\right) + \frac{2s}{c^2}\left(\frac{(d+4)f'''(s)}{c^2} + \frac{2sf^{(4)}(s)}{c^2}\right)\\
= \;& \frac{1}{c^4}\left(d(d+2)f''(s) + 4s(d+2)f'''(s) + 4s^2 f^{(4)}(s)\right).
\end{aligned}$$

Our MATLAB program

```
function mat=laplaceXYkermat(X, Y)
```

implements this as

```
mat=(dx*(dx+2)*frbf(s,2)+4*(dx+2)*s.*frbf(s,3)+4*s.^2.*frbf(s,4))/RBFscale^4;
```

### 5.6. *Other Mixed Derivatives*

The package also contains functions `laplaceXnormalYkermat.m` and `laplaceYnormalXkermat.m` for mixing normal and Laplace derivatives, but we skip over them here, for brevity.

## 6. Applications

This section provides brief sketches of a few applications of the routines of the previous sections.

### 6.1. *Collocation*

For an integral or differential equation problem on a domain, *collocation* methods write the equations down in a pointwise fashion. A typical example is the *Poisson problem* to find a smooth function $u$ on a domain $\Omega$ satisfying the equations

$$\begin{array}{rcll} \Delta u &=& f^\Omega & \text{in } \Omega \subset \mathbb{R}^d \\ u &=& f^D & \text{in } D \subseteq \Gamma := \partial\Omega \subset \mathbb{R}^d \end{array}$$

for given functions $f^\Omega$ in the interior and $f^D$ on the boundary. Collocation would take

1. points $x_j$, $1 \le j \le J$ in $\Omega \cup \partial\Omega$ for values of $\Delta u$,
2. points $y_k$, $1 \le k \le K$ on the boundary $\partial\Omega$ for values of $u$

to replace the problem by the $J+K$ equations

$$\begin{array}{rcll} \Delta u(x_j) &=& f^\Omega(x_j), & 1 \le j \le J, \\ u(y_k) &=& f^D(y_k), & 1 \le k \le K. \end{array}$$

When composing approximate solutions $u$ from kernel translates, there are two popular methods to proceed.

#### 6.1.1. *Kansa's Method*

This goes back to [23] and uses points $z_1, \dots, z_M$ in $\Omega \cup \partial\Omega$ to form the *trial space* $U_M$ of linear combinations of all kernel translates $u_m(x) := \phi(\|z_m - x\|)$, $x \in \Omega$, $1 \le m \le M$. It then poses the problem in the subspace $U_M$. This is easy to set up using routines of the previous sections. With point matrices $X$, $Y$, $Z$, the linear $(J+K) \times M$ system for a coefficient vector $c_Z \in \mathbb{R}^M$ is simply

$$\begin{pmatrix} \texttt{laplaceXkermat(X,Z)} \\ \texttt{kermat(Y,Z)} \end{pmatrix} \begin{pmatrix} c_Z \end{pmatrix} = \begin{pmatrix} f_X^\Omega \\ f_Y^D \end{pmatrix}$$

where the lower subscripts indicate the sets to which the data vectors correspond.

Of course, one needs $J+K \le M$ and would like to get away with a square matrix in case $J+K = M$. This works fine in nearly every case [34], and users should use `condest` to check for instability. However, it can be proven [21] that there is no guarantee for solvability in the square case. Convergence proofs and error estimates are available [44, 27, 46] in case of *oversampling*, i.e. $M$ sufficiently larger than $J+K$ with proper point selection. It should be clear how to add Neumann boundary conditions via `normalYkermat`.

### 6.1.2. Symmetric Collocation

This enforces symmetric positive definite $(J+K)\times(J+K)$ matrices by changing the trial space to consist of linear combinations of kernel translates $v_j(x) := \Delta\phi(\|x_j - x\|)$, $x\in\Omega$, $1\le j\le J$ and $w_k(x) := \phi(\|y_k - x\|)$, $x\in\Omega$, $1\le k\le K$. The linear system now is `(eqKansasys)`

$$\begin{pmatrix} \texttt{laplaceXlaplaceYkermat(X,X)} & \texttt{laplaceXkermat(X,Y)} \\ \texttt{laplaceXkermat(X,Y)}^T & \texttt{kermat(Y,Y)} \end{pmatrix} \begin{pmatrix} c_X \\ c_Y \end{pmatrix} = \begin{pmatrix} f_X^\Omega \\ f_Y^D \end{pmatrix} \tag{11}$$

for the coefficient vectors $c_X$ for the $v_j$ and $c_Y$ for the $w_k$. This setting [18, 17] reduces to generalized "Hermite" interpolation [61] and inherits the optimality principles of kernel-based interpolation. In particular, in the native Hilbert space of the kernel used, it realizes the smallest error norm under all linear numerical methods that use the same data [47]. On the downside, it takes derivatives of order four in `laplaceXYkermat(X,X)`.
The distribution of points, the true solution $1-x^2-y^2$, and the error for unsymmetric and symmetric collocation on 61 boundary and 301 interior points by using the multiquadric kernel with RBFpar=1/2 and RBFscale=1 are given in the Figure 7. The corresponding m-file in the package is `testcoll.m`.

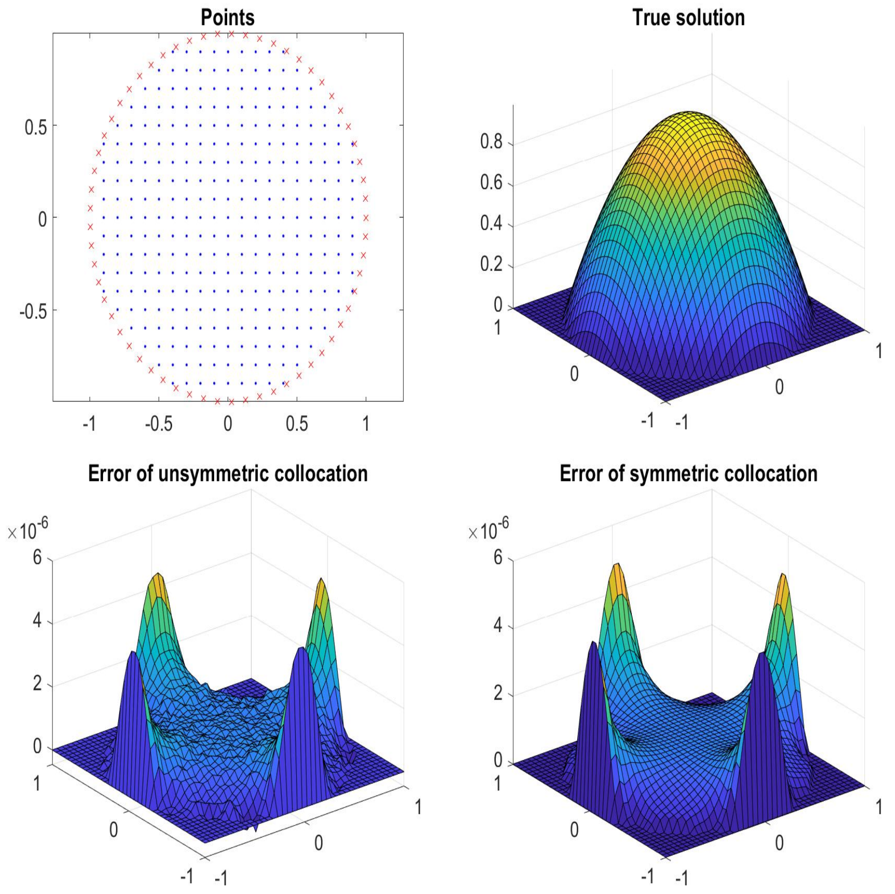

Figure 7: Points, true solution $1-x^2-y^2$, and error for unsymmetric and symmetric collocation on 61 boundary and 301 interior points by using the multiquadric kernel with RBFpar=1/2 and RBFscale=1. The corresponding m-file in the package is `testcoll.m`.

### *6.1.3. Method of Lines*

This is closely related to collocation and reduces time-dependent partial differential equations to large ordinary differential equation systems. As an example, we take the heat equation

$$\begin{array}{rcll} \Delta^x u(x,t) & = & \frac{d}{dt}u(x,t) & x \in \Omega \subset \mathbb{R}^d,\ t \geq 0, \\ u(x,t) & = & 0 & x \in \partial\Omega,\ t \geq 0, \\ u(x,0) & = & f^S(x) & x \in \Omega,\ f^S(x) = 0, x \in \partial\Omega \end{array}$$

for a given function $f^\Omega$ in the interior and a starting function $f^S$ in the domain.

The method takes a trial space $U_M = span\{u_1, \dots, u_M\}$ of smooth functions on $\Omega$ that vanish on the boundary. Then the system is approximated by

$$\begin{array}{rcl} \sum_{m=1}^{m} \Delta^x u_m(x) c_m(t) & = & \sum_{m=1}^{m} u_m(x) c'_m(t) \\ \sum_{m=1}^{m} u_m(x) c_m(0) & = & f^S(x). \end{array}$$

In the simplest case, the basis functions $u_m$ are a chosen as a Lagrange basis for a set of $J$ points in the interior of $\Omega$ forming a point matrix $X$. Then the starting coefficient vector is $c(0) = f^S(X)$, and the differential equation system is

$$c'_j(t) = \sum_{m=1}^{m} \Delta^x u_m(x_j) c_m(t),\ 1 \leq j \leq M.$$

MATLAB has various functions to solve such systems, provided that the $M \times J$ matrix $\Delta^X U_M$ with elements $\Delta^x u_m(x_j)$ is precalculated. From section 4.2 we know that the vector $u(x)$ of Lagrange basis functions solves the system

$$\texttt{kermat(X,X)}\, u(x) = \texttt{kermat(X,x)},$$

and the Laplacian of this is

$$\texttt{kermat(X,X)}\, \Delta u(x) = \texttt{laplaceXkermat(X,x)}.$$

Therefore the required matrix $\Delta^X U_M$ solves

$$\texttt{kermat(X,X)}\, \Delta^X U_M = \texttt{laplaceXkermat(X,X)}.$$

This technique has an easy error analysis and generalizes to more general equations [22, 30]

Figure 8 provides a simple example on $\Omega = [-1, +1]$ with starting function $f^S(x) = 1 - x^2$ showing the exponential decay by using the RTH kernel with `RBFpar=RBFscale=`0.1. The program in the package is `testheat.m`.

### *6.1.4. Local Kernel-Based Methods*

So far, the primary emphasis has been on the global approach for RBFs, i.e. working on a large point set $X_N := \{x_i\}_{i=1}^N$ in $\mathbb{R}^d$ as a whole.
But in numerous cases, working on small subsets of the full point set can achieve accuracy comparable to global methods.

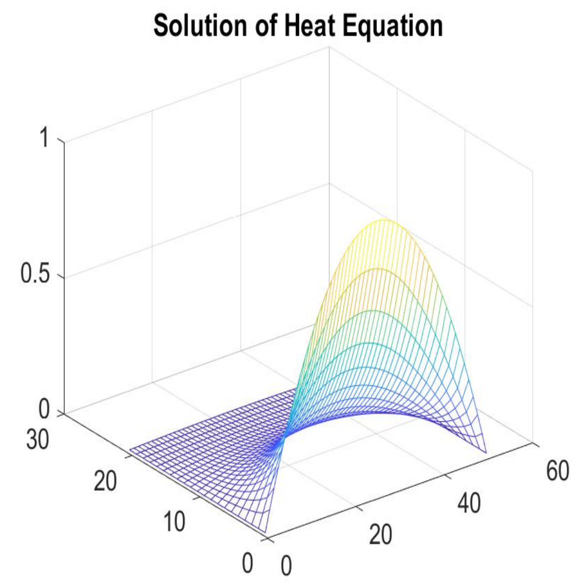

Figure 8: A solution of the heat equation with the Method of Lines by using the RTH kernel with `RBFpar=RBFscale=`0.1. The corresponding m-file in the package is `testheat.m`.

We first sketch the interpolation case and assume data $f(x_1),\dots,f(x_N)$ on the full set $X_N$. If $X(y)\subseteq X_N$ is a subset of points close to $y$, a local interpolant at $y$ can be calculated like the global one, but using only the local data. We denote it by $s_{X(y),f}$ but evaluate it only at $y$ or nearby. Since the standard error bounds for kernel-based interpolation are local anyway, this *upsampling* strategy can be comparable to the global case, at much lower computational effort, but the local pieces will not lead to a smooth global interpolant. This drawback can be overcome by *Partition-of-Unity* methods [3, 8, 26, 56, 57], but we ignore them here. The selection of good sets $X(y)$ is treated in [49] by a recursive greedy method minimizing the Power Function $P_{x_1,\dots,x_n}(y)$ with respect to $x_n$.

In most cases, it is adequate to write the solution not in terms of coefficients for a basis representation, but in terms of the given data values. The result will then be basis-independent and written in *stencil* form

$$s_{X(y),f}=\sum_{x_i\in X(y)}\alpha_i f(x_i).$$

If $L$ is a linear differential operator, the recovery of $L(f)(y)$ from values at neighbouring points can take the same stencil form, namely

$$L(f)(y)\approx\sum_{x_i\in X(y)}\alpha_i^L f(x_i). \tag{12}$$

A simple example using the package for approximation $\Delta f(y)$ from values on a set `Xy` is solved for a coefficient vector a by

```
    a=kermat(Xy,Xy)\laplaceYkermat(Xy,y).
```

This generalizes *finite differences* and has a vast literature, with [6, 7] as kernel-based examples. The above straightforward MATLAB approach uses the standard basis of kernel translates, but one can do better using the Newton basis [33, 37, 41, 42, 40] or special techniques adapted to Gaussians [24].

Optimality principles for kernel-based stencils are in [6, 7], and error bounds in [5]. The connection of stencils with Moving Least Squares is treated in [29].

The main application of stencils as finite differences arises in the discretization of partial differential equations via the *RBF-FD* method. It can be seen as a localized collocation [33, 37, 41, 42, 40] written in terms of function values and has a long history [15, 53, 60, 62] and is a meshfree nodal method [1, 48]. The values $u(x_1),\dots,u(x_n)$ of the unknown solution of $Lu = f$ on a set $X_n = \{x_1,\dots,x_n\}$ are the simultaneous input to stencils on a set $Y_m = \{y_1,\dots,y_m\}$ based on (12). The stencil coefficients then form the rows of a linear system

$$L(u)(y_j) \approx \sum_{x_i\in X(y_j)} \alpha^L_{j,i} u(x_i),\ 1 \le j \le m.$$

The program system `mFDlab` by Oleg Davydov [4] extends this package towards kernel-based finite differences.

The method can also be used within the Method of Lines to solve time-dependent PDEs. The unknowns are functions $u(x_j,t)$, and the spatial differential operators are discretized via stencils to get a system of Ordinary Differential Equations. This has a long history as well [33, 37, 41, 42, 62]

## References

[1] T. Belytschko, Y. Krongauz, D.J. Organ, M. Fleming, and P. Krysl. Meshless methods: an overview and recent developments. *Computer Methods in Applied Mechanics and Engineering, special issue*, 139:3–47, 1996.

[2] M.D. Buhmann. *Radial Basis Functions, Theory and Implementations*. Cambridge University Press, Cambridge,UK, 2003.

[3] R. Cavoretto, S. De Marchi, A. De Rossi, E. Perracchione, and G. Santin. Partition of unity interpolation using stable kernel-based techniques. *Applied Numerical Mathematics*, 116:95–107, 2017. New Trends in Numerical Analysis: Theory, Methods, Algorithms and Applications (NETNA 2015).

[4] O. Davydov. Program package mFDlab, 2020. https://bitbucket.org/meshlessFD/mfdlab/src/master/.

[5] O. Davydov and R. Schaback. Error bounds for kernel-based numerical differentiation. *Numerische Mathematik*, 132:243–269, 2016.

[6] O. Davydov and R. Schaback. Minimal numerical differentiation formulas. *Numerische Mathematik*, 140:555–592, 2018.

[7] O. Davydov and R. Schaback. Optimal stencils in Sobolev spaces. *IMA Journal of Numerical Analysis*, 39:398–422, 2019.

[8] S. De Marchi, A. MartÃ■nez, and E. Perracchione. Fast and stable rational rbf-based partition of unity interpolation. *Journal of Computational and Applied Mathematics*, 349:331–343, 2019.

[9] St. De Marchi and R. Schaback. Nonstandard kernels and their applications. *Dolomites Research Notes on Approximations*, 2:16–43, 2009.

[10] St. De Marchi and R. Schaback. Stability of kernel-based interpolation. *Adv. in Comp. Math.*, 32:155–161, 2010.

[11] St. De Marchi, R. Schaback, and H. Wendland. Near-optimal data-independent point locations for radial basis function interpolation. *Adv. Comput. Math.*, 23(3):317–330, 2005.

[12] J. Duchon. Interpolation des fonctions de deux variables suivant le principe de la flexion des plaques minces. *Rev. Francaise Automat. Informat. Rech. Opér. Anal. Numer.*, 10:5–12, 1976.

[13] G. Fasshauer and M. McCourt. *Kernel-based Approximation Methods using MATLAB*, volume 19 of *Interdisciplinary Mathematical Sciences*. World Scientific, Singapore, 2015.

[14] G. F. Fasshauer. *Meshfree Approximation Methods with MATLAB*, volume 6 of *Interdisciplinary Mathematical Sciences*. World Scientific Publishers, Singapore, 2007.

[15] Natasha Flyer, Grady B. Wright, and Bengt Fornberg. *Radial Basis Function-Generated Finite Differences: A Mesh-Free Method for Computational Geosciences*, pages 2635–2669. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[16] Bengt Fornberg and Natasha Flyer. *A primer on radial basis functions with applications to the geosciences*. SIAM, 2015.

[17] C. Franke and R. Schaback. Convergence order estimates of meshless collocation methods using radial basis functions. *Advances in Computational Mathematics*, 8:381–399, 1998.

[18] C. Franke and R. Schaback. Solving partial differential equations by collocation using radial basis functions. *Appl. Math. Comp.*, 93:73–82, 1998.

[19] Mohammad Heidari, Maryam Mohammadi, and Stefano De Marchi. A shape preserving quasi-interpolation operator based on a new transcendental RBF. *Dolomites Research Notes on Approximation*, 14:56–73, 2021.

[20] Mohammad Heidari, Maryam Mohammadi, Stefano De Marchi, et al. Curvature-based characterization of radial basis functions: application to interpolation. *Mathematical Modelling and Analysis*, 28(3):415–433, 2023.

[21] Y.C. Hon and R. Schaback. Solvability of partial differential equations by meshless kernel methods. *Advances in Computational Mathematics*, 28:283–299, 2008.

[22] Y.C. Hon, R. Schaback, and M. Zhong. The meshless kernel-based method of lines for parabolic equations. *Computers and Mathematics with Applications*, 68(12, Part A):2057–2067, 2014.

[23] E. J. Kansa. Application of Hardy's multiquadric interpolation to hydrodynamics. In *Proc. 1986 Simul. Conf., Vol. 4*, pages 111–117, 1986.

[24] E. Larsson, E. Lehto, A. Heryodono, and B. Fornberg. Stable computation of differentiation matrices and scattered node stencils based on Gaussian radial basis functions. *SIAM J. Sci. Comput.*, 35:A2096–A2119, 2013.

[25] E. Larsson and R. Schaback. Scaling of radial basis functions. *IMA Journal of Numerical Analysis*, 2023.

[26] Elisabeth Larsson, Victor Shcherbakov, and Alfa Heryudono. A least squares radial basis function partition of unity method for solving PDEs. *SIAM Journal on Scientific Computing*, 39(6):A2538–A2563, 2017.

[27] L. Ling and R. Schaback. Stable and convergent unsymmetric meshless collocation methods. *SIAM J. Numer. Anal.*, 46:1097–1115, 2008.

[28] G. Matheron. *Les variables régionaliseés et leur estimation*. Masson, Paris, 1965.

[29] D. Mirzaei, R. Schaback, and M. Dehghan. On generalized moving least squares and diffuse derivatives. *IMA J. Numer. Anal.*, 32, No. 3:983–1000, 2012.

[30] M. Mohammadi, R. Mokhtari, and R. Schaback. A meshless method for solving the 2D Brusselator reaction-diffusion system. *Computer Modeling in Engineering & Sciences*, 101(2):113–138, 2014.

[31] Maryam Mohammadi, Stefano De Marchi, and Mohammad Karimnejad Esfahani. Full-rank orthonormal bases for conditionally positive definite kernel-based spaces. *Journal of Computational and Applied Mathematics*, 444:115761, 2024.

[32] Maryam Mohammadi, Mohammad Heidari, and Stefano De Marchi. Non-oscillatory solutions of the 2d coupled burgers' equations using the rth rbf method. In *American Institute of Physics Conference Series*, volume 3094, page 320006, 2024.

[33] Maryam Mohammadi, Fahimeh Saberi Zafarghandi, Esmail Babolian, and Shahnam Javadi. A local reproducing kernel method accompanied by some different edge improvement techniques: application to the burgers equation. *Iranian Journal of Science and Technology, Transactions A: Science*, 42:857–871, 2018.

[34] Maryam Mohammadi, Alvise Sommariva, and Marco Vianello. Unisolvence of Kansa collocation for elliptic equations by polyharmonic splines with random fictitious centers, 2024.

[35] St. Müller. *Komplexität und Stabilität von kernbasierten Rekonstruktionsmethoden*. PhD thesis, University of Göttingen, 2009. https://ediss.uni-goettingen.de/handle/11858/00-1735-0000-0006-B3BA-E.

[36] St. Müller and R. Schaback. A Newton basis for kernel spaces. *Journal of Approximation Theory*, 161:645–655, 2009.

[37] Hananeh Nojavan, Saeid Abbasbandy, and Maryam Mohammadi. Local variably scaled newton basis functions collocation method for solving burgers equation. *Applied Mathematics and Computation*, 330:23–41, 2018.

[38] M. Pazouki and R. Schaback. Bases for kernel-based spaces. *Computational and Applied Mathematics*, 236:575–588, 2011.

[39] E. Porcu, M. Bevilacqua, A.A. Alegria, C. Oates, and R. Schaback. The Matérn model: A journey through statistics, numerical analysis and machine learning. *Statist. Sci.*, 39(3):469–492, 2024.

[40] H Rafieayanzadeh, M Mohammadi, and E Babolian. Solving a class of pdes by a local reproducing kernel method with an adaptive residual subsampling technique. *CMES-COMPUTER MODELING IN ENGINEERING & SCIENCES*, 108(6):375–395, 2015.

[41] Hossein Rafieayanzadeh, Maryam Mohammadi, Esmail Babolian, et al. Numerical solution of sigularly perturbed parabolic problems by a local kernel-based method with an adaptive algorithm. *Journal of Mathematical Modeling*, 7(3):319–336, 2019.

[42] Fahimeh Saberi Zafarghandi, Maryam Mohammadi, Esmail Babolian, and Shahnam Javadi. A localized newton basis functions meshless method for the numerical solution of the 2d nonlinear coupled burgers equations. *International Journal of Numerical Methods for Heat & Fluid Flow*, 27(11):2582–2602, 2017.

[43] R. Schaback. Reconstruction of multivariate functions from scattered data. Manuscript, available via http://webvm.num.math.uni-goettingen.de/schaback/teaching/rbfbook_2.pdf, 1997.

[44] R. Schaback. Convergence of unsymmetric kernel-based meshless collocation methods. *SIAM J. Numer. Anal.*, 45(1):333–351 (electronic), 2007.

[45] R. Schaback. Matlab programming for kernel-based methods. Technical report, Institut für Numerische und Angewandte Mathematik Göttingen, 2009. Preprint, available via http://num.math.uni-goettingen.de/schaback/research/papers/MPfKBM.pdf.

[46] R. Schaback. Unsymmetric meshless methods for operator equations. *Numerische Mathematik*, 114:629–651, 2010.

[47] R. Schaback. A computational tool for comparing all linear PDE solvers. *Advances in Computational Mathematics*, 41:333–355, 2015.

[48] R. Schaback. Error analysis of nodal meshless methods. In M. Griebel and M.A. Schweitzer, editors, *Meshfree Methods for Partial Differential Equations VIII*, volume 115 of *Lecture Notes in Computational Science and Engineering*, pages 117–143. Springer, 2017.

[49] R. Schaback. Greedy adaptive local recovery of functions in Sobolev spaces, 2024. https://arxiv.org/abs/2407.19864.

[50] R. Schaback and H. Wendland. Kernel techniques: from machine learning to meshless methods. *Acta Numerica*, 15:543–639, 2006.

[51] R. Schaback and Z. Wu. Operators on radial basis functions. *J. Comp. Appl. Math.*, 73:257–270, 1996.

[52] M. Scheuerer, M. Schlather, and R. Schaback. Interpolation of spatial data - a stochastic or a deterministic problem? *European Journal of Applied Mathematics*, 24:601–629, 2013.

[53] A.I. Tolstykh. On using radial basis functions in a "finite difference mode" with applications to elasticity problems. *Comput. Mech.*, 33:68–79, 2003.

[54] EW Weisstein. General identities: Differentiation. from mathworld–a wolfram web resource.

[55] H. Wendland. Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree. *Advances in Computational Mathematics*, 4:389–396, 1995.

[56] H. Wendland. Fast evaluation of radial basis functions: Methods based on partition of unity. In C. K. Chui, L. L. Schumaker, and J. Stöckler, editors, *Approximation Theory X: Wavelets, Splines, and Applications*, pages 473–483. Vanderbilt University Press, 2002.

[57] H. Wendland. *Scattered Data Approximation*. Cambridge University Press, Cambridge,UK, 2005.

[58] T. Wenzel, G. Santin, and B. Haasdonk. Analysis of target data-dependent greedy kernel algorithms: Convergence rates for $f$-,$f \cdot p$, and $f/p$ -greedy. *Constructive Approximation*, 57:45–74, 2023.

[59] Edwin G Wintucky. *Formulas for nth order derivatives of hyperbolic and trigonometric functions*. National Aeronautics and Space Administration, 1971.

[60] G.B. Wright and B. Fornberg. Scattered node compact finite difference-type formulas generated from radial basis functions. *J. Comput. Phys.*, 212(1):99–123, 2006.

[61] Z. Wu. Hermite–Birkhoff interpolation of scattered data by radial basis functions. *Approximation Theory and its Applications*, 8/2:1–10, 1992.

[62] G.M. Yao, B. Šarler, and C. S. Chen. A comparison of three explicit local meshless methods using radial basis functions. *Eng. Anal. Bound. Elem.*, 35(3):600–609, 2011.