MATLAB Programming for Kernel–Based Methods

Robert Schaback, Göttingen

Draft of October 20, 2011

This technical report contains some hopefully helpful stuff for writing MATLAB programs for kernel–based methods. Note that the book [3] by Greg Fasshauer contains a very good and competitive collection of MATLAB programs, and note that my own MATLAB is an outdated antique version, sorry...

1 Kernels and Points

We assume x, y to be vectors in $\mathbb{R}^d$, and we consider radial kernels of the form

$$K(x,y) = \phi(\|x-y\|_2), \quad \text{for all } x,\ y \in \mathbb{R}^d.$$

1.1 Point Sets

We store points in MATLAB/FORTRAN–style as rows of matrices with d columns, e.g. $x_1,\dots,x_M \in \mathbb{R}^d$ as rows of a matrix $X \in \mathbb{R}^{M\times d}$. Note that MATLAB runs through arrays in a columnwise way, like FORTRAN. Thus unsymmetric arrays should always be stored such that there are more rows than columns.

For univariate cases, note that MATLAB sequences like `t=-1:0.01:1` generate a row, not a column.

Random sets of n points in d dimensions within $[0,1]^d$ are generated via `p=rand(n,d)`. To generate random points in $[a,b]^d$, use `p=a+(b-a)*rand(n,d)`.

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

```
    [x y]=meshgrid(a:h:b,a:h:b);
```

for generating points in $[a,b]^2$ with spacing h. But these are not points in our matrix convention. Both `x` and `y` are matrices of the same shape, with identical columns or rows. Use

```
p=[x(:) y(:)];
```

to get a point matrix 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`.

1.2 Distance Matrices

Rdial kernels are usually evaluated on Euclidean distances. In 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, i.e. we want to calculate the *kernel matrix* A_{XY} with entries

$$\phi(\|x_j - y_k\|_2),\ 1 \le j \le M,\ 1 \le k \le N,$$

we have two input point matrices $X \in \mathbb{R}^{M\times d}$ and $Y \in \mathbb{R}^{N\times d}$ to generate an $M \times N$ matrix with square roots of entries

$$\|X^T e_i - Y^T e_j\|_2^2 = \|X^T e_i\|_2^2 + \|Y^T e_j\|_2^2 - 2e_i^T XY^T e_j$$

for $1 \le i \le M,\ 1 \le j \le N$. As a matrix, this is equal to $-2XY^T$ plus two matrices consisting of identical rows and columns formed by squared norms of points. Thus it is more efficient to calculate squared distances divided by two. Here is an m-file implementing the above formula in halved form, without any loops and with complexity $\mathcal{O}(MNd)$.

```
function dst=distsqh(p, q)
% calculates a np*nq matrix of halved squares of point distances
% with two point sets p and q of np and nq points each.
[np pdim]=size(p);
[nq qdim]=size(q);
if pdim~=qdim
     error('point sets of unequal dimension')
end
```

```
if pdim==1
    dst=(repmat(p,1,nq)-repmat(q',np,1)).^2/2;
    return
end
dst=p*q';
cp=sum((p.*p)')/2; % squared norms of p points, as row, halved
cq=sum((q.*q)')/2; % squared norms of q points, as row, halved
dst=repmat(cp',1,nq)+repmat(cq,np,1)-dst;
```

Note that this m-file does not do any scaling.

The standard way to build a kernel matrix thus is to form the squared and halved distances as above and to apply the unscaled kernel in the form

$$\begin{aligned} K(x,y) &:= f(\|x-y\|_2^2/2) \\ &= f(s), \\ s &:= \|x-y\|_2^2/2 \end{aligned}$$

using the function $f(s) = \phi(\sqrt{2s})$ such that $\phi(r) = f(r^2/2)$. See section 1.3.3 for the use of radial kernels in f–form. We shall use the term *S–matrix* for matrices of halved squares of point distances.

1.3 Kernel Evaluation

(`SecSubPBKE`) Radial kernels should be evaluated on halved squares of distances, as we saw, and in MATLAB they should be applicable elementwise to a full matrix. This is rather easy to do, but there are some precautions.

1.3.1 Avoiding Singularities

(`SecSUBPBKEAS`) The thin–plate spline and certain kernels involving Bessel functions have singularities at the origin. 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. *But this is still to be done ...*

1.3.2 Truncated Powers

(`SecSUBPBKETP`) 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(c)),c).^k
```

in order to avoid pitfalls and loops.

1.3.3 Kernel Routines

(`SecSUBPBKEKR`) Our standard way to calculate with kernels of the above form is to call a MATLAB m-file `frbf.m` of the form

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

which takes a matrix `dmat` of halved squared distances and evaluates the k–th derivative of the kernel on it, resulting in a matrix `resmat` of the same form as `dmat`. Users are strongly advised not to use these routines for single points. They are tailored for use on middle-size matrices. Furthermore, there are no precautions so far against evaluation of kernels for illegal parameter choices. These parameters are global MATLAB variables

```
    RBFtype

    RBFpar

    RBFscale
```

to be explained in what follows.

Control of the scaling factor c is not done within `frbf`. It is usually done via the global variable `RBFscale`, but since the m–file `distsqh.m` does not scale points, one has to apply the scaling before application of `frbf` and after `distsqh.m`. An additional parameter is defined by `RBFpar` depending on the type of kernel, while the type of kernel is selected by setting `RBFtype` to a MATLAB string like `'g'` for the Gaussian. The list of current options for `RBFtype` is

- g Gaussian $\phi(r) = \exp(-r^2/2)$ with $f(s) = \exp(-s)$, all $k \in \mathbb{N}_0$, no `RBFpar`

- `mq` Multiquadric $\phi(r) = (1 + r^2/2)^{\beta/2}$ with $f(s) = (1 + s)^{\beta/2}$, all $k \in \mathbb{N}_0$, with `RBFpar`$=\beta$. Inverse multiquadrics can be treated by choosing `RBFpar`$=\beta$ negative.
- `p` Powers $\phi(r) = r^\beta$ with $f(s) = (\sqrt{2s})^\beta$, all $k \in \mathbb{N}_0$, with `RBFpar`$=\beta$.
- `ms` Matern/Sobolev $\phi(r) = K_\nu(r)r^\nu$, all $k \in \mathbb{N}_0$, with `RBFpar`$=\nu$.
- `w` all C^{2m} Wendland functions $\phi_{3,m}$ for $m =$`RBFpar`≥ 0, all $k \in \mathbb{N}_0$
- `tp` Thin–plate splines $\phi(r) = r^{2m} \log r$ for integer $2m =$`RBFpar`> 0, all $k \in \mathbb{N}_0$.

Note that currently there are no polynomials added in case of conditional positive definiteness of positive order. Furthermore, the Wendland function class is currently restricted to kernels working in $\mathbb{R}^3$. The program `frbf.m` just calculates the formula for the k–th derivative of the kernel and it does not care for restrictions on the kernel parameters and the existence of the required derivatives. A full listing is here:

```
function y =  frbf (s, k)
% k-th derivative of standard rbf kernel in f form, i.e.
% as a function of s=r^2/2.
global RBFtype;   % 'g'=Gaussian, 'mq' = Multiquadric etc..
global RBFpar;    %  parameter depending on RBFtype
switch lower(RBFtype)
    case ('g')   % 'g'=Gaussian,
        if mod(k,2)==0
            y=exp(-s);
        else
            y=-exp(-s);
        end
    case ('mq') % 'mq' = Multiquadric, inverse or not...
        fac=1;
        ord=k;
        par=RBFpar;
        while ord>0
            ord=ord-1;
            fac=fac*par;
            par=par-2;
        end
        y=fac*(1+2*s).^(par/2);
```

```
case ('p')  % powers
    fac=1;
    ord=k;
    par=RBFpar;
    while ord>0
        ord=ord-1;
        fac=fac*par;
        par=par-2;
    end
    y=fac*(2*s+eps).^(par/2);
case ('tp')  % thin-plate
    fac=1;
    ord=k;
    par=RBFpar;
    su=0;
    while ord>0
        ord=ord-1;
        if ord==k-1
            su=1;
        else
            su=su*par+fac;
        end
        fac=fac*par;
        par=par-2;
    end
    y=(2*s+eps).^(par/2);
    y=fac*y.*log(2*s+eps)/2 +su*y;
case ('ms')  % Matern/Sobolev
    y=(-1)^k*besselk(RBFpar-k,sqrt(2*s+eps)).*...
                  (sqrt(2*s+eps)).^(RBFpar-k);
case ('w') % Wendland functions.
    % we use only those which are pos. def.
    % in dimension at most 3.
    [coeff, expon]=wendcoeff(3+2*k, RBFpar-k);
    r=sqrt(2*s);
    ind=find(r<=1);
    u=zeros(size(ind));
    sp=ones(size(ind));
    sloc=r(ind);
    for i=1:length(coeff)
        u=u+coeff(i)*sp;
```

```
            sp=sp.*sloc;
        end
        u=u.*(1-sloc).^expon;
        y=zeros(size(s));
        y(ind)=(-1)^k*u;
    otherwise
        error('RBF type not implemented')
end
```

1.4 Recursive Scalar Radial Derivatives

This section describes how the derivatives of unscaled radial kernels in the f form can be calculated. Thus it is an explanation of how `frbf(dmat,k)` works for positive derivative orders `k`.

We write an unscaled radial kernel as

$$K(\mathbf{x},\mathbf{y}) := f(\|\mathbf{x}-\mathbf{y}\|_2^2/2).$$

Besides a simplified calculation of distance matrices, this has certain advantages when calculating derivatives. But later we shall scale via

$$K_c(\mathbf{x},\mathbf{y}) := f(\|\mathbf{x}-\mathbf{y}\|_2^2/(2c^2)) = K(\mathbf{x}/c,\mathbf{y}/c)$$

with c being defined by the global variable `RBFscale`. This lets c act like a support radius, in particular when we let compactly supported unscaled radial kernels K vanish for $\|\mathbf{x}-\mathbf{y}\|_2 > 1$.

1.4.1 Gaussian

The Gaussian $K(x,y) = \exp(-\|x-y\|_2^2/2)$ uses $f(s) = \exp(-s)$ with simplest possible derivatives.

1.4.2 Multiquadrics

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

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

allowing to work with arbitrary k for $\beta \notin 2\mathbb{N}_0$. Note that this includes inverse multiquadrics.

1.4.3 Polyharmonic kernels

For the powers $\phi_\beta(r) := r^\beta$ with $\beta > 0,\ \beta \notin 2\mathbb{Z}$ we get

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

and for the thin–plate splines $\phi_\beta(r) := r^\beta \log(r)$ with $\beta \in 2\mathbb{Z}$ we find

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

Note that the second term is a polynomial in s.

1.4.4 Matern/Sobolev kernels

These are

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

with the Bessel function 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

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

Then

$$\begin{array}{rcl} f'_\nu(s) & = & K'_\nu(\sqrt{2s})\frac{1}{\sqrt{2s}}(\sqrt{2s})^\nu + K_\nu(\sqrt{2s})\nu(\sqrt{2s})^{\nu-1}\frac{1}{\sqrt{2s}} \\ & = & K'_\nu(\sqrt{2s})(\sqrt{2s})^{\nu-1} + \nu K_\nu(\sqrt{2s})(\sqrt{2s})^{\nu-2} \\ & = & \left(-K_{\nu-1}(\sqrt{2s}) - \frac{\nu}{\sqrt{2s}}K_\nu(\sqrt{2s})\right)(\sqrt{2s})^{\nu-1} + \nu K_\nu(\sqrt{2s})(\sqrt{2s})^{\nu-2} \\ & = & -K_{\nu-1}(\sqrt{2s})(\sqrt{2s})^{\nu-1} \\ & = & -f_{\nu-1}(s). \end{array}$$

1.4.5 Wendland functions

We now handle the Wendland [6, 7], `wendland:1995-1,wendland:2005-1` kernels, but note that we now have to be careful with constant factors. First, we rewrite the dimension walk operator [7], `wendland:2005-1`

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

The Wendland functions are defined via

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

and we rewrite them in the form

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

Then

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

is a derivative recursion which is easy to implement if the standard basis functions are available. We shall describe below how those can be calculated in general.

Note that the multiplicative 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:

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

An easy way to get high–degree Wendland functions with correct constants for derivatives is to start with some polynomial $w_\ell(r) := (1-r)^\ell$ and then to repeat the MAPLE statement

```
w:=-int(r*w,r);w:=factor(w-subs(r=1,w));
```

couple of times. The second part sets the correct integration constant to let the resulting polynomial vanish at 1. This process generates the above sequence backwards when started with $22176(1-r)^5$. To get $\phi_{d,k}$ one has to start with $w_\ell(r) := (1-r)^\ell$ for $\ell = \lfloor d/2 \rfloor + k + 1$ and perform k steps. The result is C^{2k} and positive definite in dimensions up to d. With `CodeGeneration` features, one can generate appropriate code for standard programming languages.

If we start from $w_\ell(r) := (1-r)^\ell$ and perform the above operation k times, one can see the above process as starting from the polynomial $p_0 = 1$ and proceeding inductively via

$$\int_r^1 s(1-s)^{\ell+n} p_n(s) ds = (1-r)^{\ell+n+1} p_{n+1}(r)$$

for $n = 0, \ldots, k-1$. This means

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

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

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

backwards for $j = n+1, \ldots, 0$. If the basis functions are evaluated on large matrices, the above $\mathcal{O}(k^2)$ 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,k)
% calculates coefficients and exponent for polynomial
% part p_{d,k}(r) of Wendland functions
% phi_{d,k}(r)=p_{d,k}(r)*(1-r)^expon
expon=floor(d/2)+k+1;
coeff=zeros(k+1,1);
coeff(1,1)=1;
for n=0:k-1
    coeff(n+2,1)=coeff(n+1,1)/(n+expon+2);
    for j=n+1:-1:2
        coeff(j,1)=(j*coeff(j+1,1)+coeff(j-1,1))/(expon+j);
    end
    expon=expon+1;
    coeff(1,1)=coeff(2,1)/expon;
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`). Note that the calculation of Wendland kernels acts only on points in the support, once they are found.

1.4.6 Remark

It seems to be a strange fact that these 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.

1.5 Multivariate Polynomials

(`SecSUBMVP`) 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^α for all j, $1 \leq j \leq M$ and all multiindices $\alpha \in \mathbb{Z}_0^d$ with $0 \leq |\alpha| := \|\alpha\|_1 < m$ 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 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....
polvalues= points(:,dim).^(order-1);
for i=2:order-1
    pe=polynomials_exact_order(points(:,1:dim-1),i);
    pp=points(:,dim).^(order-i);
    [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)];
```

These programs need enhancement wrt. to speed and different bases, e.g. Chebyshev bases.

2 Interpolation and Evaluation

(`SecSUBPBKEIE`) A standard square kernel–based interpolation system

$$\sum_{j=1}^{M} K(x_k, x_j)a_j = y_k,\ 1 \le k \le M$$

for unconditionally positive definite radial kernels is usually set up from a point matrix `X` via

```
intmat=frbf(distsqh(X,X)/RBFscale^2,0);
```

provided that the controls for the kernel are defined and the kernel is positive definite. We can use a standard MATLAB routine `kermat.m` for this, using `intmat=kermat(X,X)`:

```
function mat=kermat(X, Y)
% creates kernel matrix
% for two point sets X and Y
global RBFscale
[nx dx]=size(X);
[ny dy]=size(Y);
if dx~=dy
    error('Unequal space dimension for kermat arguments');
end
mat=frbf(distsqh(X,Y)/RBFscale^2,0);
```

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

```
a=intmat\y;
```

but it is always a good idea to check the scaling by preceding this with

```
condition=condest(intmat)
```

to avoid problems.

After finding the coefficient vector, one would usually like to evaluate the solution, e.g. for subsequent plotting. This will 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)));
```

A full sample listing of the m-file `testint.m` for interpolation of the MATLAB `peaks` function in $[-3, 3]^2$ is here:

```
clear all;
close all;
global RBFtype;
global RBFpar;
global RBFscale;
rand('state',0)
RBFscale=0.5
RBFtype='g'
RBFpar=3.5
np=15   % results in np^2 points
p=6*rand(np*np,2)-3; % use this for random points
% or the next two statements for regular points
% [xp yp]=meshgrid(-3:6/(np-1):3,-3:6/(np-1):3);
% p=[xp(:) yp(:)];
subplot(2,2,1)
plot(p(:,1),p(:,2),'.')
title('Interpolation points')
z=peaks(p(:,1),p(:,2));
[xe ye]=meshgrid(-3:0.06:3,-3:0.06:3);
pe=[xe(:) ye(:)];
ze=peaks(xe, ye);
imat=frbf(distsqh(p,p)/RBFscale^2,0);
condition=condest(imat)
coeff=imat\z;
emat=frbf(distsqh(pe,p)/RBFscale^2,0);
val=emat*coeff;
subplot(2,2,2)
surfc(xe,ye,ze)
shading interp
title('Given function')
subplot(2,2,3)
surfc(xe,ye,reshape(val,size(xe)));
shading interp
title('Reconstructed function')
subplot(2,2,4)
surfc(xe,ye,ze -reshape(val,size(xe)));
shading interp
title('Error')
```

The results are depicted in Figure 1. For conditionally positive definite ker-

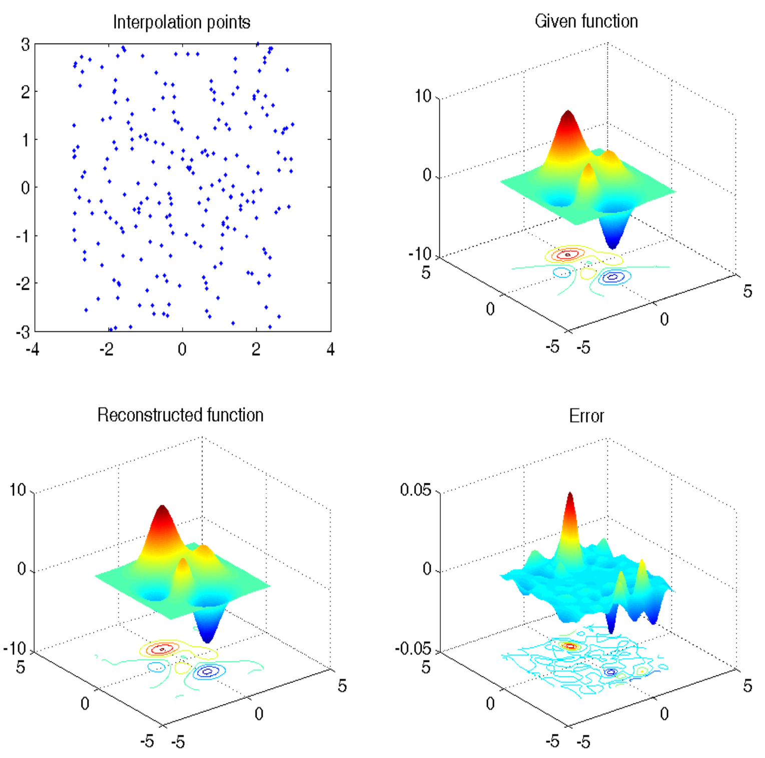

Figure 1: Resulting figure for `testint.m`

nels of order m and interpolation in M points forming a point matrix `X`, 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 $p_1,\ldots,p_Q$. The interpolation data $y_1,\ldots,y_M$ 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 below, with the result in Figure 2. See how the program testint.m was slightly modified to work in the conditionally positive definite case.

```
clear all;
close all;
global RBFtype;
global RBFpar;
global RBFscale;
rand('state',0)
RBFscale=0.5
RBFtype='tp'
RBFpar=2
order=2
np=15   % results in np^2 points
% p=6*rand(np*np,2)-3; % use this for random points
% or the next two statements for regular points
[xp yp]=meshgrid(-3:6/(np-1):3,-3:6/(np-1):3);
p=[xp(:) yp(:)];
subplot(2,2,1)
plot(p(:,1),p(:,2),'.')
title('Interpolation points')
z=peaks(p(:,1),p(:,2));
[xe ye]=meshgrid(-3:0.06:3,-3:0.06:3);
pe=[xe(:) ye(:)];
fe=peaks(xe, ye);
imat=frbf(distsqh(p,p)/RBFscale^2,0);
pmat=polynomials(p,order);
[npp q]=size(pmat);
amat=[imat pmat; pmat' zeros(q,q)];
condition=condest(amat)
ze=[z; zeros(q,1)];
coeff=amat\ze;
emat=frbf(distsqh(pe,p)/RBFscale^2,0);
pemat=polynomials(pe,order);
val=[emat pemat]*coeff;
subplot(2,2,2)
surfc(xe,ye,fe)
shading interp
```

```
title('Given function')
subplot(2,2,3)
surfc(xe,ye,reshape(val,size(xe)));
shading interp
title('Reconstructed function')
subplot(2,2,4)
surfc(xe,ye,fe -reshape(val,size(xe)));
shading interp
title('Error')
```

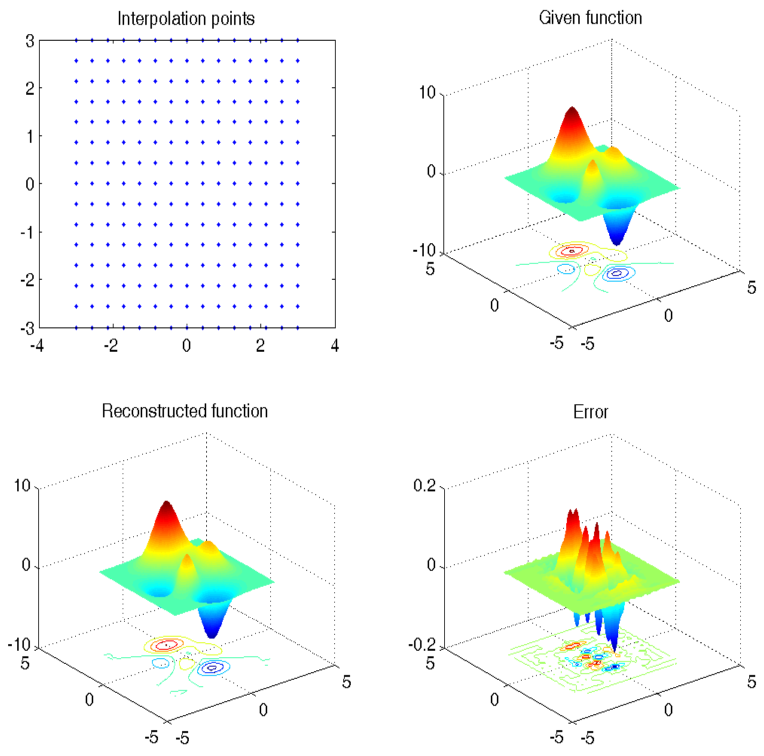

Figure 2: Resulting figure for `testintCPD.m`

2.1 Lagrange Bases

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

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

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

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

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

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

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 [5].

2.2 Power Functions

(`SecSUBPBKEPF`) Once the Lagrange basis is at hand, one can calculate the square of the optimal Power function [7] (`eqpowSPD`)

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

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, and in the stochastic setting 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. 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. [2].

Here is a program `testlag.m` for Lagrange bases and Power Functions in the unconditionally positive definite case, and its output is in Figure 3.

```
clear all;
close all;
global RBFtype;
global RBFpar;
global RBFscale;
rand('state',0)
RBFscale=0.7
RBFtype='g'
RBFpar=3.5
np=11
% p=6*rand(np*np,2)-3;
[xp yp]=meshgrid(-3:6/np:3,-3:6/np:3);
p=[xp(:) yp(:)];
[xe ye]=meshgrid(-3:0.06:3,-3:0.06:3);
pe=[xe(:) ye(:)];
intmat=frbf(distsqh(p,p)/RBFscale^2,0);
condition=condest(intmat)
evalmat=frbf(distsqh(pe,p)/RBFscale^2,0);
umat=(intmat\evalmat')';
k=floor((np*(np+1)/2))
subplot(1,3,1)
surfc(xe,ye,reshape(umat(:,k),size(xe)))
```

```
shading interp
axis square
title(sprintf('Lagrange function %d ',k))
powval=frbf(0,0)-sum((evalmat.*umat)')';
subplot(1,3,2)
surfc(xe,ye,reshape(powval,size(xe)));
shading interp
axis square
title('P. f.')
subplot(1,3,3)
plot(p(:,1),p(:,2),'.',p(k,1),p(k,2),'o')
title('Points')
axis square
```

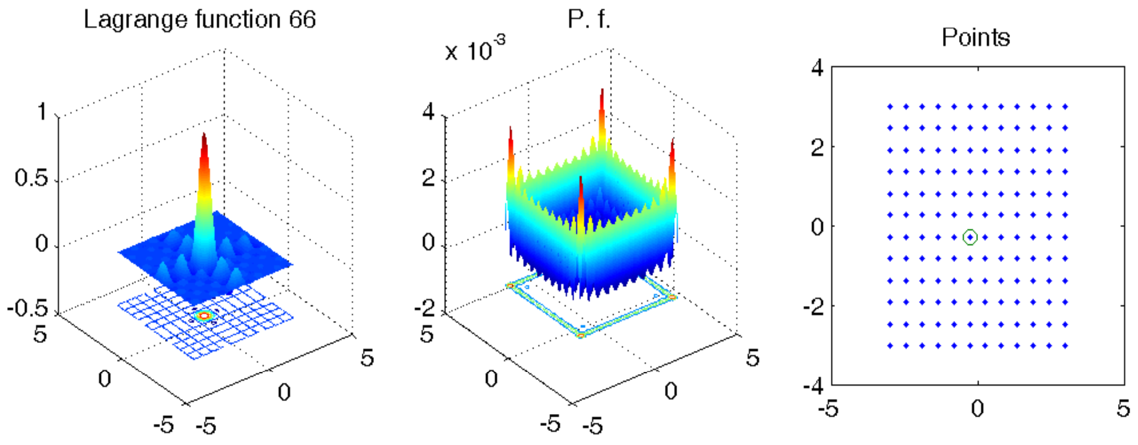

Figure 3: Resulting figure for `testlag.m`

Note that if the functions u_j are not the standard Lagrange basis using K, one has to use the formula (eqpowgen)

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

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 [1].

In the conditionally positive definite case, the Lagrange basis $u_1, \ldots, u_M$ has to be extended by additional functions $v_1, \ldots, v_Q$ 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

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

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 (2, `eqpowgen`). In the unconditionally positive definite case, the quadratic term cancels with one of the linear terms, thus simplifying to (1, `eqpowSPD`). In matrix form,

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

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

$$\begin{array}{rclcll} \texttt{umat} & = & (u_j(y_k)) & = & U_Y^T, & 1 \le k \le N,\ 1 \le j \le M \\ \texttt{vmat} & = & (v_i(y_k)) & = & V_Y^T, & 1 \le k \le N,\ 1 \le i \le Q \\ \texttt{evalmat} & = & (K(y_k, x_j)) & = & A_{Y,X}, & 1 \le k \le N,\ 1 \le j \le M \\ \texttt{pmat} & = & (p_i(x_j)) & = & P_X, & 1 \le j \le M,\ 1 \le i \le Q \end{array}$$

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

$$\begin{array}{rl} & f(0) - u^T(x)K_X(x) - u^T(x)P_X v(x) \\ = & \texttt{frbf(0,0)-sum((umat.*(evalmat+vmat*pmat'))')';} \end{array}$$

Here is an analogous program `testlagCPD.m` for the conditionally positive definite case, and its output is in Figure 4. Note the changes from `testlag.m`.

```
clear all;
close all;
global RBFtype;
global RBFpar;
global RBFscale;
rand('state',0)
RBFscale=2
RBFtype='tp'
RBFpar=2
order=2
np=11
% p=6*rand(np*np,2)-3;
[xp yp]=meshgrid(-3:6/np:3,-3:6/np:3);
p=[xp(:) yp(:)];
[xe ye]=meshgrid(-3:0.06:3,-3:0.06:3);
pe=[xe(:) ye(:)];
imat=frbf(distsqh(p,p)/RBFscale^2,0);
pmat=polynomials(p,order);
[npp q]=size(pmat);
amat=[imat pmat; pmat' zeros(q,q)];
condition=condest(amat)
pemat=polynomials(pe,order);
[nepp q]=size(pemat);
evalmat=frbf(distsqh(pe,p)/RBFscale^2,0);
cmat=amat\[evalmat' ; pemat'];
umat=cmat(1:npp,1:nepp)';
vmat=cmat(npp+1:end,1:nepp)';
k=floor((np*(np+1)/2))
subplot(1,3,1)
surfc(xe,ye,reshape(umat(:,k),size(xe)))
shading interp
axis square
title(sprintf('Lagrange function %d ',k))
powval=frbf(0,0)-sum((umat.*(evalmat+vmat*pmat'))')';
subplot(1,3,2)
surfc(xe,ye,reshape(powval,size(xe)));
shading interp
axis square
title('P. f.')
subplot(1,3,3)
plot(p(:,1),p(:,2),'.',p(k,1),p(k,2),'o')
title('Points')
axis square
```

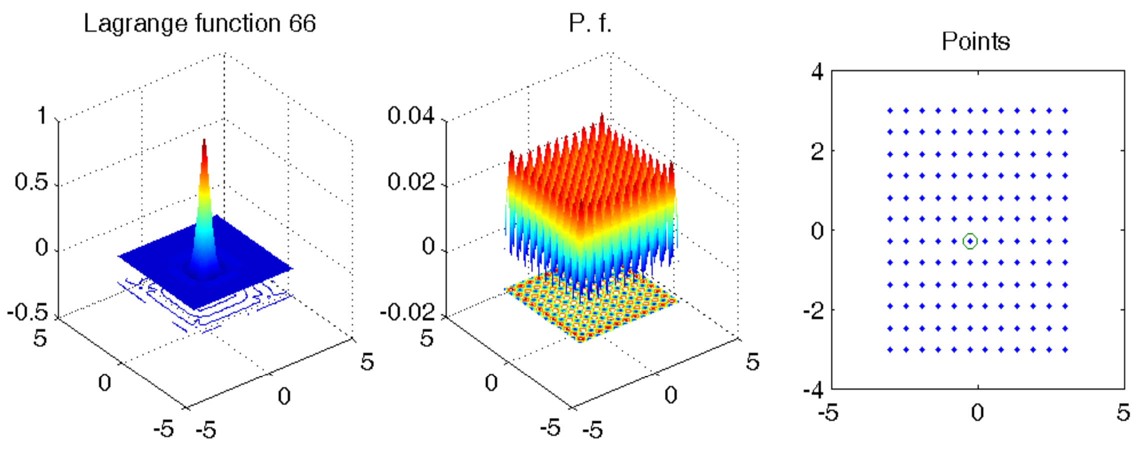

Figure 4: Resulting figure for `testlagCPD.m`

3 Explicit Multivariate Derivatives

This section is of quite some importance when radial basis functions are used for solving partial differential equations. On needs plenty of derivatives of radial kernels, and one has to care for scaling and geometry.

We assume x, y to be vectors in $\mathbb{R}^d$. We define

$$\begin{aligned} K_c(x,y) &:= f(\|x-y\|_2^2/2c^2) \\ &= f(s), \\ s &:= \|x-y\|_2^2/2c^2 \end{aligned}$$

using a positive scale factor c. We also assume M points for the x argument and N points for the y argument, stored as rows of matrices $X \in \mathbb{R}^{M\times d}$ and $Y \in \mathbb{R}^{N\times d}$. Furthermore, we assume a matrix

$$S^{XY} := \left(\|X^T e_i - Y^T e_j\|_2^2/2c^2\right)_{1\le i\le M,\ 1\le j\le N}$$

to be precalculated, e.g. by

```
SXY=distsqh(X,Y)/c^2
```

in MATLAB, and we shall provide formulae for evaluation of kernel derivatives on such a configuration. For each scalar derivative, we thus have to calculate an $M \times N$ matrix. Since `frbf` does not scale, we have to care for the scaling here.

Users will see that the following routines all boil down to calls of `frbf(S,k)` for various matrices `S` and derivative orders `k`. If the routines are used naively, there will be multiple calls for exactly the same `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.

3.1 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 $1 \leq i \leq M,\ 1 \leq j \leq N,\ 1 \leq k \leq d$:

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

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

```
function mat=gradkermatX(X, Y)
% creates kernel matrices
% for two point sets X and Y
% 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);
if dx~=dy
    error('Unequal space dimension for gradkermatX arguments');
end
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=gradkermatY(X, Y)
% creates kernel matrices
% for two point sets X and Y
% corresponding to the full gradient wrt. the Y variable.
% See gradkermatX for details.
mat=-gradkermatX(X,Y);
```

but in many applications one would prefer to call only one of these routines.

3.2 Normals

(`SecSubNormal`) Scalar normal or directional derivatives are prescribed via an additional matrix N of normals or directions as rows, with d columns. If the normals are evaluated on the X points, there are M normals, otherwise N. The pointwise case is

$$\frac{\partial}{\partial \nu_j} := \frac{\partial}{\partial \mathbf{n}_j} \quad := \quad \sum_{k=1}^{d} N_{jk}\frac{\partial}{\partial x_k}$$

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

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

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

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

We provide MATLAB routines for implementing the first formula:

```
function mat=normalXkermat(X, NX, Y)
% creates kernel matrices
% for two point sets X and Y
% corresponding to the normals NX wrt. the X variable.
% The result is a matrix of size  nx times ny.
global RBFscale
[nx dx]=size(X);
[ny dy]=size(Y);
if dx~=dy
    error('Unequal space dimension for normalXkermat arguments');
end
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)
% creates kernel matrices
% for two point sets X and Y
% corresponding to the normals NX wrt. the X variable.
% The result is a matrix of size  nx times ny.
mat=normalXkermat(Y, NY, X)';
```

3.3 Second derivatives

We start with the appropriate calculations:

$$\begin{aligned}
\frac{\partial}{\partial x_r}\frac{\partial}{\partial x_p}K_c(x,y) &= \frac{\partial}{\partial x_r}\left(\frac{f'(s)}{c^2}(x_p-y_p)\right)\\
&= \frac{f'(s)}{c^2}\frac{\partial}{\partial x_r}(x_p-y_p)+(x_p-y_p)\frac{\partial}{\partial x_r}\left(\frac{f'(s)}{c^2}\right)\\
&= \frac{f'(s)}{c^2}\delta_{rp}+\frac{x_p-y_p}{c^2}f''(s)\frac{\partial s}{\partial x_r}\\
&= \frac{f'(s)}{c^2}\delta_{rp}+f''(s)\frac{x_r-y_r}{c^2}\frac{x_p-y_p}{c^2}
\end{aligned}$$

$$\left(\frac{\partial}{\partial x_r}\frac{\partial}{\partial x_p}K_c(x,y)\right)_{ij} = \frac{f'(S^{XY}_{ij})}{c^2}\delta_{rp}+\frac{f''(S^{XY}_{ij})}{c^4}(X_{ir}-Y_{jr})(X_{ip}-Y_{jp})$$

$$\begin{aligned}
\frac{\partial}{\partial y_s}\frac{\partial}{\partial y_k}K_c(x,y) &= \frac{\partial}{\partial y_s}\left(-\frac{f'(s)}{c^2}(x_k-y_k)\right)\\
&= -\frac{f'(s)}{c^2}\frac{\partial}{\partial y_s}(x_k-y_k)-(x_k-y_k)\frac{\partial}{\partial y_s}\left(\frac{f'(s)}{c^2}\right)\\
&= \frac{f'(s)}{c^2}\delta_{sk}-\frac{x_k-y_k}{c^2}f''(s)\frac{\partial s}{\partial y_s}\\
&= \frac{f'(s)}{c^2}\delta_{sk}+f''(s)\frac{x_s-y_s}{c^2}\frac{x_k-y_k}{c^2}
\end{aligned}$$

$$\left(\frac{\partial}{\partial y_s}\frac{\partial}{\partial y_k}K_c(x,y)\right)_{ij} = \frac{f'(S^{XY}_{ij})}{c^2}\delta_{sk}+\frac{f''(S^{XY}_{ij})}{c^4}(X_{is}-Y_{js})(X_{ik}-Y_{jk})$$

$$\begin{aligned}
\frac{\partial}{\partial y_k}\frac{\partial}{\partial x_p}K_c(x,y) &= \frac{\partial}{\partial y_k}\left(\frac{f'(s)}{c^2}(x_p-y_p)\right)\\
&= \frac{f'(s)}{c^2}\frac{\partial}{\partial y_k}(x_p-y_p)+(x_p-y_p)\frac{\partial}{\partial y_k}\left(\frac{f'(s)}{c^2}\right)\\
&= -\frac{f'(s)}{c^2}\delta_{kp}+\frac{x_p-y_p}{c^2}f''(s)\frac{\partial s}{\partial y_k}\\
&= -\frac{f'(s)}{c^2}\delta_{kp}-f''(s)\frac{x_p-y_p}{c^2}\frac{x_k-y_k}{c^2}
\end{aligned}$$

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

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

3.4 Laplace operators

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

$$\begin{aligned}\Delta^x K_c(x,y) &= \frac{df'(s)}{c^2}+\frac{f''(s)}{c^4}\|x-y\|_2^2\\ &= \frac{df'(s)}{c^2}+\frac{2sf''(s)}{c^2}\\ \Delta^y K_c(x,y) &= \frac{df'(s)}{c^2}+\frac{f''(s)}{c^4}\|x-y\|_2^2\\ &= \frac{df'(s)}{c^2}+\frac{2sf''(s)}{c^2}\\ &=: g(s),\end{aligned}$$

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

In matrix form:

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

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

with a rather trivial implementation

```
function mat=laplacekermat(X, Y)
% creates Laplacian of kernel matrix
% for two point sets X and Y,
% The result is a matrix of size nx times ny
global RBFscale
[nx dx]=size(X);
[ny dy]=size(Y);
if dx~=dy
    error('Unequal space dimension for laplacekermat arguments');
end
s=distsqh(X,Y)/RBFscale^2;
mat=(dx*frbf(s,1)+2*s.*frbf(s,2))/RBFscale^2;
```

For later use, we collect derivatives of g:

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

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

3.5 Mixed Derivatives

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

3.5.1 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

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

and for matrices we get

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

Our MATLAB program is

```
function mat=normalXYkermat(X, NX, Y, NY)
% creates kernel matrices
% for two point sets X and Y
% corresponding to the normals NX wrt. the X variable
% and NY wrt. the Y variable.
% The result is a matrix of size  nx times ny.
global RBFscale
[nx dx]=size(X);
[ny dy]=size(Y);
if dx~=dy
    error('Unequal space dimension for normalXYkermat arguments');
end
smat=distsqh(X,Y)/RBFscale^2;
mat=-frbf(smat,1).*(NX*NY')/RBFscale^2-...
     frbf(smat,2)/RBFscale^4.*...
    (repmat(diag(NX*X'),1,ny)-NX*Y').*(X*NY'-repmat(diag(NY*Y')',nx,1));
```

3.5.2 Mixed Laplacians

(`SecSubMixedLaplacians`) Mixed scalar Laplacian values are

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

and they can easily be cast into matrix form by replacing s by S_{ij}^{XY}. Our MATLAB program is

```
function mat=laplaceXYkermat(X, Y)
% creates Laplacian of kernel matrix
% for two point sets X and Y,
% the Laplacian applied to BOTH arguments.
% The result is a matrix of size nx times ny
global RBFscale
[nx dx]=size(X);
[ny dy]=size(Y);
if dx~=dy
    error('Unequal space dimension for laplaceXYkermat arguments');
end
s=distsqh(X,Y)/RBFscale^2;
mat=(dx*(dx+2)*frbf(s,2)+4*(dx+2)*s.*frbf(s,3)+4*s.^2.*frbf(s,4))/RBFscale^4;
```

3.5.3 Mixed Normal and Laplacian

(`SecSubMixedNormalLaplacian`) The first scalar case is

$$\begin{aligned}
\frac{\partial}{\partial \nu_i}^{x} \Delta^y K_c(x,y) &= \sum_{k=1}^{d} N_{ik}^X \frac{\partial}{\partial x_k} \Delta^y K_c(x,y) \\
&= \sum_{k=1}^{d} N_{ik}^X \frac{\partial}{\partial x_k} g(s) \\
&= g'(s) \sum_{k=1}^{d} N_{ik}^X \frac{\partial s}{\partial x_k} \\
&= \frac{g'(s)}{c^2} \sum_{k=1}^{d} N_{ik}^X (x_k - y_k)
\end{aligned}$$

and in matrix form

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

with the implementation

```
function mat=laplaceYnormalXkermat(X, NX, Y)
```

```
% creates kernel matrices
% for two point sets X and Y
% corresponding to the normals NX wrt. the X variable
% and the Laplacian wrt. the Y variable
% The result is a matrix of size  nx times ny.
global RBFscale
[nx dx]=size(X);
[ny dy]=size(Y);
if dx~=dy
    error('Unequal space dimension for laplaceYnormalXkermat arguments');
end
S=distsqh(X,Y)/RBFscale^2;
gmat1=((dx+2)*frbf(S,2)+2*S.*frbf(S,3))/RBFscale^4;
mat=gmat1.*(repmat(diag(NX*X'),1,ny)-(NX*Y')  );
```

The other scalar case is

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

and in matrix form

$$\begin{aligned}
\left(\frac{\partial}{\partial\nu}^{y}\Delta^x K_c(x,y)\right)_{ij} &= -\frac{g'(S^{XY}_{ij})}{c^2}\sum_{k=1}^{d} N^Y_{jk}(X_{ik}-Y_{jk})\\
&= -\frac{g'(S^{XY}_{ij})}{c^2}\left((X(N^Y)^T)_{ij}-(N^Y Y^T)_{jj}\right)
\end{aligned}$$

with the implementation

```
function mat=laplaceXnormalYkermat(X, Y, NY)
% creates kernel matrices
% for two point sets X and Y
% corresponding to the normals NY wrt. the Y variable
% and the Laplacian wrt. the X variable
% The result is a matrix of size  nx times ny.
global RBFscale
[nx dx]=size(X);
[ny dy]=size(Y);
if dx~=dy
    error('Unequal space dimension for laplaceXnormalYkermat arguments');
end
S=distsqh(X,Y)/RBFscale^2;
gmat1=((dx+2)*frbf(S,2)+2*S.*frbf(S,3))/RBFscale^4;
mat=gmat1.*(repmat(diag(NY*Y')',nx,1)-(X*NY')  );
```

3.5.4 Derivatives of Normal Derivatives

(`SecSubMixedNormalGrad`) Mixed partials of normal derivatives in the scalar case for $1 \leq p \leq d$ are

$$\begin{aligned}
\frac{\partial}{\partial y_p}\frac{\partial}{\partial \nu_j}^x K_c(x,y) &= \frac{\partial}{\partial y_p}\left(\frac{f'(s)}{c^2}\sum_{k=1}^d N_{jk}(x_k-y_k)\right)\\
&= \left(\frac{\partial}{\partial y_p}\frac{f'(s)}{c^2}\right)\sum_{k=1}^d N_{jk}(x_k-y_k) - \frac{f'(s)}{c^2}N_{jp}\\
&= \frac{f''(s)}{c^2}\frac{\partial s}{\partial y_p}\sum_{k=1}^d N_{jk}(x_k-y_k) - \frac{f'(s)}{c^2}N_{jp}\\
&= -\frac{f''(s)(x_p-y_p)}{c^4}\sum_{k=1}^d N_{jk}(x_k-y_k) - \frac{f'(s)}{c^2}N_{jp}
\end{aligned}$$

$$\begin{aligned}
\frac{\partial}{\partial x_p}\frac{\partial}{\partial \nu_j}^y K_c(x,y) &= -\frac{\partial}{\partial x_p}\left(\frac{f'(s)}{c^2}\sum_{k=1}^d N_{jk}(x_k-y_k)\right)\\
&= -\left(\frac{\partial}{\partial x_p}\frac{f'(s)}{c^2}\right)\sum_{k=1}^d N_{jk}(x_k-y_k) - \frac{f'(s)}{c^2}N_{jp}\\
&= -\frac{f''(s)}{c^2}\frac{\partial s}{\partial x_p}\sum_{k=1}^d N_{jk}(x_k-y_k) - \frac{f'(s)}{c^2}N_{jp}\\
&= -\frac{f''(s)(x_p-y_p)}{c^4}\sum_{k=1}^d N_{jk}(x_k-y_k) - \frac{f'(s)}{c^2}N_{jp}
\end{aligned}$$

These formulas are not yet implemented in matrix form.

3.5.5 Derivatives of Laplace

(`SecSubMixedGradLap`) The partials of Laplace derivatives are in the scalar case

$$\begin{array}{rcl} \frac{\partial}{\partial x_j}\Delta^y K_c(x,y) & = & \frac{\partial}{\partial x_j} g(s) \\ & = & g'(s)\frac{\partial s}{\partial x_j} \\ & = & g'(s)\frac{x_j-y_j}{c^2} \\ \frac{\partial}{\partial y_k}\Delta^x K_c(x,y) & = & -g'(s)\frac{x_k-y_k}{c^2} \end{array}$$

These formulas are not yet implemented in matrix form.

4 Calculations for Poisson Problems

The following example describes how to set up a program that solves

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

via generalized Hermite interpolation [8], `wu:1992-1` or, equivalently, Kansa's collocation technique [4]. Technically, we just discretize these equations in points $x_j,\ y_k,\ z_\ell$ and do not care where the points lie. The program might also solve ill–posed problems where Dirichlet and Neumann data are prescribed on the same part of the boundary, or where function values are prescribed inside the domain. For a pure Neumann solver, one might add a single Dirichlet point for normalization of the solution. We use an unconditionally positive definite kernel throughout.

4.1 Matrices

We use three sets of points, each stored in a matrix:

1. points $x_j,\ 1\le j\le J$ for interpolation of Δ, arranged in a $J\times d$ matrix X,
2. points $y_k,\ 1\le k\le K$ for interpolation of function values, arranged in a $K\times d$ matrix Y,

3. points z_j, $1 \leq \ell \leq L$ for interpolation of directional derivatives, arranged in a $L \times d$ matrix Z, with directions ("normals") in another $L \times d$ matrix N.

We generate squared–distance matrices by the general rule

$$S^{AB} := \left(\|a_i - b_j\|_2^2/2c^2\right)_{i,j}$$

for the cases

$$S^{XX},\ S^{XY},\ S^{XZ},\ S^{YY},\ S^{YZ},\ S^{ZZ}.$$

If we collect all points into a matrix

$$P := (X^T, Y^T, Z^T)^T \in \mathbb{R}^{(J+K+L)\times d}$$

we have

$$S^{PP} = \begin{pmatrix} S^{XX} & S^{XY} & S^{XZ} \\ (S^{XY})^T & S^{YY} & S^{YZ} \\ (S^{XZ})^T & (S^{YZ})^T & S^{ZZ} \end{pmatrix}.$$

We do not store the S^{PP} matrix, but we need the above layout for the kernel matrix to arise in the next section.

4.2 Full system

The basic linear system will have a matrix of the form

$$\begin{pmatrix} A^{XX} & A^{XY} & A^{XZ} \\ (A^{XY})^T & A^{YY} & A^{YZ} \\ (A^{XZ})^T & (A^{YZ})^T & A^{ZZ} \end{pmatrix}$$

where the blocks need different derivatives of the kernel $K_c(x, y)$:

Matrix	Rows	Columns
A^{XX}	Δ on x for X	Δ on y for X
A^{XY}	Δ on x for X	Values on y for Y
A^{XZ}	Δ on x for X	Normals on y for Z
A^{YY}	Values on x for Y	Values on y for Y
A^{YZ}	Values on x for Y	Normals on y for Z
A^{ZZ}	Normals on x for Z	Normals on y for Z

A simple MATLAB implementation is

```
function [val, coeff]=fullpoissonsolver(X, fX, Y, fY, Z, NZ, fZ, E)
global RBFscale;
SXX=distsqh(X,X)/RBFscale^2;
SXY=distsqh(X,Y)/RBFscale^2;
SXZ=distsqh(X,Z)/RBFscale^2;
SYY=distsqh(Y,Y)/RBFscale^2;
SYZ=distsqh(Y,Z)/RBFscale^2;
SZZ=distsqh(Z,Z)/RBFscale^2;

% SPP=[ SXX SXY SXZ ; SXY' SYY SYZ ; SXZ' SYZ' SZZ];

AXX=laplaceXYkermat(X,X);
AYY=kermat(Y,Y);
AZZ=normalXYkermat(Z, NZ, Z, NZ);
AXY=laplacekermat(X,Y);
AXZ=laplaceXnormalYkermat(X,Z,NZ);
AYZ=normalYkermat(Y, Z, NZ);
AZZ=normalXYkermat(Z, NZ, Z, NZ);

A=[AXX AXY AXZ ; AXY' AYY AYZ; AXZ' AYZ' AZZ];
condA=condest(A)
sizA=size(A)
if condA >1.0e14
    error('Condition too large')
end
coeff=A\[fX; fY; fZ];
AEX=laplacekermat(E,X);
AEY=kermat(E,Y);
AEZ=normalYkermat(E,Z, NZ);
val=[AEX AEY AEZ]*coeff;
```

It also needs a point list `E` for evaluation, and the values on `E` will be in `val` while the coefficients wrt. the `A` matrix are in `coeff`. Here is a driver program `testfullpoissonsolver.m`:

```
clear all;
close all;
global RBFtype;
global RBFpar;
global RBFscale;
RBFscale=0.7
```

```
RBFtype='mq'
RBFpar=-2
dx=2
dy=dx;
dz=dx;
hx=0.1;
[xx, yx]=meshgrid(-1:hx:1,-1:hx:1);
X=[xx(:), yx(:)];
lenx=length(X(:,1))
hy=0.1;
yd=(-1:hy:1)';
Y=[yd -ones(length(yd),1) ; yd ones(length(yd),1)];
leny=length(Y(:,1))
Z=[-ones(length(yd),1) yd ; ones(length(yd),1) yd ];
lenz=length(Z(:,1))
NZ=[-ones(length(yd),1) zeros(length(yd),1) ;...
     ones(length(yd),1) zeros(length(yd),1) ] ;
fX=-2*cos(X(:,1)).*sin(X(:,2));
fY=   cos(Y(:,1)).*sin(Y(:,2));
fZ=  -sin(Z(:,1)).*sin(Z(:,2)).*NZ(:,1) ...
     +cos(Z(:,1)).*cos(Z(:,2)).*NZ(:,2);
he=0.05;
[xe,ye]=meshgrid(-1:he:1,-1:he:1);
E=[xe(:) ye(:)];
subplot(2,2,1);
plot(xx,yx,'b.',Y(:,1),Y(:,2),'rx',Z(:,1),Z(:,2),'go')
% legend('Laplace data','Dirichlet data','Neumann data')
hold on
quiver(Z(:,1),Z(:,2), NZ(:,1),NZ(:,2))
title('Data locations')
[val, coeff]=fullpoissonsolver(X, fX, Y, fY, Z, NZ, fZ, E);
subplot(2,2,2);
surf(xe, ye, reshape(val, size(xe)))
shading interp
title('Approximate solution')
dirval=[laplacekermat(Y,X) kermat(Y,Y) normalYkermat(Y,Z,NZ) ]*coeff;
dirichlet_norm=norm(dirval-fY)
zval=[laplacekermat(Z,X) kermat(Z,Y) normalYkermat(Z,Z,NZ) ]*coeff;
delta=0.0000001;
Zplus=Z+delta*NZ;
zneuval=[laplacekermat(Zplus,X) kermat(Zplus,Y) ...
```

```
         normalYkermat(Zplus,Z,NZ) ]*coeff;
neumann_norm_discretized=norm(fZ-(zneuval-zval)/delta)
subplot(2,2,3);
surf(xe, ye, cos(xe).*sin(ye))
shading interp
title('Exact solution')
subplot(2,2,4)
surf(xe, ye, cos(xe).*sin(ye)-reshape(val, size(xe)))
shading interp
title('Error')
```

Its output is in Figure 5. The exact solution is $u(x, y) = \cos(x)\sin(y)$. The appropriate Dirichlet and Neumann data are sampled on 42 points on each boundary line, and there are 441 sample points in the interior for interpolating the Laplacian. This makes a 525x525 matrix.

References

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

[2] Stefano 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.

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

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

[5] M. Pazouki and R. Schaback. Bases for kernel-based spaces. *Computational and Applied Mathematics*, 2011. to appear.

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

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

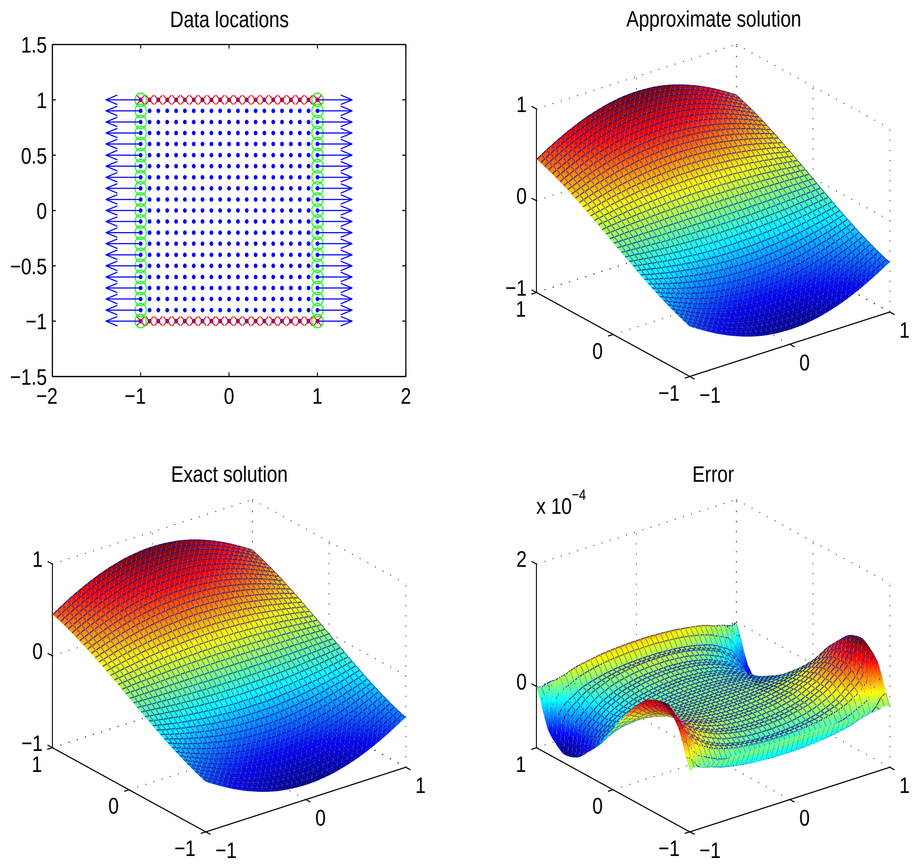

Figure 5: Resulting figure for `testfullpoissonsolver.m`

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